colesoft™

Z/XDC® COMMANDS

Z/XDC® Release V3.1 for z/OS

David B. Cole

Z/XDC®

Z/XDC® is a member of the XDC® (Extended Debugging Controller®) family of products
COPYRIGHT "Izzi Software LLC" - 2010-2026. ALL RIGHTS RESERVED

Maintenance Level CLC-2610M
Generated 11/03/2026, 11:11pm

online PDF Documents

PREFACE

PROPRIETARY LEGEND

Z/XDC® and its documentation (collectively, "Product"), including copies thereof, are the property of Izzi Software DBA Colesoft ("Owner"). Use of the product is licensed from ColeSoft Marketing, Inc. ("Licensor").

The Product may be used only by those organizations that are licensed by Licensor for such use and only in the manner so licensed. The Product may not be published, reproduced, distributed or made available to third parties for any purpose without the expressed written permission of Owner or Licensor. However, a reasonable number of copies may be made of the documentation (including the copyright notices thereon) as is necessary for the legitimate use of the Product within a licensed organization ("Customer").

Except as may be otherwise expressed in a signed agreement between Licensor and Customer, Owner and Licensor make no representations or warranties, expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose, the warranty of freedom from rightful claims by way of infringement and the like, and any warranty as to accuracy.

WARNING! Z/XDC® is a powerful tool for dynamically locating and correcting malfunctions in actively executing user programs and operating system routines. Accordingly, it is inherent in its design, that unless the use of this Product is properly controlled, then under certain conditions a malicious or careless user can use the Product to alter, subvert, counterfeit, damage or otherwise disturb the normal execution of user programs or system routines including, under certain conditions, both its own and system security routines.

Therefore, even if advised of the possibility of loss or damages, under no circumstances shall Owner or Licensor be liable for any loss or damage whatsoever (including death) arising from the Product, whether such loss or damage be direct, indirect, consequential, special or otherwise. Further, neither Owner nor Licensor shall be obligated to indemnify in any manner against any person or organization for any loss of any kind or nature which the person or organization may experience, arising out of the use or misuse of the Product.

CONTACTING COLESOFT

The **Z/XDC®** family of products is marketed by **ColeSoft Marketing, Inc**. with its principal office in Charlottesville, Virginia. If you would like more information, please contact ColeSoft Marketing as follows:

Phone:	**540-456-8210**
E-Mail:	**sales@colesoft.com**
Home Page:	**www.colesoft.com**

Our Technical Support contacts are:

Phone:	**540-456-8210**
E-Mail:	**techsupt@colesoft.com**
Home Page:	**www.colesoft.com**
FTP site:	**ftp.colesoft.com**

Our Customer Services contacts are:

Phone:	**540-456-8210**
E-Mail:	**support@colesoft.com**
Home Page:	**www.colesoft.com**

Our snail mail address is:

Address: **ColeSoft Marketing, Inc.**
440 Monticello Ave Ste 1802
PMB 380764
Norfolk, Virginia 23510-2670
USA

ONLINE PRESENCE

ColeSoft Marketing maintains the following resources on the Internet:

[Home Page] ColeSoft's Home Page is www.colesoft.com. It provides the following services:

- General information about Z/XDC
- E-mail links to both Marketing, Technical Support, and Customer Services
- FTP links for uploading diagnostic information and other files to Technical Support
- Online product delivery
- Links permitting existing customers to download a full set of Z/XDC's documentation
- 24x7 self-service for temporary, short-term, license activation codes for use in D.R. tests and other emergencies
- Occasional blog posts publicizing newly implemented features and capabilities.

[YouTube] ColeSoft's YouTube page is at youtube.com/colesoftware. This is where you will find several "how to" videos describing various aspects of using ColeSoft products. This is a wonderful resource, particularly for new Customers.

TRADEMARKS

TFS™, **XDC-TFS™**, **CDF™**, **XDC-CDF™**, **FASM™**, **base/XDC™**, **c/XDC™**, **asm/XDC™** and **dump/XDC™** are trademarks of Izzi Software.

Extended Debugging Controller®, **XDC®**, and **Z/XDC®** are registered trademarks of Izzi Software.

Other brand and product names referenced in this document are trademarks or registered trademarks of their various holders. Use of their names herein is for identification purposes only.

PRODUCT MANUALS

Z/XDC's documentation consists of the manuals shown below. The manuals are provided in PDF format. All manuals can be downloaded from www.colesoft.com under Support.

Install Guide	Describes how to install Z/XDC
User Guide	Explains how to use Z/XDC
Commands Reference	Describes all of Z/XDC's commands
Messages Reference	Explains all numbered messages issued by Z/XDC
Maintenance and New Functionality Guide	Describes what is new if Z/XDC (updated constantly)
Primer	A basic tutorial to help the beginner get started with Z/XDC
Quick Reference	A comprehensive syntax reference for all of the Z/XDC commands

ADDITIONAL MANUALS

Z/XDC customers may make as many copies of this manual as they feel is necessary for the legitimate use of Z/XDC within their organization. Existing customers may download from our web site (www.colesoft.com/product-support/zxdc-support/zxdc-documentation) printable copies of all of Z/XDC's manuals. Each manual is available in PDF format.

Contents

PREFACE . . . ii
PROPRIETARY LEGEND . . . ii
CONTACTING COLESOFT . . . ii
ONLINE PRESENCE . . . iii
TRADEMARKS . . . iii
PRODUCT MANUALS . . . iii
ADDITIONAL MANUALS . . . iv
CONTENTS . . . v
INTRODUCTION . . . 1
A Roadmap . . . 1
ONLINE PRESENCE . . . 2
Built-in Help Panels . . . 5
Help COmmands . . . 5
Help COmmands Syntax . . . 8
Help COmmands Syntax ADdressexpressions . . . 9
Help COmmands Syntax ASids . . . 10
Help COmmands Syntax Breakpoints . . . 13
Help COmmands Syntax Breakpoints Families . . . 19
Help COmmands Syntax Breakpoints Conditions . . . 22
Help COmmands Syntax Breakpoints Conditions COUntdown . . . 24
Help COmmands Syntax Breakpoints Conditions Truefalse . . . 25
Help COmmands Syntax Breakpoints Conditions COMPArisons . . . 27
Help COmmands Syntax Breakpoints Conditions COMPOund . . . 31
Help COmmands Syntax Breakpoints Conditions Null . . . 33
Help COmmands Syntax Breakpoints Autocmds . . . 33
Help COmmands Syntax Characterstrings . . . 35
Help COmmands Syntax Escapecharacter . . . 38
Help COmmands Syntax NAmes . . . 39
Help COmmands Syntax NAmes Classic . . . 39
Help COmmands Syntax NAmes Uss . . . 40
Help COmmands Syntax NUmericdata . . . 40
Help COmmands Syntax Dsnames . . . 41
Help COmmands Syntax Patternmatch . . . 44
Help COmmands Syntax Stringdata . . . 45
Help COmmands Syntax Stringdata Hexadecimal . . . 46
Help COmmands Syntax Stringdata Character . . . 48
Help COmmands Syntax Stringdata Mixed . . . 49
Help COmmands Syntax Stringdata Decimal . . . 50
Help COmmands ? . . . 52
Help COmmands ADeferred . . . 52
Help COmmands ALarm . . . 56
Help COmmands ASterisk . . . 58
Help COmmands AT . . . 59
Help COmmands ATX . . . 59
Help COmmands CD . . . 59
Help COmmands COMmentary . . . 60
Help COmmands CONsole . . . 61
Help COmmands COPy . . . 61
Help COmmands CUrsor . . . 65
Help COmmands DEFine . . . 66

Help COmmands DEFine DDntrans . . . 68
Help COmmands DEFine DDntrans Moreabout . . . 73
Help COmmands DEFine DSectlib . . . 76
Help COmmands DEFine Mmefdsn . . . 78
Help COmmands DELete . . . 81
Help COmmands DELete Cache . . . 85
Help COmmands DELete DAtaset . . . 87
Help COmmands DELete DDntrans . . . 88
Help COmmands DELete DSectlibs . . . 90
Help COmmands DELete Equates . . . 91
Help COmmands DELete Hooks . . . 94
Help COmmands DELete Ilits . . . 96
Help COmmands DELete MAPLibs . . . 97
Help COmmands DELete MAPS . . . 100
Help COmmands DELete MMefdsns . . . 105
Help COmmands DELete MOdules . . . 107
Help COmmands DELete Proxytasks . . . 109
Help COmmands DISConnect . . . 112
Help COmmands DISPlay . . . 114
Help COmmands DMap . . . 118
Help COmmands DMap Load . . . 121
Help COmmands DMap Load Csectsandbindermaps . . . 134
Help COmmands DMap Clone . . . 135
Help COmmands DOwn . . . 137
Help COmmands DRop . . . 142
Help COmmands DUb . . . 144
Help COmmands ENd . . . 144
Help COmmands EQuate . . . 146
Help COmmands EQuate Autocloning . . . 153
Help COmmands EQuate Autocloning Example . . . 162
Help COmmands EQuate Ritargets . . . 164
Help COmmands EWhere . . . 164
Help COmmands EXec . . . 174
Help COmmands FInd . . . 176
Help COmmands FOrmat . . . 190
Help COmmands FReemain . . . 198
Help COmmands GEtmain . . . 200
Help COmmands GO . . . 203
Help COmmands GO Nowhere . . . 208
Help COmmands GOT . . . 210
Help COmmands GOT Nowhere . . . 215
Help COmmands GOX . . . 217
Help COmmands GOX Nowhere . . . 223
Help COmmands HDeferred . . . 225
Help COmmands HElp . . . 229
Help COmmands HElp Absolute . . . 230
Help COmmands HElp Relative . . . 231
Help COmmands HElp Relative Examples . . . 233
Help COmmands HElp Mixing . . . 233
Help COmmands HKeys . . . 234
Help COmmands HOok . . . 235
Help COmmands IPcs . . . 243
Help COmmands ISpf . . . 245
Help COmmands Keys . . . 246

Help COmmands LEft . . . 246
Help COmmands LIBrarylists . . . 247
Help COmmands LIBrarylists SYntax . . . 251
Help COmmands LIBrarylists Add . . . 253
Help COmmands LIBrarylists Checkpoint . . . 256
Help COmmands LIBrarylists Checkpoint Restore . . . 257
Help COmmands LIBrarylists DEFault . . . 257
Help COmmands LIBrarylists DELete . . . 259
Help COmmands LIBrarylists REDirect . . . 263
Help COmmands LIBrarylists RESet . . . 267
Help COmmands LIBrarylists SHow . . . 268
Help COmmands LIBrarylists Test . . . 272
Help COmmands LIBrarylists Test Match . . . 272
Help COmmands LIBrarylists Test Try . . . 273
Help COmmands LICense . . . 275
Help COmmands LICense Panel . . . 276
Help COmmands LICense Data . . . 278
Help COmmands LISt . . . 280
Help COmmands LISt ACCESSLists . . . 285
Help COmmands LISt ACCESSRegisters . . . 287
Help COmmands LISt ACCESSRegisters Individual . . . 288
Help COmmands LISt ACCESSRegisters Registerset . . . 290
Help COmmands LISt AL . . . 293
Help COmmands LISt ALEs . . . 293
Help COmmands LISt AMode . . . 293
Help COmmands LISt AR# . . . 294
Help COmmands LISt AREgs . . . 294
Help COmmands LISt ASID . . . 294
Help COmmands LISt AUtomap . . . 296
Help COmmands LISt BAng . . . 296
Help COmmands LISt BEA . . . 297
Help COmmands LISt BELl . . . 300
Help COmmands LISt BFr# . . . 301
Help COmmands LISt BPts . . . 301
Help COmmands LISt BRAnches . . . 301
Help COmmands LISt BREakpoints . . . 303
Help COmmands LISt CAChe . . . 304
Help COmmands LISt CANdet . . . 305
Help COmmands LISt CAPs . . . 306
Help COmmands LISt CDf . . . 306
Help COmmands LISt COLORs . . . 307
Help COmmands LISt COLOUrs . . . 308
Help COmmands LISt CONSoles . . . 308
Help COmmands LISt CONSoles Mcs . . . 311
Help COmmands LISt CONTrolregisters . . . 311
Help COmmands LISt CONTrolregisters Individual . . . 312
Help COmmands LISt CONTrolregisters Registerset . . . 315
Help COmmands LISt CR# . . . 317
Help COmmands LISt CREgs . . . 318
Help COmmands LISt CRH# . . . 318
Help COmmands LISt CRHRegs . . . 318
Help COmmands LISt CRW# . . . 318
Help COmmands LISt CRWRegs . . . 318
Help COmmands LISt CUrrent . . . 318

Help COmmands LISt CWd . . . 319
Help COmmands LISt CXDc . . . 320
Help COmmands LISt CXU . . . 320
Help COmmands LISt DBc640q . . . 320
Help COmmands LISt DDntrans . . . 321
Help COmmands LISt DFr# . . . 321
Help COmmands LISt DSEctlibs . . . 321
Help COmmands LISt DSPaces . . . 322
Help COmmands LISt DSTg . . . 325
Help COmmands LISt DUB . . . 327
Help COmmands LISt DUMp . . . 327
Help COmmands LISt DXdc . . . 327
Help COmmands LISt EAR# . . . 332
Help COmmands LISt EAREgs . . . 332
Help COmmands LISt EBea . . . 332
Help COmmands LISt ENq . . . 334
Help COmmands LISt EPSW . . . 337
Help COmmands LISt EPSWE . . . 339
Help COmmands LISt EQuates . . . 340
Help COmmands LISt ER# . . . 341
Help COmmands LISt EREgs . . . 341
Help COmmands LISt ERH# . . . 342
Help COmmands LISt ERHRegs . . . 342
Help COmmands LISt ERW# . . . 342
Help COmmands LISt ERWRegs . . . 342
Help COmmands LISt ESTaes . . . 342
Help COmmands LISt ESTaes Report . . . 344
Help COmmands LISt EXits . . . 346
Help COmmands LISt FEatures . . . 347
Help COmmands LISt FEatures Zos . . . 348
Help COmmands LISt FEatures Os390 . . . 352
Help COmmands LISt FIxed . . . 356
Help COmmands LISt FLC . . . 357
Help COmmands LISt FLOAT . . . 358
Help COmmands LISt FLOATIngpointregisters . . . 359
Help COmmands LISt FLOATIngpointregisters Individual . . . 360
Help COmmands LISt FLOATIngpointregisters Registerset . . . 362
Help COmmands LISt FOrmat . . . 365
Help COmmands LISt FPc . . . 367
Help COmmands LISt FR# . . . 368
Help COmmands LISt FREgs . . . 368
Help COmmands LISt GEneralregisters . . . 368
Help COmmands LISt GEneralregisters Individual . . . 369
Help COmmands LISt GEneralregisters Registerset . . . 372
Help COmmands LISt GSR# . . . 377
Help COmmands LISt GSREgisters . . . 377
Help COmmands LISt GSREgisters Individual . . . 378
Help COmmands LISt GSREgisters Registerset . . . 379
Help COmmands LISt HElp . . . 382
Help COmmands LISt HElp Examples . . . 384
Help COmmands LISt HFr# . . . 385
Help COmmands LISt HILight . . . 385
Help COmmands LISt HKeys . . . 386
Help COmmands LISt HOoks . . . 386

Help COmmands LISt ILC . . . 387
Help COmmands LISt ILIt . . . 388
Help COmmands LISt INtensity . . . 389
Help COmmands LISt ISPf . . . 390
Help COmmands LISt ISR@prim . . . 390
Help COmmands LISt Keys . . . 391
Help COmmands LISt LIcense . . . 391
Help COmmands LISt LKedmap . . . 394
Help COmmands LISt LKedmap Classes . . . 397
Help COmmands LISt LKedmap Sections . . . 398
Help COmmands LISt LOCAtion . . . 398
Help COmmands LISt LOCKs . . . 399
Help COmmands LISt LOG . . . 400
Help COmmands LISt LOGOnid . . . 401
Help COmmands LISt LSEs . . . 401
Help COmmands LISt LSTack . . . 401
Help COmmands LISt MAIntenance . . . 406
Help COmmands LISt MAPLibs . . . 406
Help COmmands LISt MAPS . . . 408
Help COmmands LISt MEmoryobjects . . . 408
Help COmmands LISt MMefdsns . . . 412
Help COmmands LISt MObjects . . . 413
Help COmmands LISt MSgs . . . 413
Help COmmands LISt NOtes . . . 414
Help COmmands LISt NT . . . 414
Help COmmands LISt OPErands . . . 416
Help COmmands LISt OPTimization . . . 418
Help COmmands LISt PARseorder . . . 418
Help COmmands LISt PC# . . . 419
Help COmmands LISt PEt . . . 425
Help COmmands LISt PFkeys . . . 426
Help COmmands LISt PGms . . . 426
Help COmmands LISt PGms Syntax . . . 428
Help COmmands LISt PGms Automaticequates . . . 432
Help COmmands LISt PGms Examples . . . 433
Help COmmands LISt PGms Report . . . 435
Help COmmands LISt PId . . . 441
Help COmmands LISt PRIMarysize . . . 441
Help COmmands LISt PRINt . . . 442
Help COmmands LISt PROfiles . . . 442
Help COmmands LISt PSW . . . 447
Help COmmands LISt PSWE . . . 451
Help COmmands LISt Qualifier . . . 454
Help COmmands LISt R# . . . 454
Help COmmands LISt RBs . . . 454
Help COmmands LISt READ . . . 456
Help COmmands LISt READEcho . . . 457
Help COmmands LISt REFrprot . . . 457
Help COmmands LISt REGIsters . . . 460
Help COmmands LISt REGS . . . 460
Help COmmands LISt REXx . . . 460
Help COmmands LISt RH# . . . 461
Help COmmands LISt RHRegs . . . 461
Help COmmands LISt RSa . . . 461

Help COmmands LISt RSa Types . . . 465
Help COmmands LISt RSa Headings . . . 466
Help COmmands LISt RSa FLags . . . 466
Help COmmands LISt RSa F1sa . . . 467
Help COmmands LISt RSa ERrors . . . 467
Help COmmands LISt RSa Doc . . . 468
Help COmmands LISt RSa EXamples . . . 469
Help COmmands LISt RW# . . . 470
Help COmmands LISt RWRegs . . . 470
Help COmmands LISt SCBs . . . 470
Help COmmands LISt SCReen . . . 470
Help COmmands LISt SEArchorder . . . 470
Help COmmands LISt SECOndarysize . . . 473
Help COmmands LISt SECUrity . . . 473
Help COmmands LISt SESsions . . . 474
Help COmmands LISt SIGnonwait . . . 474
Help COmmands LISt SIZeof . . . 475
Help COmmands LISt SRbs . . . 476
Help COmmands LISt SSCt . . . 476
Help COmmands LISt SSRbs . . . 479
Help COmmands LISt STATIstics . . . 481
Help COmmands LISt STATS . . . 481
Help COmmands LISt STEp . . . 481
Help COmmands LISt STGlayout . . . 482
Help COmmands LISt SUbpools . . . 482
Help COmmands LISt SUbpools Reports . . . 486
Help COmmands LISt SUbpools Reports Caption . . . 487
Help COmmands LISt SUbpools Reports Spid . . . 489
Help COmmands LISt SUbpools Reports Where . . . 489
Help COmmands LISt SUbpools Reports Used . . . 489
Help COmmands LISt SUbpools Reports Notes . . . 490
Help COmmands LISt SUbpools Reports Tcb# . . . 490
Help COmmands LISt SUbpools Examples . . . 491
Help COmmands LISt TAsks . . . 495
Help COmmands LISt TCbs . . . 498
Help COmmands LISt TErminal . . . 498
Help COmmands LISt TFs . . . 499
Help COmmands LISt TIMeout . . . 499
Help COmmands LISt TIOt . . . 500
Help COmmands LISt TPut . . . 502
Help COmmands LISt TRace . . . 503
Help COmmands LISt USErid . . . 504
Help COmmands LISt USS . . . 504
Help COmmands LISt VARIables . . . 505
Help COmmands LISt VARS . . . 507
Help COmmands LISt VDisplay . . . 507
Help COmmands LISt VEctorregisters . . . 507
Help COmmands LISt VEctorregisters Individual . . . 508
Help COmmands LISt VEctorregisters Registerset . . . 510
Help COmmands LISt VR# . . . 513
Help COmmands LISt VREgs . . . 513
Help COmmands LISt VSEttings . . . 514
Help COmmands LISt VSTack . . . 514
Help COmmands LISt WIndow . . . 515

Help COmmands LISt XDc . . . 516
Help COmmands LISt XMs . . . 516
Help COmmands LISt Zap . . . 517
Help COmmands LL . . . 518
Help COmmands LOAd . . . 518
Help COmmands LOCate . . . 521
Help COmmands LOG . . . 523
Help COmmands Map . . . 524
Help COmmands Map Syntax . . . 526
Help COmmands Map Bindermaps . . . 534
Help COmmands Map Privatelyloaded . . . 534
Help COmmands Map Assemblermaps . . . 537
Help COmmands Map CMAps . . . 540
Help COmmands Map Dwarffiles . . . 543
Help COmmands Map CMDdialogs . . . 544
Help COmmands Map Findingmapdata . . . 545
Help COmmands Map Examples . . . 547
Help COmmands Note . . . 549
Help COmmands Off . . . 550
Help COmmands PAnelid . . . 551
Help COmmands POst . . . 551
Help COmmands PRInt . . . 553
Help COmmands PRInt Help . . . 555
Help COmmands PROfile . . . 559
Help COmmands PROfile REAd . . . 560
Help COmmands PROfile Save . . . 564
Help COmmands PROfile RESet . . . 567
Help COmmands PROfile Omitted . . . 569
Help COmmands Quick$init . . . 569
Help COmmands REAd . . . 570
Help COmmands RECall . . . 577
Help COmmands REFresh . . . 577
Help COmmands RELease . . . 578
Help COmmands RETRIeve . . . 578
Help COmmands RETRIeve Usage . . . 581
Help COmmands RETRIeve Helpmode . . . 583
Help COmmands RETRIeve List . . . 583
Help COmmands RETRIeve List Editing . . . 585
Help COmmands RETRIeve List Commands . . . 586
Help COmmands RETRIeve List Pfkeys . . . 587
Help COmmands RETRIeve Examples . . . 588
Help COmmands RETRY . . . 590
Help COmmands REXx . . . 590
Help COmmands RIght . . . 592
Help COmmands SCAnlog . . . 593
Help COmmands SCRname . . . 596
Help COmmands SEt . . . 596
Help COmmands SEt ACCessto . . . 599
Help COmmands SEt ACCessto Dataspace . . . 600
Help COmmands SEt ASID . . . 601
Help COmmands SEt ASN . . . 603
Help COmmands SEt AUTOmap . . . 603
Help COmmands SEt AUTH . . . 604
Help COmmands SEt BAng . . . 607

Help COmmands SEt BEll . . . 608
Help COmmands SEt BKpt . . . 609
Help COmmands SEt BPt . . . 609
Help COmmands SEt BReakpoints . . . 609
Help COmmands SEt CAndet . . . 616
Help COmmands SEt COLORs . . . 617
Help COmmands SEt COLOUrs . . . 619
Help COmmands SEt CONsole . . . 619
Help COmmands SEt CUrrent . . . 620
Help COmmands SEt CXdc . . . 623
Help COmmands SEt DUB . . . 624
Help COmmands SEt DBc640q . . . 625
Help COmmands SEt DUMp . . . 626
Help COmmands SEt EXits . . . 629
Help COmmands SEt FLc . . . 630
Help COmmands SEt FOrmat . . . 631
Help COmmands SEt HICOLOR . . . 636
Help COmmands SEt HICOLOUR . . . 637
Help COmmands SEt HILight . . . 637
Help COmmands SEt HKeys . . . 638
Help COmmands SEt ILC . . . 640
Help COmmands SEt ILIt . . . 640
Help COmmands SEt INtensity . . . 641
Help COmmands SEt ISPf . . . 643
Help COmmands SEt ISR@prim . . . 644
Help COmmands SEt Keys . . . 645
Help COmmands SEt Keys Onekey . . . 645
Help COmmands SEt Keys Keyspanel . . . 647
Help COmmands SEt Keys Pfktofkey . . . 648
Help COmmands SEt Keys Fkeytopfk . . . 649
Help COmmands SEt Keys Special . . . 650
Help COmmands SEt LInes . . . 651
Help COmmands SEt LOCks . . . 652
Help COmmands SEt LOG . . . 654
Help COmmands SEt LOG Dataset . . . 657
Help COmmands SEt LOG SYsout . . . 660
Help COmmands SEt LOG SCrollarea . . . 662
Help COmmands SEt LOG Management . . . 664
Help COmmands SEt LOGOnid . . . 665
Help COmmands SEt Maplibs . . . 666
Help COmmands SEt NOBell . . . 669
Help COmmands SEt NOHICOLOR . . . 669
Help COmmands SEt NOHICOLOUR . . . 670
Help COmmands SEt NOIlc . . . 670
Help COmmands SEt NOReadecho . . . 671
Help COmmands SEt NOWTOR . . . 671
Help COmmands SEt Optimization . . . 671
Help COmmands SEt PARseorder . . . 673
Help COmmands SEt PFkeys . . . 674
Help COmmands SEt PRIMarysize . . . 674
Help COmmands SEt PRINt . . . 675
Help COmmands SEt PROFile . . . 678
Help COmmands SEt PROXytasks . . . 679
Help COmmands SEt PSW . . . 684

Help COmmands SEt PSW AMode . . . 686
Help COmmands SEt PSW ASc-mode . . . 687
Help COmmands SEt PSW Cc . . . 688
Help COmmands SEt PSW Ext . . . 690
Help COmmands SEt PSW Io . . . 691
Help COmmands SEt PSW Key . . . 691
Help COmmands SEt PSW Prog . . . 692
Help COmmands SEt PSW State . . . 693
Help COmmands SEt PSWE . . . 693
Help COmmands SEt Qualifier . . . 694
Help COmmands SEt READ . . . 696
Help COmmands SEt READEcho . . . 700
Help COmmands SEt REFrprot . . . 700
Help COmmands SEt REXx . . . 704
Help COmmands SEt SCreen . . . 706
Help COmmands SEt SECOndarysize . . . 706
Help COmmands SEt SECUrity . . . 706
Help COmmands SEt SIgnonwait . . . 708
Help COmmands SEt STep . . . 709
Help COmmands SEt TFs . . . 710
Help COmmands SEt TImeout . . . 711
Help COmmands SEt TPut . . . 711
Help COmmands SEt TRace . . . 712
Help COmmands SEt TRace Ignore . . . 713
Help COmmands SEt TRace Local . . . 714
Help COmmands SEt TRace Roll . . . 715
Help COmmands SEt TRace Trap2 . . . 716
Help COmmands SEt USErid . . . 717
Help COmmands SEt USS . . . 718
Help COmmands SEt VARIables . . . 719
Help COmmands SEt VARS . . . 719
Help COmmands SEt VDisplay . . . 719
Help COmmands SEt VSettings . . . 720
Help COmmands SEt WIndow . . . 721
Help COmmands SEt WIndow Create . . . 723
Help COmmands SEt WIndow Delete . . . 725
Help COmmands SEt WIndow Layouts . . . 726
Help COmmands SEt WIndow Retrieve . . . 727
Help COmmands SEt WIndow Vertical . . . 727
Help COmmands SEt WIndow Horizontal . . . 729
Help COmmands SEt WTOR . . . 731
Help COmmands SEt Zap . . . 731
Help COmmands SHow . . . 732
Help COmmands SHow Sizes . . . 736
Help COmmands SHow Cdp . . . 737
Help COmmands SHow Dbcnnn . . . 739
Help COmmands SPLIT . . . 740
Help COmmands SPLITV . . . 740
Help COmmands STep . . . 741
Help COmmands SWAp . . . 742
Help COmmands SWItch . . . 743
Help COmmands TDeferred . . . 744
Help COmmands THaw . . . 744
Help COmmands TRACe . . . 745

Help COmmands TRACe I . . . 748
Help COmmands TRACe B . . . 750
Help COmmands TRACe BY . . . 752
Help COmmands TRACe BNc . . . 754
Help COmmands TRACe STar . . . 758
Help COmmands TRACe SB/sa . . . 759
Help COmmands TRACe SA . . . 760
Help COmmands TRACe W . . . 760
Help COmmands TRAP . . . 764
Help COmmands TSo . . . 769
Help COmmands UP . . . 770
Help COmmands UNdub . . . 774
Help COmmands USing . . . 774
Help COmmands USing Dsectref . . . 778
Help COmmands USing Autocloning . . . 779
Help COmmands USing Examples . . . 787
Help COmmands Verify . . . 788
Help COmmands Verify Targets . . . 790
Help COmmands Verify Address . . . 793
Help COmmands Verify String . . . 796
Help COmmands WAit . . . 799
Help COmmands WHEre . . . 800
Help COmmands WHItespace . . . 810
Help COmmands Zap . . . 810
Help COmmands Zap Targets . . . 813
Help COmmands Zap Address . . . 816
Help COmmands Zap String . . . 819
Help COmmands Zap Boolean . . . 822
Help COmmands =jump . . . 824
Help COmmands % . . . 825
Help SCripts . . . 826
Help SCripts Ourscripts . . . 829
Help SCripts Ourscripts AUTOTBnc . . . 830
Help SCripts Ourscripts Dcbmaps . . . 832
Help SCripts Ourscripts Epmaps . . . 833
Help SCripts Ourscripts Ispsetup . . . 833
Help SCripts Ourscripts Rsamaps . . . 834
Help SCripts Ourscripts SDwamaps . . . 835
Help SCripts Ourscripts SPiemaps . . . 835
Help SCripts Ourscripts STaxmaps . . . 835
Help SCripts Ourscripts SVctable . . . 836
Help SCripts Ourscripts SYsinfo . . . 837
Help SCripts Ourscripts TCbmaps . . . 837
Help SCripts Ourscripts TIotmap . . . 837
Help SCripts Ourscripts TRapstae . . . 838
Help SCripts Ourscripts TSBmaps . . . 838
Help SCripts Ourscripts TSOmaps . . . 839
Help SCripts Ourscripts XSBmaps . . . 839
Help SCripts Ourscripts XSVcmaps . . . 840
Help SCripts Ryoscripts . . . 840
Help SCripts Ryoscripts Sequencefields . . . 843
Help SCripts Ryoscripts Conditionallogic . . . 845
Help SCripts Ryoscripts Limitation . . . 846

INTRODUCTION

ColeSoft has pursued the goal of making Z/XDC's internal documentation as comprehensive as possible. Towards that end, we have devoted considerable effort to greatly expanding the amount of information available within Z/XDC and to improving the accessibility of that information and the navigability of the Help Database as a whole.

This manual is nothing more than a printout of a section of the Help Database. It is provided for those people (like myself) who steadfastly prefer looking at paper instead of glass. (It's hard to write margin notes on glass.)

The information in the Help Database has been segmented into four printed documents:

- Z/XDC® User Guide
  Contains comprehensive tutorials about the many features and capabilities of Z/XDC.
- Z/XDC® Commands
  Contains the detailed syntax, usage descriptions, and examples of all of Z/XDC's commands.
- Z/XDC® Messages
  Contains descriptions of all of the messages that can be issued by Z/XDC and all of its various components.
- Z/XDC® V3.1 Release Guide
  Contains a history of all changes and upgrades made in the current release of Z/XDC.

There are a couple of important structural differences between Z/XDC's internal Help and these manuals:

- The PDF copies of the printed manuals can be searched using typical PC-style searching commands.
- "Release Guides" for older versions and releases of Z/XDC are available only via the "HELP WHATSNEW" command.

A Roadmap

The structure of this manual follows the structure of the Help Database. A consequence of this is that the sequence of information in this book, over all, is decidedly non-sequential. For those of you who prefer to read a manual from beginning to end, please accept my apologies. However, please let me make some suggestions.

If you are an experienced Z/XDC user, then start with the **Z/XDC® V3.1 Release Guide**. This will tell you what's new in this release of Z/XDC. Within Z/XDC, the Release Guide can be reached by typing HELP WHATSNEW. You can then use hyperlinks to pursue the specific information that is of interest to you.

For new users, turn to the **Z/XDC® User Guide**, and examine its Table of Contents carefully. You will see that there are about two dozen major topics arranged alphabetically: Addressing, Attentions, Breakpoints, ..., Virtmem, XDCCALL. Information within topics is presented more or less sequentially. The following **User Guide** topics are of particular interest:

- Perhaps the first topic that should be read is named **"DEBUGGING"**. This and its subtopics give comprehensive information about whether and to what extent you may have to modify your program in order to use Z/XDC.

- Another topic that should be read early on is named **"XDCCALL"**. XDCCALL is a utility program that can be used to start a debugging session for your program.
- If you plan to debug programs that run as batch jobs or system tasks, then read the **"CDF"** topic. "Cross Domain Facility" is the component of Z/XDC that permits user terminals to connect to debugging sessions for background jobs.

For Z/XDC command information, turn to the **Z/XDC® Commands** manual. Start with the basic commands. The DISPLAY, FORMAT, and LIST commands display storage and important program related structures. The AT and TRAP commands set breakpoints. You can use the TRACE command to step execution through your program slowly. The ZAP command allows you to change storage and registers.

If you wish to play with Z/XDC's terminal and user interfaces, read the **"FULLSCREEN"** section of the **User Guide**. Also, try the PROFILE command for displaying and changing a very large number of session parameters.

Generally, the best approach is to plan your reading using the Table of Contents. And of course, if you can't find the information that you are looking for, call us. There's no charge, and we will be glad to help! Our number is 800-XDC-5150 (USA: 928-771-2003). If the information that you want is in the book, we will explain what you want to know and tell you where to find complete information. If it is not, then we will add it for our next release.

ONLINE PRESENCE

ColeSoft Marketing maintains the following resources on the Internet:

[Home Page] ColeSoft's Home Page is www.colesoft.com. It provides the following services:

- General information about Z/XDC
- E-mail links to both Marketing, Technical Support, and Customer Services
- FTP links for uploading diagnostic information and other files to Technical Support
- A dialog for downloading current maintenance for Z/XDC
- Links permitting existing customers to download a full set of Z/XDC's documentation
- Online product delivery
- 24x7 self-service for temporary, short-term, license activation codes for use in D.R. tests and other emergencies

[Facebook] ColeSoft's Facebook presence is at facebook.com/colesoftware. This is where we will from time to time post information about ColeSoft people and activities.

[LinkedIn] ColeSoft has a users group named Z/XDC Users Group. This is the "Go-To" place for all things Z/XDC. So if you want to see what's coming up with Z/XDC, then join this group. Things that we put here include:

- Notices about new releases and what they include
- Notices of new maintenance and what has been fixed, changed or added
- Notices of new training videos as we create them
- Creative ways to solve situations that our customers might encounter
- Short "how to" tips illustrating now to use Z/XDC and what it can do

But we want this group to be a two-way street. We would love it if our customers would post to the group such things as:

- Questions about how to do something with Z/XDC
- Suggestions about how to improve Z/XDC
- Interesting experiences customers have had using Z/XDC
- New ways to use Z/XDC that make you smile
- Problems encountered with Z/XDC that you would like help with
- Pretty much anything having to do with Z/XDC

Built-in Help Panels

Help COmmands

This is the root topic for reference information about **all** Z/XDC, c/XDC and dump/XDC commands. The list starting below is comprehensive.

Note, that both HELP topic names and Z/XDC command names can be abbreviated. **But** the abbreviation of the topic describing a command is **not** necessarily the same as the abbreviation of the command itself!

- The **LIST HELP** and **HELP** commands will show the minimum abbreviations accepted for the various topic names.

- Usually, when you give a command name that's too short ("L L", for example), a **DBC061E** message will be displayed showing you a list of the commands that start with that same short name.

The following Z/XDC commands are available. For specific information, please type **H COMMANDS** followed by one of the following keywords. Alternatively, you can use an **S** shortcut to select any particular topic that interests you.

SYNTAX	- Some operands having a complex syntax occur in several commands. They include address expressions, address space identifiers, breakpointing commands, dsnames, and string data. The syntax descriptions for these operands have been consolidated into this subtopic.
?	- Displays the Dialog Commands Index.
ADEFERRED	- Creates persistent breakpoints to be set into future load modules at load time.
ALARM	- Displays an alarmed message for up to five seconds.
asterisk	- * **text** is a comment command. Semicolons might (or might not) terminate a comment.
AT	- Sets traps using persistent breakpoints.
ATX	- Sets traps using super persistent breakpoints.
CD	- changes the USS Current Working Directory (CWD).
COMMENTARY	- Provides a fullscreen panel for writing commentary to the debugging session's log file.
CONSOLE	- This command has been replaced by the **SET CONSOLE** command.
COPY	- Copies data (optionally with padding/truncation) from one register or storage location to another.
CURSOR	- Moves the cursor to the "home" position of the current window. (Best used within PF key definitions)
DEFINE	- Defines DMAP and MAP override datasets (and more).
DELETE	- Deletes maps, equates, load modules, MAPLIB lists, ADATA caches (and more).
DISCONNECT	- Disconnects from a cdf/XDC debugging session without ending the session.
DISPLAY	- Displays virtual storage in a raw hex-text (dump like) format.
DMAP	- Loads or clones (i.e. copies) dsect maps for formatting and

referencing control blocks in storage.

- **DOWN** - Scrolls the display downwards.
- **DROP** - Deactivates a dsect map by clearing its base address.
- **DUB** - makes a non-USS process a USS process the z/OS **dubbing** action.
- **END** - Ends one or more instances of a debugging session, possibly aborting the entire program, possibly with a dump.
- **EQUATE** - Creates labels and assigns them to storage locations.
- **EWHERE** - Produces a formatted display of an area of storage containing the current error level PSW's ABEND address.
- **EXEC** - Run a user written REXX exec.
- **FIND** - Searches storage for a given string of hex, character, or decimal data.
- **FORMAT** - Formats storage as machine instructions and data fields, using source image maps where available.
- **FREEMAIN** - Releases allocated storage.
- **GETMAIN** - Allocates storage.
- **GO** - Clears the display screen, and resumes program execution.
- **GOT** - ("GO for tracing") Resumes program execution without clearing the display screen.
- **GOX** - Causes Z/XDC to attempt to restore a user program's fullscreen display prior to resuming program execution.
- **HDEFERRED** - Creates hooks to be set into future load modules at load time. (See also the **HOOK** command.)
- **HELP** - Discusses in detail how to using Z/XDC's **HELP** command, including **important differences** from ISPF's HELP command.
- **HKEYS** - Displays and allows you to change the PF key definitions and options that are used within Built-in Help displays.
- **HOOK** - Sets a hook into code that, when reached by execution, creates a debugging session "on the fly" for that code. Hooks can be placed into exit routines, SRB routines, locked routines and even normal code running either in the current address space or (security permitting) in most other address space, System wide.
- **IPCS** - Lets you enter one or more arbitrary IPCS commands or command procedures and capture their output.
- **ISPF** - Temporarily suspends the current debugging session and invokes ISPF.
- **KEYS** - Displays and permits changes to the PF key definitions being used for the debugging session.
- **LEFT** - Scrolls leftwards across the data displayed in a window.
- **LIBRARYLISTS** - Tells Z/XDC where to find DWARF and C source libraries when they have been renamed or moved.
- **LICENSE** - Displays the Licensing Control Data Display/Update panel.
- **LIST** - Displays various objects including registers, PSW, address space structures, and Z/XDC objects (maps, equates, etc.).
- **LL** - Alias of LIBRARYLISTS.
- **LOAD** - Brings a load module into storage.
- **LOCATE** - Scrolls the display to a specific previously issued command.
- **LOG** - Writes to a log file a copy of recently issued Z/XDC commands and their responses.
- **MAP** - Loads module, csect and source image maps for referencing and formatting storage.
- **NOTE** - Saves, in a list, an address of interest and a comment about it.
- **OFF** - Deletes breakpoints.
- **PANELID** - Tells ISPF to turn its panel name display on or off.

POST - Posts an ECB.
PRINT - Prints excerpts of the Built-in Help.
PROFILE - Loads or stores user profile data from or to a dataset.
QUICK$INIT - Internal command.
READ - Reads from a dataset and executes a script of Z/XDC commands.
RECALL - Alias of RETRIEVE.
REFRESH - Repaints the terminal display to correct possible display corruption.
RELEASE - Releases a Pause Entry Token (PET).
RETRIEVE - Brings back to the command line a recently issued Z/XDC command string. (Please notice that there are some **interesting enhancements** in Z/XDC's RETRIEVE command as compared to ISPF's.)
RETRY - Alias of RETRIEVE.
REXX - Run a user written REXX exec.
RIGHT - Scrolls rightwards across the data displayed in a window.
SCANLOG - Searches a window's scroll area for a specific character string.
SCRNAME - Tells ISPF to assign a name to the ISPF window in which Z/XDC is running.
SET - Sets various Z/XDC values and controls. Also sets/changes certain program states including several controls from the retry level PSW.
SHOW - Displays a given list of fields in storage using only one display line per field. (Uses display screen real estate efficiently)
SPLIT - Sets horizontal split screen mode. (ISPF only)
SPLITV - Sets vertical split screen mode. (ISPF only)
STEP - Steps execution through a supported High Level Language program (such as XL C/C++ and Metal C) in various ways, including one source statement at a time. (c/XDC only)
SWAP - Swaps display screens. (ISPF only)
SWITCH - Switches control of a multi-tasking debugging conversation from one task to another.
TDEFERRED - Creates one-hit breakpoints to be set into future load modules at load time.
THAW - Thaws out an address space that has become deadlocked in Z/XDC's conversation management during multi-tasking debugging.
TRACE - Steps execution through machine code in various ways, including one machine instruction at a time.
TRAP - Sets one-hit traps using transient breakpoints.
TSO - Permits the execution of arbitrary TSO commands and CLISTs.
UP - Scrolls the display upwards.
UNDUB - removes dubbing from a task.
USING - Activates a dsect by assigning a fixed or floating base address to it.
VERIFY - Verifies the contents of storage, registers, and PSWs. (For use prior to a ZAP command)
WAIT - Forces an Z/XDC task to resume waiting in conversation management.
WHERE - Produces a formatted display of an area of storage containing the current retry level PSW's resume execution address.
whitespace - Use consecutive semicolons to insert white space between displays. (Example: LIST TASKS;;LIST RBS)
ZAP - Changes storage, registers, or the resume address in the retry level PSW. (For other PSW settings, use the SET command.)
=JUMP - You can use ISPF =jump commands (=6, =X and friends) to exit a Z/XDC debugging session and go directly to other programs running elsewhere in ISPF. (ISPF only)

% - You can use the implicit execution command **%** to execute a REXX procedure as if the REXX command had been used.

For specific information, Type an **H** at the left above to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

For a general description of command syntax (including Z/XDC's special processing of dsnames), see HELP COMMANDS SYNTAX.

Help COmmands Syntax

When issuing a Z/XDC command only as much of the command's name need be given as is necessary to make it distinguishable from other Z/XDC commands. For example:

- **OFF** may be abbreviated to **O**
- But **DMAP** can be abbreviated only to **DM** (because of the DELETE and DROP commands).

In some cases ambiguous abbreviations have a **preferred** meaning. For example, **F** means **FORMAT,** not FIND.

Z/XDC command names may be preceded by blanks. If the command has operands, then one or more blanks must separate the first operand from the command name. If the command has two or more operands, then they must be separated from each other either by **single commas** or by one or more **blanks,** or (sometimes) by other special characters. In all syntactical descriptions whenever a comma is shown, a blank or blank string could be used instead.

Multiple Z/XDC commands may be typed on a single line by separating them from each other by **semicolons.** If, while processing such a string of multiple commands, Z/XDC encounters an error, then the remainder of the string is aborted.

No single Z/XDC command may ever be longer than 255 characters.

No single command may be longer then the primary command input field of the display panel from which you are entering the command. (However, Z/XDC does support **large display screen geometries,** so rather long command strings can be accommodated that way.)

In **scripts,** Z/XDC commands may **not** be continued onto multiple records. (However, script files may be **RECFM=V,** so long commands can be accommodated conveniently.)

Certain kinds of operands occur in multiple commands. These include address expressions, ASIDs (address space identifiers), numeric data and dsnames. For special descriptions of these operands, type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

ADDRESSEXPRESSIONS - The syntax of address expressions.
ASIDS - The various methods for providing address space identifiers.
BREAKPOINTS - The general syntax for all breakpointing commands: AT, ATX, TRAP, TRAP, ADEFERRED, TDEFERRED, and HDEFERRED.

CHARACTERSTRINGS	- The syntax of character strings. (See also **STRINGDATA.)**
ESCAPECHARACTER	- The syntax of escape characters in character strings and string data.
NAMES	- The syntax of names.
NUMERICDATA	- The syntax of hexadecimal and decimal operand data.
DSNAMES	- The syntax and symbolic substitutions support in Z/XDC for dataset names.
PATTERNMATCH	- The syntax of wildcard patterns.
STRINGDATA	- The syntax of string data. (See also **CHARACTERSTRINGS.)**

Help COmmands Syntax ADdressexpressions

Z/XDC command operands that refer to virtual storage are called "address expressions". A large number of Z/XDC commands use address expressions as operands. The FORMAT command, for example, expects its first operand to be an address expression.

Address expressions have three distinct parts. All of these three parts are optional under various circumstances.

- The "base term" defines a starting point for the address expression. A base term establishes an address expression's initial "location calculated so far". This may be a register, a PSW, a location in an address/data space, or one of the built-in functions that returns a location is its result. (If the base term is a register or PSW, then it **must** be followed immediately by an "indirect operator", indicating that the register or PSW is being used as a pointer.)

- "Offset terms" start with plus or minus signs and compute values to be added to or subtracted from the location calculated so far.

- "Indirect operators" may occur only between terms. They cause a new address to be loaded from the place pointed to by the old location calculated so far (from a register, from a PSW, or from storage). This new location replaces the old location calculated so far.

Z/XDC has several built-in functions that can be used in address expressions. These functions do such things as change the address space or data space to which the expression is targeted, and extract values from storage for use in the expression's computation. Built-in functions generally have operands that may, themselves, be address expressions. In other words, Z/XDC's address expression resolution process can be a recursive affair.

Address expressions can be simple **(FORMAT 10?** displays the System's CVT), or complex **(EQUATE DDTIOT 21C%+C%+X2(MYDCB.DCBTIOT)** assigns a label to represent the TIOT DD entry associated with a particular DCB). The full syntax description of address expressions is a very large topic. It can be found starting at HELP ADDRESSING.

Help COmmands Syntax ASids

Address space identifiers (ASIDs) are, of course, four-digit hexadecimal numbers that identify a specific address space. But Z/XDC supports a rather large variety of ways in which an ASID can be represented. One very common way, for example, is to specify a jobname. Z/XDC will resolve a jobname into the four digit ASID of the address space in which the job is running (assuming there are not multiple address spaces running the same jobname).

Collectively, all the ways by which Z/XDC permits references to address spaces are called **address space references.** When a command operand references an address space, that operand is represented by the word **aspaceref.** The complete syntax of aspaceref operands is described by the current topic.

There are number of Z/XDC commands that accept aspacerefs as operands. They include:

LIST ASID - Displays information relating ASIDs and jobnames.

LIST TASKS - Displays the structure of tasks found within a given address space.

SET ASID - Makes a specified address space Z/XDC's default space for displays, zaps, and other commands.

THAW - Clears a bottleneck that can sometimes occur during multitasking debugging.

And others.

In addition, aspacerefs can be used within address expressions, generally to direct the expression towards a particular address space. The following built-in functions accept aspaceref parameters:

~ASID(...) - Directs the address expression towards a particular address space.

~ASN(...) - An alias of ~ASID(...).

~XASID(...) - Resolves an aspaceref and makes it available as a pure number for use in the calculations of an address expression. (Useful, for example, for targeting a particular slot of the System's Address Space Vector Table, ASVT).

~XASN(...) - An alias of ~XASID(...).

Whenever an address space reference can be used as an operand of a command or a built-in function, the aspaceref can be specified in any of the following ways:

keyword
An aspaceref can be given as any of the following keywords:

- **HOME HASID HASN:** These resolve to the home address space.
- **PRIMARY PASID PASN:** These resolve to the retry level primary address space.
- **SECONDARY SASID SASN:** These resolve to the retry level secondary address space.
- **IFETCH IASID IASN:** These resolve to the retry level instruction execution address space.
- **EPRIMARY EPASID EPASN:** These resolve to the error level primary address space.
- **ESECONDARY ESASID ESASN:** These resolve to the error level secondary address space.

- **EIFETCH EIASID EIASN:** These resolve to the error level instruction execution address space.

- **REAL:** This resolves to real storage.

Example:

```
SET ASID ESASID
```

register name

For some commands, an aspaceref can be given as the name of any of the following registers:

- **A general register:** (R0 for example) The ASID is loaded from the lo-order two bytes of the register.
- **CR3 or CR4:** The ASID is loaded from the lo-order two bytes of these particular control registers.

Examples:

```
SET ASID CR3
```

```
LIST TASKS R0
```

jobname

An aspaceref can be given as the name of any "job" in the System. (In this context, "job" refers to any batch job, any TSO user, and any system task.) Z/XDC resolves the jobname to the ASID for the address space in which the job is running.

If multiple address spaces have the same jobname, then in most contexts, the resolution of the ASID will fail. However, for the LIST ASID command, a report will be generated identifying the ASIDs of all address spaces that matched the given jobname. For LIST ASID, the jobname may also contain wildcard character such as * and the matching is case-insensitive.

Example:

```
LIST ASID INIT
```

equate name
dsect name

For some commands, an aspaceref can be given as a pure name of an equate or a dsect. Z/XDC resolves the name to the ASID of the space to which the equate or dsect is assigned (if any).

Examples:

```
LIST ASID JCT
```

The address space to be displayed will be that in which the JCT dsect (or equate) has been assigned.

```
LIST ASID JCT+0
```

This is something different. **JCT+0** is not a pure name, so it is processed as an address expression. Then the two bytes located at the resolved address would be used as the ASID of the address space to be displayed. See below for more

information.

hexadecimal number
An aspaceref can be given as a pure hexadecimal number that resolves to a value ranging between between **0001** and **FFFE.** A hexadecimal number is given simply as a sequence of hexadecimal digits. No special punctuation or framing is necessary.

Examples:

```
SET ASID 3
LIST TASKS FACE
```

Note, however, that it may sometimes be necessary to add a leading zero to the number if it happens to start with one of the digits A through F, and another object exists whose name happens to match that number.

Example:

```
LIST TASKS 0FACE
```

decimal number
An aspaceref can be given as a pure decimal number that resolves to a value ranging between between **1** and **65534.** A decimal number is given as a sequence of decimal digits followed by the letter **N.**

Example:

```
LIST TASKS 10n
```
This resolves to ASID=X'000A' (not X'0010').

ASCB address
If the aspaceref is an address expression that resolves to the address of an Address Space Control Block (ASCB), then the ASID will be extracted from that ASCB's **ASCBASID** field.

Example:

```
LIST TASKS ASVT.ASVTFRST+32N*4?
```
This resolves to the address of the ASCB pointed to by the thirty-second (decimal) slot of the Address Space Vector Table (ASVT). The task structure within that address space is displayed.

address expression
If an address expression is given that resolves to any location other than an ASCB, then that location must be a 2-byte wide field that contains the desired ASID.

Example:

```
LIST ASID 224?+24
LIST TASKS .ASCBASID
```

Omitted
Generally, either the home space's ASID or the ASID of the space currently being targeted by Foreign Address Space Mode will be used.

Examples:

SET ASID
This restores the home address space as Z/XDC's default target space (thus ending Foreign Address Space Mode).

LIST TASKS
LIST TASKS JES2
The first of these commands displays the task structure either of the address space currently being targeted by Foreign Address Space Mode, or of the home space if not in Foreign Address Space Mode.

The second command displays the task structure located in the JES2 address space **regardless** of whether or not Z/XDC is in Foreign Address Space Mode.

Help COmmands Syntax Breakpoints

Z/XDC uses breakpoints to set traps and perform traces. Breakpoints can be...
- Transient or persistent
- Active or deferred
- Conditional or unconditional
- They can have automatic commands associated with them.
- They can be gathered together into "families" that can be manipulated as a group.

For general information, see **HELP BREAKPOINTS.**

Parenthetically, it is possible to create "breakpoints" at program assembly time. Such breakpoints are called **DEAD traps** and are created by a macro named **#DIE.** See **HELP BREAKPOINTS DEADTRAPS** for more information. (DEAD traps are not discussed further here.)

Also, related to breakpoints are **hooks.** While breakpoints require that a Z/XDC debugging environment already exist, hooks do not. Instead, they actually create the environment on the fly. For more information, see **HELP HOOKS.**

General Syntax for Breakpoint Commands:

```
verb =familyid addressexpressions (conditional) 'cmd;cmd;...' ...
     =         tracetype                        "cmd;cmd;..."
               W
```

```
... SAVE=YES  REMOVAL=PURGE    ZERO    FORCE
          =NO          =DISABLE  TRAP2   omitted
       omitted   omitted
```

Notes:

- Any or all operands can be omitted.
- Operands can be given in any order. There simply are no positional restrictions.
- **addressexpressions, tracetype** and **W** are mutually exclusive. (They distinguish between traps, traces and WATCHs, respectively.)

The Command Verb

verb
This identifies the specific breakpoint command being used. It may be any of the following:

TRAP
Sets traps using **transient** breakpoints. (Transient breakpoints are one shot breakpoints that are automatically removed upon being reached by execution.)

AT
Sets traps using **persistent** breakpoints.

ATX
Sets traps using persistent breakpoints that are **super persistent.** The **OFF** command must specifically name them or their type (ATX). In general, they cannot be removed in combination with other breakpoint types. Example: **OFF ALL** will not remove them.

ADEFERRED
Defines deferred persistent breakpoint traps. (They target load modules that have not yet been loaded into storage.)

TDEFERRED
Defines deferred transient breakpoint traps.

TRACE
Allows you to step through user program execution in various ways (instruction by instruction, or branch to branch, or etc.).

TRACE W
Creates a **watch** for a condition of some sort. (Example: R1 equal to some value.) Essentially, this is a conditional expression that is not assigned to any particular trap or trace. In effect, it behaves as if it were assigned to **all** traps and traces.

Operands

addressexpressions
For the trapping breakpoint commands (i.e. **ADEFERRED AT ATX TDEFERRED** and **TRAP),** this must be one **or more** address expressions giving the locations at which breakpoint traps are to be set. The address expressions may or may not make use of Z/XDC's Current Display Pointer in interesting ways. For specific information, see **HELP BREAKPOINTS TRAPPING.**

tracetype
For the **TRACE** command, this must be a keyword indicating the type of trace step to be performed. Permitted keywords are:

T (Instruction trace)
T I (Instruction trace)
The current instruction is permitted to execute, and control returns to Z/XDC prior to the execution of the next instruction.

T B (Branch trace)
The current instruction is permitted to execute, and then all subsequent instructions are executed until execution reaches a branch-type instruction (or an EX thereof). Note: "NOP", "BALR Rx,0", "BCTR Rx,0", etc. are **not** considered to be branch-type instructions. For more information, see **HELP BREAKPOINTS TRACING BRANCHES.**

T BY (Successful branch trace)
The current instruction is permitted to execute, and then all subsequent instructions are executed until execution reaches a branch-type instruction that will, in fact, branch. This is probably the most useful form of the TRACE command. For more information, see **HELP BREAKPOINTS TRACING BRANCHES.**

T BNC (Branch-No-Call trace)
This is a trace that is identical to **T BY** except that when a linkage instruction is reached (BAL, BASR, JAS, BAKR, etc.), **T BNC** may or may not follow the jump according to the particular way the linkage instruction is being used:

- For **point-and-jump-over** usages (for jumping over inline data), **T BNC** will follow the jump.

- For **call-with-return** usages (for calling subroutines), **T BNC** will **not** follow the jump. Instead, it will set recapture traps at the subroutine's return points, and then let the subroutine run without being followed.

For all the gory details, see **HELP COMMANDS TRACE BNC.**

T * (Loop trace)
The current instruction is permitted to execute, and then all subsequent instructions are executed until execution loops back and reaches the current instruction again.

T SB (Storage alteration trace - stop before)
T SA (Storage alteration trace - stop after)
The current instruction is permitted to execute, and then all subsequent instructions are executed until execution reaches an instruction that will alter storage. Execution will then stop either before (T SB) or after (T SA) that

instruction executes. For more information, see **HELP BREAKPOINTS TRACING STOREALTER.**

For more information about tracing, see:

HELP COMMANDS TRACE
HELP BREAKPOINTS TRACING

T W (conditional) 'cmd;cmd;...'
This form of the TRACE command creates a **WATCH.** No trace is performed. No trap is set. Instead, the conditional expression given with this command is saved and tested whenever Z/XDC receives control for any nonaborting reason.

Whenever Z/XDC receives control, all enabled WATCH conditions are evaluated. If any WATCH condition evaluates TRUE, then Z/XDC halts user program execution and queues its automatic commands string (if any) for execution.

For all WATCHs that evaluate TRUE, if they have automatic commands, then those commands are queued for processing by Z/XDC. Consequently, enabled WATCH conditions are evaluated even if Z/XDC would halt user program execution anyway. This is for the purpose of deciding whether or not the WATCH's automatic commands (if any) are to be processed.

Please note the following:

- If Z/XDC receives control due to a conditional breakpoint that evaluates FALSE, a WATCH that evaluates TRUE will still cause Z/XDC to halt user program execution.

- A **T BY** trace (for example) causes Z/XDC to receive control on all branch type instructions for the purpose of determining whether or not the instruction will, in fact, branch. Normally, if the instruction will not cause a branch, then Z/XDC silently allows the user program to resume until the next branch-type instruction. However, before resumption, Z/XDC also evaluates all enabled WATCHs. If any WATCH evaluates TRUE, then user program execution will be halted at that branch instruction after all.

- A similar statement can be made about almost all other traces.

For more information, see **HELP BREAKPOINTS CONDITIONAL WATCH.**

Special Syntax Operands

The following operands are permitted on all types of breakpointing commands. They are introduced by specific special characters.

=familyid
=
The first operand that is led by an equals sign (=) is parsed as a =familyid operand:

- **familyids** are numbers. They form the numeric portion of a breakpoint's name.

- When the **=** is followed by a decimal number, the generated breakpoint is assigned to the family of breakpoints identified by that number.

- When the decimal number is absent, the previously used family number is used again.

- When the familyid operand is entirely absent, the family number that is assigned is one greater than the previously used family number.

The point of family numbers is that it creates groups of breakpoints that can be manipulated as a group instead of just one by one.

The commands that manipulate breakpoints are the **OFF** command and the **SET BREAKPOINTS** command.

For more information, see **HELP BREAKPOINTS FAMILIES**

(conditional)

The first operand that is led by an open parenthesis (**(**) is parsed as a (conditional) operand. This operand indicates that:
- For each time the trap is reached,
- Or for each trace step taken,

A condition is to be tested.

If the condition resolves as **FALSE,** then the trap is bypassed or the trace is advanced by one step.

On the other hand, if the condition is **TRUE,** then the trap is accepted or the trace stops.

Note, if a null conditional expression is given [example: **()**], then the next previously given expression is used as a default. (But see below about the SAVE=NO operand.)

'cmd;cmd;...'
"cmd;cmd;..."

The first operand that is led by a quote character (**'** or **"**) is parsed as a list of Z/XDC commands that are to be executed when the breakpoint is reached and accepted.

Note, if a null commands string is given (example: **''**), then the next previously given commands string is used as a default. (But see below about the SAVE=NO operand.)

KEYWORD=value Operands

SAVE=NO
SAVE=YES
omitted
Normally, when you provide a conditional expression or an automatic commands string, Z/XDC automatically saves them for use as defaults on subsequent breakpointing commands (see above). But when you use the **SAVE=NO** operand, the command's

conditional expression and/or automatic commands string are not so saved.

Specifying **SAVE=YES** is the same as omitting this operand entirely. The default action is to save any provided conditional expression and/or automatic commands string as subsequent defaults.

REMOVAL=DISABLE
REMOVAL=PURGE

When a transient breakpoint (i.e. breakpoints created by the TRAP and TDEFERRED commands) is automatically removed, either it might be **disabled,** or it might be **purged:**

- When a breakpoint is **disabled,** it is removed from your code, but its definition remains:
  - It can still be displayed by the **LIST BREAKPOINTS** command.
  - It can be reenabled by:
    - The **SET BREAKPOINTS ENABLE** command,
    - Or the **SET BREAKPOINTS TOGGLE** command,
    - Or the **X** shortcut.

By default (i.e. absent use of the REMOVAL= operand)...
- Simple breakpoints are purged.
- Complex breakpoints are just disabled.

This REMOVAL= operand can be used when the default action is not desirable.

This operand has no effect upon the **OFF** command:
- **OFF yadayada** will still purge all targeted breakpoints regardless.
- **OFF yadayada NOPURGE** will still disable all targeted breakpoints regardless.

KEYWORD Operands

FORCE

The FORCE operand will allow the TRACE command to be used even when a trap save area corruption has been detected. Typically this operand would be used after altering the general register, PSW, and AR15 values. If no Trap Save Area corruption has been detected, this operand is ignored. This operand is only relevant to the TRACE command (except TRACE W) and will cause a syntax error if used on any other breakpointing command (or the TRACE W command).

ZERO

The ZERO operand will force the use of a binary zero (X'00'), regardless of the current SET TRACE ZERO|TRAP2 setting.

TRAP2

The TRAP2 operand will force the use of a TRAP2 instruction (X'01FF') regardless of the current SET TRACE ZERO|TRAP2 setting.

Legacy Support

In prior releases of z/XDC, a colon-delimited syntax was used for appending automatic commands to a breakpointing command. Example: **TRAP R1?:command:command:---** Support for that syntax has been removed.

Subtopics Index

For more detailed syntax information, type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

FAMILIES - Specifying breakpoint families.
CONDITIONS - Specifying conditional expressions.
AUTOCMDS - Specifying automatic commands.

For additional syntax information, the following may be selected directly:

- For specifying breakpoint locations for the AT, ATX, and TRAP commands, see **HELP COMMANDS TRAP.**

- For specifying breakpoint locations for the ADEFERRED, TDEFERRED, and HDEFERRED commands, see **HELP COMMANDS ADEFERRED.**

- For specifying trace step types, see **HELP COMMANDS TRACE.**

Help COmmands Syntax Breakpoints Families

When breakpoints are created, they are always assigned a "family identifier". This identifier may be either automatically generated (in which case it will probably be unique) or it may be explicitly given on the breakpoint command that creates the breakpoint. In either case if a single breakpoint command creates multiple breakpoints, then all of them are assigned the same family id.

Family identifiers are most useful for transient breakpoints. This is because when **any** breakpoint in a family is reached by user program execution and accepted by Z/XDC, then all transient members of that family are automatically deleted. In general, if the usefulness of a series of breakpoints ends when any one of them is reached, then they should all be assigned to the same family so that they can be deleted together. There are several circumstances where Z/XDC will do this **automatically:**

- When two or more traps are created by a single command, all of the created traps will be assigned to the same family.

- When multiple **A** and **T** shortcut commands are setup and then executed by a single press of the ENTER key, then all of the created traps will be assigned to the same family.

- When factory default **F11** is used to "step over" a subroutine call, all of the traps that it creates (to recapture execution upon the subroutine's return to its caller) are (a) transient and (b) assigned to a single family. Eventually, when the subroutine returns (even if it makes a vectored return) and one of the traps is reached, then all of the traps are automatically removed and the trapped instructions are all restored to their original opcodes.

Syntax:

```
verb =familyid addressexpressions ...
              tracetype ...
```

=familyid
This operand may be placed anywhere amongst other operands. **Familyid** must be a decimal number in the range of 0 to 999. The number must be preceded by an equals sign (=). Examples:

T =32,R15?
T R15? =32
These two commands do the same thing. They both set a persistent breakpoint at the location pointed to by R15 and assigns that breakpoint to family #32.

T =4
T =4 I
T I =4
These three commands all do the same thing. User program execution advances by one instruction. The transient breakpoint used for the trace step is assigned to family #4. If other transient breakpoints exist in family #4, then they are automatically deleted when the trace step completes.

=
If the equals sign is given without being followed by a decimal number, then the breakpoint(s) being created are assigned to the same family as the breakpoint(s) created by the most recent preceding breakpoint command. EXAMPLE:

```
AT R15% .TOPLOOP
T      =,R1?
```

The first command sets two persistent traps and assigns a new family identifier to them. The second command sets a transient trap and assigns it to the same family to which the first two breakpoints were assigned.

If =familyid is not given, then a familyid is automatically generated that has a high probability of being unique.

verb and **other operands**

See HELP COMMANDS SYNTAX BREAKPOINTS.

A breakpoint's family identifier is a decimal number ranging from 1 to 999. A breakpoint's label includes its family id number. If the labels of two different breakpoints contain the same decimal number, then those two breakpoints are members of the same family. A breakpoint's label can be displayed by storage formatting commands (FORMAT, WHERE, SHOW, or FIND) and by the LIST BREAKPOINTS command.

Example 1:

Suppose that user program execution has reached the following code:

```
    ...
    BAL   R14,CHECKSUB    CHK THE RESULT
    B     YES             +0 AOK
    B     NO              +4 NOK
    B     MAYBE           +8 GO CHK FURTHER
    ...
```

Assume the following:

- The retry level PSW points to the BAL instruction.
- The subroutine being branched to might return to either +0, +4, or +8 past the BAL.
- You do not want to trace the execution of the subroutine but you do want to trap the return from the subroutine.

The following commands would accomplish what you want.

```
T PSW?+4 +8 +C;GOT
```

- The **TRACE PSW?+4 +8 +C** command sets a family of three transient traps at the possible return points just past the BAL instruction.
- The **GO** command causes user program execution to resume with the BAL. When the subroutine returns, one of the traps is hit. This causes Z/XDC to regain control, whereupon it automatically clears the entire family of traps and then awaits commands.

Factory default **F11** is set to a command string that implements this example. In other words, instead of having to type the above commands, you can just press F11. See HELP BREAKPOINTS TRACING SUBROUTINES for more information.

Example 2:

Suppose the following:

- A family of breakpoints currently exists. Its family id is 23.
- You wish to set a breakpoint at the location "PSW?+10" and have it assigned to the same family (family number 23).

The following command would accomplish what you want.

```
T =23,PSW?+10
```

For related information, see HELP BREAKPOINTS FAMILIES.

Help COmmands Syntax Breakpoints Conditions

All of Z/XDC's breakpointing commands allow conditional expressions. This means that any kind of trap as well as any kind of trace can be conditional.

In addition, a kind of **floating** conditional can be defined by the **TRACE W** command. This is a conditional expression that is not associated with any particular breakpoint, but nonetheless is evaluated whenever Z/XDC receives control for any reason (including for traps and traces). For more information, see HELP COMMANDS TRACE W.

When a trap or a trace has a conditional expression associated with it, then each time the trap is reached by user program execution, or each time a trace step is taken, that condition expression (as well as all **WATCHs** that may exist) are tested. If **any** conditional evaluates as being **TRUE,** then user program execution is interrupted at that point, and Z/XDC shows its displays and awaits command input.

- If an execution location has **multiple** conditional breakpoints set, then **all** of those conditional expressions are evaluated.

- If one or more **WATCHs** are defined, then all of those conditional expressions are also evaluated.

If **any** evaluated expression resolves **TRUE,** then execution is interrupted.

If **all** expressions resolve **FALSE,** then user program execution is silently resumed as if nothing happened.

For more information, see HELP BREAKPOINTS CONDITIONAL.

When a conditional expression is assigned to a deferred breakpoint definition, it does not affect the deferred breakpoint itself. Instead, it applies to the active breakpoints that are cloned from the deferred breakpoint definition. See HELP BREAKPOINTS DEFERRED for more information.

When a breakpoint is reached, if it has a conditional expression, then that expression (as well as all enabled WATCHs) are evaluated. If any result is TRUE, then the breakpoint is "accepted". This means the following:

- If the breakpoint or any of its family members are transient, then the transient members are cleared.
- If any automatic commands are associated either with the breakpoint or with the enabled WATCHs (if any), then those automatic commands that are associated with those WATCHs and/or breakpoints whose conditions evaluated TRUE, are stacked for processing by Z/XDC.
- Automatic commands associated with breakpoints and WATCHs either that were not evaluated or that evaluated FALSE are left unprocessed.
- Z/XDC prompts you for more commands.

If the result of evaluating the conditional expression (and all enabled WATCHs) is FALSE, then the breakpoint is "bypassed". This means the following:

- If the breakpoint is associated with a trace, then the next trace address is determined and the breakpoint is moved to the new address.
- If the breakpoint is such that it doesn't move (i.e. is a trap, not a trace), then a special temporary breakpoint is placed on the following instruction and user program execution is allowed to resume. When the special trap is reached, the original breakpoint is restored to its original location and user program execution is allowed to resume again.

If the given breakpoint command sets multiple breakpoints, then the conditional expression, if any, is associated with all of them.

When multiple breakpoints are associated with a single countdown condition, the count sets the hit limit for each individual breakpoint (not for the set of breakpoints collectively).

Syntax:

```
verb ... addressexpressions (conditional) ...
         tracetype
         W
```

(conditional)
This operand may be placed anywhere amongst other operands. **(conditional)** defines one or more conditions to be tested. The individual condition tests are referred to as **comparison elements.** The rules for forming conditional expressions are as follows:

- Each given, comparison element must be enclosed within parentheses.
- Multiple comparison elements can be given. When this is the case:
  - The individual comparison elements must be enclosed within parentheses.
  - The comparison elements must be joined together by one of the following logical operators: **AND, OR, XOR, ANDNOT, ORNOT, XORNOT,** or their variations.
  - The conditional expression as a whole must also be enclosed within its own set of parentheses.
- Generally, comparison elements will be evaluated from left to right. However, additional parentheses can be used to change the order of evaluation by grouping together those comparison elements that need to be evaluated ahead of others.
- You may use as many levels of parentheses as you need to construct and organize your logical tests.
- Individual comparison elements as well as groups of comparison elements can be logically negated by prefacing them with the word **NOT** or with a logical not character (^ or !).

Four kinds of comparison elements are recognized:
- Countdown conditions
- Value testing conditions
- Boolean constants
- Null conditions

Subtopics Index

For specific syntax information, type an **H** below at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

COUNTDOWN - How to define a countdown condition.

TRUEFALSE - How to define a boolean constant.

COMPARISONS - How to define a value testing condition.
COMPOUND - How to create compound conditional expressions.
NULL - How to define and then use a default conditional expression.

For verb and other operands, see HELP COMMANDS SYNTAX BREAKPOINTS.

For related information, see HELP BREAKPOINTS CONDITIONAL.

Help COmmands Syntax Breakpoints Conditions COUntdown

Countdown conditions cause the breakpoints associated with them to be bypassed n-1 times; that is, the breakpoint is not accepted until it has been reached by user program execution for the "nth" time.

If multiple breakpoints have been associated with a countdown condition, then the condition sets hit limits for the breakpoints **individually** (not collectively).

If the countdown condition is assigned to a persistent breakpoint (via an **AT, ATX,** or **ADEFERRED** command), then the breakpoint will be accepted, not only the nth time it is reached, but also every time **thereafter**!

When a countdown condition is assigned to a deferred breakpoint definition, it does not affect the deferred breakpoint itself. Instead, it applies to the active breakpoints that are cloned from the deferred breakpoint definition. See **HELP BREAKPOINTS DEFERRED** for more information.

Syntax:

```
verb ... addressexpressions (number) ...
         tracetype          (COUNT,GE,number)
```

Note:

The conditional expression may be placed into the command at any position with respect to other operands.

Operands

(number)
(COUNT,GE,number)
(Number) is shorthand for **(COUNT,GE,number).**

Number must be a decimal value equal to or greater than 1. It prevents the breakpoint from being accepted until it has been reached by execution the specified number of times. In other words, for the first number-1 times that the breakpoint is

reached, it is silently bypassed.

verb and **other operands**

See **HELP COMMANDS SYNTAX BREAKPOINTS.**

Examples:

T (10)
This means, "Execute the next 9 instructions and stop on the 10th."

T BY,(5)
T (COUNT,GE,5) BY
These both mean, "Let 4 successful branches occur and then stop on the 5th."

AT (1000) R12?+354
This sets a persistent trap that is not accepted until it has been reached for the 1000th time and thereafter.

Related Information

For related information, see:

- **HELP BREAKPOINTS CONDITIONAL**
- **HELP BREAKPOINTS TRACING CONDITIONAL**

Help COmmands Syntax Breakpoints Conditions Truefalse

Boolean constants are simply specified as **(TRUE)** and **(FALSE).** When used as conditional expressions on breakpointing commands **(AT, TRACE, TRAP** and friends), they cause the breakpoint to be always accepted or always rejected when reached by execution.

Specifying the **(TRUE)** constant is equivalent to not specifying any conditional at all. The breakpoint is, in effect, unconditional.

Specifying the **(FALSE)** constant causes the breakpoint never to be accepted. Z/XDC still receives control when the breakpointed instruction is reached by user program execution, but **absent any other conditionals,** Z/XDC will always allow the program to resume silently as if nothing has happened.

Nonetheless, a breakpoint having the **(FALSE)** conditional expression is very useful when used in conjunction with one or more **WATCHs.** WATCHs are created by the **TRACE W** command. They are conditional expressions that are **unassociated with** any

particular breakpoint. They are, in effect, associated with **all** breakpoints. Thus, when user program execution reaches any conditional breakpoint (including breakpoints with the (FALSE) conditional), then all enabled WATCHs are evaluated:

- When any WATCH evaluates as **being TRUE,** program execution is interrupted, and the user is permitted to see displays and enter commands.
- When all WATCHs evaluate as **being FALSE,** program execution is silently resumed so that it may run to the next trap point.

For more information about WATCHs, see **HELP BREAKPOINTS CONDITIONAL WATCH.**

Examples:

```
T W (R1,EQ,00000000) 'L REGS;WHERE'
T W (R4?,NE,'I-CATCHR') 'D R4?'
T W (.DCBOFLGS,AND,10,EQ,00)

T .LOOP1 .ERR .ZPROGRAM .LOOPX (FALSE)
T .PUTMSG (R1?+2,EQ,'AHA994T')
```

The first three commands define three separate **WATCHs** to be tested for whenever Z/XDC receives control for any **conditional** trap.

The fourth command defines **four traps** at which Z/XDC will receive control. The **(FALSE)** conditional expression is assigned to all of them. So...

- **Without** the WATCHs present, execution of these four breakpoints would never cause Z/XDC to interrupt the user's program. Instead, Z/XDC would always permit program execution to resume silently.

- But **with** the WATCHs, if **any one or more** of the WATCH'd conditionals is TRUE, then the breakpoint will be accepted after all.

In other words, the (FALSE) expression allows you to create the conditionals (i.e. the WATCHs) **separately** from creating the traps.

Note that the actual position within the command of the **(FALSE)** operand makes no difference whatsoever. The conditional expression is assigned to all of the created breakpoints regardless of positioning.

The last command **(T .PUTMSG...)** defines an additional breakpoint that has a conditional of its own. When execution reaches the .PUTMSG location, this breakpoint's conditional is tested in addition to the three WATCH'd conditionals, and so this breakpoint will be accepted if either its conditional is TRUE or any of the WATCH'd conditionals are TRUE.

Automatic Commands

Multiple **WATCHs** can exist an any time. In addition, multiple traps can exist at any address. So when a trap is reached, it is quite possible for Z/XDC to accept both one or more WATCHs and one or more traps located at the reached address.

When any or all of the accepted WATCHs and traps have **automatic commands,** then those commands will all be queued for execution. Thus, it is quite possible for **multiple command strings** to be queued and executed when a trap is reached.

Those automatic commands associated with WATCHs and traps that were **not accepted** are not queued for execution.

Z/XDC will execute the default command string **(FORMAT PSW?** or **WHERE)** only when none of the accepted WATCHs and traps offer command strings of their own. (Just which default command is used is controlled by the **SET TRACE ROLL/SCROLL** setting.)

For more information, about automatic commands, see **HELP BREAKPOINTS AUTOCMDS.**

Help COmmands Syntax Breakpoints Conditions COMPArisons

Conditional expressions can be given in the form of a comparison that tests either the contents of a given location or the result of a numeric calculation. Up to eight bytes can be tested.

When a conditional expression is assigned to a deferred breakpoint definition, it does not affect the deferred breakpoint itself. Instead, it applies to the active breakpoints that are cloned from the deferred breakpoint definition. See HELP BREAKPOINTS DEFERRED for more information.

Syntax:

```
verb ... addresses (location,relation,constant) ...
         tracetype (location,operator,mask,relation,constant) ...
```

Notes:

The conditional expression may be placed into the command at any position with respect to other operands.

The operand syntax discussed here describes the minimum elements of a single comparison. These are referred to as **comparison elements.**

Multiple comparison elements may be combined together to form more complex comparison logic. When this is done, the conditional expression as a whole must be enclosed within an outer pair of parentheses.

Additional parentheses can be added in the normal way to create nests of compound comparisons of various sorts, but overall the parentheses, of course, must be balanced.

Comparison Element Suboperands

location
This specifies the location whose value is to be tested. It may be any of the following.

addressexpression
This identifies a storage location whose contents are to be tested. This address expression is evaluated each time the breakpoint is reached; therefore, if the expression is in any way indirect (e.g. if the expression contains a "?", "%", or "!"), then it could resolve to different locations each time it is evaluated.

registername
registername+offset
This identifies a particular byte of a given error level or retry level general register, access register, control register, or floating point register. This byte is the point at which a comparison is to start. If an offset is given, then it must fall in the range of:
- 0 to 3 for 4-byte wide registers
- 0 to 7 for 8-byte wide registers

PSWname
PSWname+offset
This identifies a particular byte of a particular PSW (as indicated by the PSWname you choose to use). **PSWname** may be any of the following:
- **PSW** indicates 8-byte form (the "scrunched" form) of the retry level PSW.
- **PSWE** indicates 16-byte form (the "extended" form) of the retry level PSW.
- **EPSW** indicates 8-byte form of the error level PSW.
- **EPSWE** indicates 16-byte form of the error level PSW.

If an offset is given, then it must fall in the range of:
- 0 to 7 for PSW and EPSW
- 0 to F for PSWE and EPSWE

function(...)
Those built-in functions that return numeric results can be used as the value to be tested. The following functions are supported:

XADDR(...) - (Alias: XADR) Returns the numeric value of an address expression (as opposed to the contents of the storage pointed to by that expression).
- The returned value is 8 bytes wide.

XALET(...) - Returns the numeric value of an ALET.
- The returned value is 4 bytes wide.

XASID(...) - (Alias: XASN) Returns the numeric value of an ASID.
- The returned value is 2 bytes wide.

Normally, comparisons are left-aligned (see below). But when the one of the above numeric built-in function is used as the value, a **right-aligned comparison** is done instead. This reduces the need for the constant (the value to be compared against) to have leading zeros in order to construct a valid comparison.

operator,mask
These operands are optional. If given, then they define a hexadecimal mask to be either AND'd, OR'd or XOR'd against the value extracted from the test location. By this means, particular bit patterns can be selected or set before testing.

operator
This must be one of: **AND OR XOR**
& | #

mask
This must be a data string to be used as a bit selection mask. Here are the rules:
- The mask may be given either as a hexadecimal string or a text string or a mixture of both.
- That portion of the mask that is text must be enclosed within quote characters.
- If a mask **includes** the opening quote character, then the opening quote character must be doubled up.
- That portion of the mask that is outside of any quote characters must consist of hexadecimal digits.
- Each group of hexadecimal digits must contain an even number of digits.
- The maximum length of the mask, after conversion to internal format, is 8 bytes.
- However, the length of the mask can never be longer than the test value. For example, if the location is given as "R5+1", then the mask cannot be longer than three bytes.
- If the mask is shorter than the test value, then only those bytes selected by the mask are tested. For example, if the test location is "ER3+2" and the mask is "FF", then only the third byte of error level register R3 is selected.

relation
This defines the comparison relation to be used for testing the selected value. Valid relations are:

LT	LE	EQ	NE	GE	GT
<	<=	=	≠	=>	>
	=<		<>	<=	
	^>		!=	^<	
	!>			!<	

For most **values,** the comparison is **left-aligned.** Example:
- **(RW3+1,EQ,752C)**
  - The portion of RW3 that can be tested are the 7 bytes starting at offset +1 and ending at the end of the register (i.e. the 2nd through 8th bytes).
  - However, the constant to be compared against is only two bytes wide.
  - So only RW3's 2nd and 3rd bytes are tested (since this is a left-aligned comparison).

However, when the value to be tested is provided by the XADDR, XALET or XASID function, Z/XDC uses a **right-aligned** comparison. Example:
- **(XADDR(R3%),EQ,00052C)**
  - The address expression **R3%** is resolved to a numeric address.
  - The **XADDR** function pads that address out to an 8 byte wide value and returns it as the value to be tested against.
  - The **00052C** constant is then tested against the address's **low** order three bytes (because this is a right-aligned comparison).
  - Note that in this example the first 5 bytes of the address are **not** tested.

constant
This must be a data string to be used as a comparison constant. It follows the same rules as the mask:

- The constant may be given either as a hexadecimal string or a text string or a mixture of both.
- That portion of the constant that is text must be enclosed within quote characters.
- If a constant **includes** the opening quote character, then that quote character be doubled up.
- That portion of the constant that is outside of any quote characters must consist of hexadecimal digits.
- Each group of hexadecimal digits must contain an even number of digits.
- The maximum length of the constant, after conversion to internal format, is 8 bytes.
- However, if a mask has been given, then the length of the constant must be the same as the length of the mask.
- If the mask has not been given, then the constant cannot be longer than the test location. For example, if the test location starts at the 2nd byte of a 4-byte register (example: **R4+1),** then the constant cannot be longer than 3 bytes.
- The number of bytes selected from the location for testing is limited by the length of the constant.

verb and other operands

See HELP COMMANDS SYNTAX BREAKPOINTS.

Examples:

AT R5?+10 (R3+3,AND,02,EQ,02)
This sets a breakpoint that will not be accepted until the 31st bit (i.e. bit number 30) of R3 is on.

Note that **R3** is register three's 4 byte wide name. It refers to only the low order 4 bytes of what is actually an 8 byte wide general register. So the given offset, **+3,** is relative to the lower half of the actual register.

AT R5?+10 (RW3+7,AND,02,EQ,02)
This is effectively the same as the preceding example, but in this case register three is referred to by its 8 byte wide name (RW3) instead of its 4 byte wide name (R3).

For more information about register names, see HELP COMMANDS LIST GENERALREGISTERS INDIVIDUAL.

TRACE (R1?+8,EQ,'MYPROG') *
Suppose R1 is being used to scan a queue of CDEs. This command starts a loop trace that will not be accepted until R1 points to the CDE representing the module named "MYPROG".

T BY (XADDR(NXINST()-XADDR(.CVTEXIT)),EQ,00000000)
This command starts a conditional trace that proceeds until user program execution is about to jump to the SVC 3 instruction located within z/OS's CVT. Note, NXINST is a built-in function that can be particularly useful in conditional tracing. See HELP FUNCTIONS NXINST for more information.

For more examples, see:
    HELP FUNCTIONS XADDR
    HELP FUNCTIONS XALET
    HELP FUNCTIONS XASID

Help COmmands Syntax Breakpoints Conditions COMPOund

Z/XDC allows you to organize multiple individual comparisons (the "comparison elements") into a compound boolean expression. See **HELP COMMANDS SYNTAX BREAKPOINTS CONDITIONS COMPARISONS** for details about comparison elements.

Syntax:

```
verb ... addressexpressions (compoundbooleanexpression)
         tracetype
```

Operands:

(compoundbooleanexpression)
The compound boolean expression is constructed from multiple comparison elements that must be joined together by suitable boolean operators **(AND, OR** and the like).

Each comparison element must be enclosed within a set of parentheses.

The compound boolean expression as a whole must be enclosed within its own set of parentheses.

Compound boolean expressions have the following forms:

((comparison),relation,(comparison) ...)
(NOT(comparison),relation,(comparison) ...)
- Any number of comparison elements may be given, separated from each other by logical operators ("relations").
- Individual comparison elements **must** be parenthesized.
- The logical operators **may** be separated from the comparison elements by blanks or single commas, but they need not be.
- The entire compound expression must be enclosed with parentheses.
- Normally, the comparison elements are evaluated from left to right, but parentheses, of course, may be used to nest the expressions to any depth and to influence the order in which the expressions are evaluated.

(comparison)
The individual comparison elements have the following structures:
(location,relation,constant)
(location,operator,mask,relation,constant)

These constructs are described in detail in **HELP COMMANDS SYNTAX BREAKPOINTS CONDITIONS COMPARISONS).**

(COUNT,relation,number)
This is the "(comparison)" format of the countdown condition. Its function is identical to the countdown condition:
- It triggers when its trap has been reached by execution for the nth time (and thereafter).
- But this syntax can be used in compound expressions.

(NOT(comparison) ...)
Any comparison element may be preceded by a **NOT** operator (or ^ or **!)** to invert the logical result of the element. When given, the **NOT** must immediately precede the comparison’s leading parenthesis.

... **,relation,** ...
When multiple comparison elements are given, they are connected to each other by the following logical operators:
AND OR XOR
& | #

The logical operators may be separated from the parenthesized comparison elements by blanks or single commas, but they need not be.

verb
addressexpressions
tracetype
These operands are explained in **HELP COMMANDS SYNTAX BREAKPOINTS.**

Examples:

The following are examples of valid compound comparison elements:
AT R8? ((COUNT EQ 3) OR (COUNT EQ 5))
AT R8? ((COUNT EQ 3) OR NOT(R1+2 EQ ’XY’)) ’L RBS’
AT R8? ((COUNT EQ 3) ORNOT (R1+2 EQ ’XY’)) ’L RBS’
AT R8? (NOT(COUNT EQ 3) OR (R1+2 EQ ’XY’))
AT R8? ((COUNT EQ 3) OR (NOT(R1+2 EQ ’XY’) AND (.DCBOFLGS AND 01 EQ 01)))

Help COmmands Syntax Breakpoints Conditions Null

Whenever a conditional expression is defined for any breakpoint or watch, the most recently defined expression is saved for use as a future default. This default can then be used simply by defining a null expression. Example:

```
AT .ERR (R4?,NE,'I-CATCHR')
AT .LOOP (R1,EQ,00000000)
T BY ()
```

- The first AT command creates a breakpoint located at ".ERR" that tests for whether or not R4 points to a location that contains the string, "I-CATCHR".
- The second AT tests for R1 being zeroed.
- The TRACE command specifies a null conditional expression, so the previously defined expression ("R1,EQ,00000000") is assigned to it. Thus, the trace will follow the program's execution until R1 is zeroed.

The **LIST BREAKPOINTS** command can be used to display the current default conditional expression.

Help COmmands Syntax Breakpoints Autocmds

Any one or more Z/XDC commands can be associated with any kind of breakpoint or WATCH. These commands are executed by Z/XDC only if and when the breakpoint is reached by user program execution and accepted by Z/XDC. (Conditional breakpoints are not accepted when the specified condition resolves FALSE.) WATCHs are evaluated by Z/XDC whenever Z/XDC receives control for **any** nonaborting reason.

If Z/XDC accepts multiple breakpoints located at a single address, and/or multiple WATCHs, then the automatic command strings are processed in the reverse order from which the accepted breakpoints and WATCHs were created (LIFO order). If no automatic commands are associated with any accepted breakpoints and WATCHs, then either a "WHERE" command or a "FORMAT PSW?" command (depending upon the current "SET TRACE ROLL/SCROLL" setting) is automatically executed.

Syntax:

```
verb ... 'command;command;...'
         "command;command;..."
         ''
         ""
```

'command;command;...'
"command;command;..."
When given, the automatic commands must be:

- Enclosed within quotes characters,
- Separated from each other by semicolons (;),
- And embedded opening quote characters must be doubled up ('' or "").

''
""

If just a null string is given, then this is a request to reuse the most recently previously defined automatic commands string. The **LIST BREAKPOINTS** command can be

used to display what that command string currently is.

verb and other operands
See HELP COMMANDS SYNTAX BREAKPOINTS.

In principle, any Z/XDC commands may be associated with a breakpoint; however, the END and GO commands, if used, should be placed last in the automatic commands string since any other commands placed following the END or GO will never be executed.

A string of automatic commands may include breakpointing commands which may themselves include strings of automatic commands. When strings are included within strings, all embedded quotes must be doubled and redoubled as the nesting gets deeper and deeper. See HELP BREAKPOINTS AUTOCMDS for a rather hairy example.

Examples:

AT R10!+138 'L R13;F R13!+4!+C!'
Suppose R10!+138 points into a subroutine that uses standard entry and exit linkage. Suppose also that within that subroutine R13 points to that subroutine's local save area which is chained to the caller's save area in the usual way. When the breakpoint address is reached (R10!+138), first the contents of R13 are displayed then the subroutine's return address is formatted and displayed.

AT .TOPLOOP ''
This places a persistent trap at the location named ".TOPLOOP" and associates with it the previously defined automatic commands string. In this case, that would be "L R13;F R13!+4!+C!".

T .EOD 'WHERE ,1;LIST REGS;T BY (.ANCHOR,EQ,00000000) ''ALARM ANCHOR IS LOST!''';GOT
OK, here's what's happening here:

- **T .EOD**
  A breakpoint is set at the location named ".EOD". It is both unconditional and transient. When execution reaches ".EOD", Z/XDC will receive control, accept the breakpoint, and remove it.

- **WHERE ,1;LIST REGS;T BY (.ANCHOR,EQ,00000000) 'ALARM ANCHOR IS LOST!'**
  When Z/XDC accepts the breakpoint at ".EOD", these are the commands that will be executed:
  ```
      WHERE ,1
      LIST REGS
      T BY (.ANCHOR,EQ,00000000) 'ALARM ANCHOR IS LOST!'
  ```
  Collectively, these commands (a) issue a one line display showing the current execution location (the .EOD), (b) display the general registers, and (c) kick off a BY-type trace that is conditional and has its own automatic command string. This T BY trace will follow execution until the .ANCHOR field is zeroed, and when that happens, it will issue an alarm message reporting that fact.

- **;GOT**
  This causes program execution to be resumed immediately after the T .EOD... command. In other words, the T .EOD... command causes the breakpoint to be set,

and then the GOT command command causes execution to resume.

- To summarize, the "T .EOD..." command and the "GOT" command are executed immediately. The "WHERE ,1", "LIST REGS", and "T BY..." commands are not executed until the breakpoint at .EOD is reached. And the "ALARM..." command is not executed until the T BY trace's conditional expression resolves TRUE.

For related information, see HELP BREAKPOINTS AUTOCMDS.

Help COmmands Syntax Characterstrings

Character Strings vs. String Data

Character strings are text strings that will be processed only as character data.

String data is a broader category that can include hexadecimal and decimal data in addition to character data. For more information about string data and the commands that use string data, see **HELP COMMANDS SYNTAX STRINGDATA.**

The following Z/XDC commands can accept character strings as operands:

ALARM - Displays a message and sounds the terminal's alarm for a period of time.

FIND - Searches storage for character data or hex data.

HKEYS - Defines a Built-in Help PF key.

KEYS - Defines a Z/XDC PF key.

NOTE - Saves an annotated address for display by the **LIST NOTES** command.

SCANLOG - Searches the current window's session log for a given string.

SET HKEYS - Defines a Built-in Help PF key.

SET KEYS - Defines a Z/XDC PF key.

SET PROFILE - The **DESCRIPTION=** operand defines descriptive text to be associated with the currently active session profile.

TRACE - When an automatic commands is given, it must be enclosed within quotes. Ditto for a comparison constant.

TRAP - When an automatic commands is given, it must be enclosed within quotes. Ditto for a comparison constant.

ZAP - The character data form of this command requires the character data to be enclosed withing quotes.

Enclosing Quote Characters Sometimes are Optional

All of these commands accept character string operands. For some of the commands, enclosing quotes are optional. For others, they are required, as follows:

- Required for **FIND, SET PROFILE** with **DESC=, TRACE, TRAP** and **ZAP.**
- Optional for **ALARM, [SET] HKEYS, [SET] KEYS** and **NOTE.**
- Required for **SCANLOG** only when the character data contains blanks, colons or semicolons.

When Enclosing Quote Characters are Required

Generally, when a command (such as **FIND** or **ZAP)** accepts string data:
- Quote characters are required to distinguish character data from hex data.
- Quote characters are required when a particular command's syntax permits the possibility of other operands following the character string.
- Quote characters are required to preserve instances of multiple blanks or commas or if the semicolon is present in the character string.

When quote characters are required, the parsing rules are as follows:
- The enclosing quote characters are removed.
- Embedded opening quote characters must be doubled. The pre-parser will singlize embedded doubled opening quote characters before presenting them for processing by the command.
- The character string will not be upcased.
- The **SCANLOG** command has its own rules about upcasing. For one thing, **SCANLOG** accepts a **c'string'** syntax for causing a string to be treated as being **case sensitive.** For more information, see **HELP COMMANDS SCANLOG.**

When Enclosing Quote Characters are Optional

Generally, enclosing quote characters are optional when a command's syntax dictates that the entire rest of the command string can **only** be a character string. Commands that fall into this category include **ALARM, [SET] HKEYS, [SET] KEYS** and **NOTE.**

When quote characters are optional, the only effect of their presence or absence pertains to embedded characters. The parsing rules:

- When the operand **starts** with a quote character:
  - The enclosing quote characters are removed.
  - multiple consecutive blanks or commas are not folded into one.
  - If the character string needs to contain the starting quote character, that starting quote character must be doubled-up. The parser will singlize doubled-up instances of the starting quote character. - The character string may contain semicolons (;) and these semicolons will not terminate the command.

- When the operand does **not** start with a quote character:
  - The character string may still have embedded quote characters, but they should not be doubled, and they will not be singlized.
  - a semicolon will terminate the command.

Pre-parsing.

Within Z/XDC, parsing is directed primarily by each individual command. But There also is a **pre-parser** whose role is limited to:
- Identifying the command name,
- Locating semicolons to isolate individual commands, but note that is a semicolon is within a quoted string, it **does not** terminate a command.
- Locating individual operands and creating a pointer vector for them.
- locating quoted strings.

- folding unquoted adjacent occurrences of blanks and commas into one character.

For those commands for which enclosing quotes are optional, only the command logic itself has knowledge of what actually is and is not a character string. The pre-parser only knows what is and is not enclosed within quote pairs.

Wildcards

Some commands that accept character strings also support the recognition of **wildcard** characters within those strings. For more information, see **HELP COMMANDS SYNTAX PATTERNMATCH.**

Examples - Setup

For all of the following examples, assume that an equate named **#** has been created to represent a location in storage.

Let’s create a Watch Window to keep an eye on the storage location to which the **#** equate has been assigned.

- **SET WINDOWS DELETE ALL**

- **SET WINDOW CREATE 33 CMDS=’FORMAT # DATA BYTES=40’**

Let’s create another Watch Window to display the **NOTEs** that we are about to create.

- **SET WINDOW CREATE 26 CMDS=’LIST NOTES’**

Examples - Character Strings with Optional Quoting

NOTE # This is an unquoted (’) NOTE MESSAGE
Displays "This is an unquoted (’) NOTE MESSAGE".

Notice:
- The embedded quote did not have to be doubled.
- All text following the 1st quote was displayed upcased (because it was upcased originally).

NOTE # ’THIS IS A QUOTED (’’xyz’’) NOTE MESSAGE’
Displays "THIS IS A QUOTED (’xyz’) NOTE MESSAGE".

Notice:
- The embedded quotes had to be doubled.
- Only the quoted value was displayed in lower case.

NOTE # abc’DEF’ghi Displays "abc’DEF’ghi".
NOTE # abc’’def’’ghi Displays "abc’’def’’ghi".
NOTE # ’ABC’’DEF’’GHI’ Displays "ABC’’DEF’’GHI".
NOTE # ’ABC’def’GHI’ Displays "ABC’def’GHI".

```
NOTE #    abc'def'ghi        Displays "abc'def'ghi".
NOTE #    abc''def''ghi      Displays "abc''def''ghi".
```

Examples - Character Strings with Required Quoting

```
ZAP  #='ABC''DEF''GHI'       Stores "ABC'DEF'GHI" at #.

ZAP  #='abc''def''ghi'       Stores "abc'def'ghi" at #.

SET KEYS 19        'LIST TASKS;;LIST RBS'
```
Assigns the string "LIST TASKS;;LIST RBS" to PF key shift-F7.

Help COmmands Syntax Escapecharacter

Z/XDC sometimes uses an escape character when processing entered data.

The escape character is the backslash (\).

If the character following the escape character has a special meaning, that meaning is disabled, and the character after the escape character is used literally (i.e., as itself).

For example:

- if a string is (or could be) a string with wildcard characters, if a wildcard character is preceded by the escape character, that character is treated literally (i.e., not a wildcard). LINK COMMANDS SYNTAX STRINGDATA CHARACTER
- a quote character, if preceded by the escape character, does not start a quoted string if outside a quoted string or does not terminate a quoted string if inside a quoted string. In both cases, the sequence <escape><quote> is replaced by <quote>.
- The escape character can escape itself. Specifically, if you wish to have a backslash be a part of the operand data, then specify it twice (\\).
- If the character following the escape character has no special meaning, it is processed as is. In other words, "\x" is the same as just "x".

Note that escaping a forward slash (/) in a USS path does not change the meaning of the forward slash as a separator between directory names and file names.

For **examples** of using escape characters, see **HELP COMMANDS SYNTAX PATTERNMATCH.**

Help COmmands Syntax NAmes

Names

Many items in z/OS have a 'name'. For example, member names, program names, dataset names, USS path names, USS file names, and so on.

When it comes to datasets and files, there are 2 broad classes of names: Classic z/OS dataset names and USS path/file names.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

CLASSIC - A name that follows several ancient rules regarding the valid characters and length.

USS - A name that follows rules for Unix File System names. In z/OS, these are USS names.

Help COmmands Syntax NAmes Classic

Traditional names

In z/OS, in many places, a **traditional** name is used or required:
- A segment of a dataset name.
- A program name.
- A PDS or PDSE member name.

A **traditional** name has the following restrictions: - the total length is often (but not necessarily) limited to 8 characters - The characters in the name must all be either alphabetic ('A-'Z'), numeric ('0'-'9'), or 'national' ('@', '#', '$') - Only uppercase alphabetic characters are allowed (note that many things, including Z/XDC, will upcase a provided name value before validating it) - the first character of the name must not be numeric.

In some cases, the acceptable characters will be changed, or the maximum length may be changed. For example: - An assembler label may contain alphabetic characters of any case, and the underscore('_') and the label may be up to 64 characters long.

Help COmmands Syntax NAmes Uss

USS names

Generally, items in USS (Unix System Services) are referenced by a file name.

A **file name** by itself generally does not uniquely identify a file.

A file in USS is named in a directory, which has its own file name.

This is recursive. A directory may be named in another directory, etc.

the total path to a given file is called a **path name**

Individual segments of a **path name** are separated by the '/' character (also known as the forward slash or solidus).

All segments in a path except the last one will be the file name of a directory. The last segment of a path name will be the actual file name.

A **path name** may be absolute or relative: - If the path name starts with a '/' then it is absolute and completely identifies the file. - If a path name does not start with a '/' then it is relative to the user's Current Working Directory (CWD) and this value and a '/' is prefixed onto the relative path name to obtain an absolute path name

A file name may be up to 255 characters long, and may contain any printable character except the '/' (forward slash or solidus). Note that a file name may contain embedded blanks and that case is important (so a file called 'x' and a file called 'X' are 2 distance files).

Note that IBM recommends that a file name should only contain certain characters if you ant to guarantee portability of the file name across all Unix variants, but this is only a recommendation.

A path name may be up to 1023 characters long. It must contain '/' characters to separate the individual file names. No file name's length may exceed 255 characters. Other than the first character being a '/' to signify an absolute path name, no segment can be null - that is 2 adjacent '/' characters are not allowed.

Help COmmands Syntax NUmericdata

Z/XDC command operands take dozens of forms, a couple of which are **numeric data.** Some numeric data operands must be given as pure decimal numbers. For example, the **SET LOG nnnnn** command is used to set how large (in lines) the Primary Window's scroll are is. In this case, **nnnnn** must be given as a pure decimal number.

For other operands, the numeric data is defined to be a pure hexadecimal number. Some examples are:

- A **stoken** value on a **SET ACCESSTO** command
- An **asid** number on a **LIST TASKS** command
- The **length** operand on an **EQUATE** command

- **Raw addresses** in address expressions.
- **Plus and minus offsets** also in address expressions.

More About Decimal Numeric Data

There really isn't much more to say. When a decimal value is called for, just type in the decimal digits. It's a simple as that. No commas, no framing, no nothing. Just the digits.

More About Hexadecimal Numeric Data

For hexadecimal data, it's a different story. Here, you have the option of providing either a hexadecimal number or a decimal number or a scaled decimal number:

- If you want to enter a hexadecimal value, then just do it. Just type the hex digits without framing, without commas and without spaces. But you can, however, insert **understrokes (_)** wherever you like to make large numbers more readable. The understrokes are ignored.

  Note, some load module names are syntactically indistinguishable from hex numbers. (Example: LE has a load module named **CDAEED.)** In such cases when ambiguities arise, Z/XDC will resolve the number as the module's name, not as a hex number. But to force numeric resolution, all you have to do is start the number with a **zero (0).** Example: **FORMAT 0CDAEED.**

- If you want to enter a decimal value (instead of hex), trail the digits with the letter **n.** Example: **FORMAT 16N%** resolves to the location of the System's CVT.

- If you want to enter a scaled decimal number, just trail the decimal digits with one of the following scaling factors:
  N - The number is not scaled.
  K - The number is multiplied by 2**10 (kibi-something)
  M - The number is multiplied by 2**20 (mebi-something)
  G - The number is multiplied by 2**30 (gibi-something)
  T - The number is multiplied by 2**40 (tebi-something)
  P - The number is multiplied by 2**50 (pebi-something)
  X - The number is multiplied by 2**60 (exbi-something)

For more details, see HELP ADDRESSING NUMERIC SCALINGCODES.

For more information, discussion and examples, see:
- HELP ADDRESSING NUMERIC
- HELP ADDRESSING SYNTAX OFFSETTERMS CONSTANTS

Help COmmands Syntax Dsnames

Several of Z/XDC's commands accept dataset names for operands. These commands include:
- DMAP
- LOAD

    MAP
    READ
    SET LOG
    SET MAPLIBS

When specifying a dataset name on any Z/XDC command, the following rules must be followed:

- When a preexisting dataset is being given, that dataset must be cataloged. Z/XDC does not support locating uncataloged datasets.

- The dataset name must conform to standard JCL syntax rules:
  - It must be from 1 to 44 characters long.
  - The name must consist of one or more segments each of which may be from 1 to 8 characters long.
  - Segments must be separated from each other by periods (.).
  - Each segment may contain only alphabetic (A-Z), numeric (0-9), or national (@ # $) characters.
  - The first character of each segment may not be numeric.
  - The exact syntax of a segment is described in **HELP COMMANDS SYNTAX NAMES CLASSIC**
  - For partitioned datasets, the dataset name may be immediately followed by a parenthesized member name. The member name:
    - May be from 1 to 8 characters long.
    - May contain only alphabetic (A-Z), numeric (0-9), or national (@ # $) characters.
    - The first character may not be numeric.

    For some Z/XDC commands, member names may be required. For others they may be optional, and for still others, they may be disallowed. See the particular command descriptions for specifics.

- Z/XDC does not support the syntax for referring to members of a generation data group.

- Z/XDC does not support the syntax for referring to temporary dataset.

- Z/XDC does not support TSO's conventions for dataset names. Specifically:
  - Dataset names must always be given fully qualified! Z/XDC never prefixes a name with a userid or profile prefix.
  - Dataset names must never be quoted ('dsname')!

Z/XDC supports a number of variable symbols for dataset names. They allow the dsnames to be reactive to such things as the current time, date, userid, jobname, profile prefix, etc. These variables are most useful when Z/XDC commands are executed from command scripts or DEAD traps. For more information, see:

    HELP COMMANDS READ
    HELP BREAKPOINTS DEADTRAPS

Z/XDC scans dsnames (and member names when present) for variable symbols at command parse time. When a variable symbol is encountered, it is replaced by its current value. After all substitutions have occurred, the result is rescanned for conformance to JCL syntax rules.

Variable symbols are always 2 characters long. The first character is an asterisk (*). The second character identifies the symbol. Example, In "DBCOLE.*DX.LOG", the variable symbol is "*D" (which returns the current date). After substitution, the

resulting dsname could be "DBCOLE.D981009X.LOG".

Z/XDC supports the following variable symbols:

****** - Is replaced by a single asterisk (*)

D - Is replaced by the current date formatted as "Dyymmdd".

J - Is replaced by the Home Address Space's jobname.

O - Is replaced by the Home Address Space's ownerid (from security). If no ownerid exists or if for some unknown reason the ownerid contains characters illegal for a dataset name, then Z/XDC proceeds as if *J had been specified instead.

P - Is replaced by the current TSO profile prefix string. If no profile prefix exists (as when Z/XDC is running in the batch), or if for some unknown reason the profile prefix contains characters illegal for a dataset name, then Z/XDC proceeds as if *U had been specified instead.

A TSO session's profile prefix string is set or cleared via TSO's PROFILE PREFIX(value)/NOPREFIX command. It can be displayed via TSO's PROFILE LIST command. For more information, go to ISPF's option =6 ("ISPF Command Shell") and type TSO HELP PROFILE.

Note, when Z/XDC is debugging a batch job via cdf/XDC, Z/XDC is running in the batch even though you might be connected to it from a TSO session! Accordingly, no profile prefix string exists in such cases.

T - Is replaced by the current time of day formatted as "Thhmmss".

U - Is replaced by the current TSO userid. If no userid exists (as when Z/XDC is running in the batch), or if for some unknown reason the userid contains characters illegal for a dataset name, then Z/XDC proceeds as if *O had been specified instead.

Note, when Z/XDC is debugging a batch job via cdf/XDC, Z/XDC is running in the batch even though you might be connected to it from a TSO session! Accordingly, no userid string exists in such cases.

X - Is replaced by Z/XDC's current clone name (usually "XDC").

Y - Is replaced by the current 4-digit year formatted as "Yyyyy".

You might notice, in the above descriptions, that for some of the variable symbols, a resolution failure fallback sequence has been defined. That sequence is: *P (prefix) -> *U (userid) -> *O (ownerid) -> *J. For example, if *P cannot be resolved as a TSO profile prefix, then Z/XDC tries to resolve it as a TSO userid. If that fails, etc. etc.

Examples:

SET LOG DSNAME *P.*Y.*D.*T.XDCLOG
Could refer to DBCOLE.Y1998.D981009.T091252.XDCLOG

LOAD TESTPGM *O.TESTLIB
Could refer to CSWQA.TESTLIB

Help COmmands Syntax Patternmatch

A growing number of Z/XDC commands, particularly **LIST** commands, that normally accept operands that are discrete names, are now accepting wildcard patterns as well. So instead of showing just one thing, such commands can now show all things that meet the criteria of the pattern.

So far (October 2020), the following commands accept wildcard patterns as operands:

- DEFINE DDNTRANS
- LIST ASID
- LIST DSPACES
- LIST ENQ
- LIST EQUATES
- LIST PGMS
- LIST SSCT
- LIBRARYLISTS SHOW

(Note, the data entry panel produced by the **LICENSE** command also accepts a kind of wildcard character, but those are different from what is described here. So for more information about that, see **HELP COMMANDS LICENSE DATA.)**

Wildcard characters permit a given search string to match multiple targets. Wildcard characters may be any of the following:

- **Question Mark (?)**
  This matches any single character.

- **Asterisk (*)**
  This matches any string of zero, one or more characters.

  The matching algorithms treat an asterisk as being **greedy.** When a string containing asterisks can have multiple interpretations, the match that is chosen will be the one in which the leftmost asterisk matches the longest string that does not cause the match on the entire string to fail.

- **Backslash (\)**
  The character following a backslash (\) is always treated literally. If a wildcard character (* or ?) is preceded by a backslash (\) that character is **not** treated as a wildcard character, but is treated as an instance of that character.

  Similarly, if you want to retain a backslash in the resulting string, use 2 backslashes (\\) to retain one backslash.

  It doesn’t matter whether the character following a backslash is special or normal. In either case, it is treated as is. It is processed just like any other normal character.

This means that in the case of normal characters, "\x" is the same as just "x".

Examples:

- **\?** or *****
  The result is a literal **?** or ***.**

- **GEO?***
  The result is a pattern that matches any string:
  - That is **four** characters or longer
  - And that starts with **GEO**

- **GEO\?***
  The result is a pattern that matches:
  - A string that is exactly 5 characters long
  - and that is exactly **GEO?***

- **\\?** or *****
  The result is a literal **** followed by the indicated wildcard character:
  - The **** is replaced by a literal **\.**
  - The **?** or ***** continue to be treated as wildcards.

- **\\\?** or *****
  The result is a literal **\?** or ***.**
  - The **** is replaced by a literal **\.**
  - The **\?** or ***** is replaced by a literal **?** or ***.**

- **GEO\\R\?**
  The result is a pattern that matches:
  - A string that is exactly 6 characters long,
  - and that is exactly **GEO\R?**

- **\G\E\O\\R\?**
- **\G\E\O\\\R\?**
  Both result in a pattern that matches:
  - A string that is exactly 6 characters long,
  - and that is exactly **GEO\R?**

Help COmmands Syntax Stringdata

Several Z/XDC commands expect one or more of their operands to be **String Data.** **String data** is literal data. It may be any mixture of hexadecimal data, character data, and decimal data. The following Z/XDC commands can accept **String Data:**

- **COPY** - Copies data from one storage location to another. A string may be specified for padding if the source is shorter than the target.
- **FIND** - Searches storage for the given string.
- **VERIFY** - Checks to see whether or not a given storage location already contains

the given string.

ZAP - AND's, OR's, XOR's, or stores the given string into a specified storage location.

(Note, other commands accept only **Character Strings** as operands, not string data. These commands include ALARM, HKEYS, KEYS, NOTE, SCANLOG, SET HKEYS, and SET KEYS. See HELP COMMANDS SYNTAX CHARACTERSTRINGS for more information.)

You can specify **String Data** in several different forms: Hexadecimal digits, EBCDIC characters, ASCII characters, signed or unsigned decimal integers, or a mixture of any of these. Z/XDC converts the data to internal binary for processing. Hex and character strings can be any length. Decimal integers can be 1, 2, 3, or 4 bytes wide.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

HEXADECIMAL - Syntax for hexadecimal string data.
CHARACTER - Syntax for character string data.
MIXED - Syntax for mixed hexadecimal and character string data.
DECIMAL - Syntax for decimal integer data.

Help COmmands Syntax Stringdata Hexadecimal

Syntax: HEXADECIMAL STRING DATA

Hexadecimal string data does not use any special framing characters. (Although, in the case of the **ZAP** and **VERIFY** commands, string data must be preceded by a special operator character: an equals sign.) Just type the digits at the point that the receiving command accepts them. The only restriction is that an even number of digits must be typed.

Pretty much anywhere a hexadecimal number is called for, a decimal number can be provided instead, followed by the letter **N.** Notes:

- The reverse is **not** true. when a decimal number is called for, a decimal number must be given. A hexadecimal number will not be accepted.

- When a decimal number is called for, the number must be given **without** a trailing **N.** Use a trailing **N** only when a hexadecimal number is called for but you wish to provide a decimal number instead.

Examples:

```
 +-------------------- To address
 |   +---------------- To length
```

```
     |   |     +----------- From address
     |   |     |   +------- From length
     |   |     |   |   +--- Fill string
     V   V     V   V   V
```

COPY R5? 1024N R2? 100 0700
X’100’ bytes (256 decimal) of data is copied from storage pointed to by R2. It is copied into 1024 bytes (decimal) of storage pointed to by R5. The excess 768 bytes of the target are filled with 384 repetitions of the padding string: X’0700’ (i.e. NOPRs).

FIND C481A58540C3969385,R1?
Storage is searched, starting at the location pointed to by R1, for the 9-byte wide mixed-case string, C’Dave Cole’.

FIND ’Dave Cole’,R1?
Same thing. See HELP COMMANDS SYNTAX STRINGDATA CHARACTER for more information.

VERIFY R1?=0032F8
The location pointed to by R1 is checked to see if it contains the given 3-byte wide string.

ZAP R1+3=02
The value X’02’ is stored into the lo-order byte of R1. The other 3 bytes of R1 are unchanged.

ZAP R1+3=2
This is **invalid.** The given hexadecimal string ("2") is not an even number of digits long.

ZAP R1?+4=|0180
This ORs a couple of bits into two bytes of storage located 4 bytes past the location pointed to by R1. The remaining bits at that location are left undisturbed.

HEXADECIMAL STRING DATA VS. ADDRESS EXPRESSIONS

For the **ZAP** and **VERIFY** commands, please be careful not to confuse hexadecimal string data with address expressions. The two can sometimes look alike, but how they are processed is quite different. The **ZAP** and **VERIFY** commands, can accept both kinds of data. Consider, for example, the following two **ZAP** commands:

ZAP R1=FF
ZAP R1,FF

In the first case, **FF** is string data, but in the second case, **FF** is an address expression. This is because, in the syntax of the **ZAP** command (and

the **VERIFY** command), when the operand separator is an equals sign ("="), the second operand is interpreted by the command as being a string. On the other hand, when the separator is a comma (",") or a blank (" "), then the second operand is processed as an address expression.

The first **ZAP** command (above) sets the hi-byte of R1 to X'FF' and leaves the remaining 3 bytes unchanged. The second **ZAP** command, on the other hand, sets the **lo-byte** of R1 to X'FF' and it zeros out all the remaining bits in the register except for the hi-order one or eight bits (depending upon the addressing mode of the user's program). In other words, when the hex digits are processed as an address expression, Z/XDC expands it out with zeros until it is either 24 bits or 31 bits wide, and then (in the case of the **ZAP** command) it stores that value into the lo-order 24 or 31 bits of the 4-byte field (or register) that is the **ZAP** command's target. **So be aware and be careful!**

Here's another difference between hexadecimal string data and an address expression that happens to consist of only hexadecimal digits. In the string data case, an even number of digits must be given, but in the address expression case, Z/XDC accepts either an even number or an odd number of digits. EXAMPLES:

ZAP R1=FFF <invalid>
ZAP R1,FFF <valid>

Help COmmands Syntax Stringdata Character

Syntax: CHARACTER STRINGS

The following rules and characteristics apply to character string data:

- Character string data must be enclosed within a quote character.
- The quote characters are:
  - the Apostrophe or tic (').
  - the literary quote (").
- If a character string includes the opening quote character, then it must be doubled up.
- Character data is interpreted as being either EBCDIC or ASCII according to whether SET FORMAT EBCDIC or SET FORMAT ASCII is in effect.

Examples:

COPY .MSGBUF 100 .SHORTMSG 33N " "
A short message (33 bytes long) is copied into a 256-byte long message buffer. The excess space is filled with blanks.

Note, if SET FORMAT EBCDIC is in effect, then a blank is X'40'. On the other hand, if SET FORMAT ASCII is in effect, then a blank is X'20'. See HELP COMMANDS SET FORMAT for more information.

FIND 'DAVE CODE',R1?
Storage is searched, starting at the location pointed to by R1, for the 9-byte wide string, C'DAVE COLE'.

Notice that the search string is in upper case. If you want to search for lower case letters, then the following example will work just fine.

FIND 'Dave Cole',R1?

VERIFY R1?='IT''S UGLY'
The location pointed to by R1 is checked to see if it contains the given 9-byte wide string, "IT'S UGLY".

ZAP R1+3=''''
One single-quote character ("'") is stored into the lo-order byte of R1. The other 3 bytes of R1 are unchanged.

Z/XDC's interpretation of character strings is affected by the "SET FORMAT ASCII/EBCDIC" command. If SET FORMAT EBCDIC is in effect, Z/XDC will leave character strings that it receives from the terminal unchanged. Otherwise (i.e. if SET FORMAT ASCII is in effect), Z/XDC will translate them to ASCII codes before using them. See HELP COMMANDS SET FORMAT for more information.

Example:

When **SET FORMAT ASCII** is in effect:
- FIND 'Dave Cole',R1?"

(from the example given above) becomes equivalent to:
- FIND 4461766520436F6C65,R1?

Help COmmands Syntax Stringdata Mixed

Syntax: MIXED HEXADECIMAL AND CHARACTER STRINGS

A mixed hexadecimal and character string can be created simply by enclosing the character portions within quote characters leaving the hexadecimal portions outside of the quote characters. The hexadecimal portions of the string conform the hexadecimal syntax rules, while the character portions conform to the character string syntax rules.

One important restriction to remember: Hexadecimal strings must always be an even number of digits long. Therefore, the switching from hexadecimal syntax to character syntax can occur only after an even number of hex digits has been given.

Z/XDC allows you to switch back and forth between hexadecimal syntax and character syntax as often as you like.

Examples:

FIND 'D'81A585" C"969385,R1?
Storage is searched, starting at the location pointed to by R1, for the 9-byte wide

mixed-case string, C’Dave Cole’.

VERIFY R1?=0008’PARMTEST’
The location pointed to by R1 is checked to see if it contains 2 bytes containing X’0008’ followed by 8 bytes containing the word "PARMTEST".

ZAP R1+1=02"HI"
The value X’02C8C9’ is stored into the 3 lo-order bytes or R1. The hi-order byte is left unchanged.

ZAP R1?=12D’PARMTEST’
This is invalid because "12D" is a hexadecimal string containing an odd number of digits.

Help COmmands Syntax Stringdata Decimal

Syntax: DECIMAL INTEGER NUMBERS

Decimal integer number syntax allows you to enter binary data as a decimal number. You can give either a signed or unsigned number, and you can tell Z/XDC that you want it to be converted to a 1-byte, 2-byte, 3-byte, or 4-byte wide value. Z/XDC processes the given decimal number as a binary integer (signed or unsigned), and converts it to binary according to the rules of binary integer arithmetic. See IBM’s various **Principles of Operation** manuals for more information:

- In SP/370 its GA22-7000
- In XA/370 its SA22-7085
- In ESA/370 its SA22-7200
- In ESA/390 its SA22-7201
- In z/OS its SA22-7832

(Yeah, I’ve still got them all.)

Generally, a decimal number is given as a quoted string of signed or unsigned decimal digits preceded by a single-character "keyword" that indicates how many bytes wide the converted value is to be. Example: **F’1307’** represents a 4-byte wide decimal number whose binary value is X’0000051B’.

Note, Z/XDC does **not** support embedding commas or periods amongst the decimal digits.

The following decimal number formats are recognized by Z/XDC:

```
    F’number’ - "Number" is converted to a 4-byte wide value.
    4’number’ - "Number" is converted to a 4-byte wide value. (Same as F)
    3’number’ - "Number" is converted to a 3-byte wide value.
    H’number’ - "Number" is converted to a 2-byte wide value.
    2’number’ - "Number" is converted to a 2-byte wide value. (Same as H)
    1’number’ - "Number" is converted to a 1-byte wide value.
```

In all cases and for all widths, "number" may be given either as a signed or unsigned string of decimal digits. In all cases, the maximum and minimum values that

can be given depend **both** upon the field width and upon whether or not the given number is signed or unsigned. Note the following:

- Unsigned numbers cannot, of course, be negative.
- The maximum positive value of an unsigned number is twice that (plus 1) of the maximum permitted positive value of signed numbers.
- The binary values of the high half of the unsigned positive number range are identical to signed negative numbers. (That's not my fault. That's just how binary integer representations work inside computers.)
- The binary values of signed positive numbers are identical, of course, to the low half of the unsigned number range.

More precisely, the following table gives the exact minimum and maximum values that Z/XDC accepts for both signed and unsigned decimal numbers for all supported field widths.

FIELD WIDTH	UNSIGNED MINIMUM VALUE	UNSIGNED MAXIMUM VALUE	SIGNED MINIMUM VALUE	SIGNED MAXIMUM VALUE
1-BYTE	0	256	-128	+127
2-BYTES	0	65,535	-32,768	+32,767
3-BYTES	0	16M-1	-8M	+8M-1
4-BYTES	0	4G-1	-2G	+2G-1

Examples:

FIND 3'-4000000',R1?
Storage is searched, starting at the location pointed to by R1, for the 3-byte wide value X'C2F700'.

VERIFY R1?=H'8'
The location pointed to by R1 is checked to see if it contains the 2-byte wide value, X'0008'.

ZAP R1+3=1'255'
ZAP R1+3=1'-1'
These two commands are identical: The 1-byte wide value, X'FF' is stored into the lo-order byte of R1. The other 3 bytes of R1 are unchanged.

ZAP R1+3=1'+255'

This is invalid. The maximum permitted **signed** value for a 1-byte wide field is "+127".

COPY AR5? 5000,,,1'255'
Five pages of storage (i.e. 20K bytes) located at the address pointed to by **general** register R5 in the address/data space designated by **access** register AR5 are filled with X'FF's. The copy source is null (as indicated by the three consecutive commas), so the copy sink is filled completely from the padding string: 1'255'.

Help COmmands ?

Command Dialogs are comprehensive dialog panels that provide both commentary and fill-in fields for constructing and issuing some of Z/XDC's more complex commands. For more information, see **HELP CMDDIALOGS.**

To display a selection list of all available Command Dialogs, simply type a lone **?** (question mark) on the command line.

Syntax:

?
[a lone question mark all by itself]

Operands: none

Help COmmands ADeferred

The **ADEFERRED** command is used to set deferred **persistent** breakpoints.
The **TDEFERRED** command is used to set deferred **transient** breakpoints.
The **HDEFERRED** command is used to set deferred **hooks.**

The ADEFERRED and TDEFERRED commands are documented here. The **HDEFERRED** command, although quite similar, **is documented separately** at HELP COMMANDS HDEFERRED.

A **deferred** breakpoint is one that is to be set into a load module that has not yet been brought into storage. Eventually, when the module is loaded, the breakpoint will be set at that time.

Note that at this time, deferred breakpoints (of any type) cannot be set in programs that reside in a hierarchical file system (i.e., loaded by the loadhfs and loadHFSextended USS system calls).

Modules are loaded into storage as a result of LINK(X), LOAD, ATTACH(X), and XCTL(X) macros or the Loadhfs and LoadHFSextended USS system calls executed by your program. When a target module is read into storage as a result of one of these macros or

system calls, then a Z/XDC support routine receives control from the System. After making appropriate security checks, this routine "clones" (i.e. makes a copy of) the deferred breakpoint definition in order to create an active breakpoint in the target module. All the attributes of the deferred definition are copied and assigned to the active breakpoint.

If the System decides to satisfy the LINK(X), LOAD, ATTACH(X), or XCTL(X) macro or the Loadhfs and LoadHFSextended USS system calls with a copy of the target module that **already resides** in storage, then **the deferred breakpoint definition is ignored!** No new active breakpoint is created. Deferred breakpoints affect **only new copies** of the target module that are read from a library on disk (or from System cache).

If a copy of the target module already resides in storage at the time that the ADEFERRED or TDEFERRED command is issued, then that copy remains unaffected by the command. In order to set a breakpoint into such a copy, you must use the **AT, ATX,** or **TRAP** commands.

A **persistent** deferred breakpoint definition remains in existence for the life of the debugging session or until it is removed via the **OFF** command. This means that if additional copies of the target module are loaded into storage by subsequent LINK(X), LOAD, ATTACH(X), or XCTL(X) macros or the Loadhfs and LoadHFSextended USS system calls then additional active breakpoints will be generated by the deferred breakpoint definition until that definition is removed via an **OFF** command.

A **transient** deferred breakpoint definition remains in existence either for the life of the debugging session or until one of its generated active breakpoints is reached during program execution or until it is removed via the **OFF** command. This means that active breakpoints will be set into the target module **each** time a copy is brought into the Home Address Space’s storage from disk. This will continue until one of those active breakpoints is reached by user program execution, at which time both that breakpoint, and all of its siblings (if any), and its parent deferred breakpoint definition will all be automatically deleted.

Deferred breakpoints are supported only for modules that are loaded into the Home Address Space’s **private** area. Deferred breakpoints are not set in modules that are loaded into common storage.

Deferred breakpoints also are not set when **both** of the following are true:
- The module is loaded by the System into **protected** storage.
- Z/XDC was running **non-authorized** when it created the breakpoint definition.

The general syntax of the ADEFERRED and TDEFERRED commands is:

ADEFERRED =familyid addressexpressions (conditional) ’cmd;cmd;...’
TDEFERRED

addressexpressions
This must be a list of one or more address expressions separated from each other by commas or blanks. In this context, only certain types of address expressions are valid. Specifically, the following restrictions must be met:

- The **base** of the expression must be the name of a load module (primary or alias). It **may not be** a register, a PSW, an equate or dsect name, or any field name. The named module either may or may not already be loaded into

storage. Note, however, that the breakpoint will **not** be set into any currently loaded copies of the module. (You will have to use the AT, ATX or TRAP command for that.) Deferred breakpoint definitions are effective **only for future** loadings of the module.

- If a module map of the load module has previously been loaded via either the MAP or **DMAP** command, then the given load module may be qualified by a csect name (e.g. "AD SUBMIT.IKJEFF10"). For more information about using the **DMAP** command to load module maps, see HELP MAPS CSECTSASDSECTS.

- If the given load module name is **not** qualified by a csect name, then the name refers to the module's entry address **(NOT!** its first byte of storage, i.e. not its load point).

- If you wish to refer to a module's load point, then qualify the module's name with **.X#1** (e.g. **TD SUBMIT.X#1).** This is useful if the module's entry address is different from its load point.

- The address expression may be modified by offset values (e.g. **AD OFFLOAD+1DC").** Both positive and negative offsets are permitted.

- However, the expression may **not** contain indirect operators (% ? !) or built-in functions.

Addressexpressions is a required operand. All other operands may be omitted. For detailed syntax information about all operands other than addressexpressions, see **HELP COMMANDS SYNTAX BREAKPOINTS.**

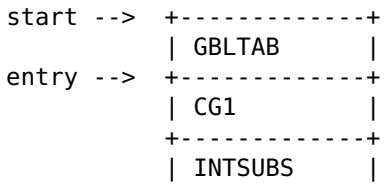
If a family identifier or a conditional expression or automatic commands are specified, then they become attributes of the deferred breakpoint definition, but they do not affect the operation of the definition in any way. Instead, these attributes are copied to all active breakpoints generated by the deferred breakpoint definition. Thus, these attributes apply to the generated active breakpoints, not to the deferred breakpoint.

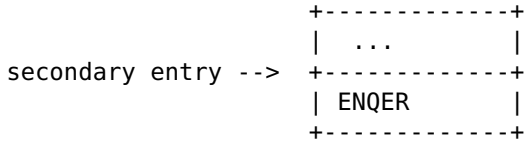
Examples:

Suppose that a module named CROSSGUN exists and has the following characteristics:

- It has not yet been loaded into storage.
- Its entry point is at a csect named GC1, and that csect is **not** first in the load module.
- The csect that is physically first in CROSSGUN is GBLTAB.
- The module has an alias named ENQER pointing to a csect of the same name.
- It has another csect named INTSUBS.
- However, a module map has **not** yet been loaded by either the MAP or DMAP command.

Schematically, the load module looks like this:





Then the following commands will have the following results.

AD CROSSGUN+3DC
When the CROSSGUN module is eventually loaded into storage via its primary name (CROSSGUN), a persistent breakpoint will be set at +X'3DC' past the start of the GC1 csect.

TD ENQER
When the CROSSGUN module is eventually loaded into storage via its alias name (ENQER), a transient breakpoint will be set at the start of the ENQER csect.

DM CROSSGUN.
TD CROSSGUN.ENQER
The DMAP command reads CROSSGUN's module map as if it were a dsect. Once the map is available, Z/XDC can refer to it in order to learn where CROSSGUN's csects are located. Later, when the CROSSGUN module is loaded into storage via its primary name, a transient breakpoint will be set at the start of the ENQER csect.

TD CROSSGUN.X#1+2C
TD CROSSGUN.GBLTAB+2C
Both of these commands have the same result: When the CROSSGUN module is brought into storage via its primary name, a transient breakpoint will be set at +X'2C' past the start of the GBLTAB csect. The difference is that the second command requires the availability of the module's module map ("DM CROSSGUN."), while the first command does not.

TD ENQER.X#1+2C
TD ENQER.GBLTAB+2C
If the CROSSGUN module is brought into storage via its alias name, the one or the other of these commands will be necessary for setting the trap at the GBLTAB+2C location.

AD CROSSGUN.INTSUBS
If a module map is not available, then this command will fail because Z/XDC will not know where in the load module to find the INTSUBS csect.

Related information can be found by typing HELP BREAKPOINTS.

Information about deferred hooks can be found at HELP COMMANDS HDEFERRED.

Help COmmands ALarm

The **ALARM** command starts a persistent alarm. A special alarm message panel is displayed, and the terminal's audible alarm is sounded repeatedly for up to five seconds. You may specify the message to be displayed, or you can let a default message be displayed.

The alarmed message will display and sound continuously for up to five seconds; however, you can abort the message sooner than that with an attention signal **(ATTN** key on remote terminals, **PA1** key on locally attached terminals).

Syntax:

```
ALARM message
      'message'
      omitted
```

message
'message'
The **ALARM** command verb may be followed by any message. The message will be displayed when the alarm is sounded.

- **message**
  If the message is **not** framed by quote characters, then it may not contain semicolons (;) or colons (:). Also, it may contain embedded quote characters. If it does, then they need not be doubled.

- **'message'**
  If the message **is** framed by quote characters, it may contain semicolons and colons. The framing quote characters will be removed, and embedded doubled quote characters will be singlized.

For more information, see **HELP COMMANDS SYNTAX CHARACTERSTRINGS.**

omitted
If no message is given, then a default message will be displayed.

Using ALARMs With Conditional Expressions

One good use of the **ALARM** command is to notify you when a long running process completes. For example, you may want to include an **ALARM** command among the automatic commands associated with a **conditional breakpoint** to notify you when the conditional is finally met.

Another very good way to use the **ALARM** command is in conjunction with the **TRACE**

W command. If **ALARM** is made an automatic command associated with a **TRACE W,** then when the WATCH'd condition becomes TRUE, the alarm will be triggered. Example:

> **TRACE W (R8+3,NE,45) 'ALARM R8 HAS CHANGED!'**
> When Z/XDC detects that the lo-order byte of register R8 no longer contains X'45', the alarm will sound, and the message, "R8 HAS CHANGED!" will be displayed.

A related subject: To control whether or not error messages cause an alarm to be sounded, see **HELP COMMANDS SET BELL.**

Using PINGs With Conditional Expressions

PING is not a valid Z/XDC command! But then the automatic commands associated with a breakpoint do not have to be valid commands.

Personally, I often find it useful for a WATCH's automatic command to be an **invalid command!** Why? Because when the WATCH condition is met:

- The command is queued for execution.
- Its execution fails with a parse error.
- This causes Z/XDC to abort all incoming command streams from all sources. In other words, things stop!
- The invalid command is displayed.
- And the workstation's **bell rings!**

That generally gets my attention... Especially when it's been awhile since I created the WATCH, and I've gotten all wrapped up in tracing through the code waiting for the WATCH to trigger.

For a deeper discussion of using invalid commands as a trap's automatic command, see **HELP COMMANDS TRACE W.**

Examples:

T BNC (R1+3,EQ,05) 'ALARM Hey! Time to wake up.'
This command is a conditional trace. User program execution proceeds until the lo-order byte of R1 contains X'05'. (The test is made only at branch instructions that will in fact branch.) When the condition is met, user program execution stops, the alarm message is displayed, and the terminal's alarm (bell, buzzer, beep, or custom designed wisecrack) is sounded at one second intervals for either up to five seconds or until you send an attention signal from your terminal.

T BNC (R1+3,EQ,05) 'ALARM ''Hey! Time to wake up; Bad things are happening!'''
When the message itself is quoted, it can include colons and semicolons.

T BNC (R1+3,EQ,05) 'PING! R1''s been messed with!'
This example shows an alternative to using an **ALARM** command. Here, an invalid command is used instead.

When the given conditional is met, and the given automatic command is invalid, the

following happens:
- The **PING** "command" is queued for execution.
- But **PING** fails because it is not a valid command.
- This causes Z/XDC to abort all incoming command streams from all sources.
- In other words, all things stop!
- The invalid command is displayed.
- And the workstation's bell rings.

Help COmmands ASterisk

Z/XDC recognizes, echos, and otherwise ignores a comment command. Its syntax is:

```
*blah blah blah ...
* blah blah blah ...
*blah;*another comment ...   [The semicolon ends the first comment]
*blah;* another comment ...  [The semicolon ends the first comment]
*blah;a new command ...      [The semicolon ends the comment]
*blah; blah blah ...         [The semicolon blank does not end the comment]
```

blah
This is a text string that may consist of any characters. Sometimes, the text may include semicolons (Z/XDC's normal command delimiter), but sometimes not.

A comment string may include semicolons (;) as part of the comment, but sometimes a semicolon instead can be used to delimit a comment so that it can be followed by a command or another comment in the same command string. Here are the rules:

- When a comment string contains a semicolon followed directly by an asterisk **(;*),** the semicolon ends the current comment string, and the asterisk starts a new one. Example:

  ```
  *** comment string 1 ***;*** comment string 2 ***
  ```

  This would be displayed as:

  ```
  *** comment string 1 ***
  *** comment string 2 ***
  ```

- When a comment string contains a semicolon followed directly by an alphabetic character (e.g. ***blah;LIST TASKS),** the semicolon ends the current comment string, and the alphabetic characters starts a new command (presumedly). Example:

  ```
  *** Here is a LIST TASKS display ***;list tasks
  ```

  This would be displayed as:

  ```
  *** Here is a LIST TASKS display ***
  TCB#  PROGRAM NAME (ASID 003F, DBCOLE3)
  1     IEAVAR00
  2     . IEESB605
  3     . . IKJEFT01, JOBSTEP (STEPLIB)
  ```

- When a comment string contains a semicolon followed directly by anything else (by a blank, by a digit, by whatever, e.g. ***blah; more text**), the semicolon is considered to be part of the commentary and does **not** delimit the comment string.

Example:
***** Here is a LIST TASKS display ***; list tasks**

This would be displayed as:
***** Here is a LIST TASKS display ***; list tasks**

Being able to delimit comment strings is useful when presenting commands to Z/XDC via lengthy command strings, as would be the case when using command-type **#DIE** macros.

Help COmmands AT

The **AT** command is **identical** to the **TRAP** command except:
- The **AT** command creates persistent breakpoints.
- The **TRAP** command creates transient breakpoints.

For complete information, see HELP COMMANDS TRAP.

Help COmmands ATX

The **ATX** command is **identical** to the AT command except:

- The **ATX** command creates persistent breakpoints that are **super persistent:** They must be specifically referenced, either by name or by type, by an **OFF** command operand.

- They also can sometimes be removed by **O** (off) and **X** (toggle) shortcuts. See **HELP SHORTCUTCMDS O** for more information.

The **AT** command, on the other hand, creates persistent breakpoints that can be removed by any **OFF** command that references them either specifically or generically. For complete information, see **HELP COMMANDS OFF.**

Help COmmands CD

The **CD** command can be used to alter the USS Current Working Directory (CWD).

Note that this command may cause the current task to be dubbed. (An operand controls this)

If no specific CWD has been set, the user's home directory is the initial CWD.

Note: **The Unix Shell command** CD - **is** NOT **supported.** Z/XDC does not have access to the current environment variables and as a result is unable to toggle the CWD.

Syntax:

CD **[directory-name]**

[DUB=option]

directory-name
The new directory name.

The new directory name can be absolute (starts with '/') in which case the new CWD is set to the specified directory name.

The new directory name can be relative (does not start with '/') in which case the new CWD is set to the existing CWD suffixed with '/' and the specified name.

If the new directory name is omitted the user's home directory is used as the new CWD (as if **CD ~/** was specified)

DUB=option
This operand is parsed and removed from the command before other operands are processed, regardless of its position in the command. It (or its default) controls the processing if Z/XDC determines that dubbing is required.

For a description of the possible values, their meaning, and the interaction with the DUB profile facility please see **HELP DEBUGGING USS OPERANDS DUB=**

Examples:

CD mydir
the directory mydir is suffixed (along with '/') to the current CWD and set as the new CWD.

CD
the user's home directory is made the CWD.

Help COmmands COMmentary

The **COMMENTARY** command causes a blank fullscreen panel to be displayed into which you may type up to 50 lines of arbitrary commentary. Whatever you write is recorded in the debugging session's log file (see HELP FULLSCREEN LOGGING). By this means, you can annotate the session log with whatever information you choose.

Syntax:

COMMENTARY

Operands - None

When you are finished entering your commentary, you can do either of the following:

- Use the **END** command or **QUIT** command to return to your normal debugging session and cause Z/XDC to write the commentary to the session log. You will then see your commentary displayed in your debugging session's main window.

- Use the **CANCEL** command to return to your normal debugging session and cause Z/XDC to discard your commentary.

Help COmmands CONsole

This command has been replaced by the **SET CONSOLE** command. See HELP COMMANDS SET CONSOLE for more information.

Help COmmands COPy

The **COPY** command copies data from a source location to a target location with optional padding from a "fill string". The target location must be qualified by an explicit length. The source location, if given, may be qualified by an explicit length, but it need not be. The lengths (or length) may be any non-negative value. They need not be the same value, and they may be variable.

The target can be any virtual storage location, real storage location, or a retry level register (general, access or floating point). The source can be any storage location (virtual or real), any register (any at all), a PSW, or null.

The source and target lengths can either be a specified value, or they can be taken from the contents of any general register, or they can be the resolved value of an address expression (Example: The resolved value of "PSA.PSATOLD" is X'21C'-hex or 540-decimal.) The source length may be omitted, the target length must be given.

If the target length is shorter than the source, then the source is truncated, only as much of the source is copied as will fit into the target.

If the source length is shorter than the target, then padding from a fill string is copied to the target location immediately following the data copied from the source.

An omitted source operand, or a source operand having a length of zero causes the target location to be filled solely by padding from the fill string. The source location is not referenced.

If the source operand is omitted, then the source length must also be omitted; however, a fill string can still be given. Just separate the padding operand from the target length operand by three commas. Example: **COPY HERE 100,,,'PAD'**

A target length of zero causes the COPY command to NOP, no reference is made to either the source or target location.

The default fill string is X'00'. The fill string can be one byte long or longer. When padding is needed, Z/XDC behaves is if it had prefilled the entire target with as many replications of the pad string as will fit, and then it overwrites the start of the target with the source data. In other words, the replicated pad string aligns with the start of the target, **not** with the end of the source data.

Copying is done by successively reading from the source location (or fill string), followed by writing to the target location. The amount of data read or written in each iteration is unpredictable. Destructive overlap is not detected and will cause unpredictable results.

If Z/XDC is running non-authorized, then the PSW protect key used by COPY is that of the user program's retry level environment (usually key 8). When Z/XDC is running authorized, the PSW protect key used is zero. This allows reading and writing of most storage locations. Regardless of the PSW key used, Z/XDC checks System Security for each storage access. (See HELP SECURITY for more information.)

Syntax:

```
COPY tgt-addr tgt-len src-addr src-len fill-string
C                     omitted  omitted omitted
```

The operands are positional. Each operand is separated from the next operand by a comma or blanks. The first two operands are required. The last three operands are optional. To omit an embedded operand, represent the omission with a comma. Example: "COPY .BUFFER R0,,,40" clears a buffer (whose length is specified in R0) to blanks.

tgt-addr
This is an address expression for the target location. The target location may also be a retry level register: general, access, or floating point.

tgt-len
This is the length of the target location. This can be anything from a constant, to an expression, to a value from storage. Examples:

- A hexadecimal number. Examples:
  44 (equals 68, decimal)
  DC9
  0E

- A decimal number. Example:
  45N

- A scaled decimal number. Example:
  4K (same as 4096N and 1000)

- The contents of a register. Examples:
  R4 (The 4-byte wide contents of general register 4)
  R4? (The lo-order 31 bits of R4)
  R4% (The lo-order 3 bytes of R4)

0+X3(R4+1) (ditto)
0+X2(R4+2) (The lo-order 2 bytes of R4)

- An arithmetic expression. Examples:
  24N+X1(R2+3)
  2G-1 (same as 7FFFFFFF)

- The contents of a location in storage. Example:
  0+X1(.MSGLEN)

- The resolved value of an address expression. Example:
  .BUFSIZE - (This resolves to the **address** of .BUFSIZE.)
  .BUFSIZE? - (This resolves to the **contents** of .BUFSIZE, assuming that .BUFSIZE is 4 bytes wide.)
  0+X2(.BUFSIZE) - (This resolves to the contents of .BUFSIZE, assuming that .BUFSIZE is 2 bytes wide.)

src-addr
This is an address expression for the source location. The source location may also be any register or PSW specification. If src-addr is omitted, then the target will be filled entirely from the given or defaulted fill string. If src-addr is omitted, then src-len also must be omitted.

src-len
This is the length of the source location. Like "tgt-len" described above, this can be anything from a constant, to an expression, to a value from storage.

If src-addr is given and src-len is omitted, then tgt-len will be used as the default src-len.

If src-addr and src-len are both omitted, then zero will be used as the default src-len, thus causing the target to be filled entirely from the fill string.

If src-addr is omitted, then src-len must also be omitted.

fill-string
This is an optional padding string of up to 256 characters. If omitted, then X'00' will be used as the default fill string. For specific syntax information, see HELP COMMANDS SYNTAX STRINGDATA.

When padding is needed, Z/XDC behaves is if it had prefilled the entire target with as many replications of the pad string as will fit, and then it overwrites the start of the target with the source data. In other words, the replicated pad string aligns with the start of the target, **not** with the end of the source data.

Examples:

COPY .BUFFER 0+X2(.BUFLEN)
Assuming that .BUFLEN is a 2-byte wide field containing the length of .BUFFER, this

command clear the buffer to zeros.

COPY R4?ASID(PASID) 200N R3?ALET(AR5)
This command copies 200 bytes (decimal) from the location pointed to by general register R3 in the address/data space indicated by access register AR5. The data is copied to the location pointed to by general register R4 in the retry level Primary Address Space.

COPY R7?ASID(PASID) 3000 R6?ASID(SASID) 2000 'THIS IS PADDING'
This command copies 2000 bytes (hexadecimal, i.e. 2 pages) from the location pointed to by general register R6 in the retry level Secondary Address Space. The data is copied to the location pointed to by general register R7 in the retry level Primary Address Space. The target location is 3 pages long ("3000" hex), and the source data is only 2 pages long, so the remaining page is filled with repetitive copies of the string: C'THIS IS PADDING'.

The padding repetitions are aligned with respect to the start of the target; consequently, in this case the first repetition is truncated on the left by two characters ("IS IS PADDING" remains), and the last repetition is truncated on the right by 12 characters (only "THI" remains).

GETMAIN 1000
COPY AREA 1000,,,47000000
The GETMAIN command:
- Obtains a page of storage from the Home Address Space.
- Zeros it.
- Automatically assigns an equate named AREA to represent the obtained storage's location.

The COPY command fills the entire page with "NOP 0" instructions.

COPY R1 4 AR2
This command copies the entire contents (4 bytes) of retry level access register AR2 into retry level general register R1.

COPY R14 4 PSW+4;T R14!;GOT R15?
The COPY command copies the rightmost 4 bytes of the retry PSW (the resume address) to retry level register R14. (The entire resume address, including the AMODE indicator, is copied.) The next command, "T R14!", sets a transient breakpoint at the current resume address. Subsequently, the "GOT R15?" command is executed, which resumes the program at the location specified in retry level register R15.

In effect, these commands perform a CALL to the subroutine pointed to by register R15, with the current resume address as the return address in R14. When the called subroutine returns, the transient breakpoint is encountered and control returns to Z/XDC.

TIP: This sequence of commands can be assigned to a PF key or placed in a READ file

for convenient use.

COPY R5 1 R4+3
This command copies 1 byte from the rightmost byte of retry level R4 to the leftmost byte of retry level R5.

COPY R4! R5 R6! R7+1 ABCDEF
COPY R4! X4(R5) R6! X3(R7+1) ABCDEF
These two commands function identically. They both copy data from the location pointed to by retry level register R6 to the target location pointed to by retry level register R4. The source length is extracted from the rightmost 3 bytes of retry level register R7. The target length is extracted from the retry level register R5. The padding string, if needed, is X'ABCDEF'.

COPY R4! 24n+X4(R2) R6! R7
This command copies data from the location pointed to by retry level register R6 to the target location pointed to by retry level register R4. The source length is extracted from the retry level register R7. The target length is computed from the 4-byte extraction of retry level register R2 plus the decimal value 24n. The padding string, if needed, defaults to binary zero.

COPY R4! X2(R2!+22) R6! R7+2
This command copies data from the location pointed to by retry level register R6 to the target location pointed to by retry level register R4. The source length is extracted from rightmost 2 bytes of the retry level register R7. The target length is extracted from the 2-byte location specified by the address expression R2!+22.

COPY RW4 8 XADR(R6!+X4(R7))
This command copies the numerical value of the address expression R6!+X4(R7) (the sum of the base register R6 address plus the entire contents of index register R7) to 8-byte wide retry level general register RW4. The computed storage location is not referenced; only the address of the storage location is copied into register RW4. Note, this can be done even on OS/390 systems. RH4 will be set to zeros, and R4 will contain the resolved address.

TIP: This technique can be used for calculating any arithmetic expression and assigning the result to a register or a storage location.

Help COmmands CUrsor

The **CURSOR** command moves the cursor to the home position of the window that currently contains the cursor. (A window's "home" position is the first input location on that window's command line.)

The CURSOR command is valid **only** when it is issued via a PF key. It cannot be typed directly on a command line.

Syntax:

CURSOR

This command accepts no operands.

Help COmmands DEFine

Background

The **DEFINE** command creates overrides to the **MAP** and **DMAP** commands' normal methods of finding mapping data.

Typically, the overrides to mapping are only relevant to and done for dump/XDC.

The **MAP** and **DMAP** commands are used to build maps of various sorts:
- **Assembler** programs and control block maps,
- **C** program maps,
- **Binder maps** for load modules and program object,
- System and user **control blocks** and **data area** maps.

The maps are built from mapping data consisting of **ESD** data, **SYM** data, **ADATA, DWARF** data and **C source** files.

When debugging programs running within a live system, the needed mapping data can be found in various files and libraries also located on that same system.
The **MAP** and **DMAP** commands already have numerous mechanisms for finding that mapping data from those local libraries. **HELP COMMANDS DMAP LOAD** gives a pretty good discussion of those mechanisms.

About Mapping Data for Dumps

When debugging a dump, consideration has to be given to where the dump comes from.

- When a dump/XDC session is being conducted on the same system from which the dump came, the mapping data can generally be found in the same libraries and files that would be used when debugging your program live, on that same system.

- On the other hand, if the dump originated from a foreign system (such as, for example, from a downstream customer of yours), then the local libraries and files would not contain mapping data matched to the programs, control blocks and data areas found within the dump. This is where the **DEFINE** command can be helpful.

About the DEFINE Command

The **DEFINE** command comes into play when setting up access to mapping data that is matched to a dump that originated on a foreign system.

Normally, the mapping commands (MAP and DMAP) will look for mapping data on Z/XDC's host system. However, when dump/XDC is being used, and the dump being analyzed came from a third party system (such as from a downstream customer), at least some of the mapping data that is matched to the dump must also come from that same system. (See **HELP DUMPS MAPPINGDATA** for further discussion.)

Once you have gathered the necessary libraries and extraction files from the foreign system, you can use the **DEFINE** command to connect them to the dump.

dump/XDC **journals** DEFINE commands. This so that when you end your dump/XDC session and then return to it later, your previously issued DEFINE commands will be automagically reissued. Thus, once issued, you will not have to manually reissue them again. For more information, see **HELP DUMPS MAPPINGDATA JOURNALING.**

The **DEFINE** command is used to create the following tokens:
- DDname translation definitions
- Dsect library definitions
- Module Mapping Extraction File (MMEF) definitions

Syntax:

```
DEFINE DDNTRANS  definition
       DSECTLIB  libraryname
       MMEFDSN   dsname
```

Operands:

The **DEFINE** command's first operand identifies the type of token being created. More exhaustive discussions are in the subtopics.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

DDNTRANS - About defining ddname translation tokens
DSECTLIB - About defining overriding dsect library definition tokens
MMEFDSN - About defining Module Mapping Extraction File tokens

Help COmmands DEFine DDntrans

Background

When the **MAP** command is used to build maps for load modules, the command first needs to find and build the Module's **Binder** map. Binder maps are built from a load module's (or program object's) **ESD** records. Binder maps are needed so that Z/XDC can know where, within a module, its csects are. For more information, see **MAPS BINDERMAPS.**

When the module being mapped is located in a dump, and that dump comes from a foreign system (such as one owned by a downstream customer), the customer may have provided, along with the dump, an **MMEF** file containing extractions of ESD data from the customer's own load libraries and system libraries. MMEF files allow a customer to send to you load module mapping data, matched to a dump **without** having to send the entire load modules themselves. For more information, see **HELP DUMPS MAPPINGDATA MMEFFILES.**

MMEF files are sequential files that contain compressed mapping data (ESD data mostly) for all load modules (and program objects) contained within any number of load libraries. MMEF files are created when a customer runs a publicly available utility named **XDCMMEF.** The libraries to be extracted from are identified to the utility via ddnames of the form **LIB1, LIB2,** etc. For further details, see **HELP UTILITIES XDCMMEF.**

For a more thorough discussion of obtaining mapping data matched to a dump from a foreign system, see **HELP DUMPS MAPPINGDATA.**

What are DDNTRANS Tokens?

DDNTRANS tokens come into play when dump/XDC is trying to find a mapping data source for load modules located within a dumped **private area.** The lookup process is approximately as follows:

- (1) dump/XDC presumes private area load modules were loaded from task libraries such as **JOBLIB, STEPLIB, ISPLLIB** etc.
- (2) The **XDCMMEF** utility, when executed on a foreign system, extracts from that system mapping data for concatenations of load libraries identified to the utility via **LIBn** ddnames.
- (3) Typically, a **LIBn** concatenation will contain the same libraries matching the concatenations of the task libraries and linklist libraries actually used by the dumped address space (for task libraries) and by the system within which the address space was running (for the linklist libraries).
- (4) **DDNTRANS** tokens (with the help of **MMEFDSN** tokens) inform dump/XDC which **LIBn** ddname was used by the **XDCMMEF** utility for the load libraries corresponding to a given **task library** ddname from the dump.

  In other words, the DDNTRANS tokens translate task library ddnames from the dump to LIBn ddnames from the MMEF file creation run.
- (5) The LIBn libraries (in the MMEF file) will then be the libraries whose

extracted mapping data is searched for a given task library identified in the dump.

- (6) Note, the linklist libraries (extracted by XDCMMEF) are scanned only after a scan of all relevant task libraries fails to find a match.

DDNTRANS tokens connect task library ddnames (found in a dump) to **LIBn** ddnames allocated to the **XDCMMEF** run that extracted the needed mapping data from the foreign system. This **DEFINE DDNTRANS** command creates the DDNTRANS tokens that make those connections.

When are DDNTRANS Tokens NOT Needed?

However in truth, DDNTRANS tokens are **only a fallback mechanism.** Most dumps will already contain (in JFCBs found via TIOT entries) the library names that dump/XDC needs in order to find mapping data within MMEF files. That library will then be searched without regard to under which LIBn it was concatenated when the MMEF file was built.

dump/XDC will need DDNTRANS tokens only when the necessary JFCBs are (for one reason or another) absent from the dump.

In other words, unless you are in a situation where you specifically need DDNTRANS tokens, there is no need for you to read further in this topic. This is further discussed in **HELP DUMPS MAPPINGDATA DEFINE**

The DEFINE DDNTRANS Command

The **DEFINE DDNTRANS** commands create a list of **DDNTRANS** tokens that link task library ddnames (found in a dump) to **LIBn** ddnames used in a **XDCMMEF** mapping data extraction run. Each LIBn ddname is linked (within the MMEF file) to a list of library names for the load libraries that were concatenated under that LIBn ddname.

In other words, a DDNTRANS token is a **connection** from a **task library ddname** found in a dump to a list of **candidate library extractions** found in an MMEF file.

Sometimes a dump will contain multiple address spaces. And of course, different aspaces may have different STEPLIBs (etc.) defined to them. So DDNTRANS tokens can be (but don’t have to be) qualified by address space names (or ASIDs).

You can use the **LIST DDNTRANS** command to display the current list of ddname translation tokens.

Syntax:

DEFINE DDNTRANS definition [POS={FRONT|BACK}]

Notes:

Only one definition may be provided per **DEFINE DDNTRANS** command. But you may use as

many DEFINE commands as necessary to define all the translation tokens that you require.

Parts of the **definition** operand accept values that are **patterns** that can have multiple matches. A discussion of pattern match rules can be found at **HELP COMMANDS SYNTAX PATTERNMATCH.**

Operands:

Definition
This operand describes a ddname translation token. Its general syntax is:

sdef=tdef

sdef
The "source definition" describes the aspace and task library ddname that the token is to translate from.

tdef
The "target definition" describes (by **LIBn** ddname) the concatenation of libraries within the MMEF file that is to be searched when mapping a load module presumed to have been loaded from a candidate task library. (If the map data is **not** found, then the presumption is proven incorrect, so the search moves on to an ancestor's task library or to the linklist library.)

Important! In order for **tdef** operands to be meaningful, **MMEFDSN** tokens need to be created that assign **labels** (of the user's choosing) to specific MMEF datasets. Those labels can then be used in **tdef** operands of DEFINE DDNTRANS commands to link DDNTRANS tokens to MMEF file dsnames.

sdef Detailed Syntax:

[aspace.]ddname

aspace.
This is optional. If present, it limits the token to applying to only one (or more, if a pattern) address space. It may be any of the following:
- An aspace's name (jobname, STC name, or TSO userid),
- A pattern that matches one or more aspace names.
- An ASID:
  - This is always a hex value that resolves to X'FFFF' or less.
  - Prepend a leading 0 when needed to distinguish aspace names (A349 and FFFF) from hex ASIDs (0A349 and 0FFFF).

If omitted, then the token applies to any aspace in the dump.

ddname
This is required. It specifies one or more task library ddnames to which the token applies. It may be either of the following:
- The ddname of a **task library** concatenation. Typically, such ddnames are

STEPLIB, JOBLIB, ISPLLIB, etc.
- A **pattern** (such as *) that matches one or more task library ddnames.

tdef Detailed Syntax:

[label.]ddname

label.
This is optional. If present, it selects by identifier one **MMEFDSN** token for the purpose of linking the here-given ddname to the there-given dsname (amongst the MMEFDSN tokens) of a particular MMEF file. That will then be the file whose LIBn entries are searched for the needed mapping data.

MMEFDSN tokens are created by the **DEFINE MMEFDSN** command. The tokens identify (to dump/XDC) the dsnames of MMEF files.
- The MMEFDSN tokens may (but don't have to) assign labels to those dsnames.

When **label.** is omitted from tdef, **all** defined MMEFDSN tokens are searched.

When **label.** is given, it may be any of the following:

- A **specific label** assigned (by DEFINE MMEFDSN) to a MMEFDSN tokens. Only that token will be examined.

- The value **-** or **none:** This means that only MMEFDSN tokens that were **not** assigned a label will be examined.

- The value * or just **omitted** altogether: This means that **all** MMEFDSN tokens will be examined.

ddname
This is a **LIBn** ddname originally defined in the JCL used to run the XDCMMEF utility when it extracted the mapping data from the foreign system. Within the MMEF file that XDCMMEF created, the LIBn value heads a list of dsnames of the libraries that were concatenated under the LIBn ddname.

POS={FRONT|BACK}
This operand describes where the DDNTRANS token is to be placed in the DDNTRANS queuing order:
- First (POS=FRONT)
- Or last (POS=BACK)
- The default is last.

Patterns

As noted above, on the **sdef** side of a DDNTRANS token definition, both the aspace name and the task library ddname can be given as **patterns.** A full discussion of valid patterns and the match rules behind them can be found at **HELP COMMANDS SYNTAX PATTERNMATCH.**

Examples:

DEFINE DDNTRANS MYJOB.ISPLLIB=MYLMI1.LIB3

The **IFs:**
- **If** a dump contains an aspace named **MYJOB,**
- And **if** that aspace had a task library allocated to it via the ddname **ISPLLIB,**
- And **if** the DEFINE MMEFDSN command has been used to create an MMEFDSN token labeled **MYLMI1**

Then: When a **MAP** command is issued, and the command needs to build a **Binder map,** Z/XDC will do the following:

- First, it will locate the the MMEF export file whose dsname was given in the MMEFDSN token identified by MYLMI1. (See **HELP COMMANDS DEFINE MMEFDSN.)**

- Then it will search that MMEF definition to ensure the the LIB3 definition was in the original JCL for the XDCMMEF program. If the LIB3 definition is not found, an error will be raised.

- Then for each library that had been concatenated to LIB3, Z/XDC will scan the list of load module names from that library.

- If a match occurred against the name of the load module (in the dump) to be mapped, then mapping data (ESD entries in this case) will be extracted from the MMEF file, and a Binder map will be built.

DEFINE DDNTRANS *=LIB2

The **FORs:**
- **For** any task library ddname, [Because the sdef is "*".]
- **Regardless** of the dumped aspace, [Because "aspace." is omitted from sdef.]

When a MAP command is issued, and the command needs to build a Binder map, Z/XDC will do the following:

- In this case, **all** MMEFDSN tokens will be eligible. [Because "label." is omitted from tdef.]

- So Z/XDC will run the **entire** MMEFDSN list,

- And process each listed **MMEF file,** one by one.

- It will see if the file includes a **LIB2** allocation. the first MMEF definition found that has a a **LIB2** definition in it will be used. Other MMEF definitions are not considered.

- If found, then for each library that had been concatenated to LIB2, Z/XDC will scan the list of load module names from that library.

- If a match occurs against the name of the load module (in the dump) to be mapped, then mapping data will be extracted from the MMEF file, and a Binder map will be built.

DEFINE DDNTRANS *=NONE.LIB2 [or]
DEFINE DDNTRANS *=-.LIB2

The **FORs:**
- **For** any task library ddname, [Because the sdef is "*".]
- **Regardless** of the dumped aspace, [Because "aspace." is omitted from sdef.]

When a MAP command is issued, and the command needs to build a Binder map, Z/XDC will do the following:

- In this case, Z/XDC will examine only those MMEFDSN tokens that do **not** have labels. [Because "label." is "NONE." or "-." in the tdef.]

- And process each eligible **MMEF file,** one by one.

- It will see if the file includes a **LIB2** allocation. the first MMEF definition found that has a a **LIB2** definition in it will be used. Other MMEF definitions are not considered.

- If found, then for each library that had been concatenated to LIB2, Z/XDC will scan the list of load module names from that library.

- If a match occurs against the name of the load module (in the dump) to be mapped, then mapping data will be extracted from the MMEF file, and a Binder map will be built.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

MOREABOUT - Technical details about how DDNTRANS tokens affect the MMEF scan process.

Help COmmands DEFine DDntrans Moreabout

Background

Basically, **DDNTRANS** tokens provide a method for finding **MMEF** provided mapping data for private area load modules when a task library **ddname** can be discovered, but a load library **dsname** cannot. If a dump contains the necessary control blocks (specifically, JFCBs) for the task library dsnames to be discoverable, then **DDNTRANS** tokens are not needed.

This topic is intended to provide insight as to when **DDNTRANS** tokens are and are not needed.

When building Binder maps for load modules located in an address space's private area, the load library from which the module was loaded needs to be identified (because that's where the Binder map's mapping data [ESD records] is). Such load libraries fall into three possible categories:

- When a load module is brought into storage via a user program issued **LOAD** or **LOADX** macro, the macro might designate an arbitrary library from which the module is to be loaded.

- Otherwise, it might be loaded from a task library such as JOBLIB, STEPLIB, ISPLLIB or whatever other library has been indicated via a **TASKLIB=** operand on the **ATTACH** macro that created the current (or ancestor) task.

- Otherwise, it will be loaded in from a system linklist library.

About Task Libraries

In the following descriptions, mention is made of something called **task libraries.** To clarify:

- A task library is a TCB connected load library which z/OS will automatically search when it needs to bring load modules into storage.

- Any task may have a task library associated with it.

- A task library (if any) is assigned to a task when the task is created. If the creating **ATTACH** macro has a **TASKLIB=** operand, then a task library is assigned; otherwise, one is not.

- When a load module needs to be loaded into storage, z/OS will search probably multiple task libraries to find that module. Both the **current** task's task library (if any) and the task libraries of all **ancestor** tasks are searched.

- If a **JOBLIB** or **STEPLIB** exists, then it will be the task library for the **jobstep** task. (Note, if both exist, then only STEPLIB will be assigned.)

- The TCB (Task Control Block) contains a field named **TCBJLB,** which, if not 0, points to an open DCB (data Control Block), whose DCBTIOT field contains the offset into the task's TIOT (Task Input-Output Table) of the task library's ddname.

- If the definition is for concatenated libraries, extra TIOT entries follow the DCBTIOT indicated entry, and these extra entries have ddname fields that are blank.

- If the TCBJLB Field is 0, then that task has no specific task library.

Searching for a Load Module's Mapping Data

When a specific dataset name is **not** specified on the Z/XDC **MAP** command, dump/XDC

uses the same search logic that z/OS uses to determine the library where the load module definition resides.

Once the appropriate dataset name has been determined, mapping can proceed.

dump/XDC identifies all task library ddnames in a dumped address space. It does this by accessing the following control blocks:

- The TCB (Task Control Block) so that it can retrieve the value of TCBJLB (0 or the pointer to an open DCB).
- The DCB pointed to by TCBJLB (If TCBJLB is not 0) (so that it can get the TIOT offset from that DCB's DCBTIOT field).
- The task's TIOT, using the offset gotten in the previous step (DCBTIOT), which will have the DDname that the DCB was opened with.

At this stage, dump/XDC knows the ddname that the task library's DCB uses.

What happens next depends on whether or not there is a **DDNTRANS** token that matches this address space and ddname.

Finding Mapping Data in MMEF Files when DDNTRANS Tokens DO NOT Exist

If there is no matching **DDNTRANS** token, dump/XDC will use the TIOT, starting at the entry located above (which will have the task library ddname), and using any TIOT entries that follow and have a blank ddname (which indicates a concatenated dataset), and use the TIOEJFCB field to locate the JFCB (which has the dataset name in it).

To do this, dump/XDC must be able to retrieve all of the relevant sections of the TIOT (the first entry for the DCB, and all following concatenated dataset entries, and, if the ddname is not the last ddname in the TIOT, the next entry (so that it can check the ddname field in the TIOT entry for blanks) as well as the relevant control blocks that allow the value in TIOEJFCB to be translated to a JFCB address. (The TIOT is in the 'LSQA' section of the dump). (the control blocks associated with translation and the JFCB are in the 'SWA' section of the dump).

The dataset name is looked up in the defined MMEFDSN tokens. It must have been one of the datasets defined in the JCL used to run the XDCMMEF program specifically, in one of the **LIBn** groups. The dataset name must be found, otherwise an error is raised.

The load module name is looked up in each library's directory. If not found, the next dataset in the concatenation is searched. If there are no more concatenated datasets, the task library search continues with the next older ancestor task.

Finding Mapping Data in MMEF Files When DDNTRANS Tokens DO Exist

If there is a matching **DDNTRANS** token, that token provides the **MMEF** file's **LIBn** group that is linked to the current task library ddname.

The load module name is looked up to determine if it is in the indicated **LIBn** group, and if so, that **LIBn** group will be used. If not found, the task library search continues with the next older ancestor task.

Recommendation

Basically, **DDNTRANS** tokens provide a method for finding **MMEF** provided mapping data when a task library **ddname** can be discovered, but a load library **dsname** cannot. If a dump contains the necessary control blocks (specifically, JFCBs) such that the task library dsnames **can** be discovered, then **DDNTRANS** tokens are **not** needed.

Help COmmands DEFine DSectlib

The **DEFINE DSECTLIB** command is used to create a list of **DSECTLIB** tokens that name libraries that contain DSECT mapping data in the form either of ADATA side files or of load modules (or program objects) containing either ADATA or SYM data.

When DSECTLIB tokens exist, the mapping data search process is significantly altered for the **DMAP** command (but not for the MAP command).

But before getting into the details, let me first provide some background.

Background - The Types of Mapping Data Used to Build Dsect Maps

The **DMAP** command builds dsect maps from three kinds of mapping data:

- **ADATA** found both in load modules and in ADATA side files,

- **SYM** data found only in load modules,

- **ESD** data found only in load modules.

Notes:

- In most cases, my use of the term "load modules" is intended to refer both to load modules and to program objects.

- ESD data is used in only one special case. It is used for building floating versions of Binder maps. The present topic, however, is about creating DSECTLIB tokens. Such tokens affect searches only for SYM data and ADATA, not ESD data. Accordingly, this topic does not further discuss ESD data.

- See **HELP COMMANDS DMAP LOAD** for a detailed syntax and functional description of the **DMAP** command.

Background - The DMAP Command’s Normal Map Data Search Process

When a **DMAP** command is asked to build a dsect map, you have to provide it with (or it will have to otherwise figure out) at least two pieces of information:

- The name of the DSECT whose map is to be built
- The name of the library member within which the command is to search for the necessary mapping data:
  - In the case of ADATA, this member name can be:
    - Either a member of an ADATA side file library,
    - Or the name of a load module from a load library.

- In the case of SYM data, this member name can only be the name of a load module from a load library.

Absent the presence of DSECTLIB tokens, when searching for the member to be examined, the **DMAP** command will search as follows:

- First, it will look for the target member name in each of the ADATA side file libraries listed in the currently active **MAPLIBS** list.

- Failing that, it will then search the target aspace's task libraries, PLPA libraries and linklist libraries for the first one it finds that contains the target member name (load module name).

If an ADATA side file member is found in a MAPLIBS-listed library, that member is searched for ADATA (but not for SYM data).

If a load module is found in a load library, then that library member is searched, first for ADATA, then for SYM data.

The Changes Wrought by DSECTLIB Tokens

When DSECTLIB tokens exist, the MAPLIBS searches still occur, but the load library searches are replaced. The load libraries and ADATA side file libraries, listed by the DSECTLIB tokens, are searched instead.

If you want to see what DSECTLIB tokens are already defined, you can use the **LIST DSECTLIBS** command to do that.

The Changes NOT Wrought by DSECTLIB Tokens

When DSECTLIB tokens exist, the MAPLIBS searches still occur.

When the **SYMDATA=, ADATA=** and **ESDDATA=** operands are used on the DMAP command, the existence of DSECTLIB tokens has no effect upon accesses to the libraries named by the operands. They will still function as documented in **HELP COMMANDS DMAP LOAD.**

When the DMAP command needs to find ESD data (for loading Binder maps as dsects), DSECTLIB tokens are ignored, and task libraries etc. are searched instead.

Syntax:

DEFINE DSECTLIB dsname [POS={FRONT|BACK}]

Note:

Only a single dsect library name may be provided per **DEFINE DSECTLIB** command. However, you may use as many commands as necessary to define all the libraries that you require.

Operands:

dsname
The name of the library that is to be defined. It may be either a load library or an ADATA side file library. It must:
- Be catalogued
- Be DSORG=PO (a PDS or PDSE)
- Be RECFM=U (for a load library) or RECFM=VB (for an ADATA side file library)

If **RECFM=U,** the dataset must be a load library (a PDS or PDSE) containing load modules or program objects. The modules/objects should contain embedded **SYM data** or **ADATA.**

If **RECFM=VB,** the dataset must be an ADATA side file library (a PDS or PDSE) with each member containing **ADATA** records.

POS={FRONT|BACK}
This operand describes where the DSECTLIB definition is to be placed in the DSECTLIB queuing order:
- First (POS=FRONT),
- Or last (POS=BACK),
- The default is last.

Examples:

DEFINE DSECTLIB DDEFS.ZOS.V1R13.SYM

This definition will cause the **DMAP** command to search the nominated library instead of using its normal load libraries search when looking for dsect mapping data. (DMAP’s MAPLIBS search remains unaffected and continues to occur prior to the DSECTLIBs search.)

The definition is positioned last on the DSECTLIB tokens queue.

DEFINE DSECTLIB DDEFS.ZOS.V1R13.ADATA,POS=FRONT

This definition adds a second library to the overriding mapping data search process for DMAP.

The definition is positioned first on the DSECTLIB tokens queue.

Help COmmands DEFine Mmefdsn

Background

- When the **MAP** command is used to load maps for load modules, the command first needs to find and load the Module's **Binder** map. Binder maps are built from a load module's **ESD** records. Binder maps are needed so that Z/XDC can know where, within a module, its csects are. For more information, see **MAPS BINDERMAPS.**

- When the module being mapped is located in a dump, and that dump comes from a foreign system (such as one owned by a downstream customer), the customer may have provided, along with the dump, an **MMEF** file containing extractions of ESD data from the customer's own load libraries and system libraries. MMEF files allow a customer to send to you load module mapping data, matched to a dump (without having to send the entire load modules themselves). For more information, see **HELP DUMPS MAPPINGDATA MMEFFILES.**

- MMEF files are sequential files that contain compressed mapping data (ESD data mostly) for all load modules (and program objects) contained within any number of load libraries. MMEF files are created when a customer runs a publicly available utility named **XDCMMEF.** The libraries to be extracted from are identified to the utility via ddnames of the form **LIB1, LIB2,** etc. For further details, see **HELP UTILITIES XDCMMEF.**

- For a more thorough discussion of obtaining mapping data matched to a dump from a foreign system, see **HELP DUMPS MAPPINGDATA.**

What are MMEFDSN Tokens?

MMEFDSN tokens connect MMEF datasets to a dump/XDC session. Each MMEFDSN token identifies an MMEF file's **dsname** and (optionally) assigns a user chosen **label** to the dsname.

- Then the **MAP** command can search either all or label-selected MMEF files for mapping data matched to the dump being examined.

- You can use the **LIST MMEFDSN** command to display the current list of MMEFDSN tokens.

Syntax:

```
DEFINE MMEFDSN dsname
               [ID=label]
               [POS={FRONT|BACK}]
               [DIFFSYSOK={MAYBE|NO|YES}]
               [DLPA={IPL|YES|NO}]
```

Note:

Only a single MMEF dataset name may be provided per **DEFINE MMEFDSN** command. However, You may use as many commands as necessary to define all the datasets that you require.

Operands:

dsname
This must be the name of a **Module Mapping Extraction File** (MMEF file) as created by the **XDCMMEF** utility. The file must have been created on the same system from which the dump was produced, and it should contain compressed extractions of **ESD** data (and possibly other mapping data) for load modules and program objects contained within the dump.

ID=label
This (optional) operand allows you to assign a short label by which the MMEF dsname can be represented in **DDNTRANS** tokens. The label can range up to 8 characters in length.

POS={FRONT|BACK}
This operand describes where the MMEFDSN token is to be placed in the MMEFDSN queuing order:
- First (POS=FRONT)
- Or last (POS=BACK)
- The default is last.

DIFFSYSOK={MAYBE|NO|YES}
This operand describes whether the MMEFDSN will be accepted if a system mismatch is detected:
- MAYBE (DIFFSYSOK=MAYBE, the default).
  Message DBC793 will be displayed, and based on the reply to that message, the MMEF dataset may be used.
- NO (DIFFSYSOK=NO)
  The MMEF dataset will not be used. (the command fails)
- YES (DIFFSYSOK=YES)
  The MMEF dataset will be used.

If this command is journaled by dump/XDC, and this operand is omitted (causing the default of DIFFSYSOK=MAYBE to be used) or DIFFSYSOK=MAYBE is specified, then the playback suppresses asking the question, and DIFFSYSOK=NO is assumed. If this is not wanted, use DIFFSYSOK=YES on the command.

DLPA={IPL|YES|NO}
This operand controls whether DLPA information on the MMEF dataset will be used (if the MMEF dataset contains DLPA information, otherwise this operand is parsed but ignored).
- IPL (DLPA=IPL)
- If the MMEF dataset has a different IPL date/time, any DLPA information on the MMEF dataset is ignored. This is the default behaviour if this operand is omitted and the MMEF dataset system is the same as the current system.
- YES (DLPA=YES)

```
  -       any DLPA information on the MMEF dataset
          will be used.
          This is the default behaviour if
          this operand is omitted and
          the MMEF dataset system is not the same as
          the current system
          (and the mismatch was accepted).
  - NO    (DLPA=NO)
  -       any DLPA information on the MMEF dataset
          will not be used.
```

Examples:

DEFINE MMEFDSN DUMPS.CUST21.D052.MMEF

This definition creates an MMEFDSN token that adds the given dataset to the list of MMEF datasets known to dump/XDC. No label is assigned. The token is positioned last in the queuing order.

DEFINE MMEFDSN DUMPS.CUST10.D003.MMEF ID=STEPLIB1 POS=FRONT

This definition creates an MMEFDSN token that adds the given dataset to the list of MMEF datasets known to dump/XDC. The token is assigned **STEPLIB1** as its label. The token is positioned first in the queuing order.

Help COmmands DELete

The **DELETE** command can be used to remove the definition of many Z/XDC related objects from Z/XDC’s information structures. It can also be used to delete datasets and libraries (but not individual library members).

Syntax:

```
   DELETE modulenames              SILENT=YES   USS=option   DUB=option
          addressexpressions              NO
          CACHE       cacherefs
          DATASET     dsname
          DDNTRANS    tokenrefs
          DSECTLIBS   tokenrefs
          DSN         dsname
          EQUATES     equaterefs
          HOOKS       hookrefs
          ILITS       ipcslitrefs
          MAPLIBS     maplibrefs
          MAPS        maprefs
          MMEFDSNS    tokenrefs
```

MODULES	**modulerefs**
PGMS	**modulerefs**
PROGRAMS	**modulerefs**
PROXYTASKS	**tcbaddresses**

Note:

Generally, the DELETE command's first operand is the **deletion type** operand. It indicates the type of object to be deleted. Below, each typing operand is briefly mentioned. The more exhaustive discussions will be in the type-specific subtopics.

Deletion Types

CACHE
This indicates that ADATA cache storage is to be purged.

DDNTRANS
This indicates that ddname translation tokens are to be purged.

DATASET
DSN
This indicates that a dataset is to be deleted.

DSECTLIBS
This indicates that DSECTLIB tokens are to be purged.

EQUATES
This indicates that equate names are to be deleted.

HOOKS
This indicates that hooks are to be removed.

ILITS
This indicates that IPCS literals are to be deleted.

MAPLIBS
This indicates that MAPLIBS lists or list entries are to be purged.

MAPS
This indicates that csect maps, dsect maps, and/or Binder maps are to be deleted.

MMEFDSNS
This indicates that MMEFDSN tokens are to be purged.

MODULES **modulenames** or **addressexpressions**
PROGRAMS **modulenames** or **addressexpressions**
PGMS **modulenames** or **addressexpressions**
omitted **modulenames** or **addressexpressions**
These are all aliases of each other. They indicate that one or more load modules are to be deleted from storage.

During command parse, if the DELETE command's first operand is not a recognized type code, then Z/XDC assumes it's an address expression or random module name. A syntax error is not recognized unless either the address expression fails to parse correctly, or the module name to be deleted is not found.

If the module to be deleted has the same name as a type code (or an abbreviation thereof), then either append a **+0** to the module name or use the **PGMS** (or similar) type code.

PROXYTASKS
This indicates that Formal Proxy Tasks are to be deleted.

SILENT=YES
SILENT=NO
Most of the various DELETE commands normally display progress messages and/or completion summary messages. If you wish to suppress those messages, you can add **SILENT=YES** (or just **S=Y)** to the DELETE command. (SILENT=NO is the default action.)

Command aborting error messages will not be suppressed - only progress and completion summary messages.

The SILENT=value can be placed in **any** operand position. It can be first or last or anywhere in between. Its position has no effect on any other parsing. Examples:

```
DEL S=Y EQU RB* TCB*
DEL EQU RB* SILENT=YES TCB*
DEL MAPS SIL=Y
DEL PGMS myprogrm SIL=Y
DEL S=Y myprogrm
```

USS=option

This operand is optional, and can be specified anywhere on the command. It is always processed before any other operands.

Only some DELETE command options make use of the **USS=option** operand. These are:
- DELETE MAPS
- DELETE MODULES (etc.)

If a **DELETE** command option does not make use of the **USS=option** operand, it is still processed but any option is ignored (Note that it is still validated, and must be correct).

If this operand is omitted, the **DELETE** command behaves as if **USS=PROFILE** was specified.

More information about this operand is in the **HELP DEBUGGING USS OPERAND USS=** help topic.

DUB=option
This operand is optional, and can be specified anywhere on the command. It is always processed before any other operands.

Only some DELETE command options make use of the **DUB=option** operand. These are:
- DELETE MODULES (etc.)

If a **DELETE** command option does not make use of the **DUB=option** operand, it is still processed but any option is ignored (Note that it is still validated, and must be correct).

If this operand is omitted, the **DELETE** command behaves as if **DUB=PROFILE** was specified.

More information about this operand is in the **HELP DEBUGGING USS OPERAND DUB=** help topic.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

CACHE - How to purge cached ADATA.

DATASET - How to delete datasets.

DDNTRANS - How to delete DDNTRANS tokens.

DSECTLIBS - How to delete DSECTLIB tokens.

EQUATES - How to delete equates.

HOOKS - How to delete hooks.

ILITS - How to delete IPCS literals.

MAPLIBS - How to purge MAPLIB lists and MAPLIB list entries.

MAPS - How to delete symbol maps.

MMEFDSNS - How to delete MMEFDSN tokens.

MODULES - How to delete load modules.

PROXYTASKS - How to delete Formal Proxy Tasks.

To delete breakpoints, use the **OFF** command.

Help COmmands DELete Cache

Because of its sheer massiveness, when Z/XDC reads ADATA, it caches an extraction of the data into storage just in case it might be needed again. A cached ADATA file can run to several megabytes of 31-bit storage. So its possible that the presence of the cached data could cause a storage shortage.

When Z/XDC detects a storage shortage during its internal GETMAIN requests, it will automatically try to relieve the shortage by purging cache on an LRU basis. However, Z/XDC has no way to detect storage shortages that the user program might experience. Accordingly, if you anticipate that a shortage might occur, you can use the **DELETE CACHE** command to purged cached ADATA that you know you will no longer be need.

Note, cached ADATA side files are "owned" by MAPLIB list entries. Whenever a MAPLIB list entry is purged, all ADATA cache files owned by that entry also are purged. See **HELP COMMANDS DELETE MAPLIBS** for more information.

You can use the LIST CACHE command to display how much storage is being taken by which cache files.

Syntax:

```
DELETE CACHE dsname          ...
             dsname(member)  ...
             #nnn            ...
             ALL
```

Notes:

- Any number of and any combination of operands may be given in any order (except that it doesn’t make much sense to give ALL in combination with other operands).

- As with any other DELETE command, a **SILENT=YES** operand can be added to the command (in any position) to suppress the display of progress messages and completion messages. See **HELP COMMANDS DELETE** for details.

dsname
dsname(member)
These operands identify by name which cache file to purge.

If the cache file is associated with a sequential dataset, then the **dsname** form has to be used. Similarly, if the cache file is associated with a member of a partitioned dataset, then the **dsname(member)** form has to be used.

The **dsname** form of this operand **cannot** be used to purge all ADATA side files associated with various members of a partitioned dataset. That function can be accomplished, however, by purging the MAPLIB list entry for the PDS(E) from which

the cached ADATA was read. See **HELP COMMANDS DELETE MAPLIBS** for more information.

#nnn
"nnn" must be a decimal number. This operand permits you to purge **cache files** from storage by referencing its sequence number as displayed by the LIST CACHE command.

#nnn may be used only if the set and ordering of cached ADATA has not changed since the last time a LIST CACHE command was used to display the cache files.

If **multiple** #nnn operands are given within one DELETE CACHE command, then the association of "#nnn" values to cache files remains as displayed by the last preceding LIST CACHE command throughout the processing of the DELETE CACHE command.

After a DELETE CACHE command has completed, the association of **#nnn** values to cache files is discarded and not reestablished until the next issuance of a **LIST CACHE** command. In fact, any command or event that changes the cached ADATA side files causes this association to be lost too. Examples:

- **MAP** commands,
- **DMAP** commands,
- **DELETE MAPLIBS** commands,
- Certain **SET MAPLIBS** commands,
- And a detected storage shortage.

ALL
This causes all cached ADATA side files to be purged from storage.

Examples:

Note, in the following examples, **hlq** represents the High Level Qualifier assigned to Z/XDC's Product Libraries when Z/XDC was installed. Ask Systems Programming what the hlq is in your shop. [In ours, It's CSW.]

Assume that a **LIST CACHE NOSECTIONS** command produces the following display:

```
THE FOLLOWING MAPPING INFORMATION HAS BEEN CACHED:
#1 (2,199K)   hlq.XDCV31.XDCADATA(DBCDISPL)
              MAPLIB: (#1) hlq.XDCV31.XDCADATA
#2 (2,499K)   hlq.XDCV31.XDCADATA(DBCINIT)
              MAPLIB: (#1) hlq.XDCV31.XDCADATA
#3 (34,770K) hlq.XDCV31.XDCADATA(DBCMAPS)
              MAPLIB: (#1) hlq.XDCV31.XDCADATA
```

Then the following command sequences would have the following effects:

DEL CACHE #1 #3
This would remove cache files number 1 and number 3, leaving only number 2. The operands "#1" and "#3" would not be accepted unless a preceding LIST CACHE command had been issued. After this command completes, the next issued LIST CACHE command would show what had been the 2nd cache file to now be cache file #1.

DEL CACHE hlq.XDCV31.XDCADATA(DBCMAPS) #2
This would remove cache files number 3 and number 2.

DEL CACHE hlq.XDCV31.XDCADATA
This command would fail. There is no cache file having this name.

DEL MAPLIBS hlq.XDCV31.XDCADATA
This command would do both of the following:
- The MAPLIBS list entry named hlq.XDCV31.XDCADATA would be removed from the currently active MAPLIBS list.
- All three ADATA cache files (#1 #2 and #3) would be purged since all three of those files are owned by the purged MAPLIBS list entry.

For more information, see **HELP COMMANDS DELETE MAPLIBS.**

DEL CACHE ALL
This command would purge all of the cached ADATA side files.

Help COmmands DELete DAtaset

The **DELETE DATASET** command can be used to delete a fully-qualified cataloged dataset. The dataset name must contain a period. Note that a member of a partitioned dataset (PDS or PDSE) cannot be deleted. The TSO command may be used to delete a member of a PDS or PDSE if the TSO environment is active.

Syntax:

```
DELETE DATASET dsname  ...
       DSN     dsname  ...
```

Notes:

- As with any other DELETE command, a **SILENT=YES** operand can be added to the command (in any position) to suppress the display of progress messages and completion messages. See HELP COMMANDS DELETE for details.

dsname
This is the name of a dataset to be deleted. It must be cataloged and not a member of a PDS or PDSE. It must contain at least one period.

Example:

To delete dataset ABC.SCRIPT.OUTPUT, issue:

DE DSN ABC.SCRIPT.OUTPUT

Providing the dataset exists, can be allocated with DISP=OLD, and the user has the authority to delete it, ABC.SCRIPT.OUTPUT will be deleted.

Help COmmands DELete DDntrans

The **DELETE DDNTRANS** command can be used to delete one, or more, or all DDNTRANS tokens.

Syntax:

```
DELETE DDNTRANS #nnn  ...
                ALL
```

Notes:

- Any number of and any combination of operands may be given in any order. (Except that it doesn't make much sense to give ALL in combination with other operands).

- As with any other DELETE command, a **SILENT=YES** operand can be added to the command (in any position) to suppress the display of' progress messages and completion messages. See **HELP COMMANDS DELETE** for details.

Operands

ALL
All existing DDNTRANS tokens are purged.

#nnn
The specified DDNTRANS token is purged. **nnn** must be a decimal number.

This operand permits you to purge a **DDNTRANS token** from storage by referencing its sequence number, as displayed by the **LIST DDNTRANS** command.

#nnn may be used only if the set and ordering of DDNTRANS entries has not changed since the last time a LIST DDNTRANS command was used to display the entries.

If **multiple** #nnn operands are given within one DELETE DDNTRANS command, then the association of **#nnn** values to DDNTRANS tokens remains as displayed by the last preceding LIST DDNTRANS command throughout the processing of the DELETE DDNTRANS command.

After a DELETE DDNTRANS command has completed, the association of **#nnn** values to DDNTRANS entries is **discarded** and is not reestablished until the next issuance of a **LIST DDNTRANS** command. In fact, any command or event that changes the definition of DDNTRANS entries causes this association to be lost too.

Examples:

DEL DDN ALL #4 #3
This given combination of operands works, but only sort of.
- **ALL** causes all existing DDNTRANS tokens to be deleted.
- **#4** fails because by the time processing reaches it, token #4 no longer exists. Also, the failure causes all further processing of pending commands to abort.
- **#3** is never reached because of the abort caused by the failure of the #4 operand.

Assume that a **LIST DDNTRANS** command produces the following display:

```
seq  value
#1   X.A=lib1
#2   X.B=lib2
#3   X.C=lib3
#4   X.D=lib4
```

Then the following command sequences will have the following effects:

DEL DDNTRANS #1 #3
This will remove DDNTRANS tokens number 1 and number 3.

The operands #1 and #3 would not have been accepted but for the preceding **LIST DDNTRANS** command.

After this DELETE DDNTRANS command completes, the next issued LIST DDNTRANS command will show what had been token #2 to now be token #1.

Also, after this DELETE DDNTRANS command is processed, **#nnn** operands can not be used on subsequent DELETE DDNTRANS commands unless an intervening LIST DDNTRANS command is issued.

DEL DDNTRANS #2 #4
This command fails if no **LIST DDNTRANS** command is issued since the prior DEL DDNTRANS command.

DEL DDNTRANS ALL
This command succeeds regardless. It purges all remaining DDNTRANS tokens (if any).

Help COmmands DELete DSectlibs

The **DELETE DSECTLIBS** command can be used to delete DSECTLIB definitions. All definitions can be deleted or a specific one. Multiple operations can be performed with one command.

NOTE: Only the DSECTLIB definition is purged from Z/XDC. The actual DSECTLIB dataset is NOT deleted!

Syntax:

```
DELETE DSECTLIBS dsname  ...
                 #nnn    ...
                 ALL
```

Notes

- Any number of and any combination of operands may be given in any order. (except that it doesn't make much sense to give ALL in combination with other operands).

- As with any other DELETE command, a **SILENT=YES** operand can be added to the command (in any position) to suppress the display of progress messages and completion messages. See **HELP COMMANDS DELETE** for details.

dsname
"dsname" must be a dataset name that matches the dataset name of one of the defined DSECTLIB entries. A dataset name is distinguished from "ALL", or a number ("#nnn"), by the fact that most dataset names consist of at least 2 levels, separated by a period ("."). If any periods are present in a value, it is assumed to be a dataset name. If none, then it is the value "ALL", or a number (starting with "#").

#nnn

"nnn" must be a decimal number. This operand permits you to purge **DSECTLIBS definitions** from storage by referencing its sequence number as displayed by the LIST DSECTLIBS command.

#nnn may be used only if the set and ordering of DSECTLIBS entries has not changed since the last time a LIST DSECTLIBS command was used to display the entries.

If **multiple** #nnn operands are given within one DELETE DSECTLIBS command, then the association of "#nnn" values to DSECTLIBS definitions remains as displayed by the last preceding LIST DSECTLIBS command throughout the processing of the DELETE DSECTLIBS command.

After a DELETE DSECTLIBS command has completed, the association of **#nnn** values to DSECTLIBS entries is discarded and is not reestablished until the next issuance of

a **LIST DSECTLIBS** command. In fact, any command or event that changes the definition of DSECTLIBS entries causes this association to be lost too.
Examples:

- **DEFINE DSECTLIB** commands

ALL
This causes all DSECTLIBS definitions to be purged from storage.

Examples:

Assume that a **LIST DSECTLIBS** command produces the following display:

```
seq dsn                                          fm
#1  userid.TEST.LOAD                             U
#2  userid.TEST.LOADPO                           U
#3  userid.Z22.SYMLINK                           U
#4  userid.Z22.SYMPLPA                           U
```

Then the following command sequences would have the following effects:

DEL DSECTLIBS #1 #3
This would remove DSECTLIBS definitions number 1 and number 3. The operands "#1" and "#3" would not be accepted unless a preceding LIST DSECTLIBS command had been issued. After this command completes, the next issued LIST DSECTLIB command would show what had been the 2nd definition to now be the 1st definition. Also, after this command has been processed, #nnn operands could not be used on subsequent DELETE DSECTLIBS commands unless an intervening LIST DSECTLIBS command was issued.

DEL DSECTLIBS userid.TEST.LOADPO
This would remove the definition for the DSECTLIB named ’userid.TEST.LOADPO’.

DEL DSECTLIBS ALL
This command would purge all of the DSECTLIBS definitions

Help COmmands DELete Equates

The **DELETE EQUATES** command is used to remove equate names from Z/XDC’s storage. Only user created equates can be deleted. **(Built-in** equates cannot be deleted.)

- See **HELP COMMANDS EQUATE** for how to create equates.
- See **HELP EQUATES BUILTIN** for a discussion of built-in equates.
- Use the **LIST EQUATES** command to display information about all (or selected) equates, including which are built-in vs. user created.

Syntax:

```
DELETE EQUATES equatenames         ...
               equatenameprefixes* ...
               addressexpressions  ...
               omitted
```

Notes:

- Any number and combination of operands may be given.

- Equates may also be automatically deleted as a side effect of the **FREEMAIN** command.

- As with any other **DELETE** command, a **SILENT=YES** operand can be added to the command (in any position) to suppress the display of progress messages and completion messages. See **HELP COMMANDS DELETE** for details.

Operands

equatenames
These are the names of one or more equates to be deleted.

equatenameprefixes*
These are the start of one or more equate names followed by an asterisk (**RB*** for example). This causes all equates to be deleted whose names have matching starting characters (**RB#1, RB#2, RBwhatever,** for example).

addressexpressions
These are storage addresses at which one or more fixed-location equates have been defined. All such equates are deleted. (Floating equates that happen at the moment to resolve to the same locations are not deleted.)

...
The command will accept any number of any of the above operands for selecting additional equates to be deleted.

omitted
This causes **all** nonbuilt-in equates to be deleted for all address spaces.

For additional information, see:

HELP EQUATES

HELP COMMANDS EQUATE
HELP COMMANDS FREEMAIN

Examples:

Let's start by defining some equates:

- **LIST TASKS:** As a side effect, this command creates a series of automatic **TCB#n** equates that label the locations of each TCB in the current aspace.

- **LIST RBS TCB#5:** As a side effect, this command creates a series of automatic **RB#n** equates that label the locations of each request block queued from **TCB#5.**

- **EQUATE CTCB 21C?:** This creates an equate that labels the current task's TCB.

- **EQUATE CURRENTTCB 21C? FLOAT:** This creates an equate that also labels the current task's TCB, but this one floats. That means that if you are engaging in multitasking debugging, and you switch Z/XDC's focus from one task to another, this equate will automagically move to represent the TCB for the task being switched to. (See **HELP MULTITASK** for more information.)

- **EQUATE TCBOFINTEREST TCB#7:** This creates a fixed-location equate named **TCBOFINTEREST** located at the same location labeled by **TCB#7.**

Given the above, the following **DELETE EQUATES** commands will have the following results:

DEL EQU RB#*
All equates whose names start with **RB#** will be deleted. This will include (but not be limited to) all of the automatic equates created by the most recent **LIST RBS** command.

DEL EQUATES CTCB
This specifically deletes the **CTCB** equate. (Other equates that happen to locate to the same address are untouched.)

DEL EQUATES CTCB+0
The **CTCB** equate and whatever **TCB#n** equate that labels the same TCB are both deleted. That's because **CTCB+0** is interpreted as an address expression, not just a pure equate name.

Note, however, that the **CURRENTTCB** equate is **not** deleted regardless of whether or not it happens to currently resolve to the same location as **CTCB+0.** That's because it is a floating equate, and floating equates cannot be deleted by address (only by name).

DEL EQUATES TCB* CURRENTTCB
This deletes a bunch of equates:

- All automatic equates created by the most recent **LIST TASKS** command,
- The **TCBOFINTEREST** equate because, of course, it starts with **TCB,**
- The **CURRENTTCB** equate because it is explicitly named on the **DELETE** command.

DEL EQUATES
All equates (except those that are built-in) are deleted.

Help COmmands DELete Hooks

The **DELETE HOOKS** command can be used to remove dynamic hooks and to restore original program code to the hook location.

Dynamic hooks can be removed regardless of whether they were created by the current debugging session or by a different or prior debugging session.

Static hooks cannot be removed.

A dynamic hook is one that has been created by the **HOOK** or **HDEFERRED** command. A static hook is one that has been assembled into program code by the **#XDCHOOK** macro. For comprehensive information about hooks, see HELP HOOKS.

The **LIST HOOKS** command can be used to display information about only those dynamic hooks that were created by the current debugging session. If you need to delete a dynamic hook that was created by another debugging session, you are going to have to know either where that hook is located or where its **HOOKCBE** control block is located. Z/XDC is not going to be able to tell you. (The **FIND** command could prove useful for finding such hooks. See below for an example.)

When a hook is created by the **HOOK** command, it is assigned a name of the form **HOOKnnnn.** Such names are displayable by the **LIST HOOK** command. Note however, that Z/XDC's knowledge of hook names survives only for the debugging session within which the hook was created. Hooks from other debugging sessions are unnamed as far as the current debugging session is concerned.

Syntax:

```
DELETE HOOKS hookaddress     ...
             hookcbeaddress
             hkodeaddress
             omitted
```

Notes:

- Any number of operands may be given, one for each hook to be deleted.

- As with any other DELETE command, a **SILENT=YES** operand can be added to the

command (in any position) to suppress the display of progress messages and completion messages. See HELP COMMANDS DELETE for details.

hookaddress
This operand is an address expression that gives the address of the dynamic hook that is to be removed. The address expression may resolve to any virtual address located in any address space in the System (System Security and program authorization permitting). See HELP ADDRESSING for a comprehensive discussions of address expressions.

The address expression must resolve to a location that contains a dynamic hook (i.e. a hook created by the **HOOK** command). The hook may have been created either by the current debugging session or by any other prior or concurrent debugging session.

hookcbeaddress
This operand is an address expression that gives the address of the Hook Control Block (HOOKCBE) that is associated with the hook to be removed.

A dynamic hook is a sequence of 1 or more instructions that jump to a custom instance of Hook Processing code. A HOOKCBE is a control block that precedes the instruction sequence target. It starts at an eyecatcher that is the string **xxxHOOK** (where **xxx** is your Z/XDC's current clone name). However, it is not really expected that customers will know or care much about HOOKCBEs. This form of the DELETE HOOK command was created principally for Z/XDC's own internal use.

hkodeaddress
This operand is an address expression that gives the address of the HOOK Processing code (HKODE) that is associated with the hook to be removed. This is the location that the hook's instruction sequence jumps to.

It is not expected for customers to know or care about HKODE. This form of the DELETE HOOK command was created principally for Z/XDC's own internal use.

...
The **DELETE HOOKS** command accepts multiple address expressions for deleting multiple hooks.

omitted
If no address expressions are given, then **all** dynamic hooks created by the current debugging sessions (and displayable by the LIST HOOKS command) are removed.

Related commands:
- HOOK
- LIST HOOKS

Related topics:

- HELP HOOKS: A comprehensive discussion of hooks and how to use them in a debugging session.

Examples:

```
HOOK .PUTRET
LIST HOOKS
DELETE HOOKS HOOK0001 .PUTRET
```

- The **HOOK** command places a hook at the address labeled **PUTRET.**
- The **LIST HOOKS** command displays all dynamic hooks that have been created so far in the current debugging session.
- The **DELETE HOOKS** command deletes two hooks, one labeled **HOOK0001** and the other located at **PUTRET.**

Help COmmands DELete Ilits

The **DELETE ILITS** command can be used to delete IPCS literals. This can only be done under dump/XDC. Multiple deletions can be performed with one command.

Syntax:

DELETE ILITS name ...

Notes

- Any number of and any combination of operands may be given in any order. (except that it doesn't make much sense to give * in combination with other operands).
- As with any other DELETE command, a **SILENT=YES** operand can be added to the command (in any position) to suppress the display of progress messages and completion messages. See **HELP COMMANDS DELETE** for details.

name
"name" must be an IPCS literal name. An IPCS literal name is from 1 to 31 characters long, and must start with an alphabetic or national character, and the remaining characters must be alphanumeric or national.

Examples:

DEL ILIT MYL1 MYL2
This would delete IPCS literals called MYL1 and MYL2

Help COmmands DELete MAPLibs

MAPLIBs is Z/XDC's name for those sequential and partitioned datasets in which the MAP and DMAP commands can find ADATA for building source image maps of programs and control blocks. The MAPLIB datasets may be any of the following:

- SYSADATA output created by any mainframe Assembler that supports producing IBM-compatible ADATA. As of this writing, there are three such assemblers that I know of:
  - IBM's High Level Assembler (https://www.ibm.com/products/high-level-assembler-and- eature) Use PARM=ADATA.
  - Tachyon Software's Cross Assembler and z/Assembler (www.tachyonsoft.com/txaover.html).
  - Dignus' Systems/ASM (www.dignus.com).

- A GOFF file containing embedded ADATA. (If using IBM's High Level Assembler, then specify "PARM='GOFF(ADATA)'".)

- A file containing streamed ADATA produced by a PC based mainframe Assembler and then binary-uploaded to the mainframe into a RECFM=FB file of arbitrary LRECL.

A **MAPLIB list** is a list of MAPLIB files. In Z/XDC multiple, named MAPLIB lists can be defined, but at most only one list can be active. For more information, see **HELP MAPS ADATA MAPLIBS.**

MAPLIB lists...
- Can be created and managed by the **SET MAPLIBS** command.
- Can be displayed by the **LIST MAPLIBS** command. command.
- Can be purged by the **DELETE MAPLIBS** command.

The **DELETE MAPLIBS** command purges either entire lists or individual entries from a MAPLIB list. It does **not** delete MAPLIB files themselves, it only purges Z/XDC's knowledge of those files.

When ADATA is read from a MAPLIB, Z/XDC caches an extraction of that ADATA into 31-bit storage just in case the same ADATA file might have to be read again. Thus, a MAPLIB entry can "own" one or more cached ADATA files. For more information, see **HELP MAPS ADATA CACHING.**

When a MAPLIB entry is purged, all cached ADATA files that it owns are also purged.

Syntax:

```
DELETE MAPLIBS dsname          ...
               dsname(member)  ...
               #nnn            ...
               listname        ...
               ALLSAVED        ...
               ALLACTIVE       ...
               ALL
```

Notes:

- Generally, any number and combination of operands can be given in any order, although some combinations of operands make less sense than others.

- As with any other DELETE command, a **SILENT=YES** operand can be added to the command (in any position) to suppress the display of progress messages and completion messages. See **HELP COMMANDS DELETE** for details.

dsname
dsname(member)
These operands purge, by dataset name, individual entries from the active MAPLIB list.

#nnn

"nnn" must be a decimal number. This operand permits you to purge entries from a MAPLIB list by referencing its sequence number as displayed by the LIST MAPLIBS command.

#nnn may be used only if the set and ordering of the active MAPLIB list has not changed since the last time a LIST MAPLIBS command was used.

If multiple **#nnn** operands are given within one DELETE MAPLIBS command, then the association of "nnn" values to MAPLIB list entries remains as displayed by the last preceding LIST MAPLIBS command throughout the processing of the DELETE MAPLIBS command.

After a DELETE MAPLIBS command has completed, the association of "nnn" values to MAPLIB list entries is lost and not reestablished until the next issuance of a LIST MAPLIBS command. Any event that changes the set and ordering of the active MAPLIB list causes this association to be lost. Examples: SET MAPLIBS command, PROFILE READ command, etc.).

listname

This purges the named MAPLIB list from the user’s session profile. Warning, in order for this purge to be made permanent, a PROFILE SAVE command needs to be issued following this command.

ALLSAVED
This purges all MAPLIB lists from the user's session profile. (It does not affect the active MAPLIB list.) Warning, in order for this purge to be made permanent, a PROFILE SAVE command needs to be issued following this command.

ALLACTIVE
This purges all entries from the active MAPLIB list. It does not affect the named lists saved in the user's session profile. Note, this is the same as the SET MAPLIBS RESET command.

ALL
This purges all entries from the active MAPLIB list and all lists saved in the user's session profile. Warning, in order for this purge to be made permanent, a PROFILE SAVE command needs to be issued following this command.

Examples:

Assume that a LIST MAPLIBS command produces the following display:

```
MAPPING INFORMATION IS AVAILABLE FROM THE FOLLOWING LIBRARIES AND DATASETS:
#1   (SEQ) DBCOLE.S20.ADATA(DBCMAPS)
#2   (PDS) DBCOLE.S20.ADATA
#3   (PDS) DBCOLE.S10.ADATA

THE FOLLOWING MAPLIB LISTS ARE SAVED:
    DEFAULT  VERS10   VERS20
```

Then the following command sequences would have the following effects:

DEL MAPLIBS #1 #3
This would remove active MAPLIB files number 1 and number 3, leaving only number 2. In addition, any ADATA cache files owned by these MAPLIBS also would be purged.

The operands "#1" and "#3" would not be accepted unless a preceding LIST MAPLIB command had been issued. After this command completes, the next issued LIST MAPLIB command would show what had been the 2nd MAPLIB file to now be MAPLIB file #1.

DEL MAPLIBS hlq.XDCV31.XDCADATA #3
This would remove active MAPLIB files number 2 and number 3 only. Note that "hlq.XDCV31.XDCADATA" does **not** match MAPLIB file number 1.

DEL MAPLIBS DEFAULT VERS10
PROFILE SAVE

The DELETE MAPLIBS command removes the MAPLIB lists named "DEFAULT" and "VERS10" from Z/XDC's session profile. The "PROFILE SAVE" command then writes the profile to DASD thereby hardening the change.

DEL MAPLIBS ALLSAVED S=Y
This command purges all of the named lists (PROFILE, VERS10, and VERS20) from Z/XDC's session profile. But if a PROFILE SAVE command is not issued, this change will evaporate at the end of the debugging session. A display of the changes is suppressed. Note, the active MAPLIB list is not affected by this command.

Help COmmands DELete MAPS

The **DELETE MAPS** command is used to remove one or more dsect maps, csect maps, source level maps, and/or module maps from Z/XDC's storage. Maps can be selected for deletion either by name or by address.

Syntax:

```
DELETE MAPS name                ...
            prefix*             ...
            mname.cname         ...
            mname.cprefix*      ...
            mname.*             ...
            mprefix*.cname      ...
            mprefix*.cprefix*   ...
            mprefix*.*          ...
            addressexpression   ...
            omitted
            *
            *.*
```

Notes:

- Any number of operands may be given in any combination and any order.

- As with any other DELETE command, a **SILENT=YES** operand can be added to the command (in any position) to suppress the display of progress messages and completion messages. See HELP COMMANDS DELETE for details.

- the value of the **USS=option** operand (which may not have been specified on the **DELETE** command) could be relevant. See the section named **MAP name match rules** for more information.

name
This is a pure name. It may be the name of a specific module map or dsect map to be deleted.
- If a module map is found that matches this name, then it is deleted. (see below

for the match rules)
- If that module has one or more csect maps within it, then all of those csect maps are deleted as well.
- If that module map has one or more aliases associated with it, then they are all deleted as well.
- If both a module map and a dsect map are found having this same name, then they both are deleted.
- Note, if "name." is given instead of "name", that is **not** a pure name, so it is treated as being an address expression and is handled very differently. See below for more information.

prefix*
This is a pure name suffixed by an asterisk.
- All module maps and all dsect maps are deleted whose names match the given prefix. (see the section named **MAP name match rules** for more information)
- If any matching module has one or more csect maps within it, then all of those csect maps are deleted as well.
- If any matching module map has one or more aliases associated with it, then they are all deleted as well.

mname.cname
This is a pure module map name followed by a pure csect map name. It identifies a specific csect map to be deleted. (see the section named **MAP name match rules** for more information) It does **not** delete a module map, only a csect map. If the named module contains the named csect map, then that map is deleted.

mname.cprefix*
This is a pure module map name followed by a prefix name, suffixed by an asterisk. (see the section named **MAP name match rules** for more information)
- If the named module contains any csect maps whose names match the given prefix, then those csect maps are deleted.

mname.*
If the named module contains any csect maps at all, then all of those csect maps are deleted. (The module map is **not** deleted.) (see the section named **MAP name match rules** for more information)

mprefix*.cname
mprefix*.cprefix*
mprefix*.*
These operands are similar to the various "mname.whatever" operands described above except that the action is performed for csect maps that are within **all** modules whose names match the given mprefix. (see below for the match rules)

addressexpressions (for deleting maps by location instead of by name)

If an operand does not fit the syntax of a specific map name, then Z/XDC attempts to parse it as an address expression. If successful, then Z/XDC deletes **all** maps (csect, dsect, source level, and module) that contain the specified address.

However, before deleting a module map, Z/XDC first checks to see if the mapped load module contains other mapped csects. If so, then the module map is not deleted.

omitted
*****
.
ALL maps are deleted for all address spaces.

MAP name match rules
When determining if a map name matches a provided name, the following rules are used. (the first rule that is true is used):

- If the map name does not contain any lowercase characters, the map name (or prefix) is compared against the (uppercased) supplied value
- If **USS=NO** is in effect, The map name is uppercased (in a work area!) and the uppercased map name (or prefix) is compared against the (uppercased) supplied value
- (the map name has lowercase characters and **USS=YES** is in effect), the mixed-case map name is compared against the (asis) supplied value

For more information, see:
HELP MAPS
HELP COMMANDS MAP
HELP COMMANDS DMAP
HELP COMMANDS SET ASID
HELP ADDRESSING

Examples:

DEL MAPS GEORGE
In this example, "GEORGE" is pure name. Suppose a load module named GEORGE has been mapped. Suppose also that several csects within GEORGE have also been mapped. Then the following will occur:

- All csect maps within GEORGE will be deleted.
- GEORGE’s module map will also be deleted.
- If there happens also to be a dsect map named GEORGE, then that too will be deleted.
- No other dsect maps (regardless of location) will be deleted.

DEL MAPS GEORGE.
DEL MAPS GEORGE+0
In these cases, "GEORGE." and "GEORGE+0" are treated as being address expressions (not pure names). Therefore, Z/XDC interprets this as a request to delete maps by location, not by name, and Z/XDC resolves these expressions to the load module’s

entry point address. The following results are possible:

- If GEORGE contains no csect maps, then GEORGE’s module map is deleted.
- If the control section located at GEORGE’s entry point address is mapped, then that csect map is deleted. Then if GEORGE contains no other csect maps, then GEORGE’s module map is also deleted.
- On the other hand, if GEORGE does contain other csects (other than the one located at GEORGE’s entry point address), and if any of those csects are mapped, then GEORGE’s module map is **not** deleted.
- If any dsect maps happen to be based such that they extend at or across the module’s entry point address, then they too are deleted.

DEL MAPS GEORGE.*
This causes all csect maps within GEORGE to be deleted. The module map for GEORGE itself is **not** deleted.

DEL MAPS GEORGE.XYZ*
This causes all csect maps within GEORGE whose names start with XYZ to be deleted. The module map for GEORGE itself is **not** deleted.

DEL MAPS GEORGE.XYZ
If GEORGE contains a csect named XYZ, and if that csect has been mapped, then that csect map is deleted. The module map for GEORGE itself is **not** deleted.

DEL MAPS GEORGE.XYZ.
DEL MAPS GEORGE.XYZ+0
In these cases, the operands are treated as being address expressions. Therefore, these expressions are resolved to the start of the csect named XYZ. The following results are possible:

- If GEORGE contains no csect maps, then GEORGE’s module map is deleted.
- If the XYZ csect is mapped, then that csect map is deleted. Then if GEORGE contains no other csect maps, then GEORGE’s module map is also deleted.
- On the other hand, if GEORGE does contain other csects (other than XYZ) and if any of those csects are mapped, then GEORGE’s module map is **not** deleted.
- If any dsect maps happen to be based such that they extend at or across the start of the XYZ csect, then they too are deleted.

DEL MAPS LBR*.Q*
All csect maps (if any) whose names start with "Q" are deleted from all load modules whose names start with "LBR". No module maps are deleted.

**DEL MAPS LBR*
**
All module maps are deleted for all load modules and program objects whose names start with "LBR". If any of those modules also had csect maps, then they are all deleted as well.

**DEL MAPS
**
All maps (all module maps, all csect maps, and all dsect maps) are deleted **regardless** of the address space or data space to which they might be assigned.

Now for the following examples, suppose that the IKJEBIN1 and IKJEBIN2 csects of the TSO editor (EDIT) have been mapped. This means that the module map for **EDIT** also exists and that the alias name **E** exists for that map. Suppose further that the Task Control Block dsect map (named TCBFIX) has also been loaded.

**DE MAPS EDIT.IKJEBIN1
**
This deletes the csect map for IKJEBIN1.

**DE MAPS EDIT TCBFIX
**
This deletes the csect maps for IKJEBIN1 and IKJEBIN2 and the module map for EDIT and the module map for the alias named E and the dsect map named TCBFIX.

**DE MAPS
**
This deletes **all** dsect, csect, source level, and module maps currently defined to Z/XDC.

Another Example:

"HASJES20" is the name of JES2’s primary load module. The csect located at HASJES20’s load point is named HASPNUC. In addition, there is a control block named the "HCT" (JES2’s "HASP Control Table") located at the start of HASPNUC.

Suppose that the module map for HASJES20 exists and the csect maps for HASPNUC, HASPPRPU, and HASPRDR also exist. Also suppose that the dsect map for the HCT also exists and a USING command has been issued to assign it to represent the HCT control block located at the start of HASPNUC. Then the following commands will have the following effects:

**DEL MAPS HASJES20.X#1
**
"HASJES20.X#1" is an address expression that refers to the load point of the HASJES20 load module. This is the location of both the HASPNUC control section and

the HCT control block. Therefore, this command will cause Z/XDC to attempt to delete all maps that cover the address represented by "HASJES20.X#1". Therefore, Z/XDC will delete both the dsect map for the HCT and the csect map for HASPNUC. However, the module map for HASJES20 will **not** be deleted because of the existence of the HASPPRPU and HASPRDR csect maps.

DEL MAPS HASJES20
"HASJES20" is a pure name. It is the name of the module map for the load module of the same name. Therefore, this command will force Z/XDC to delete the module map. However, before it can do that, Z/XDC must first delete all csect and source level maps that are associated with HASJES20. Accordingly, Z/XDC first deletes the HASPNUC, HASPPRPU, and HASPRDR csect maps. Then it deletes the HASJES20 module map. Note, however, that Z/XDC does **not** delete the HCT dsect map because Z/XDC maintains no linkage between dsect maps and other kinds of maps.

Help COmmands DELete MMefdsns

The **DELETE MMEFDSNS** command can be used to delete MMEFDSN definitions. All definitions can be deleted or a specific one. Multiple operations can be performed with one command.

NOTE: Only the MMEFDSN definition is purged from Z/XDC. The actual MMEF dataset is NOT deleted!

Syntax:

```
DELETE MMEFDSNS dsname  ...
                id      ...
                #nnn    ...
                ALL
```

Notes

- Any number of and any combination of operands may be given in any order. (except that it doesn't make much sense to give ALL in combination with other operands).

- As with any other DELETE command, a **SILENT=YES** operand can be added to the command (in any position) to suppress the display of progress messages and completion messages. See **HELP COMMANDS DELETE** for details.

dsname
"dsname" must be a dataset name that matches the dataset name of one of the defined MMEFDSN entries. A dataset name is distinguished from "ALL", a number ("#nnn"), or an id by the fact that most dataset names consist of at least 2 levels, separated by a period ("."). If any periods are present in a value, it is assumed to be a dataset name. If none, then it is the value "ALL", a number (starting with "#"), or an ID value.

id
"id" must be a id value assigned to one of the defined MMEFDSN entries. (Note that valid ID values cannot be "ALL" or start with "#")

#nnn
"nnn" must be a decimal number. This operand permits you to purge **MMEFDSN definitions** from storage by referencing its sequence number as displayed by the LIST MMEFDSNS command.

#nnn may be used only if the set and ordering of MMEFDSNS entries has not changed since the last time a LIST MMEFDSNS command was used to display the entries.

If **multiple** #nnn operands are given within one DELETE MMEFDSNS command, then the association of "#nnn" values to MMEFDSNS definitions remains as displayed by the last preceding LIST MMEFDSNS command throughout the processing of the DELETE MMEFDSNS command.

After a DELETE MMEFDSNS command has completed, the association of **#nnn** values to MMEFDSNS entries is discarded and is not reestablished until the next issuance of a **LIST MMEFDSNS** command. In fact, any command or event that changes the definition of MMEFDSNS entries causes this association to be lost too.
Examples:
- **DEFINE MMEFDSN** commands

ALL
This causes all MMEFDSNS definitions to be purged from storage.

Examples:

Assume that a **LIST MMEFDSNS** command produces the following display:

```
    seq dsn                                          id
    #1  userid.TEST.OFBC2                            -
        D=2020/11/12 T=13.44.24 Mb=2 C=N NUC=N PLPA=N LL=N LIB=Y-4
    #2  userid.TEST.OFCC2                            XYZZY
        D=2020/11/12 T=13.52.42 Mb=4 C=Y NUC=Y PLPA=Y LL=N LIB=N
    #3  userid.TEST.OFCX1                            -
        D=2020/11/12 T=13.52.42 Mb=4 C=Y NUC=Y PLPA=Y LL=N LIB=N
```

Then the following command sequences would have the following effects:

DEL MMEFDSN #1
This would remove MMEFDSN definition number 1. The operand "#1" would not be accepted unless a preceding LIST MMEFDSNS command had been issued. After this command completes, the next issued LIST MMEFDSN command would show what had been the 2nd definition to now be the 1st definition. Also, after this command has been

processed, #nnn operands could not be used on subsequent DELETE MMEFDSNS commands unless an intervening LIST MMEFDSNS command was issued.

DEL MMEFDSNS XXXX.TEST.OFCX1
This would remove the definition for the MMEFDSN named 'userid.TEST.OFCX1'

DEL MMEFDSNS XYZZY
This would remove the definition for the MMEFDSN named 'userid.TEST.OFCC2' (which has an ID of XYZZY)

DEL MMEFDSNS ALL
This command would purge all of the MMEFDSNS definitions

Help COmmands DELete MOdules

The **DELETE MODULES** command is used to remove load modules from the Home Address Space. Only load modules (or program objects) that have been brought into storage via:
- a LOAD assembler macro
- the loadhfs[_exended] USS assembler calls

issued under the current TCB can be deleted.

Note that the **USS=option** and **DUB=option** operands (or their default values if omitted) are relevant here.

This command cannot be used under dump/XDC.

Syntax:

```
DELETE MODULES  entry    entry    ...
DELETE PROGRAMS
DELETE PGMS
DELETE entry    entry    ...
```

Where:

MODULES PROGRAMS PGMS
The deletion type keywords **(MODULES PROGRAMS** or **PGMS)** are optional. But if the (unquoted) name in the first entry happens to match any of the typing keywords, then you will have to use one or another of **MODULES PROGRAMS** or **PGMS** (or their permitted abbreviations). For a complete list of the deletion type keywords, see HELP COMMANDS DELETE.

entry

Indicates what is to be deleted. Any number of entries may be specified on a single **DELETE MODULES** command.

The syntax of an entry is as follows:

name
'name'
"name"
address-expression

where:

name
'name'
"name"
the name of the item to be deleted.

A quoted value is always treated as a USS-format file or path name, regardless of whether **USS=YES** is in effect.

An unquoted value is treated as:
- an 8-character PDS(E) member name if **USS=NO** is in effect.
- a USS file or path name if **USS=YES** is in effect.

address-expression
An address that can be anywhere within a loaded module that is to be deleted.

If Z/XDC detects that an address expression has been used, it issues either a DELETE assmbler macro or the deletehfs assembler call against the primary name and a DELETE macro against all aliases of the primary name.

You can force the value to be treated as an address expression by:
- not using quotes and
- suffixing **+0** onto the value.

For example **iefbr14** will be treated as a name but **iefbr14+0** will be treated as an address expression.

USS notes
If Z/XDC detects that the module to be deleted is a USS-loaded module (i.e., loaded by the **loadhfs[_extended]** assembler call) then deletion is performed by the **deletehfs** assembler call.

If the file or path name does not start with '/' then the user's CWD (Current Working Directory) is prefixed onto the file or path name.

If the file or path name contains characters that have a special meaning to Z/XDC, the name **must** be quoted. Refer to **HELP DEBUGGING USS NAMES** for more information.

If Z/XDC detects that the current environment needs to be dubbed then the DUB option is relevant. If dubbing is not allowed the command is aborted with an error message.

Examples:

DE PGMS MAPS MYPROG
The use counts for modules named MYPROG and MAPS are decremented by 1. If a module's count reaches 0, then that module is removed from storage.

DE MAPS+0
"MAPS+0" is interpreted as an address expression, not just a keyword or a pure load module name. Consequently, the use counts for the major name AND all minor names for the load module named "MAPS" are each decremented by 1. If the use count for any name reaches 0, then that name is removed from storage. If the use counts for all names reach 0, then the module too is removed from storage.

DE PGMS pgms/myprog USS=YES
the CWD for the current user is assumed to be set to **/users/u1** ...).
The use count for the USS-loaded module **/users/u1/pgms/myprog** is decremented by 1.
If the use count reaches 0, then that module is removed from storage.

Help COmmands DELete Proxytasks

The **DELETE PROXYTASKS** command is used to terminate **Formal Proxy Tasks.**

Proxy Tasks are tasks used to run the **Remote Phase** of Z/XDC processing when Z/XDC is running as an **FRR** (in order to debug **SRB** routines and **FRR** protected Task Mode routines). For more information, see **HELP DEBUGGING FRR.**

Formal Proxy Tasks are those that have been created...
- By **XDCCALLA** pursuant to the presence of a **//xxxFPTnn DD DUMMY** allocation.
- Or through use of the **SET PROXYTASKS** command.

The Proxy Tasks to be deleted must meet particular requirements:
- They must be **Formal** Proxy Tasks. Such tasks have a particular structure that is needed by the DELETE PROXYTASKS command.
- They must have been created either by or on behalf of the current Z/XDC clone (not by other clones).
- They must not have subtasks ATTACH'd to them.
- They **may** have IRB routines queued and running under them (in which case the task will terminate when the IRB terminates).
- They may reside in **any** accessible address space.

This command does not directly DETACH Proxy Tasks. Instead, the ECB upon which a Proxy Task is waiting is POST'd. Then when the task awakes, it immediately terminates itself via an **SVC 3** instruction.

This DELETE PROXYTASKS command cannot be used unless Z/XDC is running authorized.

A single use of this DELETE PROXYTASKS command can delete multiple Proxy Tasks.

Syntax:

DELETE PROXYTASKS tcbaddress ...

Notes:
- At least one operand is required.
- Any number of TCB addresses may be given on the command.

tcbaddress

This must be an address expression that resolves to the location of a Task Control Block for the Proxy Task to be deleted.

The TCB can be located in any accessible address space.

Eligible TCBs must be running Formal Proxy Tasks for the current Z/XDC clone.

Any number of **tcbaddress** operands may be given. They are processed left to right either until all have been processed successfully, or until an error is encountered (in which case all unprocessed operands are discarded).

The TCB Address Expressions

Any address expression is valid so long as it resolves to the address of a Task Control Block for a task that is running a **Formal Proxy Task.** For examples, see below.

As a side effect, the **LIST TASKS** command creates a series of **TCB#n** equates labeling the locations of all listed TCBs. Every time the LIST TASKS command is reissued, the TCB#n equates are recreated. These TCB#n equates are very convenient for use as **tcbaddress** operands on this DELETE PROXYTASKS command.

Keep in mind that this DELETE PROXYTASKS command results in changing the address space's Subtask Tree. That means that the next time you issue a LIST TASKS command, the values of at least some TCB#n equates will change! This can be confusing if you're not paying attention.

However, the values of TCB#n equates do not change within the processing of this DELETE PROXYTASKS command. (They are changed only by LIST TASKS commands.) So no problems arise from using multiple TCB#n equates within a DELETE PROXYTASKS command. See below for some examples:

Examples

DELETE PROXYTASKS 21C?
21C? certainly is a valid address expression for referencing the current task's TCB; however, is the current task a Proxy Task? Well that depends...

If Z/XDC is currently being used as an FRR to debug an SRB routine, then yes, the **Remote Phase** of Z/XDC processing is currently running, and that phase is running in the current task, so yes, in this case the true current task is a Proxy Task.

So this command succeeds, but what happens? Well, immediately, nothing. But the task has been posted, so once Z/XDC releases control (i.e. when a **Go** or **TRACE** or **END** command is issued) the Proxy Task will terminate.

That won't cause problems, though, because the next time SRB debugging needs a Proxy Task, it will just use another one.

Suppose that a **LIST TASKS** command shows the following Task Tree:

```
_ CBD ===> list tasks
_    TCB#  PROGRAM NAME (ASID 00A8, DBCOLE3)
_    1     IEAVAR00
_    2     . IEESB605
_    3     . . IKJEFT01, JOBSTEP
_    4     . . . IKJEFT02
_    5     . . . . XXXCALLA
_    6     . . . . . "CBDCALL" (XXXT0004)
_    7     . . . . . . "XDCTESTS" (ABEND: s0C1) [awaiting CBD]
_    8     . . . . . . . "PROXYCBD" (INUSE), SRB-FRR PROXY
_    9     . . . . . . . "PROXYCBD"
_    10    . . . . . . . . "PROXYCBD"
_    11    . . . . . . . . "PROXYCBD"
_    12    . . . IKJEFT02
_    13    . . . . IKJEFT09
_    14    . . . . . ISPF
_    15    . . . . . . ISPTASK (ISPLLIB)
_    16    . IEAVTSDT
```

There's a lot going on here...

- Two debugging sessions are present:
  - A Z/XDC clone named **XXX** (see TCB#5) is being used to debug a 2nd clone named **CBD** (TCB#6),
  - While **CBD** is being used to debug XDCTESTS (TCB#7).
- **CBD** currently is in control and has produced this display.
- The hilighted tasks (TCB#6 thru TCB#11) are within the scope of the **CBD** debugging session. (The scope of the XXX debugging session is not shown.)
- All the Formal Proxy Tasks shown (TCB#8 thru TCB#11) are owned by the **CBD** debugging session. (The XXX debugging session does not own any Proxy Tasks.)
- The CBD clone of Z/XDC is currently being used to debug an SRB (started by the XDCTESTS program), so CBD's Remote Phase is currently running in TCB#8. So TCB#8 is the TCB that is pointed to by **21C?** .
- The LIST TASKS command has created 16 equates named **TCB#1** thru **TCB#16** labeling each of the 16 TCBs in this address space:
  - Prior **TCB#n** equates, if any, have been deleted.
  - The equates can be used as simple and convenient address expressions in any command that needs to reference the address of a TCB.

Now suppose I issued **DELETE PROXYTASKS 21C? TCB#10 TCB#11 TCB#9.** This would cause TCBs #8 (pointed to by 21C?), #10, #11 and #9 (in that order) to be POST'd for termination.

- TCBs #9, #10 and #11 would terminate immediately.
- But TCB#8 would not (because CBD is running as an IRB within that task).
- But when a GO, TRACE or END command is issued, the IRB would terminate which would allow the task to terminate as well. (If the DELETE PROXYTASKS command had not been issued, then TCB#8 would have persisted.)

Thus a subsequent **LIST TASKS** command (issued immediately after the DELETE PROXYTASKS command) would show the following:

```
_ CBD ===> list tasks
_    TCB#  PROGRAM NAME (ASID 00A8, DBCOLE3)
_    1     IEAVAR00
_    2     . IEESB605
_    3     . . IKJEFT01, JOBSTEP
_    4     . . . IKJEFT02
_    5     . . . . XXXCALLA
_    6     . . . . . "CBDCALL" (XXXT0004)
_    7     . . . . . "XDCTESTS" (ABEND: s0C1) [awaiting CBD]
_    8     . . . . . "PROXYCBD" (INUSE), SRB-FRR PROXY (TERMINATION PENDING)
_    9     . . . IKJEFT02
_    10    . . . . IKJEFT09
_    11    . . . . . ISPF
_    12    . . . . . ISPTASK (ISPLLIB)
_    13    . IEAVTSDT
```

And the set of **TCB#n** equates would be rebuilt. This would result in:

- TCB#1-TCB#8 remaining unchanged,
- While tasks 9 thru 13 (which previously had been tasks 12 thru 16) are now represented by the equates TCB#9 thru TCB#13).

Help COmmands DISConnect

Users can connect to background program debugging sessions using cdf/XDC. The background program can be a batch job or a system task. cdf/XDC is a component of Z/XDC that supports fullscreen interactive debugging sessions with batch jobs and system tasks. For more information, see **HELP XDCSRVER CDF.**

The **DISCONNECT** command is used to disconnect a user's terminal from a background debugging session. **DISCONNECT** allows a user to terminate his connection to a background debugging session without terminating the debugging session itself. Instead, the session simply reissues its "awaiting user signon" WTOR **(DBC640Q)** and then waits for a new user connection.

The **DISCONNECT** command can be used only when the terminal is connected via cdf/XDC to a debugging session. The command fails with an error message when it is used outside of an cdf/XDC environment.

Syntax:

```
DISCONNECT KEEP
           DROP
           NEWUSER
           omitted
```

Operands:

KEEP
omitted
The user's terminal is disconnected from the debugging session but not from cdf/XDC. The terminal is returned to cdf/XDC's **Job Selection Menu** from which the user may reconnect to the same debugging session, connect to another debugging session, or disconnect from cdf/XDC entirely.

When the **DISCONNECT** command is given without operands, **KEEP** is the default action taken by cdf/XDC.

DROP
The user's terminal is disconnected both from the debugging session and from cdf/XDC entirely.

- If the user had logged on to cdf/XDC from and idle VTAM terminal, then the terminal is returned to its logged off state.
- If the user had connected to cdf/XDC from a TSO ISPF session, then control of the terminal is returned to ISPF.

NEWUSER
The user's terminal is disconnected from the debugging session. cdf/XDC then displays a Logon Panel from which the user may log back into cdf/XDC using a different TSO userid. Note, if you have connected to cdf/XDC from a TSO session, this relogon does **not** affect your TSO session.

DISCONNECT;GO

The **DISCONNECT** command, by itself, does not cause the user's program to resume execution. Instead, the program remains within the debugging session, and the session just waits for you or another user to connect back to it.

If you want **both** to disconnect from the debugging session **and** to let your program resume execution, then you will need to issue both a **DISCONNECT** command and a **GO** command in the same command string, specifically: **DISC;GO**

Caution: Reversing the two commands (GO;DISCONNECT) will not work. That's because when a program resumes execution (the **GO** command), Z/XDC's input command stream is flushed. So the **DISCONNECT** command will not be seen by Z/XDC, and your terminal will remain connected to the debugging session with the keyboard locked.

Attention Processing

If it happens that you did resume your program without disconnecting from it, and now your keyboard is locked, you can use the attention key unlock the keyboard at which point, cdf/XDC will display a "Yeah? wha d'ya want?" panel. One of the choices offered by that panel is to disconnect you from the session. See **HELP XDCSRVER CDF CONNECTION** for more information.

Help COmmands DISPlay

The **DISPLAY** command produces raw displays of storage using a hex-text format similar to that which you would see in a storage dump. The text portion of a display may be set to show either an EBCDIC or an ASCII interpretation of the storage.

Syntax:

```
DISPLAY addressexpression lines           ADDRESSES ASCII  WIDE
D       omitted           LINES=lines     OFFSETS   EBCDIC NARROW
        ,                 BYTES=hexnumber
```

Shortcut: D

Notes:

Operands appearing above within columns are mutually exclusive.

All operands are optional.

If the addressexpression operand is given, then it must be first. All other operands may be given in any order.

If the addressexpression operand is omitted but other operands are given, then a comma must be used to indicate the omission.

Operands:

addressexpression

This gives the starting address of the storage to be displayed. When the command finishes, the **Current Display Pointer** is set to this address. The **Next Display Pointer** is set to point past the last byte displayed. (For more information, see HELP ADDRESSING IMPLICIT.)

omitted

The display starts with the address pointed to by the **Next Display Pointer.** Usually, this points just past a previous display. When the **DISPLAY** command finishes, the **Current Display Pointer** is set to this address. The **Next Display Pointer** is reset to point past the last byte displayed. This facility simplifies the displaying of successive storage locations.

If the address expression is omitted and other operands are given, then a **comma** must be used to show the omission.

Note: The following operands may be given in any order. They must, however, follow the given address expression, if any.

Also, all of the following operands (except **lines, LINES=** and **BYTES=)** serve to override the corresponding default values established via the **SET FORMAT** command.

lines
LINES=lines
BYTES=hexnumber
These control the size of the display. The **lines** and **LINES=** operands do so in terms of the number of lines (rows) of data the display is to contain. The **BYTES=** operand does so in terms of the amount of storage that is to be displayed:

lines
LINES=lines
These two operands function identically; however the **LINES=lines** form is preferred because eventually support for the **lines** form may be discontinued.

The **lines** value must be a decimal number specifying the number of display lines to generate containing data (i.e. not including the Location Interpretation header lines). This value must be in the range of 0 to 65,535 (64K-1). When omitted, either the default value set by the **SET LINES** command is used (for nonfullscreen displays), or the length of the current window is used (for fullscreen mode displays).

BYTES=hexnumber
This must be a number giving the amount of storage to be displayed. It can be either of the following:

- A hexadecimal number (Example: **BYTES=C8)**
- A decimal number followed by the letter **N** (Example: **BYTES=200N)**
- A scaled decimal number (Example: **BYTES=4K,** but don't make it too big)

The given number must be in the range of **0** to **1FFFE0:**

- The very strange X'1FFFE0' limit is (64K-1)*32:
  - 64K-1 is the LINES= limit.
  - 32 is because one line can display up to 32 bytes of storage.

The number of display lines needed to produce a BYTES= driven display can vary.

The **lines, LINES=** and **BYTES=** operands are mutually exclusive.

ADDRESSES
OFFSETS

These operands control whether the far left column of the display (the "address" column) will show storage addresses or offsets:

ADDRESSES
Causes each line of data displayed to be prefixed by that data's virtual address.

OFFSETS
Causes each line of data displayed to be prefixed by that data's offset from the start of an appropriate including object (load module, csect, dsect, or equate).

ASCII
EBCDIC
These operands control whether the text portions of data displays are to be interpreted using the **ASCII** or **EBCDIC** character set:

ASCII
Causes data displays to show the **ASCII** interpretation of the displayed storage. To visually indicate this, the text portion of each display line will be framed with **vertical bars (|)** instead of asterisks.

EBCDIC
Causes data displays to show the **EBCDIC** interpretation of the displayed storage. To visually indicate this, the text portion of each display line will be framed with **asterisks (*).**

More information on EBCDIC and ASCII can be found at **HELP CHARACTERSETS**

WIDE
NARROW
These operands control whether data displays are to be wide or narrow:

WIDE
This causes data displays to show up to **32 bytes** of storage per line (not just 16). This option works best when your terminal's display is at least 136 columns wide. For more information, see HELP FULLSCREEN TERMINALS.

NARROW
This causes data displays to show only **16 bytes** of storage per line. This width is appropriate for terminals with 80 character wide display lines.

Storage Protect Key Display

The **DISPLAY** command shows the storage protection key as a single hexadecimal digit appearing to the right of the address/offset field, running down the entirety of the display.
Following the storage protect key may be additional characters:
f - The 'f' character indicates that the storage is fetch protected.
s - The 's' character indicates that the storage is store protected. (there are additional comments about store protected storage below).
r - The 'r' character indicates that the storage may be redacted. (this character

can only be seen when processing a redacted dump in dump/XDC). A redacted area is indicated by the storage being all binary zero. Note that it is impossible to distinguish storage that has a value of binary 0 from storage that has been redacted.

The "Store Protected" indicator, **s,** is displayed whenever Z/XDC detects that the storage being displayed has one or another of the following special protections:

- **Low Address Protection** (LAP): This is a special kind of protection that the Operating System uses to specifically protect the first 512 bytes of both real and virtual storage. It prevents the storage from being altered by any program, regardless of that program's state or key. (LAP does not apply to data space storage.)

- **Store Protection:** This is a kind of protection that applies to the following areas of storage:
  - The Pageable Link Pack Area (PLPA)
  - The read-only areas of the System Nucleus (IEASYS0x)
  - Any area of storage that has been set to "read-only" by the PGSER or IARVSERV macros.

  When a page of storage has been marked as being "read-only", that storage cannot be directly altered by any program, regardless of that program's state or key. (Note, "Store Protection" does not apply to real storage. Thus, it cannot protect a virtual storage page when that page is accessed via its real address.)

- **Access List Entry Protection:** When an address space or a data space is accessed via an ALET that "points to" an Access List Entry (ALE) having its "fetch only" flag on, then accesses via that ALE are store protected. Note, accesses to that same space via a different ALE might not be store protected.

Examples:

D MYPROG 4 ASCII
D MYPROG LINES=4 ASCII

Both of these commands display four lines of raw storage located at the start of **MYPROG.** The text portions of the display show the ASCII interpretation of the storage. Each display line will show either 16 bytes of storage or 32 bytes of storage, depending upon the current **NARROW/WIDE** setting in the **SET FORMAT** command.

D

This displays storage following the storage shown in the previous display. The default number of display lines is generated. The text portions of the display revert to showing the EBCDIC interpretation (assuming that **SET FORMAT EBCDIC** is in effect).

D +0 8
D +0 LINES=8

These redisplay the same storage shown in the previous display, but now 8 lines are generated. (The **+0** refers to the **Current Display Pointer.)**

D +0 ADDR 8
D +0 ADDR LINES=8
These redisplay the same storage shown in the previous display, but now the storage addresses are shown as virtual addresses instead of (possibly) offsets.

D +0 WIDE BYTES=C00
D +0 WIDE BYTES=3072N

These redisplay the same storage, but now 8 words will be displayed per line, not just four. (This works best if your terminal has been set up to display 136 characters per line or more.) Also, as many lines will be generated as is needed to display 3,072 bytes (equals X'C00' bytes) of data.

D ,WIDE
This displays storage following the storage displayed by the preceding DISPLAY, FORMAT, or WHERE command. Eight words will be displayed per line, not just four. The comma is necessary to indicate that the addressexpression operand has been omitted.

Other commands that can be used to display storage are:

FORMAT - Formats a contiguous chunk of storage as either source code displays, machine instructions, or data fields, as appropriate.

SHOW - Accepts a list of address expressions and formats a 1-line display for each expression given.

WHERE - Formats an area of storage containing the current **retry** level PSW's resume execution address.

EWHERE - Formats an area of storage containing the Current **error** level PSW's resume execution address.

FIND - Scans storage for a given string of data.

Help COmmands DMap

There are two forms of the **DMAP** command. One loads dsect maps into Z/XDC's storage from disk resident load libraries (PDS or PDSE). The other creates copies of dsects that have already been loaded in. (This is called **cloning.)**

Dsect maps can be built from **ADATA** or **SYM data** for Assembler maps, or from **ESD** information for module maps (load modules and program objects):
- Maps built from **ADATA** are called **source level maps.**
- Maps built from **SYM data** are called **symbol maps.**
- Maps built from **ESD** information are called **module maps.**

Unless otherwise coerced, whenever ADATA is available, the DMAP command will build a

source level map in preference to a symbol map. Generally though, the **SET MAPLIBS** command needs to be used in order to make ADATA available to the DMAP command. For more information, see HELP MAPS.

Licensed Features Required for the DMAP Command

The **asm/XDC Feature** must be licensed, in order to use the DMAP command for building maps from ADATA or SYM data.

The ability to build dsect maps from a load module's ESD data is a part of **base/XDC,** so no special licensing is needed for that.

Dsects do not apply to XL C/C++ programming, so no support exists in the DMAP command for building dsect from their DWARF data.

For more information, see HELP SUPPORT FEATURESANDCAPS.

Source Level Maps

The **DMAP** command builds source level dsect maps from **ADATA** located in:

- Files and libraries containing **SYSADATA** output created by any mainframe Assembler that supports producing IBM-compatible ADATA. As of this writing, there are three such assemblers that I know of:
  - **IBM's** High Level Assembler (https://www.ibm.com/products/high-level-assembler-and- eature). Use PARM=ADATA.
  - **Tachyon Software's** Cross Assembler and z/Assembler (www.tachyonsoft.com/txaover.html).
  - **Dignus'** Systems/ASM (www.dignus.com).

- **GOFF** files containing embedded ADATA. (If using IBM's High Level Assembler, then specify **PARM='GOFF(ADATA)'.)**

- Files containing **streamed ADATA** produced by a **PC based** mainframe Assembler and then binary-uploaded to the mainframe into a RECFM=FB file of arbitrary LRECL.

- **ADATAnnnn classes** found within program objects. These classes will be present in a program object when the Binder finds that ADATA is present in the GOFF file produced by the Assembler.

The **SET MAPLIBS** command is needed for making SYSADATA and GOFF files and libraries available to the DMAP command. However, if it is intended for ADATA to be read from ADATAnnnn classes within a program object, then the SET MAPLIBS command is **not** needed.

The **SET MAPLIBS** command also is not needed when the **ADATALIB=** operand is used with the DMAP command. (See below.)

The **asm/XDC Feature** must be licensed in order to use the DMAP command for building source level dsect maps.

Symbol Maps

The **DMAP** command builds symbol dsect maps from **SYM data** located in a load module or program object when:

- The program is **assembled** with PARM=TEST specified.
- And the program is **linkedited** with PARM=TEST specified.
- And ADATA is **not present** or available.
- And the the **asm/XDC Feature** has been licensed.

Module Maps

The **DMAP** command builds load module **dsect** maps from **ESD data** located within load modules or program objects.

A load module dsect map is exactly like a normal module map (as might be loaded by the **MAP** command) except that Z/XDC does not automatically assign it to any particular location in storage. Instead, like any other dsect map, a **USING** command has to be issued, allowing the user to assign it to represent any storage he sees fit. See HELP MAPS CSECTSASDSECTS for more information about this.

Cloning Existing Dsect Maps

Once a dsect map has been brought into storage, any number of uniquely named copies can be created and then independently manipulated. This is useful when there is a need to map multiple copies of a given control block (TCBs, for example).

Mixed-Case Support

Module maps may or may not be flagged as being **case-sensitive.** If the map contains mixed-case names, then it will be flagged as being case-sensitive; otherwise, it will not. For more information about the consequences of this, see HELP ADDRESSING PARSERS ASM MIXEDCASE.

Dsect maps built from ADATA or SYM data are **never** flagged as being case-sensitive. This is because even though the Assembler supports mixed-case names, it treats those names in a case-insensitive way: When two names are identical except for case, the Assembler considers them to be the exact same name.

Command Dialog

As you can see, the DMAP command is a pretty powerful but complex command. If you would like Z/XDC to construct a DMAP command for you, you can use a Command Dialog. See **HELP CMDDIALOGS** for more information.

Syntax:

?
DMAP

DMAP ?

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

LOAD - How to load a new dsect map. Also, how to load a csect or a module map as if it were a dsect.
CLONE - How to clone an existing dsect map.

Help COmmands DMap Load

The **load form** of the **DMAP** command loads a new dsect map into storage from disk resident data. This data may be in the form of a SYSADATA file, a GOFF file, a load module, or a program object.

Unlike the MAP command, the DMAP command does not require that the corresponding load module or program object actually exist in storage.

Dsect maps can be built from ADATA or SYM data for Assembler maps, or from ESD information for Binder maps.

- Maps built from ADATA are called **source level maps.**
- Maps built from SYM data are called **symbol maps.**
- Maps built from ESD information are called **Binder maps.** They show where the csects are within load modules and program objects.
- Any map loaded by the DMAP command is considered to be a "dsect". This is regardless of what sort of "section" the underlying mapping data represents. They can be built from:
  - Assembly time DSECT data,
  - Assembly time CSECT data, or
  - Bind time ESD data.
- Maps loaded by the DMAP command are not assigned to represent storage until a **USING** command is issued.

Source Level Maps

Unless coerced otherwise, whenever ADATA is both allowed and available, the DMAP command will build a source level map in preference to a symbol map. Note, you can:

- Use the **SET MAPLIBS** command to make ADATA available.
- Use the DMAP command's **ADATA** or **ADATA=** operand to coerce using ADATA.
- Use the **ADATA=** operand both to coerce using ADATA and to provide the name of the ADATA dataset or library from which it is to be loaded (instead of from a MAPLIBS library).
- Use any of the following operands to coerce **against** using ADATA: **ESDDATA, ESDDATA=, SYMDATA** or **SYMDATA=**

dump/XDC Considerations

When Z/XDC is running within a dump/XDC debugging session, if the dump originated from a third party system, then the mapping data for modules and cblocks within the dump may also have to originated from that third party system. For more information, see:

- **HELP DUMPS EXPECTEDUSES ISVUSAGE:** For discussions regarding dumps received from third party systems.
- **HELP DUMPS MAPPINGDATA:** For discussions of setting up mapping data for third party dumps.
- **HELP DUMPS:** For complete information about dump/XDC debugging sessions.

So mechanisms exist by which dump/XDC can set up overrides of DMAP's normal process for searching for mapping data. There are two such mechanisms:

- **MMEFDSNs** are tokens that cause accesses to load libraries to be intercepted at the I/O level and redirected to **Module Mapping Extraction Files.** These are files that are created on third party systems and then are FTP'd (along with a dump) to you, the person tasked with solving the dump.

  MMEF files are intended to contain mapping data that is matched to the dump. They contain extractions of ESD data (mainly) and SYM data and ADATA (when present and requested). These extractions are taken from load modules and program objects only (never ADATA side files).

  MMEFDSN tokens are created by the **DEFINE MMEFDSN** command. For more see:
  - **HELP COMMANDS DEFINE MMEFDSN**
  - **HELP DUMPS MAPPINGDATA MMEFFILES**

- **DSECTLIBs** are tokens that provide the names of libraries that are to be searched instead of task libraries, PLPA libraries and linklist libraries. (MAPLIB searches remain unaffected by the presence of DSECTLIBs.)

  DSECTLIB libraries may be:
  - Load libraries containing ESD data (and perhaps SYM data and/or ADATA),
  - ADATA side file libraries.

  When DSECTLIBs are present:
  - MAPLIBS will still be searched for ADATA.
  - But System Search Order load libraries (task libraries, etc.) will be skipped.
  - DSECTLIB libraries will be searched instead of Search Order load libraries.

  When DSECTLIBs are absent:
  - MAPLIBS may be read for ADATA (if appropriate).
  - Load libraries may be read for SYM data, ESD data or ADATA (as appropriate).

  DSECTLIB tokens are created by the **DEFINE DSECTLIB** command. For more see:
  - **HELP COMMANDS DEFINE DSECTLIB**
  - **HELP DUMPS MAPPINGDATA**

Note, you can prevent the use of DSECTLIBs and MMEFDSNs by using the **USELDS** operand on the DMAP command. (See below.)

If all of the following are true:
1: Neither the **USELDS** or **NOUSELDS** operands have been specified.
2: A dataset name was specified on the command that could be defined in an MMEF dataset (i.e. a positional dataset name or the **SYMDATA=** operand or the **ESDDATA=** operand but not the **ADATA=** operand).
3: This is not the 'host' system (i.e. this is a dump)

4: No MMEF datasets have been defined.
then processing of the command behaves as if the **USELDS** operand was specified.

Command Dialogs

A Command Dialog is available for assisting with constructing and issuing DMAP commands. Just issue the command either with no operands or with a lone question mark as its sole operand.

Licensing

Before the DMAP command attempts to build maps from ADATA or SYM data, it first checks to see if the **asm/XDC Feature** has been licensed. If not, then the attempt is bypassed. For more information, see **HELP SUPPORT FEATURESANDCAPS.**

Syntax:

?
DMAP
DMAP ?

[Invokes a Command Dialog]

DMAP membername.dsectname [llorpathname] ...
membername.csectname
modulename.
.dsectname
.csectname

... ADATA
ADATALIB=dsorlibname
ADATAPATH=pathname
ADATAFILE=pathname

SYMDATA
SYMDATALIB=dsorlibname
SYMDATAPATH=pathname
SYMDATAFILE=pathname ...
ESDDATA
ESDDATALIB=dsorlibname
ESDDATAPATH=pathname
ESDDATAFILE=pathname

... RENAME=newname ZEROPOINT=offset ...

... [NO]ALLMESSAGES

... [NO]USELDS

... **[USS=option]**

... **[DUB=option]**

Operands:

membername.dsectname [the 1st positional operand]
membername.csectname [the 1st positional operand]
The **membername** part of this operand gives the name of the member of a partitioned dataset wherein Z/XDC can find mapping data (ADATA or SYM data).
- In the case of load libraries, it will the the name of a load module or program object.
- In the case of an ADATA side file library, it will be whatever member name is necessary to reach the correct ADATA.

The **.dsectname** or **.csectname** part of this operand gives the name of the map to be loaded as a dsect (for .dsectname) or as if it were a dsect (for .csectname).

.dsectname [a form of the 1st positional operand]
.csectname [a form of the 1st positional operand]
If the membername part of the operand is **omitted** from the DMAP command, then the default membername used depends upon whether or not a membername has been explicitly given on a **prior** DMAP command issued during the current debugging session:
- If all prior DMAP commands also have omitted the membername, then the current DMAP command will use the load module name currently defined via the **SET QUALIFIER** command.
- Otherwise, if any prior DMAP command explicitly specified a membername, then the most recently specified name will be used as the default.

If you omit the membername from the DMAP command, you must still specify the dot **(.)** that would have followed the membername. Example: **DMAP .TIOT**

You can use the **LIST QUALIFIER** command to display what DMAP's default membername currently is.

If you give a membername, then you **must** trail the name with a dot. Examples:
DMAP XDCMAPS.TIOT
DMAP CICSPGM.

If you give a membername and a dot **without** also giving a dsect name, then Z/XDC presumes that membername is the name of a program object or load module and that you want Z/XDC to read that program's Binder map **as if it were a dsect.** Doing this allows you to use the **USING** command to assign the Binder map to any location you want. Example:
- **DMAP MYCICSPG.** [Notice the trailing dot]
- **USING MYCICSPG R12?**

See **HELP COMMANDS DMAP LOAD CSECTSANDBINDERMAPS** for more information.

. [a dot]
The presence of this character in the command's first operand is what distinguishes

this **map loading and building form** of the DMAP command **from the cloning form.**

.dsectname [a form of the 1st positional operand]
.csectname [a form of the 1st positional operand]
This is the name of the dsect map **or csect map** that is to be loaded (as a dsect). This **must** be the name that appears on the DSECT statement or CSECT statement at assembly time.

When loading a dsect map of a standard system control block, sometimes the name that must be given for **dsectname** is not the same as the name by which the control block is commonly known. For example, in XDCMAPS the name of the dsect that maps the TCB is **TCBFIX** (because the map has been generated to include the control block's prefix section).

(To help ease this inconvenience, for the **specific case** of loading maps from the **XDCMAPS** load module (or the XDCMAPSx ADATA side files), the DMAP command supports an automatic **nickname** capability that allows you to give the control block's common name instead of its dsect name. For detailed information about this, please see **HELP MAPS XDCMAPS.)**

In place of the dsectname operand, you may instead give a **csectname.** When you do this, Z/XDC reads the csect as if it were a dsect. This allows you to use the **USING** command to assign the map to any location you want. See **HELP COMMANDS DMAP LOAD CSECTSANDBINDERMAPS** for more information.

llorpathname [the second positional operand]
Notice: This positional operand is **deprecated.** Its functionality has been completely replaced by the **ESDDATA** and **ESDDATA=** operands. Those operands are documented further below.

In brief, use of the positional loadlibrary operand does a couple of things:
- It coerces the DMAP command to disregard ADATA when searching for mapping data.
- The command will consider only SYM data and ESD data for building maps.

But then, the ESDDATA= and SYMDATA= operands also do these things.

ADATA
ADATALIB=dsorlibname
ADATAPATH=pathname
ADATAFILE=pathname
These operands coerce the **DMAP** command towards building dsect maps **from ADATA.** (SYM data is not considered.)

If the dataset or library **is** specified, then it may be any of the sequential files or libraries listed above under the heading: **Source Level Maps. (a USS pathname may also be specified. In this case the ADATA in the program object is used).**

If a dataset or library **is not** specified, then the currently active **MAPLIBS list** is searched. See **HELP MAPS ADATA MAPLIBS** for more information.

When a dataset or library is specified, it is automatically **added** to the currently active MAPLIBS list. If it is already in the list, then it is moved to the top of

that list's search order.

The **ADATA** and **ADATA=** operands have no effect upon the loading of Binder maps.

When **both** the ADATA= and SYMDATA= operands are given, then:
- The ADATA= library will be searched first.
- It the dsect map is not found in the ADATA= library, then the SYMDATA= load library is searched.

SYMDATA
SYMDATA=dsorlibname
SYMDATALIB=dsorlibname
SYMDATAPATH=pathname
SYMDATAFILE=pathname
ESDDATA
ESDDATA=dsorlibname
ESDDATALIB=dsorlibname
ESDDATAPATH=pathname
ESDDATAFILE=pathname
These operands all coerce the **DMAP** command towards building maps from **SYM** data or **ESD** data. (ADATA is not considered.)
Note that you may use **ESD...** operand names - they are aliases of the **SYM...** Operands.

The operands **all** have the following common behaviors:
- They all coerce searching for **both** SYM data and ESD data (as per the requested map type indicated by the command's 1st positional operand [not per the keyword's name]).
- The SYM... and ESD... operands can be used interchangeably.
- They all prevent the consideration of ADATA.
- They all cause the first positional operand to be considered to be naming (or to imply the name of) a load module (not the member name of an ADATA library) or USS path.

The differences between the xxxDATA and xxxDATA= operands are these:
- **xxxDATA=** explicitly name a load library or USS path to be searched instead of the aspace's normal search order.
- **xxxDATA** does not override DMAP's normal load library search order.

For the SYMDATAx= operands:
- The given **dsorlibname** or **pathname** must be either a fully qualified, UNquoted dataset name or a quoted USS path name.
- The name may contain **variable symbols** for which Z/XDC will make substitutions:
  - This allows for the dsname or pathname to be reactive to the current time, date, userid, jobname, etc.
  - These variables are most useful when the DMAP command is executed from a **command script** or a **DEAD trap.**
  - For more information, see:
    - **HELP COMMANDS SYNTAX DSNAMES**
    - **HELP COMMANDS READ**
    - **HELP BREAKPOINTS DEADTRAPS**

If a load library name is provided, the dataset must be a PDS or PDSE containg load modules, and catalogued.

If a path name is provided, it must be the path of a program object. The name does not have to be given more than once; Z/XDC will remember it for future references to the same load module.

When **both** the ADATA= and SYMDATA= operands are given, then:
- The ADATA= library will be searched first.
- If the dsect map is not found in the ADATA= library, then the SYMDATA= load library or path (program object) is searched.

RENAME=newname
This operand allows you to assign to the map being loaded a name other than its original name. This can be particularly useful for a variety of reasons. For instance, if there are multiple instances of a control block, you may want to chose a name that is relevant to the particular instance in which you are interested. For example, if you wish to load a TCB map to represent your address space's jobstep TCB, then you might want to issue: **DMAP XDCMAPS.TCB RENAME=JSTEPTCB**

Newname may be any name that is valid to the Assembler. It may contain underscores **(_),** and it may be up to **64** characters long.

ZEROPOINT=offset
Sometimes, a control block has what is known as a **prefix section.** This is a group of fields that come prior to what is normally considered to be the control block's **zero point,** ie, that field to which other cblocks point.

A classic example of this is the System's Communications Vector Table **(CVT).** The CVT's zero point is the field named **CVTTCBP.** That's because CVTTCBP is the field that is pointed to by the CVT anchor field (storage location X'00000010'). The CVT contains several field that precede CVTTCBP (such as CVTPRODN, CVTVERID, etc.). Those fields are part of the CVT's **Prefix Section.**

The problem is, the Assembler knows nothing of all this. It knows **nothing** about "Prefix Sections". All it knows is what the **CVT macro** tells it, and that is:

- There is a dsect named **CVTFIX.**
- This dsect contains a bunch of fields, one of which is named **CVTTCBP.**
- There are several fields that come prior to CVTTCBP, the first of which is **unnamed** and is **X'D8'** bytes long.
- Altogether, the named and unnamed fields preceding **CVTTCBP** come to **X'100'** bytes (256 decimal) in length.
- Therefore, the "location" (offset actually) that the Assembler assigns to **CVTTCBP** is X'00000100', not zero, as one might prefer.

The **ZEROPOINT=** operand can be used to cope with this problem. For example, in above scenario, you would issue the command: **DMAP XDCMAPS.CVTFIX ZEROPOINT=.CVTTCBP**

ZEROPOINT=offset

Offset must be any "location" that is relative to any field in the map. **Offset** is constructed as follows:

ZEROPOINT={[.fieldname][+or-][numericoffset])
All elements are optional; however, one element (other than just a + or -) must be given.

.fieldname
This must be the name of any field in the map. If omitted, then the start of the map is used by default.

+ or **-**
If both a .fieldname and a numericoffset are given, then a **+** or **-** must be present to separate the two. If only a numericoffset is given (and a .fieldname is not given), then a **+** or **-** is optional, with **+** being the default effect in their absence.

numericoffset
This offset is applied either against the .fieldname (if given) or against the start of the map otherwise. **Numericoffset** can be either **hex** or **decimal** as follows:
- For **hex,** just provide hex digits.
- For **decimal,** provide decimal digits followed directly by the letter **N.**

Note, there is no requirement that the offset resolve to any location within the map. As long as it makes sense to do so, you could choose an offset that resolves to **any distance** prior to or following the map.

ZEROPOINT= Examples: The following all do the identical thing: They load a dsect map for the CVT and set its zero point to the **CVTTCBP** field:

```
DMAP XDCMAPS.CVTFIX Z=.CVTTCBP
DMAP XDCMAPS.CVTFIX Z=100
DMAP XDCMAPS.CVTFIX Z=+100
DMAP XDCMAPS.CVTFIX Z=256N
DMAP XDCMAPS.CVTFIX Z=.CVTLINK-8
DMAP XDCMAPS.CVTFIX Z=.CVTPROD+28
DMAP XDCMAPS.CVTFIX Z=.CVTPROD+40N
```

DMAP XDCMAPS.CVT [Note, this works because of the **automatic nicknaming** support described both earlier in this topic and in **HELP MAPS XDCMAPS.]**

ALLMESSAGES
When searching for map data, the **DMAP** command may try and fail in a lot of locations before finding the data it is looking for. Normally, when a map is loaded successfully, the error messages generated by one or more unsuccessful look-see's are discarded. This **ALLMESSAGES** operand causes those transitory error messages to be displayed instead of being discarded.

NOALLMESSAGES
This is the default action: All transitory error messages are discarded unless

the **DMAP** command was unable to find the map data anywhere.

USELDS [use local dataset]
this operand can also be specified using one of the following aliases:
USELFILE
USELPATH
When this operand is given, the possible existence of **MMEFDSN** tokens and **DSECTLIB** tokens is ignored.

- It prevents I/O intercepts and redirections caused by the **MMEFDSN** tokens.
- It prevents search order overrides caused by the the **DSECTLIB** tokens.

In other words, the DMAP command's mapping requests will be allowed to be go against local libraries, not third party libraries matched to the dump being examined.

This operand is effective only when using dump/XDC to debug a dump.

NOUSELDS
this operand can also be specified using one of the following aliases:
NOUSELFILE
NOUSELPATH
This operand is the default action. When dump/XDC is being used to debug a dump, and when **MMEFDSN** tokens and **DSECTLIB** tokens are present:

- All DMAP command I/O against load libraries is intercepted and redirected against MMEF files defined via the **DEFINE MMEFDSN** command.
- DSECTLIB libraries are searched for SYM data and ADATA instead of MAPLIB libraries and load libraries.

For further information, see above under the heading: **dump/XDC Considerations.**

USS=option
This operand is parsed and removed from the command before other operands are processed, regardless of its position in the command. It (or its default) affects the meaning of some of the operands on the command.

For a description of the possible values, their meaning, and the interaction with the USS profile facility please see **HELP DEBUGGING USS OPERANDS USS=**

DUB=option
This operand is parsed and removed from the command before other operands are processed, regardless of its position in the command. It (or its default) controls the processing if Z/XDC determines that dubbing is required.

For a description of the possible values, their meaning, and the interaction with the DUB profile facility please see **HELP DEBUGGING USS OPERANDS DUB=**

The Normal Search Order

When searching for mapping data, absent overriding operands (SYMDATA, ADATA, etc.) and absent dump/XDC related override tokens (MMEFDSNs and DSECTLIBs), the DMAP command uses its normal search order as follows:

First, the datasets present in the currently active **MAPLIBS list** are searched for ADATA in the order displayed by the **LIST MAPLIBS** command. Exceptions:

- If an operand is given that coerces searching load modules for mapping data (SYM data or ESD data), then the MAPLIBS search for ADATA is bypassed. The operands that coerce load library searches are:
  - The **loadlibraryname** positional operand
  - The **SYMDATA** and **SYMDATA=** operands
  - The **ESDDATA** and **ESDDATA=** operands

- If the **ADATA=** operand is given (see below), then:
  - The MAPLIBS search is bypassed in favor of searching the specified ADATA library.
  - But under the covers, the given library name is added to the current MAPLIBS list so that ADATA= does not have to be given again.

For details concerning the MAPLIBS search, see **HELP MAPS ADATA SEARCHORDER.**

For those MAPLIB datasets that are partitioned, the library is searched **twice:**
- The member whose name matches the given **dsectname** or **csectname** is searched first.
- If that member does not exist, or if the desired map is not found within the member, then the member whose name matches the given **membername** is searched.

If either dsectname or csectname or membername is not syntactically valid as a PDS member name, then that search is bypassed.

If suitable ADATA is not found by the MAPLIBS search, then the given membername is treated as being a load module or program object name, and the module or object is searched for as follows:

- If:
  - A loadlibraryname positional operand, or
  - A SYMDATA= operand, or
  - An ESDDATA= operand

  was given, then that library is searched for the given name. Otherwise:

- The load module or program object is searched for in system load libraries according to the address space’s load module search order. The search stops with the first instance is found.

  The load module search order is as follows:
  - The current task’s task library (if any).
  - The task libraries for all ancestor tasks in the order of newest to oldest.
  - The address space’s STEPLIB or JOBLIB libraries.
  - The System’s PLPA libraries.
  - The System’s linklist libraries.

Once the load module or program object is found, it is OPEN’d.
- If it is a program object, it is scanned for ADATA.
- If it is a load module, or if no ADATA is found, then it is scanned, for SYM data.
- If no SYM data is found, then the command fails.

Regarding MMEFDSN Tokens

If any **MMEFDSN** tokens exist, and if the DMAP command eventually OPENs and reads a load library to obtain the necessary mapping data (be it SYM data, ESD data or ADATA), then the I/O against the library will be intercepted and redirected against the corresponding library extraction data found in an MMEF file.

This interception is prevented when the **USELDS** operand is used.

MMEFDSN tokens are created by the **DEFINE MMEFDSN** command. For more information, see:

- **HELP COMMANDS DEFINE MMEFDSN**
- **HELP DUMPS MAPPINGDATA MMEFFILES**

Regarding DSECTLIB Tokens

If any **DSECTLIB** tokens exist, the listed libraries are searched for the given member name, and the DMAP command's normal mapping data search process is bypassed.

Use of DSECTLIB tokens is prevented when the **USELDS** operand is used.

DSECTLIB tokens are created by the **DEFINE DSECTLIB** command. For more information, see:

- **HELP COMMANDS DEFINE DSECTLIB**
- **HELP DUMPS MAPPINGDATA**

The use of DSECTLIB tokens is prevented when any of the following are true:

- The **USELDS** operand is used.
- The **loadlibraryname** positional operand is used.
- The **SYMDATA=** or **ESDDATA=** operand is used.
- The **ADATA=** operand is used.

Additional Information

For information about using the DMAP command to load csect maps and Binder maps as if they were dsects, type **HELP COMMANDS DMAP LOAD CSECTSANDBINDERMAPS.**

For information about using the DMAP command to copy ("clone") existing dsect maps, type **HELP COMMANDS DMAP CLONE.**

Examples:

For these examples, please assume the following:

- The DMAP command has not yet been used in the current debugging session.
- My TSO userid is "MYUID".
- "MYLOAD" has been loaded into storage from "MYUID.MYLIB.LOAD".
- "MYUID.MYLIB.LOAD" is currently one of the libraries forming this address space's "search order" (as, perhaps, a JOBLIB, STEPLIB, or task library).
- "XDCMAPS" is a load module residing in Z/XDC'S XDCLINK library. (The factory default name is hlq.XDCV31.XDCLINK. Ask your Systems Programmer for the

library's actual name at your Data Center.)
- "XDCMAPS" has **not** been loaded into storage.
- The asm/XDC Feature has been Licensed for use.

SET QUALIFIER MYLOAD.MYCSECT
DM .RBPRFX
The SET QUALIFIER commands establishes MYLOAD as the default load module name (for address expressions) and MYCSECT as the default csect name. Also because no DMAP command has yet been issued in this debugging session, MYLOAD is also established as the default load module for DMAP commands.

The DM .RBPRFX command loads the dsect map for Request Blocks from the "MYLOAD" member of the MYUID.MYLIB.LOAD dataset. The map is built either from ADATA or SYM data, depending upon which is present in the "MYLOAD" program object or load module. The map is read from "MYLOAD" because no prior DMAP commands have been issued during this debugging session and "MYLOAD" is the default csect established via the "SET QUALIFIER" command. "MYLOAD" is found in the "MYUID.MYLIB.LOAD" dataset because that dataset is part of the Home Address Space's search order.

DM XDCMAPS.RB
Because of a limited nickname support in the DMAP command, this command will succeed at:
- Loading the RBPRFX dsect map.
- Renaming it to RB.
- Assigning the map's zero point to the the RBBASIC field.

For more information, see **HELP MAPS XDCMAPS.**

DM XDCMAPS.ASCB hlq.XDCV31.XDCLINK
This loads the dsect map for ASCBs from the "XDCMAPS" load module located in the "hlq.XDCV31.XDCLINK" library. When a library's dataset name is given on the DMAP command, the address space's search order is not checked. Also because the library name was given, the search of MAPLIB libraries was bypassed as well.

DM .PSCB
This loads the dsect map for PSCBs from the "XDCMAPS" load module located in the "hlq.XDCV31.XDCLINK" library. The map was read from "XDCMAPS" because that is the load module that was used on the prior DMAP command. "XDCMAPS" was found in "hlq.XDCV31.XDCLINK" because that is where it was found by a previous DMAP command and no dataset name was given on this DMAP command to override that. Also note that Z/XDC does **not** check the search order in this example because it remembers where the previous DMAP command found "XDCMAPS".

SET MAPLIBS hlq.XDCV31.XDCADATA SAVE
PROFILE SAVE
DM XDCMAPSP.PSA
USING PSA 0

The SET MAPLIBS command establishes the hlq.XDCV31.XDCADATA library as a source for ADATA information.

The PROFILE SAVE command (in combination with the SAVE operand on the SET MAPLIBS command) hardens that setting, making it automatically available to all future debugging sessions for the current user.

The DM XDCMAPSP.PSA command loads a source level map of the PSA from ADATA found in the XDCMAPSP member of the hlq.XDCV31.XDCADATA library.

The USING PSA 0 command assigns the PSA map to represent low storage.

DM MYDOC.MYCBLOCK *U.MYLIB.LOAD
This loads a dsect map named MYCBLOCK. The map was read from the MYDOC load module. That module was found in the MYUID.MYLIB.LOAD load library. (*U was replaced by Z/XDC by the TSO session's userid string. For more information, see **HELP COMMANDS SYNTAX DSNAMES.)**

DM IGC0001I.
USING IGC0001I R1?
This reads the Binder map of the load module named IGC0001I and "converts" it into a dsect map. The USING command then assigns the IGC0001I map to the location pointed to by R1.

SET MAPLIBS DBCOLE.ADATALIB
DM IDEAMON.XYZNAME
This is an example involving mixed-case csect names. For this example, suppose that the IDEAMON program includes a csect named **XYZname.** Now, even though the user may have defined that name as a mixed-case name, when the Assembler passes that name to the Binder, it actually pass an upcased copy of the name: "XYZNAME".

If, however, you want the Assembler to pass the name to the Binder with its original mixed casing, you have to include in your program an ALIAS statement. Examples:

```
    xyzname ALIAS C'XYZname'
    XYZname ALIAS C'XYZname'
    XYZNAME ALIAS C'XYZname'
```

These examples all yield the identical result: The csect that the Assembler knows as **XYZNAME** will be known to the Binder as **XYZname.** (Note, the case of the ALIAS command's left-side operand does not matter, but the case of its right-side operand matters a lot!)

Then as long as there are no other Binder-time variations of the **xyzname** name, then references to the name need not match its case. So **dm ideamon.xyzname** would work just fine.

However, suppose that the Assembler program contained the following two statements:

```
    EP1 ALIAS C'XyZnAmE'
    EP2 ALIAS C'xyzNAME'
```

This creates two Binder-time names that differ only by case. Then:

- **dm ideamon.xyzname** would fail.
- **dm ideamon.XyZnAmE** would reference **EP1.**

- **dm ideamon.xyzNAME** would reference **EP2.**

DM IDEAMON.Weird_Long_Name
The Assembler's ALIAS statement can also be used to change a csect's name to strings that are longer than the normal eight characters. The ALIAS statement used to create this name might have been: **NORMALNM ALIAS C'Weird_Long_Name'**

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

CSECTSANDBINDERMAPS - For information about using the DMAP command to load csect maps and Binder maps as if they were dsects.

Help COmmands DMap Load Csectsandbindermaps

Sometimes it may be useful to load a **Csect Map** and even a **Module Map** as if it were a Dsect Map; that is, it may be useful to be able to make clones of a Csect Map or a Module Map and to be able to move such maps around in storage (via the USING command) in the same way a Dsect Map can be moved around.

Normally, Csect Maps and Module Maps are loaded via the MAP command; however, Z/XDC supports a syntax whereby you can load these maps via the DMAP command as well:

- For Csect Maps, simply specify the name of the desired csect instead of a dsect. Example: **DMAP MYPGM.MYCSECT**

- For Module Maps, just give the load module name followed by a dot (".") and **not** followed by any csect or dsect name. Example: **DMAP MYPGM.**

In both cases, a dataset name may be given as a second operand, if necessary. Example: **DMAP MPGM. libraryname**

For detailed information about the loading and management of Module Maps and Csect Maps (when loaded as Dsect Maps), and about the **reasons why** you might want to do this, see HELP MAPS CSECTSASDSECTS.

When a USING command is used to assign a DMAP-loaded Module Map to represent a **Privately Loaded** load module, Z/XDC is suddenly made aware of that module's existence and location. Thereafter, information about that module shows up in all the same displays as does any other module in the System. Examples:

- The module shows up in **LIST PGMS** displays,
- It shows up in the header lines of **FORMAT** and **WHERE** commands,
- It can displayed by the **LIST LKEDMAP** command,
- etc.

For more information, see HELP MAPS PRIVATELYLOADED.

Help COmmands DMap Clone

The **cloning form** of the **DMAP** command creates an independent copy of an already loaded dsect map. This copy must have a different name, and it can have a different "zero point".

A map's "zero point" is that relative location that is referenced when the map's name is used without being qualified by a specific field name. Consider, for example, TCBFIX, the dsect map for TCBs. It maps both the TCB proper **and** the TCB's prefix which is X'20' bytes long. Consequently, the label TCBRBP, which is normally thought of as having an offset of +0, REALLY has an offset of +X'20' relative to the map's name (TCBFIX). The cloning form of the DMAP command can be used to redefine the zero point for new copies of the TCBFIX map, moving it from the start of the map to the TCBRBP field where it belongs.

A Command Dialog is available for assisting with constructing and issuing DMAP commands. Just issue the command either with no operands or with a lone question mark as its sole operand.

Syntax:

?
DMAP
DMAP ?

[Invokes a Command Dialog]

DMAP newmap oldmap.labelname+offset

... **[USS=option]**

... **[DUB=option]**

newmap
This gives the name of the new map being created. This name may be up to 64 characters long.

The new map name may be quoted.

Note, the absence of a dot (.) from this first operand distinguishes this form of the DMAP command from the "load" form.

oldmap.labelname+offset
This is a reference expression that is relative to the zero point of an existing dsect map. The reference MUST contain either **oldmap** or **labelname** or both.

oldmap
This gives the name of an existing map to be copied. If given without

either **labelname** or **offset,** then this map's zero point value will be copied to the new map. If given with **labelname** and/or **offset,** then the new map's zero point will be computed based on them.

The old map name may be quoted.

.labelname
This must be the name of a field label found within a dsect. If **oldmap** is given, then **labelname** must be found within that dsect. If **oldmap** is omitted, then all defined dsects (both active and inactive) are searched until one containing the given label is found. This dsect then becomes the one that is copied. The dsects are searched in the same order in which they are displayed by the "LIST MAPS" command. The new map's zero point is set to the value of this label's relative address (+ or - an offset, if given).

+offset or **-offset**
This is a hexadecimal value that adjusts the zero point value to be assigned to the new map. If **labelname** is given, then **offset** is added to (or subtracted from) the label's relative address. If **labelname** is omitted, then **offset** is applied against the old map's zero point. The resulting value becomes the new map's zero point.

USS=option
This operand is parsed and removed from the command before other operands are processed, regardless of its position in the command. It (or its default) affects the meaning of some of the operands on the command.

For a description of the possible values, their meaning, and the interaction with the USS profile facility please see **HELP DEBUGGING USS OPERANDS USS=**

DUB=option
This operand is parsed and removed from the command before other operands are processed, regardless of its position in the command. It (or its default) controls the processing if Z/XDC determines that dubbing is required.

For a description of the possible values, their meaning, and the interaction with the DUB profile facility please see **HELP DEBUGGING USS OPERANDS DUB=**

Note that, this operand, while supported, is not used.

Examples:

Suppose various maps, including TCBFIX, have already been loaded.

DM TCBCRNT .TCBRBP
This searches all maps for the label TCBRBP. The first map found to contain this label is TCBFIX. TCBCRNT becomes a copy of TCBFIX but with its zero point set to

TCBRBP.

DM TCBCRNT TCBFIX+20
DM TCBJSTP TCBCRNT
The first of these commands uses a different process to accomplish essentially the same thing shown in the preceding example. The second of these commands creates another TCB map named TCBJSTP. It is assigned the same zero point that TCBCRNT has (which is the label TCBRBP).

Help COmmands DOwn

The **DOWN** command causes a display window to be scrolled downwards (usually). Subsequent messages are brought into view. The window that is scrolled is always the one that contains the cursor at the time the command is issued.

DOWN Commands and PF keys

F8's factory default value is **DOWN -.** The factory default setting for F20 is **DOWN M.**

In the definition of F8, the dash is important: It permits you to optionally place an operand on a window's command line and have that operand merged with the PF key's definition. For example, If you placed a **6** on the command line and then pressed **F8,** then a **DOWN 6** command would occur.

If the dash were not included in the PF key's definition, then any potential operand placed on the command line would be ignored. For more information about this, see **HELP FULLSCREEN PFKEYS.**

DOWN Commands and the Primary Window

The Primary Window has associated with it a Scroll Area that keeps a history of your debugging session. This typically is up to **10,000** (or so) lines long. The **UP DOWN LOCATE** and **SCANLOG** commands can be used to navigate freely through this recent history. For more information, see **HELP FULLSCREEN WINDOWS PRIMARYWINDOW.**

As you scroll around through the history, you will quickly see that every command string that you have issued from the command line has been given a sequence number and has been saved into the Scroll Area. Following each command string, you will see all of the messages generated by that string's commands.

You may also see command strings that you did not type in. Usually, these are commands that Z/XDC has generated internally in order to effect some Point-and-Shoot command (or other fullscreen action) that you have issued. **ProTip:** You should take the time to examine and understand these generated commands. They sometimes can be quite instructive...

Unlike Watch Windows (see below), the concept of display stability does not apply to the Primary Window. Every time a command string that generates messages is executed,

the window is always repositioned to show the first of the newly generated messages at the top of its display area.

Also, the command line is erased in preparation for you to type in your next commands.

Dual Use - Scrolling Thru Storage

For the Primary Window, when its Scroll Area is already positioned at its bottom, and storage is being displayed, then the **DOWN** command will cause additional storage to be added to the Scroll Area and then displayed. This creates a **scrolling through storage** effect.

More precisely:

- **If,** when a DOWN command is issued, the window was already position at the bottom of its Scroll Area,
- **And if** a display of storage is the last thing in the Scroll Area,

- **Then** the DOWN command will cause a FORMAT or DISPLAY command to be issued such that the display of additional storage will be appended to the Scroll Area,
- **And then** the Scroll Area will be positioned to display that additional storage.

DOWN Commands and Watch Windows

Unlike the Primary Window, Watch Windows do not keep histories of their activities. The only messages available for scrolling are those that have been generated by the most recent issuances of their command strings. So there is no opportunity to use scrolling commands to peruse the window’s recent history. You can scroll only within the current instance of the display generated by the window’s current command string.

For storage displaying commands **(FORMAT, [E]WHERE** and **DISPLAY),** generally the size of the generated display will exactly fit the Watch Window, so most **DOWN** commands will have no apparent effect since there are no additional, out-of-view messages to scroll to. However, if you issue these commands **(FORMAT** and friends) with a **LINES=** operand, you can force the command to generate a potentially large number of additional lines that **DOWN** will be quite happy to show you.

Another difference between Watch Windows and the Primary Window is that the command line’s contents are retained so that its displaying-type commands can be re-executed each time you press the ENTER key (or a PF key). This means that the general form of the display remains reasonably stable while the data itself can change in near real time.

If you use scrolling commands to reposition downwards, this scroll position will be retained either until you issue more scrolling commands or until you change the content of the command line.

Now, depending upon the particular displaying-type commands you’ve placed on the command lines, it is possible for the number of generated messages to change from one ENTER press to the next... But for most DOWN command operands (see below), the position that will be maintained in the display will be relative to the top of the

window's Scroll Area.

But for **DOWN MAX-nnn** type scrolling, positioning will be maintained relative to the **bottom** of the scroll area.

Syntax:

```
DOWN CURSOR+nnn-nnn...
     FULL  |   |
     PAGE  |   |
     DATA  |   |
     HALF  |   |
     MAX   |   |
     CMD   |   |
     TRACE |   |
     nnn   |   |
     #nnn  |   |
     omitted V   V
```

Notes:

- Every operand **(including the omitted operand)** can optionally be appended by one or more plus or minus adjustments that adjust the display upwards or downwards by some number of lines. Examples:
  - DOWN 10+5 [Works like DOWN 15]
  - DOWN +15 [Ditto]
  - DOWN -15 [Works like UP 15]
  - DOWN MAX-5
  - DOWN MAX+3 [Try it in a Watch Window when the window is not large enough to show all of its messages]
  - DOWN HALF-12+3-34 [Works like DOWN HALF-43]
  - DOWN #10+3

  See below for more information.

- It is possible to form DOWN command operands that would cause positioning to change in an upwards direction (like an UP command). This occurs when operand resolution results in a negative scroll distance.

- Command operands that result in a positive scroll distance will scroll downwards. Negative distances result in upwards scrolling.

- Any scrolling that would result in positioning completely below the bottom of the Scroll Area will be converted into a simple **DOWN MAX** command.

- Similarly, any scrolling that would result in positioning above the top of the Scroll Area will be converted (effectively) into a simple **UP MAX** command.

DOWN **CURSOR**
If the cursor is within a window's display area, then that window is scrolled downwards so as to move the line containing the cursor to the top of the window. If the cursor is located on the window's command line, then **DOWN CURSOR** functions

like **DOWN FULL:** The window is scrolled downwards so as to bring the next full window's worth of information into view.

DOWN **FULL**
DOWN **PAGE**
The window containing the cursor is scrolled downwards so as to bring the next full window's worth of information into view.

DOWN **DATA**
The window containing the cursor is scrolled downwards so as to move the previously bottom display line to the top of the window.

DOWN **HALF**
The window containing the cursor is scrolled downwards half a window's worth.

DOWN **MAX**
The window containing the cursor is scrolled downwards to the bottom of its scroll area. The last message currently saved in the scroll area is positioned at the last line of the window's display area.

For Watch Windows, most DOWN command operands establish a scroll position that is relative to the top of the Scroll Area. DOWN **MAX,** on the other hand, establishes a scroll position that is relative to the **bottom** of the Scroll Area.

Further, when the MAX operand is followed by **+nnn** or **-nnn,** that creates an adjusted positioning that is relative to the bottom. This matters for Watch Windows because they contain persistent displays whose sizes may change from one instance of the displays to the next. (See **HELP FULLSCREEN WINDOWS WATCHWINDOWS** for more information.) Examples:

- DOWN **MAX-10**
  Once the window is scrolled to the bottom of the Scroll Area, it is then bounced back upwards by 10 lines. For Watch Windows, the display will remain at this **MAX-10** position even as the number of generated display lines might change over time.

- DOWN **MAX+7**
  In this case, after scrolling the window down to the bottom of the Scroll Area, it is then scrolled down further, thus creating seven lines of whitespace between the last displayed line and the bottom of the window's display area. Note that **+nnn** needs to be a smaller value than the size of the window's display area.

DOWN **CMD**
The window containing the cursor is scrolled downwards so as to bring into view the next logged command string and its generated display. The command string is shown at the top of the window's display area.

DOWN **TRACE**
This command is valid only only when issued within the Primary Window. It functions like DOWN MAX when issued from a Watch Window.

This command presumes that you have already scrolled upwards in the Scroll Area to some point in time prior to **GO** or **TRACE** commands (i.e. prior to returns to the program being debugged.)

This command scrolls the Primary Window downwards to the first display generated by the next time that Z/XDC received control from the user program. Usually, this will be a display generated by an internally issued **WHERE** command.

Usually, Z/XDC receives control as a result either of a Hook, a Trap or a Trace action. Usually, those commands will cause a **WHERE** command to be issued in order to display the logic at the location where user program execution reached the trap or hook.

Well if, for example, you have used a **T BY** command repeatedly to step through your code, and you've issued an UP MAX (for example) to position to the top of the Scroll Area, then you can repeatedly use a **DOWN T;RETRIEVE** command to replay execution forwards just is if you were watching a movie!

Even easier, if you have the default PF keys in effect, then you can run your movie just by repeatedly typing **T** on the command line and pressing **F8** (or the **Page Down** key, depending upon your keyboard setup).

In practice, you'd probably intermix UP T and DOWN T commands to run the "movie" forwards and back.

DOWN **nnn** [Example: DOWN 10]
The window containing the cursor is scrolled downwards the specified number of lines. This number may range from 0 to 32,767.

DOWN **#nnn** [Example: DOWN #10]
As you scroll around through the Primary Window's scroll area, you will notice that the recorded copies of the Z/XDC commands that you have issued have been assigned sequence numbers. This form of the DOWN command scrolls the Primary window **either** upwards or downwards so as to bring the desired command string into view at the top of the window's display area.

Before you ask, the following three command forms all function **identically:**

- UP #nnn
- DOWN #nnn
- LOCATE #nnn

This form of the DOWN command is valid only for the debugging session's Primary Window. It cannot be issued from any Watch Window.

omitted [Example: DOWN]

The window containing the cursor is scrolled downwards by that window's default scroll amount. This default can be set by the SET WINDOW VERTICAL command:

- It can be set to CURSOR, FULL, DATA, or HALF.
- It **cannot** be set to MAX, CMD, or #nnn.

For more information, See **HELP COMMANDS SET WINDOW VERTICAL.**

+nnn
-nnn
Every one of the above described operands can be modified by appending one **or more** plus or minus adjustments to it. They work by adjusting the scroll position determined so far either downwards (for +nnn) or upwards (for -nnn) the specified distance.

Examples of this usage can be found above.

It certainly is questionable why one would want to use adjustments such as these, but the **DOWN MAX-nnn** adjustment is particularly useful because (as already discussed above) for Watch Windows this particular command form will establish display stability relative to the **bottom** of the Scrollable Area (whereas all other forms are relative to the top).

Scrolling Through Storage

Normally, when use of the **DOWN** command would cause the display to move beyond the bottom of the session log, the session log is repositioned such that the very last message is located at the very bottom of the window, thus filling the window with messages from there upwards. Also a note is displayed in the display screen's upper right-hand corner that says **BOTTOM-OF-DATA.**

However, if the last display in the session log was of storage; then instead, a new **DISPLAY** or **FORMAT** command (as appropriate) is issued to append a display of the next chunk of storage to the bottom of the log, thereby extending the log and bringing that next chunk of storage into view. This creates a "scrolling through storage" effect, and a note is displayed in the display screen's upper right-hand corner that says **MORE STORAGE APPENDED.**

A similar change has **not** been made to the **UP** command, but if **UP** commands are issued after a series of scroll extending DOWN commands, then a limited scrolling upwards through storage effect is achieved.

Help COmmands DRop

The **DROP** command releases the base addresses for all or selected dsect maps. In other words, the dsect maps are deactivated. The maps are **not** purged from storage. (Use the DELETE MAPS command if that's what you want to do.)

Unlike the USING command, the DROP command does **not** affect the dsect map search order.

Syntax:

```
DROP mapnames
     addressexpressions
     omitted
```

mapnames
These must be the exact names of one or more dsects to be dropped.

addressexpressions
These must be address expressions that resolve to locations that are within areas that are mapped by one or more dsects. The bases addresses for all such dsects will be dropped.

omitted
When no operands are given, the base addresses for all dsects are dropped.

The DROP command may be given with as many dsect names and/or address expressions as you like.

All errors that are encountered are reported; however, for most errors (except syntax errors), Z/XDC will continue processing with the DMAP command’s next operand.

Examples:

DR TCBFIX
The dsect map named TCBFIX is deactivated. Clones of this map, if any, are not affected.

DR TCBFIX+0 PSA RB#3
In this example "TCBFIX+0" is not a pure name of a dsect map, so instead Z/XDC parses it as an address expression. Then Z/XDC deactivates **all** dsect maps that contain the resulting address.

In addition, "PSA" is the pure name of a dsect map, so if that map exists and has a base address assigned to it, then it is deactivated.

Finally, assuming "RB#3" is the name of an equate symbol (which it usually is in Z/XDC), after determining that no dsect map named "RB#3" exists, Z/XDC then attempts to interpret "RB#3" as an address expression. If successful, Z/XDC will drop all maps that contain the resulting address.

DR
Since no operands are given, all dsect maps are deactivated.

Help COmmands DUb

The **DUB** command can be used to **dub** Z/XDC. Dubbing is a z/OS term whereby the current task is made a USS process and assigned a process ID.

You cannot dub a formal USS process or a task that has already been dubbed.

A task is always dubbed as a process, never as a thread of an existing process.

Syntax:

DUB

This command has no operands.

Examples:

DUB
If the task being debugged is not a formal USS process and has not already been dubbed then it will be dubbed as a USS process and a process ID will be assigned to it.

Help COmmands ENd

The **END** command under dump/XDC, works differently. Please see below.
The **END** command is used to terminate Z/XDC processing under the current ESTAE (or ESTAI) and percolate the ABEND to the next older ESTAE/I. If there are no older ESTAE/Is, then the current task is terminated. On the other hand, if there are older ESTAE/Is, then the current task will be terminated or resumed according to what they do. If one (or more) of the older ESTAE/Is specifies Z/XDC, then Z/XDC will come up again if and when that ESTAE/I gains control. If this happens, then you may have to issue the END command again if you still want to terminate the debugging session.

If Z/XDC is executing as a TRAP handler, there is no ABEND to percolate and so execution of the current RB is terminated (by issuing an SVC 3 (EXIT)) (This may cause the task to terminate if there are no other RBs). If other ESTAEs (or ESTAIs) exist (that drive Z/XDC), Z/XDC may get control again.

If a **REXX** command has been used during the debugging session, and if a **rexx/XDC Interface** still exists, (and if the END command's **KEEP** operand is neither given nor implied), then REXX's IRXTERM routine is called to take down the instance of the rexx/XDC Interface that was created by the REXX command. (See HELP REXX for more information.)

If dump/XDC is executing, only the **ASK NOASK** and **COMPLETELY** operands have an effect. Other operands are validated but ignored. If there is any REXX executing, it is terminated.

Syntax:

END	**DUMP**	**KEEP**	**ASK**	**NOMSG**
	NODUMP	**PURGE**	**NOASK**	**omitted**
	IGNORE	**COMPLETELY**	**omitted**	
	omitted	**omitted**		

Operands may be given in any order.
Operands within columns are mutually exclusive.

DUMP

This causes Z/XDC to signal the System's RTM (Recovery/Termination Manager) to produce a dump. If neither a SYSABEND nor a SYSMDUMP nor a SYSUDUMP file exists, then a SYSUDUMP file is dynamically allocated to the System's SYSOUT queue; in which case, Z/XDC will use the SYSOUT class that has been established by the SET PRINT command. You can use the LIST PRINT command (or the PROFILE command) to see what that class is currently set to.

If Z/XDC is executing as a TRAP handler, this operand is ignored.

NODUMP (or omitted)
This causes Z/XDC to suppress the production of a dump.

IGNORE
This prevents Z/XDC from altering RTM's dump flag. A dump may or may not occur according to how the flag has been set by other processes and by whether or not a SYSABEND, SYSUDUMP, or SYSMDUMP file is currently allocated. (Z/XDC will not automatically allocate such a file when it executes an "END IGNORE" command.)

PURGE or **COMPLETELY**
PURGE and **COMPLETELY** are synonyms. They both cause Z/XDC to purge all currently defined breakpoints, maps, and equates before ending. Also, a flag is set signaling that the debugging session has ended. If Z/XDC should receive control again, it will immediately return to its caller (usually the RTM) to let ABEND processing continue as if Z/XDC had not been called.

Note that purging breakpoints involves restoring the original opcode to the instructions where the breakpoints had been located. This is an important thing to do for modules that might remain in storage after the debugging session has ended (such as reentrant modules, reusable modules and especially common storage modules).

KEEP (or omitted)
Z/XDC does not purge its breakpoints, maps, and equates before ending under the current ESTAE (or ESTAI). This may be desirable if you expect Z/XDC to gain control again under a different task or ESTAE/I.

ASK

Before processing the END command, Z/XDC will issue message DBC848Q to ask the user whether or not his issuing the END command was intentional. (It is too easy to press F3 by mistake.) The user must respond with either "Y" or "N" to indicate whether or not END command processing should proceed. For more information, see HELP MESSAGES DBC848.

NOASK
This indicates that the user really does mean to issue the END command and that he

does not want to be queried about it. Z/XDC goes ahead and percolates the current ESTAE/I.

Both **ASK** and **NOASK** omitted
If other operands are given on the END command, then Z/XDC proceeds as if **NOASK** has been specified. Otherwise, if the END command is given without operands, then Z/XDC proceeds as if **ASK** has been specified.

NOMSG
When Z/XDC starts up, it begins generating messages and sending them to Z/XDC's Fullscreen Support for display. The Fullscreen Support, however, does not display any messages until Z/XDC is ready to receive a command from the user's terminal. If for some reason Z/XDC ends without having requested a command from the terminal, then the Fullscreen Support is unable to display any pending messages that it may have accumulated. Normally when this happens, the Fullscreen Support sends those messages back to Z/XDC so that he can display them himself (just before terminating) via linemode TPUTs (or WTORs, as appropriate). If you do not want any pending messages to be displayed, then you can specify **END NOMSG** to suppress them.

Example: You may want to write a Z/XDC command script that performs some sort of zap (for example) and then terminates Z/XDC without further processing. If you don't care to see any messages from Z/XDC, then you can use an **END NOMSG** command as the command script's last command.

Examples:

END DUMP
Z/XDC terminates the current ESTAE/I without asking the user if he really means it. (NOASK is the default when other operands are given.) A dump is produced. (SYSUDUMP is allocated if necessary). Z/XDC's global data is not purged. The ABEND is percolated to the next older ESTAE/I, if any. If Z/XDC is executing as a TRAP handler, then (as mentioned above) the DUMP operand is ignored. If dump/XDC is executing, then (as mentioned above) all operands are ignored and Z/XDC terminates.

END
Z/XDC issues message DBC848Q to ask the user whether or not he really meant to issue the END command. If the user responds with "Y", then Z/XDC will terminate the current ESTAE/I and cause the ABEND to be percolated. No dump will be produced. Z/XDC's global data will not be purged. If Z/XDC is executing as a TRAP handler, there is no ABEND to percolate. If dump/XDC is executing, Z/XDC just terminates.

Help COmmands EQuate

The **EQUATE** command assigns a symbol to a given location in any chosen address space, dataspace or even real storage. Equates will show up in formatted displays of storage, and they can be used in address expressions when referencing storage.

Floating Equates

An equate can be assigned to either a fixed or floating location in any storage:
- For **fixed** locations, the given address expression is resolved at the time that the **EQUATE** command is issued, and that resolved value is used each time the equate is referenced.
- For **floating** locations, the given address expression is saved in an unresolved form so that it can be recomputed each time the equate is referenced. Consequently, the address may resolve to different locations at different times. (See **HELP EQUATES FLOATING** for more information.

Area Equates

Equates can represent **points** or **areas.** When you provide a length, they represent areas. Otherwise, they represent points.

Area Equates can be used to control address offsets displayed by the **FORMAT, DISPLAY** and other commands. When **SET FORMAT OFFSET** is in effect, Z/XDC will display locations, not as addresses, but as offsets within some significant object (load module, csect, dsect, and the like). But if you think Z/XDC has not made the best choice, and you would prefer that the offsets shown be relative to something else, you can use an Area Equate to **coerce** Z/XDC's choice.

An Equate's Scope

Equates that resolve to a location in common storage are automatically assigned the **COMMON** scope:
- When a formatted display includes the location of an equate, that equate will be displayed regardless of what the current address space is. (They will not be displayed for dataspaces and real storage.)
- When an address expression uses a **COMMON-scoped** equate, the expression's target aspace will not be changed by the equate.

Equates that resolve to locations in a private area (or in a dataspace or in real storage) are assigned the **PRIVATE** scope. They have no meaning in other spaces:
- When a formatted display includes the location of an equate, that equate will be displayed only when the address space, dataspace or real storage is the same as that assigned to the equate.
- When an address expression uses a **PRIVATE-scoped** equate, the expression's target aspace **will** be changed by the equate to the space (aspace, dataspace or real storage) to which the equate is assigned.

Equates that are assigned to the **GLOBAL** scope are like COMMON-scoped equates but on steroids. They can show up in displays regardless of what type of space is being displayed.

An equate's **PRIVATE/COMMON/GLOBAL** attribute can be overridden. See **Scope Setting Operands** (below) for more information.

Autocloning

Equates can be **autocloned.** An entire series of equates can be automatically generated to represent multiple entries in a table or on a chain. For tables (and of course chains), the entries can be either fixed length or variable.

Like other equates, the locations of autocloned equates can be either **fixed** or **floating.** However, care must be taken not to use autocloning to create "too many" floating equates! This is because the resolution of floating equates is extremely CPU intensive, so when too many floating equates and maps exist, Z/XDC's responsiveness will become unacceptably sluggish.

Autocloning is discussed in detail in the **AUTOCLONING** subtopic of the current topic.

Syntax:

```
    EQUATE  name  address  length  PRIVATE  FLOAT    INSTRUCTION ...
                                   COMMON   omitted  DATA
                                   GLOBAL            NOBIAS
                                   omitted           omitted

            autocloning
            operands        chain-related    list-related
        ... AUTOCLONE       CFOFFSET=offset  LFOFFSET=offset
            1STCLONE=nnn    CFWIDTH=width    LFWIDTH=width
            MAXCLONE=count  CFMASK=mask      TESIZE=size
                            ZCVALUE=address  ZTVALUE=address
                            ZTPTR=address
```

Rules:

- The **name** operand is required. All other operands are optional.
- The **name** operand must be given first.
- If an **address** operand is given, then it must be given second.
- All remaining operands, if given, may be given in any order.
- If the **address** operand is omitted, then only the **name** operand may be given. All other operands must be omitted as well.
- All **chain** related operands are **mutually exclusive** with **list** related operands.

- Autocloning occurs only when one or more of the autocloning operands are given. When none of them are given, then only a single equate is generated. The autocloning operands are described in **HELP COMMANDS EQUATE AUTOCLONING.**

Operands

name
This gives the name to be assigned to the equate. (For autocloning, it gives the root name to be used for the generated equates. Sequence numbers will be appended to

this name to form the names of the individual equates.) **Name** may consist of from 1 to 64 alphanumeric or national characters (@ # $). The underscore character (_) also may be used. The first character must be alphabetic or national.

Certain names (e.g., R5) are reserved and cannot be used.

If **name** has already been defined as an equate name, then the prior definition is automatically replaced.

If **name** matches a built-in equate name, then the built-in equate is superseded: Its definition will be restored when the superseding equate is deleted.

Name may be the same as a load module name. In this case, subsequent references to **name** will resolve to the equate. (To reference the load module, "name." must be used.)

Location and Length Operands

address
This is an address expression that gives the storage location to which the name is to be assigned. This location may any of the following:

- A common storage location,
- A private storage address located in any accessible address space,
- A location in any data space owned either by the host address space or any other accessible address space,
- A location in real storage,
- A location in a dump.

- For autocloning, the address expression provides the location of the first table or chain entry to be labeled.

The **address** operand is optional. When **not** given:

- The generated equate is assigned to the location pointed to by the **Current Display Pointer (CDP).**

- All other operands (except the **name** operand) must also be omitted!

- The generated equate will automatically be assigned a formatting bias **(INSTRUCTION** or **DATA** or **NOBIAS)** according to what the **CDP** currently points to. For more information about the Current Display Pointer, see **HELP ADDRESSING IMPLICIT.**

length
This operand is optional. If given, then it must be a number equal to or greater than zero.

Numbers can be given in any of the following ways:

- A string of hex digits (no quotes, no framing, just the digits)
- A string of decimal digits followed by the letter **N** (Example: Both **10** and **16N** mean sixteen.)
- A scaled decimal number (Examples: **4K, 12G** etc.).

The **length** operand provides the area's length, in bytes, to be associated with the name being defined. If omitted, then a length of zero is assigned.

When an Area Equate's length is **32 bytes or longer,** Z/XDC makes use of that length when it generates offset information in storage displays that fall within the range of the equate. When **SET FORMAT OFFSETS** is in effect, Area Equates can be used to coerce the offset display when you don't like Z/XDC's choice of an offset base.

Z/XDC's **FREEMAIN** command can free (among other things) areas of storage represented by Area Equates. See **HELP COMMANDS FREEMAIN** for details.

FLOAT
This causes the address expression to be saved in an unresolved form and recomputed whenever the equate is referenced. As a result, the equate can resolve to different locations at different times, depending upon the possibly changing values of pointers that the basing address expression might reference.

omitted
When **FLOAT** is omitted, the equate's defining address expression is computed immediately upon being parsed. Accordingly, the equate's location remains fixed to a specific address.

Warning! When creating a large number of equates through **autocloning,** care must be taken not to create **"too many"** floating equates! This is because the resolution of floating equates is extremely CPU intensive, so when too many floating equates and maps exist, Z/XDC's responsiveness will become unacceptably sluggish.

Scope Setting Operands

PRIVATE [alias: LOCAL]
This operand overrides Z/XDC's determination of the scope of the equate, forcing the equate to be treated as a private area equate even if it actually is assigned to a common area location.

Normally when common storage is displayed that has an equate assigned to it, that equate name will show up in the display regardless of the address space that is being displayed. However, when an equate in common storage has been assigned the **PRIVATE** attribute, it will show up in the display only when that display is of the address space to which the equate has been assigned.

COMMON [alias: SHARED]
This operand overrides Z/XDC's determination of the scope of the equate. It causes the equate to be treated as being owned by all address spaces even if it resolves to a private storage location. This means that when a FORMAT command displays the location to which the equate is assigned, the equate will be displayed regardless of what address space is being displayed.

(A COMMON-scoped equate will not be displayed when real storage or data space storage is being displayed.)

GLOBAL
A global equate is owned by **all** spaces: address spaces, data spaces, and real storage. A global equate that's been assigned to a particular address will be displayed in any space for which that address is being displayed. It will be displayed regardless of whether the space is an address space, a data space, or real storage.

omitted
When **GLOBAL, COMMON** and **PRIVATE** are all omitted, the default depends upon where the equate is assigned:
- If the equate resolves to a common storage location, then it is assigned the **COMMON** scope.
- Otherwise, it is assigned the **PRIVATE** scope.

Formatting Bias Operands

INSTRUCTION
DATA
NOBIAS
omitted

These operands are mutually exclusive. If given, then they influence the **formatting bias** of displays produced by the **FORMAT, WHERE,** and **SHOW** commands. If the these commands encounter an equate having the **INSTRUCTION** attribute, then they will attempt to switch to instruction formatting mode. On the other hand, if an encountered equate has the **DATA** attribute, then the storage formatter will switch to data formatting mode.

The **DATA** and **INSTRUCTION** operands are useful when Z/XDC's formatter misinterprets an area of storage. This can happen because machine instruction opcodes and other data are often indistinguishable.

Also, such misinterpretations are much more likely to happen in unmapped areas of storage than in mapped ones. However, mapped areas can sometimes still be misinterpreted. So, placing equates having the **D** or **I** attribute at appropriate locations in the misinterpreted area of storage will cause subsequent displays of that area to be formatted correctly.

Note, **HEXDATA** is an alias for **DATA;** however, it is a legacy operand that may be removed in a future release.

NOBIAS

This prevents a formatting bias from being assigned to an equate. Such equates, when encountered by the **FORMAT** command (and friends), do not change the formatting bias of the display.

omitted
When **INSTRUCTION, DATA** and **NOBIAS** are all omitted, the default varies as follows:

- For equates created through **autocloning,** the default is **DATA.**
- For regular equates, the default is **NOBIAS.**

The Autocloning Operands

For descriptions of the autocloning related operands, see **HELP COMMANDS EQUATE AUTOCLONING.**

Examples:

EQ CVT 10? D
This assigns an equate named CVT to the location of the System's Communications Vector Table. It is assigned the **DATA** attribute to ensure that storage formatting commands will not try to interpret any data at that location as being machine instructions.

EQ TCB CVT?+4? F D
EQ TCB 21C? F D
These commands both assign an equate named TCB to the location of the current Task Control Block. It is assigned the **FLOAT** attribute, **not** because the TCB might move, but because when debugging multitasking programs Z/XDC might gain control under more than one task. In this event using the FLOAT attribute causes the TCB equate to represent the current TCB no matter which task might be current at any given point in time.

EQ JSTCB TCB+7C? 130 D
This assigns an equate named **JSTCB** to the location of the current job's jobstep TCB. A length of **X'130'** is assigned to this equate. If the **FORMAT** command is then used to display within this jobstep TCB, then the **130** operand will cause the storage locations to be shown as offsets from the start of the JSTCB equate.

EQ NIA PSW? I F
This assigns an equate named **NIA** to the location of the user program machine instruction that will be executed next when you issue Z/XDC's **GO/GOT/GOX** or **TRACE** command. Z/XDC's **FORMAT** command will always try to format this location as a machine instruction.

The **NIA** label will float according to the location pointed to by the retry level PSW. Thus, as you use the **TRACE** command to let user program instructions execute one by one (or few by few), the **NIA** equate will dutifully follow along. (Note, similar floating equates can easily be setup to follow the execution path of any program running under any RB queued from any TCB located in any address space.)

L TASKS
L RB TCB#3

EQ HEREHEIS RB#1+14? F I

This series of commands creates an equate named **HEREHEIS** that floats according to the current execution location of the program running under the oldest RB (RB#1) of task #3. That program's current execution location can then be displayed just by issuing a **FORMAT HEREHEIS** command.

EQ X @CDP 1000 F

Recently, I had a customer who wanted all storage displays to show offsets relative to the start of the display. That seemed like a rather unique request, but after a bit of thought, I came up with this **EQUATE** command. It creates a floating equate named **X** that **floats** according to the location of a built-in equate named **@CDP,** which almost always labels the most recently displayed storage.

1000 is just a sufficiently large number (4K) so that the range of the equate will always be larger than the amount of storage to be displayed.

F causes the location of the equate to **float** such that for every new display, when **@CDP** changes, **X** will change as well.

Help COmmands EQuate Autocloning

Autocloning is a method by which you can automatically generate an entire series of equates or dsects maps to represent either each entry in a table or each control block on a chain. The equates or dsects that are created all have unique sequence numbers but share a common root name that you must provide.

This topic describes those operands of the **EQUATE** and **USING** commands that specifically relate to autocloning. The remaining operands are described in the parent topic **(HELP COMMANDS EQUATE** or **HELP COMMANDS USING).**

Chains

The **EQUATE** and **USING** commands have operands that let you define the following about chains:

- Where the chain field is.
- How wide the chain field is (3, 4, or 8 bytes).
- A mask for selecting a chain field's significant bits.
- The chain field value that signifies end-of-chain (zero, usually).
- The maximum number of chain entries to label or map.

Tables

Z/XDC's autocloning supports contiguous tables of either fixed length or variable length entries. When variable, the table entries will (presumedly) have a length field. This length field can be of any reasonable width, and it may define either the table entry's entire length or only the length of the entry's variable portion

(in which case, all entries in the table are presumed to consist partly of common fields of a fixed size). In support of all this, the **EQUATE** and **USING** commands have operands that let you define the following about tables:

- For fixed length table entries, or for variable table entries containing a fixed length root section, you can specify the length either of the entries or of the root section.
- For variable length table entries, you can specify where the length field is and how wide it is (1 through 8 bytes).
- You can specify either the length field value (if any) that identifies the end of the table or the address at which the table ends.
- Finally, you can specify the maximum number of table entries to label or map.

Whenever a series of equates or maps is created via autocloning, Z/XDC first checks to see whether or not a series of previously created equates or maps, having the same root name, already exists. If so, then that series is purged in its entirety prior to the creation of the current series.

The equates or dsects that are generated through autocloning can all have the same attributes and characteristics that any other equate/dsect might have. In particular, they can be assigned either fixed bases or **floating bases,** thus the equates/dsects can be created to represent either particular instances of or generic templates for table or chain entries.

Command Dialog

A Command Dialog is available for assisting with constructing and issuing USING commands (but not EQUATE commands yet). Just issue the command either with no operands or with a lone question mark as its sole operand.

Syntax:

?
USING
USING ?

[Invokes a Command Dialog]

EQUATE name address length PRIVATE FLOAT INSTRUCTION ...
USING COMMON omitted DATA
GLOBAL NOBIAS
omitted omitted

	autocloning operands	chain-related	table-related
...	**AUTOCLONE**	**CFOFFSET=offset**	**LFOFFSET=offset**
	1STCLONE=nnn	**CFWIDTH=width**	**LFWIDTH=width**
	MAXCLONE=count	**CFMASK=mask**	**TESIZE=size**
		ZCVALUE=address	**ZTVALUE=address**
		ZTPTR=address	

Notes:

- The name and address operands are required, the rest are all optional.
- The name and address expression must be given in the order shown. The optional operands can be given in any order.
- All **chain** related operands are **mutually exclusive** with **table** related operands.
- Autocloning occurs only when one or more of the autocloning operands are given. When none of them are given, only a single equate is generated.

Autocloning Operands Related to Both Chains and Tables

1STCLONE=number
This sets the start of the range of sequence number to be assigned to the generated equates. The default first sequence# is **1.**

The given value may range from **0** to **999999.**

The value set for this operand affects the maximum value that can be used for the **MAXCLONE** operand.

MAXCLONE=number
This sets a limit on the maximum number of equates or dsect maps to be created through autocloning. **Number** may range from 1 to **1000000,** but adjusted for whatever you might provide with the **1STCLONE=** operand. More precisely, the maximum allowed value for **MAXCLONE=** is **(1000000-1STCLONE).** Example: -
When **1STCLONE=1,** then **MAXCLONE=** may range from 1 up to **999999.**

Notes:
- The default limit is **100.**
- The number of significant digits in the highest permitted clone# determines the width of the sequence number that is appended to the equate's or dsect map's root name in order to create uniquely named equates/maps.
- Example, if **1STCLONE=10** and **MAXCLONE=99,** then the highest possible clone# will be 109, which is 3 digits wide. So all generated sequence numbers will have 3 digits.
- The highest permitted clone# is **999999** This limits the sequence# field width to never more than six digits.
- When **FLOAT is given,** the number of equates or maps that will be created will **always** be the **MAXCLONE=** number. (So be careful of large numbers!)
- On the other hand, when **FLOAT is NOT given,** the number of equates/maps created will be limited to the number of chain elements (or table entries) that actually exist.

AUTOCLONE
This activates autocloning. Normally, it is not needed since specifying any of the other autocloning operands will also activate autocloning. But due to defaults, this operand can be used to activate autocloning without providing any of the other autocloning operands. When this happens, the following defaults will be in effect:

```
CFOFFSET=0
CFWIDTH=4
CFMASK=7FFFFFFF
ZCVALUE=0
1STCLONE=1
MAXCLONE=100
```

In other words, specifying **AUTOCLONE** without providing any of the other autoclone operands will generate equates or dsects for labeling/mapping up to the first 100 instances of any chain whose chain field is 4 bytes wide and located at the start of each chain entry.

Chain-Related Autocloning Operands

CFOFFSET=offset
This provides the offset, from the start of each chain entry, of the entry's chain field. This must be a hex number or an N-trailed decimal number whose value ranges from 0 to 7FFFFFFF. (Example: **10** and **16N** both mean sixteen.) The default offset is 0.

CFWIDTH=width
This defines the width of the chain field. Permitted values are **3, 4,** or **8.** The default width depends upon whether or not the CFMASK= operand is given. If it is given, then the default width is the width of the mask provided by CFMASK=. If CFMASK= is not given, then the default width is 4.

CFMASK=mask
This defines a selection mask that is AND'd against the chain field's value to eliminate bits that are not part of the chain pointer. The width of the mask must match the width of the chain field (as defined by the CFWIDTH= operand). The default mask depends upon the width of the chain field as follows:

WIDTH	DEFAULT MASK
3	FFFFFF
4	7FFFFFFF
8	FFFFFFFFFFFFFFFF

ZCVALUE=address
ZCVALUE=NEGATIVE
ZCVALUE=NOTPOSITIVE
Z/XDC **always** considers that a chain has ended when it encounters a zeroed chain field. But some chains (request block queues for example) actually have a different end-of-chain value. This ZCVALUE= operand can be used to provide such an alternate

value. The following ZCVALUE= values are supported:

ZCVALUE=address
The given address expression is resolved, and the resulting address is compared to the contents of each chain entry's chain field. A match identifies the chain's last entry.

ZCVALUE=NEGATIVE
ZCVALUE=NOTPOSITIVE
Both of these keywords cause any zero or negative value to signify end of chain. **Important!** When testing for a negative value, the CFMASK= mask is ignored. In other words, the **field's** hi-order bit is checked. If it is on, then the chain is ended.

The default ZCVALUE= value is 0.

Table-Related Autocloning Operands

LFOFFSET=offset
If the table entries have a length field, then this operand provides the offset of that field from the start of each table entry. **Offset** must be a hex number or an N-trailed decimal number whose value ranges from 0 to 7FFFFFFF.
(Example: **10** and **16N** both mean sixteen.)

The default is to presume that the table entries do not have a length field, and therefore, that all of the table entries have the same size. (See TESIZE= below.)

Notes:
- If **LFOFFSET=** is given, then **LFWIDTH=** also must be given.
- The table entry's length field, if present, can provide either the entire length of each table entry or just the length of a variable portion of each table entry:
  - In the first case, be sure to also specify (or default to) **TESIZE=0** (see below).
  - In the second case, you will have to use the **TESIZE=** operand to provide the length of the nonvariable part of each table entry.

LFWIDTH=width
If the table entries have a length field, then this operand provides the width of that field. **Width** may range in value from 1 to 8.

There is no default for LFWIDTH=, since if LFOFFSET= is given, then LFWIDTH= also must be given (and vice-versa).

TESIZE=size
If the table to be labeled/mapped has entries that are all the same size, or if the entries are variable but also have a fixed section of a constant size, then this operand provides the length of the fixed entries or of the fixed section. **Size** must be a hexadecimal number or an N-trailed decimal number having a value that ranges from 0 to 7FFFFFFF. (Example: **10** and **16N** both mean sixteen.) The default value is 0.

ZTVALUE=address
ZTVALUE=NEGATIVE
ZTVALUE=NOTPOSITIVE
Either **ZTVALUE=** or **ZTPTR=** or **MAXCLONE=** can be used to provide end-of-table information. **ZTVALUE=** and **ZTPTR=** are mutually exclusive. **(MAXCLONE=** is not.)

ZTVALUE= can be used when the table entries lengths are defined by a length field. It causes Z/XDC to examine the contents of the length fields to look for a specified end-of-table condition.

When **TESIZE=0,** then a length field must be present, and a length field containing a zero will always end a table. However, when **TESIZE>0** and length fields also are present, then a zeroed length field does **not** necessarily end a table.

There is no default value for **ZTVALUE=.** When **ZTVALUE=** is omitted, the length field's value does not participate in end-of-table determination (except as noted in the preceding paragraph).

ZTVALUE= may be specified as follows:

ZTVALUE=address
The given address expression is resolved and the resulting address is compared with the contents of the length field. A match identifies the last entry in the table. (The resolved address expression may not be wider than the length field.)

ZTVALUE=NEGATIVE
This keyword notifies Z/XDC that the length field for the last entry in the table will contain a negative value (i.e. the field's hi-order bit will be on).

ZTVALUE=NOTPOSITIVE
This keyword notifies Z/XDC that the length field for the last entry in the table will contain a zero or negative value.

ZTPTR=address
Either **ZTPTR=** or **ZTVALUE=** or **MAXCLONE=** can be used to provide end-of-table information. **ZTPTR=** and **ZTVALUE=** are mutually exclusive. **(MAXCLONE=** is not.)

ZTPTR= can be used to provide a table's ending address. Z/XDC will recognize the end of the table when it attempts to create an equate or dsect map that is located at or beyond the ending address.

There is no default value for **ZTPTR=.** When **ZTPTR=** is omitted, an ending address check is not performed.

Selected Non-Autocloning Related Operands

All of the remaining operands are not autocloning operands, so they are not described here. (They are described in the parent topic, **HELP COMMANDS EQUATE** or **HELP COMMANDS USING).** Nevertheless, some of them require a comment or two

with respect to autocloning.

FLOAT
This causes the generated equates or maps to be assigned floating bases instead of fixed bases. This is a very powerful and useful capability, but you need to be aware that the resolution of floating bases is extremely **CPU intensive** and has to be done repeatedly during typical Z/XDC processing. Therefore, if the number of equates and maps having floating bases grows "too large", then **Z/XDC's performance can be impacted severely!**

Be aware that when **FLOAT** is specified, the autocloning process will **always** create the number of equates/maps equal to the **MAXCLONE** value. This means that if **MAXCLONE** is too high, Z/XDC's responsiveness will become terrible!

Usually, specifying **MAXCLONE=10** will give you enough floating equates/maps to meet your needs without impacting performance noticeably.

name
For autocloning, **name** gives the root name to be used for the generated equates or dsect maps. Sequence numbers will be appended to this name to form the names of the individual equates or maps.

address
For autocloning, **address** provides the location of the first table or chain entry to be labeled or mapped.

All remaining operands
All operands that can be given with normal **EQUATE** and **USING** commands can also be given with the autocloning forms of these commands. The attributes are assigned to all equates or maps created by the autocloning process.

Examples:

EQ RB 21C%% CFO=1C CFM=00FFFFFF ZCV=21C% MAX=100
This causes a series of equates (named RB001 through RBnnn) to be created to label the current task's request blocks. The RBs will be labeled from newest (RB001) to oldest. (Note, this is the opposite sequence by which the LIST RBS command labels request blocks.) Not more than 100 equates will be created. The last labeled RB will be the one whose link field points back to the TCB.

DMAP XDCMAPS.RB
USING RB 21C%% CFO=1C CFM=00FFFFFF ZCV=21C% MAX=99
This sequence of commands accomplishes the same thing as the preceding example, but uses dsect maps instead of equates. Here are the details:

DMAP XDCMAPS.RB
This loads a copy of a request block's dsect map from Z/XDC's XDCMAPS module. (See **HELP MAPS XDCMAPS.)**

USING RB 21C%% CFO=1C CFM=00FFFFFF ZCV=21C% MAX=99
This command first assigns the RB dsect map to represent the newest request block. It then generates up to 99 copies of the RB map, with names ranging from RB01 to RBnn. It then assigns the first generated map (RB01) to represent the very same request block to which the RB map was assigned. It then generates additional copies (named RB02 through RBnn) to represent the remaining request blocks queued from the current task. The **ZCVALUE=21C%** operand lets Z/XDC recognize that the last queued request block will have a link field that points back to the TCB.

EQUATE TIOE 21C%+C%+18 LFO=0 LFW=1 MAX=1000
This command creates a series of equates (named TIOE0001 through TIOEnnnn) to label the DD entries in the current task's TIOT (Task I/O Table). Not more than 1000 equates will be created. The last labeled DD entry will be the one whose length field contains X'00'. (Note, LFVALUE=0 did not have to be specified because a zeroed length field is the default end-of-table condition for tables that contain variable length entries and for which the length fields specify the **entire** length of each table entry.)

LIST TASKS
DMAP XDCMAPS.TCB
USING TCB TCB#3 FLOAT
EQUATE RSA .TCBFSA? DATA 48 CFO=8 MAX=100 FLOAT
This series of commands creates a set of 100 floating equates that will map up to 100 standard 72-byte register save areas that are chained from a task's TCBFSA field.

- Because the **FLOAT** operand is given, the full maximum number of equates permitted by the **MAXCLONE=** operand are created. This has the following results:
  - When the chain being mapped has 100 or more elements, the first 100 elements will be labeled by the generated equates.
  - When the chain has fewer than 100 element, the excess labels simply won't resolve.
  - When the length of the chain changes, the labels will automatically resolve or not so that the then-current elements on the chain will always be labeled.

- The **LIST TASKS** command, as a side effect, automatically creates equates named **TCB#1, TCB#2,** [...] to label all of the Task Control Blocks that are displayed by that command. (See **HELP COMMANDS LIST TASKS** for more information.) So the **USING** command takes advantage of that fact to assign the TCB map to the third listed Task Control Block.

- If you wish to map the RSA chain for a different task, simply issue another **USING** command to reassign the TCB map to a different TCB. Example: **USING TCB TCB#7 FLOAT**

- Because the TCB map floats, you can automatically map the RSA chain of the nth task in any other address space simply by issuing a **SET ASID asname** command

(security and authority permitting, of course) followed by another **LIST TASKS** command (to rebuilt the TCB#n equates). Example:

```
SET ASID JES2
LIST TASKS
```

```
DMAP XDCMAPS.TCB
USING TCB 21C%
DMAP .SCB
USING SCB .TCBSTAB% AUTOCLONE FLOAT
```

This series of commands first loads a dsect map for the TCB and positions it to represent the current task's Task Control Block. It then loads an SCB map (STAE Control Block), and then autoclones it to represent all SCBs that are currently queued from the current task's TCB.

The **USING ... AUTOCLONE** command uses the following defaults, all of which are appropriate for mapping the STAE Control Block queue:

CFOFFSET=0
The SCB's chain field (SCBCHAIN) is located at the start of the control block.

CFWIDTH=4
The width of the SCBCHAIN field is 4 bytes.

CFMASK=7FFFFFFF
The SCBCHAIN field contains 31-bit addresses.

ZCVALUE=0
The end of the SCB queue is signaled when SCBCHAIN is zeroed.

MAXCLONE=100
Since I am creating floating maps, The default **MAXCLONE** value of 100 is a bit high. Z/XDC response time is likely to be affected. So next time, you'll probably want to stipulate a lower number. **MAXCLONE=10** or **=20** might be good choices.

Since I am satisfied with the defaults for **all other** autoclone related operands, I am not using them in my command string. But I still need something to notify Z/XDC that I am intending autocloning. So I have to use this **AUTOCLONE** operand to provide that notification.

A Real World Example

The following links to an example of using autocloning to help diagnose queue corruption we encountered in a storage dump we received. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

Example - Diagnosing a corrupted DFE chain

Help COmmands EQuate Autocloning Example

Recently, we received a dump where Z/XDC's **LIST SUBPOOLS** command was failing and issuing a message reporting that its scan of a DFE queue had failed **(DBC159-24).**

After some investigation, we determined that the **LIST SUBPOOLS** command's logic was fine, and that one of the System's DFE queues was, in fact, corrupted.

What are DFEs?

Briefly, DFEs are Virtual Storage Management (VSM) cblocks used by z/OS to track unallocated chunks of storage located within allocated storage blocks located in SQA or LSQA.

In the dump I was examining, the DBC159E-24 message I received indicated that a DFE queue was corrupted for subpool 245. There are four DFE queues for subpool 245. They are anchored in the GDA at 4-word fields labeled GDADFEQ5, GDAEDFE5, GDADFEQ5R64 and GDAEDFE5R64. The corrupted DFE could have been on any one of thee queues.

There is a lot more to know about DFEs, but for the purposes of this example, the above information is all that matters. But if you're curious, take a look at **HELP MESSAGES DBC159 19-24.**

Using Autocloning to Identify the Broken DFE Chain

Using the commands illustrated below, I was able to very quickly examine each of the four DFE queues one by one, and discovered that the damaged DFE was on the GDAEDFE5 anchored queue. This queue is for subpool 245 storage blocks that are located in 31-bit virtual storage and backed by 31-bit real storage.

But GDAEDFE5 was the second queue I examined. The first was GDADFEQ5 (for subpool 245 storage blocks that are located in 24-bit virtual storage and backed by 31-bit real storage).

I started by loading maps and created equates for labeling a DFE chain.

DM XDCMAPS.CVT;U CVT 10?
DM .GDA;U GDA .CVTGDA?
These commands load and place the CVT and GDA maps at their proper locations in storage.

EQ Q .GDADFEQ5 18 DATA AUTOCLONE CFO=0 ZCV=.GDADFEQ5 1ST=0 MAX=999
This is the autocloning form of the **EQUATE** command. It does the following:
- The chain being labeled is the DFE queue's address ordered, forward chain.
- The command creates up to 999 equates to label up to 999 elements of the chain.
- If an end condition is met, then fewer equates are created.
- In this case, the end condition is a chain field that points back to the GDADFEQ5 anchor field (ZCV=.GDADFEQ5).

- The first equate created is named Q000 (1ST=0) and labels the anchor fields themselves.
- Consequently, Q001 labels the 1st actual DFE.

Analysis

If fewer than 999 equates are created, then there are two possibilities:
- The chain is intact (likely).
- Or the last chain field points into unallocated storage (0C4 storage).

To see which it is, issue **FORMAT Qnnn?** where **nnn** is **one less** than the number of equates reported created.
- If it displays the anchor field (GDADFEQ5), then you're good.
- If anything else happens, then that last DFE or some prior DFE on the chain is the broken DFE.

If 999 equates are created, then there are two possibilities:
- The chain actually contains more than 999 elements (unlikely). But to see if this is the case, then try reissuing the command with **MAX=9999.**
- Or the chain is broken, and the equates creation process either has fallen into a loop (likely) or has wandered off into never-never land. If the chain is looped, then a **FORMAT Qnnn?** (where **nnn** is one less than the number of equates created) will show multiple Qnnn labels at the displayed location.

When a Chain is Not Broken

If the GDADFEQ5 chain is **not** broken, then move on to the next (GDAEDFE5) as follows.

EQ Q .GDAEDFE5 18 DATA AUTOCLONE CFO=0 ZCV=.GDAEDFE5 1ST=0 MAX=999

Notes:
- Both references to **GDADFEQ5** have been changed to **GDAEDFE5.**
- This command will purged the previously created Qnnn equates before creating a new set.

Wash, rinse, repeat until you find the broken DFE chain.

When a Chain is Broken

If the chain **is broken,** then you may now want to find exactly which DFE is the culprit. To do that, the easiest way is to create a **Watch Window** to display a consecutive set of DFEs and see if they make sense.

One way to create a Watch Window is to cursor down to the desired point on your display screen and press the **shift-F1** key.

Then enter the following command onto the new command line:

SH Q001 ? ?? ??? ????? ?????? ??????? ???????? ????????? ADD DAT

This creates A 10-line display showing the first ten DFEs by following the chain fields. Notes:

- The first DFE displayed will be labeled **Q001.**
- The last one displayed should be labeled **Q010.**
- If the last is labeled anything else, then the broken DFE is one of the ones shown.

If all is well, then move the display down the chain by reissuing the above **SHOW** command but change **Q001** to, say, **Q500:**
- The first DFE displayed will be labeled **Q500.**
- If the 2nd is **not Q501,** then the broken DFE resides prior to Q500.
- On the other hand, if the last displayed DFE is labeled **Q510,** then the broken DFE resides further down the chain.

Wash, rinse, repeat until you zero in on the broken DFE.

The DFECURNT and DFEPRIOR Equates

When a **DBC159E-19** thru **-24** error occurs, several automatic equates are created to help resolve the problem. (See **HELP MESSAGES DBC159 19-24** for details.)

Two of those equates are name **DFECURNT** and **DFEPRIOR.** If you create a Watch Window and set the command line to **FORMAT DFECURNT B=18;;FORMAT DFEPRIOR B=18,** then when you issue the above suggested autocloning commands, you will see **Qnnn** labels appear when you get to the right queue.

(Remember, the current DFE and prior DFE are not necessarily on the same queue.)

Help COmmands EQuate Ritargets

This topic has not yet been written.

Help COmmands EWhere

The **WHERE** command generates a formatted display of storage showing the current **retry** level PSW's resume execution location.

The **EWHERE** command generates a formatted display of storage showing the current **error** level PSW's **(EPSW's)** ABEND location.

Display Positioning

The **WHERE** and **EWHERE** commands will always produce a display that shows the **PSW's** or **EPSW's** execution location; however, that location generally will **not** be at the top of the display. Instead, it will most often be embedded several lines down

into the display.

But the display line showing the execution location will be both flagged and hilighted, so it will be easy to spot.

When **SET TRACE ROLL** is in effect, the **TRACE** command internally uses the **WHERE** command to produce its displays such that the resume address (which, of course, advances during tracing) has the appearance of rolling down the display screen.

The **WHERE** and **EWHERE** commands accept an address expression for an operand. This can be used to change the start of the generated display relative to the **[E]PSW's** execution address.

But only addresses which result in a display that **includes** that resume address are valid. If this rule is violated, then the given address expression is silently **ignored,** so that Z/XDC can still generate a display that will contain the appropriate execution address.

The CDP and NDP

Both the **WHERE** and **EWHERE** commands set the **CDP (Current Display Pointer)** to point to the **[E]PSW's** execution address (as flagged and hilighted in the **[E]WHERE** command's display). Note, **unlike** all other storage displaying commands, this CDP setting usually is **not** at the start of the display.

Personally, I hate this exception, but I had to create it because it **simplifies** references by subsequent commands to the **[E]PSW's** resume address.

Example: Suppose a WHERE command display is up, and somewhere down the middle of the display it shows that the next instruction to be executed is a **BAS** to subroutine. you might want to issue **T +4;GOT** to allow that subroutine to run without your having to step through it. The fact that the **CDP** points to the BAS in this example (and not to the start of the display) is what makes the **T +4** command possible.

The **NDP** (the **Next Display Pointer)** is more normal. It is set to point to the **end** of the display.

WHERE and EWHERE Display Similarities and Differences

Generally, the **WHERE** and **EWHERE** commands will produce similar displays **except** when the retry level and error level environments are **different.** Then the **WHERE** command will show the **retry level resume address,** while the **EWHERE** command will show the **error level ABEND address.** For more information, see HELP EXECUTIONLEVELS.

When the Retry level and Error Level execution environments are the same, the **WHERE** and **EWHERE** commands may still show resume addresses that **differ** (usually by two bytes). This difference arises from the fact that for most program checks, the hardware leaves the PSW pointing **past** the failing instruction. This is what will be retained in the Error level PSW and what will be shown by the **EWHERE** command.

On the other hand, for the Retry Level PSW, Z/XDC will adjust the resume instruction address to point to the **start** of the failing instruction, hence the difference. For

more information, see HELP EXECUTIONLEVELS.

When the Error and Retry Levels are the Same

Notice: Even when the error and retry environments are the same,
the **WHERE** and **EWHERE** commands will usually still produce differing displays when Z/XDC has received control due to a program check. This is because for most program checks, the hardware will leave the PSW pointing **past** the failing instruction, not at it. This has the following effects:

- The **error level** PSW (represented within Z/XDC as **EPSW** or **EPSWE)** will be set to the actual PSW, exactly as the hardware left it at program check time or ABEND time.

- The **retry level** PSW, on the other hand, will be adjusted by Z/XDC to point to the **start** of the failing instruction (not its end).

Examples: [when the error level and retry level environments are the same]

- **WHERE** produces a display guaranteed to include your program's resume address.

- While **EWHERE** does not. Instead, the display will be guaranteed to include the program check or ABEND location, not the resume address.

- **WHERE PSWE?** will produce a display that **starts** at the resume address.

- To produce the same display with the **EWHERE** command, you would have to use either of the following:
  - **EWHERE EPSWE?-2** [usually]
  - **EWHERE PSWE?**

  (Note, the presumptions here are:
  - That the error level and retry level environments are the same,
  - And Z/XDC received control as a result of a program check in which the failing instruction was 2 bytes long,
  - And the hardware left the error time PSW pointing past that instruction.

  - Note that the hardware perceives all Z/XDC breakpoints as being 2-byte wide machine instructions. Thus, whenever Z/XDC receives control due to a breakpoint, the EPSW will always be set to 2 bytes past the start of the machine instruction even when that instruction is a 4-byte or 6-byte instruction.)

When Z/XDC Receives Control from a Breakpoint

Breakpoints can be constructed either using an illegal opcode
(specifically, **X'00')** or TRAP2 instructions **(X'01FF'):**

- In the former case, a code 0001 program check will occur which eventually will cause Z/XDC to received control as an ABEND Recovery routine. In this case the error and retry levels **may be the same** or they **may be different** (which may be OK, but usually is not).

- In the latter case, a Trap Exception will occur which causes Z/XDC to receive

control as a Trap Handler routine. In this case, the error and retry level environments **will always be the same.**

For more information, see HELP BREAKPOINTS TYPES.

For both types of breakpoints, when the error and retry levels are the **same,** there will **always be a 2-byte difference** between the locations pointed to by **PSWE** and **EPSWE.** This will result in the **EWHERE** issues discussed above in the preceding section.

The 2-byte difference occurs **regardless** of the length of the instruction upon which the breakpoint is placed. This is because the hardware doesn't see that instruction. Instead, it sees:

- Either the illegal opcode **X'00'** which the hardware regards as a 2-byte wide instruction,
- Or a **TRAP2** instruction **(X'01FF')** which, in fact, is a 2-byte wide instruction.

Local Overrides of SET FORMAT Options

The **[E]WHERE** commands accept several keyword operands that allow you to do one-time overrides of the various format control settings defined by the **SET FORMAT** command.

Syntax:

```
[E]WHERE addressexpression lines           SOURCE  ADDRESSES  DECIMAL ...
EW or W  omitted           LINES=lines     OBJECT  OFFSETS    HEXADECIMAL
         ,                 BYTES=hexnumber BOTH

     ... ASCII   WIDE   INSTRUCTION SHOWMCODE   POINTERS
         EBCDIC  NARROW DATA        HIDEMCODE   NOPOINTERS
                        NOBIAS      CURRENTMCODE
```

Shortcuts: E
W

Notes:

Operands appearing above within columns are mutually exclusive.

All operands are optional.

If the addressexpression operand is given, then it must be first. All other operands may be given in any order.

If the addressexpression operand is omitted but other operands are given, then a **comma** must be used to indicate the omission.

Operands:

addressexpression
This operand is optional. If given, then it must resolve to a location that is **at or preceding** the execution address for either the **PSW** (for the **WHERE** command) or the **EPSW** (for the **EWHERE** command). **But** the address must still be close enough to the **[E]PSW's** execution address so that the resulting display will include that address. (Note, the size of the display usually is determined by the size of the window from which the **[E]WHERE** command was issued.) If the given address expression fails to follow these rules, then it is silently ignored, and the **[E]WHERE** command is processed as if the address expression had been omitted.

omitted
If the **addressexpression** is omitted, then the **[E]WHERE** command chooses a starting display address automatically.

When **SET TRACE ROLL** is in effect, then when possible, Z/XDC will choose the same address as was used by the last preceding [E]WHERE display, but if execution tracing has resulting in the [E]PSW moving beyond the range of the display, then the command will instead choose an address that repositions the [E]PSW to the top of the display.

,
If the **addressexpression** is omitted and other operands are given, then a **comma** must be used to show the omission.

Format Controls

The following operands may be given in any order. They must, however, follow the given address expression (if any).

Also, all of the following operands (except **lines, LINES=** and **BYTES=)** serve to override the corresponding default values established via the **SET FORMAT** command.

lines
LINES=lines
BYTES=hexnumber
These control the size of the display. The **lines** and **LINES=** operands do so in terms of the number of lines (rows) of instructions, data and comments the display is to contain. The **BYTES=** operand does so in terms of the amount of storage that is to be displayed:

lines
LINES=lines
These two operands function identically; however the **LINES=lines** form is preferred because eventually support for the **lines** form may be discontinued.

The **lines** value must be a decimal number specifying the number of display lines to generate containing instructions, data and comments (i.e. not including the Location Interpretation header lines). This value must be in the range of 0 to

65,535 (64K-1). When omitted, either the default value set by the **SET LINES** command is used (for nonfullscreen displays), or the length of the current window is used (for fullscreen mode displays).

The **bytes** value must be a number giving the amount of storage to be displayed. It can be either of the following:

- A hexadecimal number (Example: **BYTES=C8)**
- A decimal number followed by the letter **N** (Example: **BYTES=200N)**
- A scaled decimal number (Example: **4K)**

Regardless of the number's form, it must resolve to a value between **0** to **1FFFE0:**

- The very strange X'1FFFE0' limit is (64K-1)∗32:
  - 64K-1 is the LINES= limit.
  - 32 is because one line can display up to 32 bytes of storage.

The number of display lines needed to produce a BYTES= driven display can vary.

The **lines, LINES=** and **BYTES=** operands are mutually exclusive.

SOURCE
OBJECT
BOTH
These operands control the use of maps that might be available for controlling the formatting of the storage being displayed:

SOURCE
Causes storage to be formatted under the control of a source level map, if available.

OBJECT
Causes storage to be formatted by disassembly. Any labels generated by equates, dsect maps, or csect maps will also be displayed where applicable. Information from source level maps will **not** be displayed.

BOTH
Causes storage to be formatted by both SOURCE and OBJECT; source level statements will be interspersed with statements generated by disassembly.

Note, when **DATA** is explicitly given, both **SOURCE** and **BOTH** are ignored in favor of **OBJECT.**

POINTERS
NOPOINTERS
When storage is being formatted via disassembly (i.e. when either **OBJECT** or **BOTH** is in effect), if a line of display is showing a snippet of data that is 3, 4 or 8 bytes wide, then this **POINTERS** operand will cause Z/XDC to attempt to resolve that snippet as a storage location. Then it will display that resolution as a comment on the display line. Notes:

- The **NOPOINTERS** operand prevents these resolutions.
- The factory default is **POINTERS.**
- When very complex structures of **floating maps and equates** are present, the **POINTERS** setting may result in slower displays. In that case, you may wish

to use **NOPOINTERS** instead. (For more information about floating maps and equates, see **HELP MAPS FLOATING.)**

SHOWMCODE
HIDEMCODE
CURRENTMCODE
These operands control whether or not source image displays formatted via **ADATA** maps will include source code create by certain macro expansions:

- **SHOWMCODE:** The expansions of all macros are shown.
- **HIDEMCODE:** The expansions of all code generating macros are suppressed. (Data generating macros, such as control block mapping macros, are unaffected.)
- **CURRENTMCODE:** The expansions of all code generating macros are suppressed **except** if the macro expansion currently being displayed includes the error level or retry level PSW address. In this case, the expansion is shown. (Data generating macros, such as control block mapping macros, are unaffected. They are never suppressed.)

Note the following:

- The intent of the **CURRENTMCODE** operand is to improve the display of code written using structured programming macros.

- Also, the hiding of macro expansions is effective only for storage formatted via **ADATA** maps. Storage formatted via SYM data maps is unaffected.

- Finally, the hiding of macro expansions does not occur for dsect maps that map data areas and control blocks.

For detailed information about what happens when macro expansions are suppressed, see HELP MAPS ADATA MACROS.

ADDRESSES
OFFSETS
These operands control whether the far left column of the display (the "address" column) will show storage addresses or offsets:

ADDRESSES
Causes each line of data displayed to be prefixed by that data’s virtual address.

OFFSETS
Causes each line of data displayed to be prefixed by that data’s offset from the start of an appropriate including object (load module, csect, dsect, or equate).

DECIMAL
HEXADECIMAL
For disassembled machine instruction displays, these operands control whether the displacement fields are formatted as decimal or hexadecimal numbers:

DECIMAL
Causes the displacement fields to be displayed as decimal numbers.

HEXADECIMAL
Causes the displacement fields to be displayed as hexadecimal numbers.

ASCII
EBCDIC
For data displays, these operands control whether the text portions of data displays are to be interpreted using the **ASCII** or **EBCDIC** character set:

ASCII
Causes data displays to show the **ASCII** interpretation of the displayed storage. To visually indicate this, the text portion of each display line will be framed with **vertical bars (|)** instead of asterisks.

EBCDIC
Causes data displays to show the **EBCDIC** interpretation of the displayed storage. To visually indicate this, the text portion of each display line will be framed with **asterisks (*).**

More information on EBCDIC and ASCII can be found at **HELP CHARACTERSETS**

WIDE
NARROW
These operands control whether data displays are to be wide or narrow:

WIDE

This causes data displays to show up to **32 bytes** of storage per line (not just 16). This option works best when your terminal's display is at least 136 columns wide. For more information, see HELP FULLSCREEN TERMINALS.

NARROW
This causes data displays to show only **16 bytes** of storage per line. This width is appropriate for terminals with 80 character wide display lines.

INSTRUCTIONS
DATA
NOBIAS
The **WHERE** and **EWHERE** commands generally attempt to format storage contents as instructions, but this bias can be influenced by many factors, such as whether or not a PSW **(PSW** or **EPSW)** points to the storage being formatted, as well as the attributes of maps, fields and equates that label the storage being displayed. As a part of the management of this process, the **[E]WHERE** commands maintain a concept called the **formatting bias** which, for a given displayed line, retains information about how the preceding line was formatted and, therefore, contributes information about how the current displayed line should be formatted. In other words, if a display line was formatted as data, then the **[E]WHERE** commands will be biased towards formatting the following display line also as data, unless new information (such as label attributes) suggests otherwise.

The **INSTRUCTION, DATA,** and **NOBIAS** operands set the initial formatting bias for the **[E]WHERE** commands as follows:

INSTRUCTION
The **[E]WHERE** commands' initial formatting bias will be to interpret storage contents as being machine instructions. This will be done even if other factors might suggest that the storage contents should be interpreted as data. This will **not** be done, however, if the initial opcode is invalid.

DATA
The commands' initial formatting bias will be to interpret storage contents as being data. This will be done even if other factors might suggest that the storage contents should be interpreted as machine instructions. Note, when **DATA** is explicitly given, both **SOURCE** and **BOTH** are ignored in favor of **OBJECT.**

NOBIAS
The commands' initial formatting bias will not be forced. Absent other factors, the storage contents will initially be interpreted as machine instructions, but if other factors suggest otherwise, then those factors won't be ignored.

These **INSTRUCTION, DATA,** and **NOBIAS** operands set only the **initial** formatting bias. Once the display gets going, the formatting bias for the second and subsequent display lines will be influenced only by the normal factors discussed above.

Storage Protect Key Display

The **[E]WHERE** commands show the storage protection key as a single hexadecimal digit appearing in the display's header line, just to the right of the address field. Following the storage protect key may be additional characters:
f - The 'f' character indicates that the storage is fetch protected.
s - The 's' character indicates that the storage is store protected. (there are additional comments about store-protected storage below).
r - The 'r' character indicates that the storage may be redacted. (this character can only be seen when processing a redacted dump in dump/XDC). A redacted area is indicated by the storage being all binary zero. Note that it is impossible to distinguish storage that has a value of binary 0 from storage that has been redacted.

The "store protected" indicator, **s,** is displayed whenever Z/XDC detects that the storage being displayed has one or another of the following special protections:

(Note than when processing a dump (using dump/XDC), many store-protection areas are not indicated. This is because in many cases the store-protection flag is in the page table and that information is not available to dump/XDC).

- **Low Address Protection** (LAP): This is a special kind of protection that the Operating System uses to specifically protect the first 512 bytes of both real and virtual storage. It prevents the storage from being altered by any program, regardless of that program's state or key. (LAP does not apply to data space storage.)

- **Page Protection:** This is a kind of protection that applies to the following areas of storage:
  - The Pageable Link Pack Area (PLPA)
  - The read-only areas of the System Nucleus (IEASYS0x)
  - Any area of storage that has been set to "read-only" by the PGSER or

IARVSERV macros.
When a page of storage has been marked as being "read-only", that storage cannot be directly altered by any program, regardless of that program's state or key. (Note, "Page Protection" does not apply to real storage. Thus, it cannot protect a virtual storage page when that page is accessed via its real address.)

- **Access List Entry Protection:** When an address space or a data space is accessed via an ALET that "points to" an Access List Entry (ALE) having its "fetch only" flag on, then accesses via that ALE are store protected. Note, accesses to that same space via a different ALE might not be store protected.

Examples:

W

A formatted display is generated that shows the current retry level PSW's resume execution location somewhere within the display. The display will be either wide or narrow according to the current SET FORMAT command setting. Note, if you then start (or resume) using Z/XDC's TRACE command to step through program execution, Z/XDC attempts to continue using this positioning for as long as it can for the displays generated by the TRACE commands that you issue.

W ,WIDE

This causes a wide display to be generated without specifying a new display address. Note, the WIDE/NARROW operand won't have any effect on storage being formatted as machine instructions. It primarily affects displays that include large data fields.

EW EPSW?

A formatted display is generated that shows the current error level PSW's ABEND location at the top of the display.

F EPSW?

Like the preceding example, a formatted display is generated that shows the current error level PSW's ABEND location at the top of the display. **However,** unlike the preceding example, the **EWHERE** command's display position information is **not** changed. Thus, a subsequent **EW** command, without operands, probably would not show the execution location at the top of the display.

W PSW?+10

This is not valid. The given address expression resolves to a location that **follows** the resume PSW's address, and so the display that would be generated cannot possibly include the resume address. Accordingly, Z/XDC ignores the address expression and proceeds to process the **WHERE** command as if no address expression had been given.

W

W +0

The first **WHERE** command generates a display that includes the retry level PSW address somewhere within that display. It also sets the CDP (Current Display Pointer) to point to that retry level address.

The second **WHERE** command references the CDP ("+0") to reposition the display so that the retry level PSW address is shifted to the top of the display. In other words, this sequence of two commands has basically the same effect as the "W PSW?" command.

W ,OFFSETS SOURCE

This generates a display that (a) shows offsets instead of virtual addresses and (b) formats the display using a source level map instead of disassembly (assuming that a source level map has been loaded and positioned to include the retry level PSW address).

Other commands that can be used to display storage are:

- **DISPLAY** - Displays a contiguous chunk of storage in a hex-text (dump like) format.
- **FORMAT** - Formats a contiguous chunk of storage as either machine instructions or data fields, as appropriate.
- **SHOW** - Accepts a list of address expressions and formats a 1-line display for each expression given.
- **FIND** - Scans storage for a given string of data.

Help COmmands EXec

EXEC is a command that allows the user to run user-written subcommands written in REXX.

Syntax:

```
EXEC STARTREXX
     ENDREXX
     LISTREXX
     execname argumentstring
```

STARTREXX

This command initializes a rexx/XDC Interface. It causes Z/XDC to call IRXINIT to do the following:

- Locate Z/XDC's REXX interface function package (load module xxxEFMVS) and load it into storage.
- Locate the user's REXX exec library (//xxxREXX ddname) and open it.
- Build a REXX Environment Block as well as all other structures necessary for

running REXX execs.

Notes:

- This command is optional. If a REXX exec is invoked prior to this command having been issued, then Z/XDC will perform the STARTREXX processing anyway.

- This command will fail if the user's REXX exec library(ies) have not been preallocated to ddname //xxxREXX (where "xxx" is Z/XDC's current clone name).

ENDREXX
This is an optional command to explicitly shut down a rexx/XDC Interface. It does the following:
- It deletes Z/XDC's REXX interface function package (load module xxxEFMVS) from storage.
- It closes the REXX exec library (//xxxREXX ddname).
- It purges the REXX Environment Block and all related structures.

Notes:

- This command is optional. If an **END** command is issued (except for END KEEP) and the rexx/XDC Interface still exists, then Z/XDC will perform the ENDREXX processing automatically.

- If the debugging session should end by any means other than an **END** command, then the rexx/XDC Interface (if any) is **not** automatically taken down.

LISTREXX

This command is identical to the LIST REXX command. It displays information about the rexx/XDC Interface. See HELP COMMANDS LIST REXX for more information.

execname
This is the name of the REXX exec to be run.
the following formats are recognized:
  name
  name(member)
if **name** has at least 1 fullstop within it, the value is always regarded as a dataset name.
if **name** has no fullstops within it, the value is regarded as **simple** and the value could be:
  1: A procedure name.
  2: A DDname.
The rules used to decide which applies are:
- If immediately followed by an open parenthesis ('(') then the value is treated as a ddname.
- If **SET REXX NAME=PROCEDURE** is in effect then the value is treated as a procedure name.
- Otherwise, the value is checked to see if it is an allocation:
  - If a current allocation is **not found** the value is treated as a procedure name.
  - If a current allocation is **found** the value is treated as a ddname.

Based on the above, an execname could be:
  1: A procedure name - naming a member in the xxxREXX library
  2: A ddname - defining a sequential file allocated to that ddname.
  3: A dsname - defining a sequential file by dataset name.
  4: A member of a library defined by a specific ddname.

5: A member of a named PDS(E) library.
Note that, in all cases, the REXX execs library must be allocated (ddname: xxxREXX, see below).

argumentstring
This is the argument string (if any) that may be required by the REXX exec. The user's exec can receive this string via an **ARG** or **PARSE ARG** instruction.

Note, due to Z/XDC's own syntax rules, argumentstring may not contain semicolons or colons unless the string is enclosed within quotes ('). See COMMANDS SYNTAX CHARACTERSTRINGS for more information.

Requirements

The following are the requirements for enabling the ability to run REXX execs as Z/XDC commands:

- The REXX execs may reside in any library or sequential dataset. However the **xxxREXX** ddname must be allocated in all cases (to a library or concatenation of libraries). If all the procedures are explicitly executed by dataset name (and member name), the **xxxREXX** library can be empty. (the prefix **xxx** must match Z/XDC's current clone name, usually **XDC,** see HELP XDCCLONES for details).

Implicit REXX execution

The **%** command can be used instead of the **EXEC** command to request execution of a REXX procedure.

Z/XDC Services Available to User-Written Execs

The rexx/XDC Interface provides several built-in functions by which user-written execs can call a selection of Z/XDC's internal services. Generally, these services do such things as:
- Control the source of REXX input and output
- Parse address expressions
- Extract or zap data in storage
- Extract or zap data in registers
- Extract or alter the retry level PSW
- Send messages to Z/XDC's Display Management for display at the user's terminal.
- Request data from a user

For complete and detailed information, see HELP REXX XDCSERVICES.

Help COmmands FInd

The **FIND** command searches the storage of any space for one or more occurrences of a given string of data. The string may be any mixture of hex data, character data, and decimal integers. The search starts at a given address and ends either when a specified number of hits have been found, or when a given or defaulted search distance or limit has been reached, or when the user presses the **ATTN** key at his

terminal.

The search can test for matches at every byte within the search range or at every nth byte.

Specified strings can be AND'd, OR'd, and/or XOR'd into the trial data before each test for a match, and/or the trial data can be EBCDIC and/or ASCII upcased before each test for a match.

Numbered equates can be automatically generated to mark the location of each match to the search string.

When the search ends, if any matches have been found, then the last found match is displayed via either a **DISPLAY** command or a **FORMAT** command. (If no matches were found, then no display of storage is produced.)

The Current Display Pointer **(CDP)** will be set either to the last match or to the end of the search range if no match is found.

The **FIND** command can be used with **dump/XDC,** but be aware that it will need to fetch storage being examined from the IPCS dump dataset. This process will be significantly slower than referencing main storage (as it involves doing I/O).

Storage that was already in the **dump/XDC** storage cache will be used but storage retrieved from the dump (via IPCS) as a result of the **FIND** command will not be cached, as it is unlikely to be referenced again.

dump/XDC uses various techniques to speed up **FIND** command processing against dump storage. Note that only storage that is actually present on the dump can be scanned.

Syntax:

```
    FIND string starting-address ending-address ...
                                 DISTANCE=

         ... ANDMASK= ORMASK= XORMASK= ...
         ... UC       EUC     AUC ...

         ... INTERVAL= OFFSET= HITLIMIT= EQUATENAMES= ...

         ... DISPLAY  NARROW DATA        STATISTICS   ...
             DISPLAY= WIDE   INSTRUCTION STATS
             FORMAT          NOBIAS      NOSTATISTICS
             FORMAT=                     NOSTATS

         ... SHOWMCODE
             HIDEMCODE
             CURRENTMCODE

         ... NOCB
             CB
             CB=hexlength

         ... PAGEOPS=option
```

```
... XAMO     IAMO
    XCMO     ICMO
    XPMO     IPMO
    XSMO     ISMO

... ALLABOVEBAR
```

Operands

all operands omitted
When the **FIND** command is given without operands, a **repeat FIND** operation is performed. A search is performed, using all the same operands as were specified or used by the preceding **FIND,** except that:

- If the preceding FIND finished searching storage, the **repeat FIND** is not done.
- The search starts at the Current Display Pointer (CDP) incremented by the interval of the original search (which, if omitted, will default to 1) (note that the **FIND** command always sets the **CDP** when it's done).
- If the starting address was just an equate name that had a nonzero length specified and no ending address or **DISTANCE=** was specified the **repeat FIND** acts as if an ending address had been specified and only searches the remaining part of the range.

string
This gives a string of data to be searched for. If omitted, then the string given by the previously issued **FIND** command, if any, is used again.

The string may be given as either hexadecimal digits, EBCDIC or ASCII characters, or a decimal integer number, or any combination thereof. Brief examples:

```
FIND F0D93B72,...     a hex string
FIND 'DBCOLE',...     a character string
FIND "DBCOLE",...     a character string
FIND C4'BCOL'C5,...   a mixture
FIND F'-1234557',...  a 4-byte wide integer
FIND H'32767',...     a 2-byte wide integer
FIND 1'139',...       a 1-byte wide integer
```

For more complete information, see **HELP COMMANDS SYNTAX STRINGDATA.**

starting-address
This must be the starting address for the search. The **starting-address** may reference **any** accessible storage: The Target Address Space's private area, common storage, data space storage, foreign address space storage, or real storage. Storage is searched rightwards from this location for the given string.

If the starting-address is omitted, then the **CDP** (Current Display Pointer) plus one is used.

ending-address
DISTANCE=number
Either of these operands can be used to specify how much storage is to be searched. They are mutually exclusive.

- **ending-address:** If the given **FIND** command contains a second address expression, then that is taken to be the search's ending address. It must resolve into the same space as the first address expression. If it resolves to a lower address than the first address expression, then the meanings of the two address expressions are switched: Whichever specifies the lower address will be the starting address, and whichever specifies the higher address will be the ending address.

  The ending-address points to the last byte of storage that is to be included in the search. During the search, the highest address that will be tested will be such that the test string's last byte will be at or below the ending address.

- **DISTANCE=number:** Alternatively, the search limit can be specified as a distance from the starting address. **Number** can be any of the following:
  - A string of hex digits (no quotes, no framing, just the digits)
  - A string of decimal digits followed by the letter **N** (Example: Both **10** and **16N** mean sixteen.)
  - A scaled decimal number (Examples: **4K, 12G** etc.)

  Regardless of the number's form, it must resolve to a value between **1** and **(2**64)-1.**

  If the given distance would cause the search to extend past the end of storage, then the search will stop when the end of storage is reached.

- **omitted:** When neither an ending address nor a **DISTANCE=** operand is given, if the starting address is just an equate name (either user-defined or builtin), and that equate has a nonzero length associated with it, then that length is used as the amount of storage to be searched, else the default amount of storage searched will be eight mebibytes.

ANDMASK=string
ORMASK=string
XORMASK=string
UC
EUC
AUC
These operands can be used to apply masks to and/or upcase the trial data at each compare point, prior to its being compared against the string being searched for.

These operands are not mutually exclusive. Any or all of them may be given in any order. Each mask operand can, of course, specify a different mask. They are applied in the order in which the operands are given.

The masks follow the same rules of syntax as any other string data operand supported by Z/XDC. Briefly, hexadecimal data is specified without punctuation, while character data must be enclosed within quotes ('). The complete rules can be found at HELP COMMANDS SYNTAX STRINGDATA.

The masks **must** be exactly the same length as the search string.

The **UC** operand will upcase the trial data, either EBCDIC or ASCII depending on the current value set by the **SET FORMAT** command.

The **EUC** operand will EBCDIC upcase the trial data.

The **AUC** operand will ASCII upcase the trial data.

INTERVAL=number
INTERVAL=keyword
Normally when searching, a compare is performed at every byte within the range of storage being searched. However, the **INTERVAL=** operand can be used to restrict the compares to only every nth byte. For example, if you know that what you are looking for is doubleword aligned, then specifying **INTERVAL=8** (in combination with an aligned starting address) will cause the search to perform compares only at doubleword boundaries.

number: This can be any of the following:

- A string of hex digits (no quotes, no framing, just the digits)
- A string of decimal digits followed by the letter **N** (Example: Both **10** and **16N** mean sixteen.)
- A scaled decimal number (Examples: **4K, 12G** etc.)

Regardless of the number's form, it must resolve to a value between **1** and **(2**64)-1.**

keyword: For your convenience, certain interval values have nicknames. They are:

KEYWORD	DECIMAL VALUE
BYTE	1
HWORD	2
FWORD	4
DWORD	8
QWORD	16
HALFWORD	2
FULLWORD	4
DOUBLEWORD	8
QUADWORD	16
PAGE	4096

Examples:

INTERVAL=1000
INTERVAL=4096N
INTERVAL=4K
INTERVAL=PAGE
All four of these cause the search to check for a match, starting at the starting address, and then at every 4096th byte after the starting address.

INTERVAL=BYTE
INTERVAL=1
These are the same as not specifying an interval. They both cause the search to test for a match at every byte in the search range.

INTERVAL=7
This causes the search to test for a match at every seventh byte, starting from the starting address.

OFFSET=number
An offset can be specified to cause the search to test for a match at a specific offset.

number: This can be any of the following:
- A string of hex digits (no quotes, no framing, just the digits)
- A string of decimal digits followed by the letter **N** (Example: Both **10** and **16N** mean sixteen.)
- A scaled decimal number (Examples: **4K, 12G** etc.)

Regardless of the number's form, it must resolve to a value between **0** and (2**64)-1. The default offset is zero.

Example:

INTERVAL=DWORD OFFSET=20
This causes the search to test for a match at eight-byte intervals from the starting address, but at +20 bytes from the start of each interval. (this allows you to look for a particular value in a field in a control block that is always on an 8-byte boundary).

HITLIMIT=decimalnumber
HITLIMIT=NONE
Normally, if a search finds a match, the search will stop at that first match. However with this operand, you can cause the search to continue until it has found the nth match.

decimalnumber: This must be a decimal number specifying the number of hits that have to occur before the search will stop. It must be in the range of 1 to 1000. (It may not be a hexadecimal number, nor a scaled decimal number, and it must not be trailed by an N.) The default value is one.

NONE this indicate that no limit on the number of hits is to be applied. The **FIND** command will only stop at the end of the search range. If the **EQUATENAMES=rootname** operand is also used, no leading zero's are inserted in the names of the generated equates.

EQUATENAMES=rootname
If you like, you can cause the **FIND** command to assign numbered equates to each match that the search finds. For each match found, an equate will be created formed by appending a sequence number to the given root name. The rootname must be valid for equate names:
- It must consist entirely of alphanumeric or national (@ # $) characters.
- underscore characters (_) also can be used.
- The first character may not be numeric.
- The maximum length of an equate name is 64 characters. So the rootname must be short enough that after the sequence number is appended, the 64 character limit

is not exceeded. (The sequence number length ranges from one to four characters, if the **HITLIMIT=** value is decimal, and 1 to 6 characters with no leading zeroes if **HITLIMIT=NONE** has been specified)
Note that only the first 999999 hits will have an equate generated.

Each time the **FIND** command is issued, the sequence numbers restart at one. (This is true for the **repeat FIND** process too.)

Each time the **FIND** command is issued, all previously existing equates (if any) that match the given rootname are purged.

The width of the sequence number that is appended will match the width of the number given for the **HITLIMIT** value. (if **HITLIMIT=NONE** is specified, the sequence number will be as wide as necessary; in this case there will be no leading zeros)

Example:

HITLIMIT=37 EQUATENAMES=HIT
The search proceeds until one of the following occurs:
- The search distance or ending address has been reached.
- The user has sent an attention signal.
- 37 matches have been found.

At each match location, an equate of the form **HITnn** is created. The first such will be **HIT01.** If any prior **HITccccc** equates existed, then they are purged before the search starts. Note, this purge will purge **ALL** equates that start with the string **HIT,** not just numbered equates.

DISPLAY or **DISPLAY=nnnnn**
FORMAT or **FORMAT=nnnnn**
When the **FIND** command completes, if at least one match has been found, then a display of storage will be produced, showing the last match's location.

The **FIND** command will use either a **DISPLAY** command or a **FORMAT** command to produce its final display. These operands allow you to control which command is used.

Normally, the size of the display generated by the **DISPLAY** or **FORMAT** command will be sufficient to fill up the display window in which it appears. (See **HELP FULLSCREEN WINDOWS** for information about display windows.) However, you can override that by appending a number **(=nnnnn)** to the DISPLAY or FORMAT keyword. **nnnnn** may be a decimal number ranging from 0 to 65,535 (64K-1). It gives the size (in screen lines) of the display to be produced.

If these operands are omitted, then the default is to use the **DISPLAY** or **FORMAT** command that was used by the previously issued **FIND** command. If this is the first **FIND** command, then the default is to use the setting saved in the session profile. This setting can be viewed and changed via the Profile Menuing System. See **HELP PROFILES MENU** for more information.

SHOWMCODE
HIDEMCODE
CURRENTMCODE
These operands control whether or not source image displays formatted via

ADATA maps will include source code create by certain macro expansions:
- **SHOWMCODE:** The expansions of all macros are shown.
- **HIDEMCODE:** The expansions of all code generating macros are suppressed. (Data generating macros, such as control block mapping macros, are unaffected.)
- **CURRENTMCODE:** The expansions of all code generating macros are suppressed **except** if the macro expansion currently being displayed includes the error level or retry level PSW address. In this case, the expansion is shown. (Data generating macros, such as control block mapping macros, are unaffected. They are never suppressed.)

Note the following:

- The intent of the **CURRENTMCODE** operand is to improve the display of code written using structured programming macros.

- Also, the hiding of macro expansions is effective only for storage formatted via **ADATA** maps. Storage formatted via SYM data maps is unaffected.

- Finally, the hiding of macro expansions does not occur for dsect maps that map data areas and control blocks.

For detailed information about what happens when macro expansions are suppressed, see **HELP MAPS ADATA MACROS.**

INSTRUCTIONS
DATA
NOBIAS
When the **EQUATENAMES=** operand is given, these operands cause the **DATA** or **INSTRUCTION** attribute to be assigned (or **not** assigned) to the generated equates. **NOBIAS** is the default. See **HELP COMMANDS EQUATE** for more information.

These operands also may affect the storage display produced in the event that a match is found. If such a display is produced, and if the **FORMAT** command is used to produce that display, then these operands affect the initial formatting bias of that display. (If these operands are all omitted, then the initial formatting bias will be controlled by the **SET FORMAT** command settings.) For more information, see **HELP COMMANDS FORMAT.**

Note, **HEXDATA** is an alias for **DATA;** however, it is a legacy operand that may be removed in a future release.

NARROW
WIDE
These allow you to control whether the **FIND** command's final storage display will be wide or narrow. If **WIDE,** then the display will show up to 8 words (32 bytes) of storage per line. This is suitable if your display terminal is set to be at least 136 columns wide.

If **NARROW,** then the display will show only 4 words (16 bytes) of storage per line. This is suitable for displays that are set to show only 80 characters per line. (For information about setting up wide displays, see **HELP FULLSCREEN TERMINALS.)**

If omitted, then the default width, as established by the **SET FORMAT** command, will

be used. (This default can be displayed by the **LIST FORMAT** command.) See **HELP COMMANDS SET FORMAT** for more information.

STATISTICS or **STATS**
NOSTATISTICS or **NOSTATS**
Z/XDC maintains several statistics about the search process. These operands allow you to control how much statistical information is displayed:

NOSTATISTICS or **NOSTATS:** Only two lines of information are displayed, the amount of storage scanned and the number of matches found.

STATISTICS or **STATS:** Additional information is displayed.

NOCB
CB
CB=hexlength
These operands control the location used when recording a hit.

If the **CB** or **CB=hexlength** operands have been specified then:
- The location equated is the start of the control block, which is the hit location backed up by any offset value.
- Any storage display of the last hit will also be backed up from the hit location by any offset value.
- when processing a **repeat FIND** where the new **FIND** command picks up from where the previous **FIND** command finished, the starting position calculation **does not** subtract any offset value from the **CDP** address.

Additionally, if the **CB=hexlength** operand is in effect, the storage display of the last hit will be limited in length by the control block length and any number of lines specified by the DISPLAY= or FORMAT= operands will be ignored.

If the **NOCB** operand is in effect then no changes (as documented above) are done.

If no CB related operands are specified, processing is as if the **NOCB** operand was specified.

PAGEOPS=option
When scanning virtual storage in the host system, the **FIND** command will reference many pages of storage that would otherwise not be referenced.

To reduce the effect of the **FIND** command on the set of referenced pages, the **PAGEOPS=option** operand can be used to control how these references are processed.

Note that this operand only has an effect when the **FIND** command is directly referencing storage. If scanning a copy of storage, no paging operations are done. (If the **STATS** option is specified, the output will indicate the strategy used).

If paging operation are to be done, each 4M piece of storage that has had no hits found will be processed as requested. If that piece of storage has had any hits, no paging operations are done.

The following options can be specified:
ALL - All paging operations (page out and page release) will be done.
NONE - No paging operations will be done.
OUT - Only page out operations will be done.
O - The same as **OUT.**
OU - The same as **OUT.**
RELEASE - Only page release operations (where a page is all binary 0) will be done.
OUTONLY - Page out operation will be done, and page release operations will also cause a page out to be done.
OO - The same as **OUTONLY.**

The default processing if this operand is not specified is as if **PAGEOPS=NONE** was specified.

This operand can also be specified as **PO.**

Options can be abbreviated as long as they remain unique.

XAMO
XCMO
XPMO
XSMO
IAMO
ICMO
IPMO
ISMO
These operands can be used to control which memory objects will be searched. If none of these operands are specified, processing is as if **IAMO** is in effect.

The **XcMO** operands are used to control exclusion, and the **IcMO** operands are used to control inclusion. (c represents a memory object class. Memory object classes are explained below.)

If a particular class of memory object is neither included or excluded, it is treated as being excluded.

The memory object classes are:
A - All memory object classes.
C - Common memory objects.
P - Private memory objects.
S - Shared (into this region) memory objects.

ALLABOVEBAR
This operand can be used to make all storage above the bar (i.e. > (2**32)-1) eligible for scanning (of course, the in-use start and end addresses are relevant to whether or not the above the bar storage is scanned at all).

If this operand is used, the **IcMO** and **XcMO** operands are ignored.

Scan technique

The **FIND** command will scan storage using the fastest technique that it can.

If scanning the host system (i.e. not a dump) and one of the following cases applies, the **FIND** command will scan the actual storage without copying it:
- The home address space.
- a dataspace owned by the home address space.
- real storage.
- common storage only.

In all other cases scanning will retrieve storage from the target aspace/dataspace/etc. and scan the copy.

The search string (and any masks or upcasing operations) is pre-processed to determine:
- The value(s) and number of first-byte matches.
- The value(s) and number of second-byte matches.

Based on the pre-processing information, a scan technique is chosen as follows:

- For searches against actual (i.e. not copied) storage:
  - The first byte of a possible match is located using one of these instructions:
    - **SRST** (=1 possible match value and **INTERVAL=1)**
    - **TRTE** (>1 possible match value and **INTERVAL=1)**
    - **CLI** (=1 possible match value and **INTERVAL>1)**
    - **TRT** (>1 possible match value and **INTERVAL>1)**

- For searches against copied storage:
  - The first byte of a possible match is located using one of these instructions:
    - **CLI** (=1 possible match value)
    - **TRT** (>1 possible match value)

- If a first byte match is found, the remaining part of a possible match has the masks and upcasing applied (in a work area!) and then that value is compared to the search string:
  - If a match occurs then the hit is counted (and the **FIND** command may terminate).
  - If no match is found then the search continues.

Note that the remaining part of a possible match is optimized:

- If the search string is only 1 byte in length, there is no additional compare.

- If the search string is exactly 2 bytes long, the second byte match will be either a **CLI** or **TRT** instruction.

- In other cases (the search string is at least 3 bytes long) the value is only copied to a work area if masks must be applied and/or upcasing must be done.

- If a compare must be done, the compare always starts at the second byte of the (possibly copied to a work area) value.

Note that when storage is being retrieved from a dump the storage will not be cached by **dump/XDC,** and the **FIND** command will be sensitive to the actual storage contained on the dump.

Long Running Searches

The storage search performed by the **FIND** command can take a rather long time. There are several reasons for this, one of which is that **FIND** processing is both CPU intensive and storage intensive, and so your address space may become discriminated against by the System's Workload Manager.

However, for some searches, Z/XDC does try to help with the page load created by the search process. Provided that paging operations have been enabled, After every 4 megabytes have been scanned (or at the end of an area), provided that no hits were found in the piece of storage being processed, Z/XDC may either page out or page release pages of storage that it has referenced. Hopefully, this will keep your program's working set size from growing beyond reasonable limits.

Nevertheless, If you feel that a search is taking more time than it's worth, then you can terminate the search by pressing the **ATTN** key (or **PA1** key) at your terminal. This causes the **FIND** command to abort, causing Z/XDC to display a message and then await further commands.

When the search ends or is aborted, the "Current Display Pointer" is set to point to the last address tested. This is regardless of whether the search succeeds, fails, or is aborted.

Spurious Matches

Sometimes the **FIND** command will suffer spurious matches. This can occur when FIND's scan point reaches an ISPF buffer, a display buffer, a log buffer or some other system buffer that happens to contain a copy of the **FIND** command text that you typed. This is especially true if you typed your search argument as a character string.

In some cases the buffer in which a spurious match occurred might be of such a transient nature that by the time Z/XDC's display routines read it out for display, its contents have changed. When this occurs, the user might very understandably ask the question: "HUH!?!?". Well, don't worry, you're not seeing things, and Z/XDC's not broken. That's just the kind of thing that can happen when displaying volatile storage.

Z/XDC has a considerable amount of code to recognize and reject those spurious matches that might occur within its own processing buffers; however, there really isn't any reasonable way for Z/XDC to recognize and reject all spurious matches. So you will have to use your own judgment as to which matches matter and which do not.

If spurious matches become a significant problem, then you can reduce the problem by coding the search target as hex data instead of text characters.

Areas:
The **FIND** command only scans allocated areas.

Within a virtual address space, or common storage, these areas include:
- The PSA.
- low private.

- CSA.
- SQA.
- (F/M/P)LPA.
- low nucleus.
- high nucleus.
- E(P/M/F)LPA.
- ESQA.
- ECSA.
- high private.
- memory objects (above the bar). (there is a separate area for each memory object).

Within a dataspace, the entire dataspace is processed as one area.

Within real storage, the entire real storage range is processed as one area.

Examples:

FI '00C',IEANUC01 XSQA
This searches the Nucleus for the string C'00C'. The search will probably find the UCB for a card reader. (Huh? What's that you say? "What's a card reader??!" My, how times change.)

The search starts at the beginning of the Nucleus and ends at the start of 31-bit SQA.

FI 0A0A PSW? FORMAT;FI;FI

The effect of these three commands is to find the third occurrence of the REGMAIN SVC at or following the location pointed to by the retry level PSW. For each match, a display will be generated via an internally issued **FORMAT** command. In the first search, a maximum of 8 mebibytes of storage will be scanned. The second and third searches will start at the first byte following the point where the first and second searches stopped, and will still search 8 mebibytes.

FIND 0A0A PSW? INT=2 HITL=3 EQU=SVC10_# INST
This also will find the third occurrence of the REGMAIN SVC, but there are some differences:
- Only one **FIND** command is needed, not three.
- Thus, only one display will be produced, not three.

- Equates will be created at each of the three hits. They will be named SVC10_#1, SVC10_#2, and SVC10_#3, and they all will have the INSTRUCTION attribute.
- Only instances of X'0A0A' that are halfword aligned will be found. Instances of X'0A0A' that start on odd boundaries will be ignored.

FIND 90 MYPROG AND=F0 INT=2 HITL=1000 EQU=SCON_ HEX
This will find up to a thousand occurrences of halfword aligned bytes (assuming the starting address is halfword aligned) that have a nine in their first nibble and an arbitrary digit in their second nibble. The search will start at MYPROG's entry point and run for up to a 8 mebibytes (the default search distance). Each match will be labeled by an equate of the form SCON_nnnn. The first such equate will be

SCON_0001.

I use this sort of search sometimes to scan a program for s-cons based upon a particular register.

FI 'COLE' PRIVATE~ASID(JES2)
This finds the first occurrence of the string C'COLE' following the beginning of the private area in JES2's address space. **(PRIVATE** is a built-in equate name that points to the start of an address space's private area.)

FIND F'-738',R1? DISPLAY=2
This searches storage, starting at the location pointed to by R1, for the 4-byte hex value "FFFFFD1E" (the result of converting the decimal integer "-738" to a 4-byte wide binary value). If the value is found, then storage is displayed via Z/XDC's **DISPLAY** command. The generated display will show two display lines worth of storage.

FI 91040014 IEANUC01.CSVGETMD INT=2 ANDMASK=FFFF0FFF HIT=99 EQ=HIT INST
This scans, as follows, the CSVGETMD csect of the system nucleus for certain TM instructions:

- The following things are known about the desired machine instruction:
  - It is a TM instruction.
  - The TM is testing for an X'04' flag.
  - The flag is located at an offset of +X'14' into a control block.

  The following is not known:
  - What the base register is that points to that control block.

  Accordingly, an **ANDMASK** has to be used to remove the base register digit from the compares performed by the search.

- Search compares are performed starting at every second byte. Since the search starts on a halfword aligned boundary, the search checks only for halfword aligned strings.

- Up to 99 matches will be searched for.

- Numbered equates will be created labeling each hit. The equate names will start with the characters **HIT.** But first, all prior equates starting with **HIT** will be deleted.

- The equates will be assigned the **INSTRUCTION** attribute. (See **HELP COMMANDS EQUATE** for more information.)

FIND 3'+738',R1?
This searches storage for the 3-byte hex value "0002E2".

Other Commands

Other commands that can be used to display storage are:

- **DISPLAY** - Displays a contiguous chunk of storage in a hex-text (dump like) format.
- **FORMAT** - Formats a contiguous chunk of storage as either source code displays, machine instructions, or data fields, as appropriate.
- **SHOW** - Accepts a list of address expressions and formats a 1-line display for each expression given.
- **WHERE** - Formats an area of storage containing the current retry level PSW's resume execution address.

Help COmmands FOrmat

The **FORMAT** command formats and displays virtual storage. In general, the command attempts to decode and display machine instructions, but if the current address does not contain a valid opcode, then the display automatically switches to a raw hex-text (dump like) format.

The generated display is strongly affected by whether or not a map is defined for the area being formatted. When this is the case, information from the map is used to substantially improve the accuracy of the formatting.

The FORMAT command understands and uses the following kinds of maps when available:

- SYM data maps allow FORMAT to add source code defined statement labels and field names to the generated displays. They also aid Z/XDC in distinguishing Assembler statements from data fields.
- ADATA maps allow FORMAT to incorporate your actual Assembler program's source images into storage displays.
- DWARF data maps allow FORMAT to incorporate your XL C/C++ and Metal C program's source images into storage displays.

In order to use SYM data and ADATA maps, you must license Z/XDC's **asm/XDC** feature. In order to use DWARF data maps, you must license Z/XDC's **c/XDC** feature. See HELP SUPPORT FEATURESANDCAPS for more information about this.

You can use the **LIST XDC** command to see what features are actually licensed.

Syntax:

```
FORMAT addressexpression lines           SOURCE ADDRESSES DECIMAL     ...
F      omitted           LINES=lines     OBJECT OFFSETS   HEXADECIMAL
       ,                 BYTES=hexnumber BOTH
```

```
... ASCII  WIDE   INSTRUCTION  SHOWMCODE     POINTERS
    EBCDIC NARROW DATA         HIDEMCODE     NOPOINTERS
                  NOBIAS       CURRENTMCODE
```

Shortcuts: F

Notes:

Operands appearing above within columns are mutually exclusive.

All operands are optional.

If the addressexpression operand is given, then it must be first. All other operands may be given in any order.

If the addressexpression operand is omitted but other operands are given, then a comma must be used to indicate the omission.

Operands:

addressexpression
This gives the starting address of the storage to be displayed. When the command finishes...
- The **Current Display Pointer** (CDP) is set to this address.
- The **Next Display Pointer** (NDP) is set to point past the last byte displayed.

omitted
,
The display starts with the address pointed to by the **Next Display Pointer.** Usually, this points just past a previous display.

When a FORMAT command (or other storage displaying command) completes:
- The **Current Display Pointer** is set to the display's starting address.
- The **Next Display Pointer** is set to point past the last byte displayed.

This **NDP** facility simplifies the displaying of successive storage locations: Just repeatedly issue an **F** command with no operands. This will produce successive displays of storage, all under the control of the initial FORMAT command's operations. For more information, see HELP ADDRESSING IMPLICIT.

If the **addressexpression** is omitted and other operands are given, then a comma must be used to show the omission.

The following operands may be given in any order. They must, however, follow the given address expression, if any.

Also, all of the following operands (except **lines, LINES=** and **BYTES=)** serve to override the corresponding default values established via the **SET FORMAT** command.

lines
LINES=lines
BYTES=hexnumber
These control the size of the display. The **lines** and **LINES=** operands do so in terms of the number of lines (rows) of instructions, data and comments the display is to contain. The **BYTES=** operand does so in terms of the amount of storage that is to be displayed:

lines
LINES=lines
These two operands function identically; however the **LINES=lines** form is preferred because eventually support for the **lines** form may be discontinued.

The **lines** value must be a decimal number specifying the number of display lines to generate containing instructions, data and comments (i.e. not including the Location Interpretation header lines). This value must be in the range of 0 to 65,535 (64K-1). When omitted, either the default value set by the **SET LINES** command is used (for nonfullscreen displays), or the length of the current window is used (for fullscreen mode displays).

BYTES=hexnumber
The **number** must give the amount of storage to be displayed. It can be any of the following:

- A string of hex digits (no quotes, no framing, just the digits)

- A string of decimal digits followed by the letter **N** (Example: Both **10** and **16N** mean sixteen.)

- A scaled decimal number (Examples: **4K, 12G** etc.)

Regardless of the number's form, it must resolve to a value between **0** to **1FFFE0:**
- The very strange X'1FFFE0' limit is (64K-1)∗32:
  - 64K-1 is the LINES= limit.
  - 32 is because one line can display up to 32 bytes of storage.

The number of display lines needed to produce a BYTES= driven display can vary.

The **lines, LINES=** and **BYTES=** operands are mutually exclusive.

SOURCE
OBJECT
BOTH
These operands control the use of maps that might be available for controlling the formatting of the storage being displayed:

SOURCE
Causes storage to be formatted under the control of a source level map, if available. Notes:

- ADATA and DWARF data maps are source level maps. SYM data maps are not. See the various subtopics of HELP MAPS for more information.
- When **SOURCE** is given or defaulted, and a source level map is not available for the storage being displayed, then the storage is displayed as if **OBJECT** had been specified.

OBJECT
Causes storage to be formatted by disassembly. The disassembly will be corrected by and annotated by symbols (if any) from the following sources:

- Equates created by the **EQUATE** command
- Symbols from **SYM** data maps
- Symbols from **ADATA** maps

BOTH
Causes storage to be formatted by both SOURCE and OBJECT; source level statements will be interspersed with statements generated by disassembly.

Note, when **DATA** is explicitly given, both **SOURCE** and **BOTH** are ignored in favor of **OBJECT.**

POINTERS
NOPOINTERS
When storage is being formatted via disassembly (i.e. when either **OBJECT** or **BOTH** is in effect), if a line of display is showing a snippet of data that is 3, 4 or 8 bytes wide, then this **POINTERS** operand will cause Z/XDC to attempt to resolve that snippet as a storage pointer. If successful, it then will display that resolution as a comment on the display line. Notes:

- The **NOPOINTERS** operand prevents these resolutions.
- The factory default is **POINTERS.**
- When very complex structures of **floating maps and equates** are present, the **POINTERS** setting may result in slower displays. In that case, you may wish to use **NOPOINTERS** instead. (For more information about floating maps and equates, see **HELP MAPS FLOATING.)**

SHOWMCODE
HIDEMCODE
CURRENTMCODE
These operands control whether displays of the underlying machine code are intermixed into source image displays. Their effects differ a bit between **ADATA** maps and **DWARF** data maps:

- For **ADATA** maps:
  - **SHOWMCODE:** The expansions of all macros are shown.
  - **HIDEMCODE:** The expansions of all code generating macros are suppressed. (Data generating macros, such as control block mapping macros, are unaffected.)
  - **CURRENTMCODE:** The expansions of all code generating macros are suppressed **except** if the macro expansion currently being displayed includes the error level or retry level PSW address. In this case, the expansion is shown. (Data generating macros, such as control block mapping macros, are unaffected. They are never suppressed.)

- For **DWARF** data maps:
  - **SHOWMCODE** and **HIDEMCODE** have no effect other than to cancel the effect of CURRENTMCODE.
  - **CURRENTMCODE** causes the underlying machine code to be displayed only for the current language statement (i.e. only for the statement to which the retry level PSW points).

- For **SYM** data maps, these setting have no effect.

Note the following:

- The intent of the **CURRENTMCODE** operand is to improve the display of code written using structured programming macros.

- Also, the hiding of macro expansions does not occur for dsect maps that map data areas and control blocks.

For detailed information about what happens when macro expansions are suppressed, see HELP MAPS ADATA MACROS.

ADDRESSES
OFFSETS
These operands control whether the far left column of the display (the "address" column) will show storage addresses or offsets:

ADDRESSES
Causes each line of data displayed to be prefixed by that data’s virtual address.

OFFSETS
Causes each line of data displayed to be prefixed by that data’s offset from the start of an appropriate including object (load module, csect, dsect, or equate).

DECIMAL
HEXADECIMAL
For disassembled machine instruction displays, these operands control whether the displacement fields are formatted as decimal or hexadecimal numbers:

DECIMAL
Causes the displacement fields to be displayed as decimal numbers.

HEXADECIMAL
Causes the displacement fields to be displayed as hexadecimal numbers.

ASCII
EBCDIC
For data displays, these operands control whether the text portions of data displays are to be interpreted using the **ASCII** or **EBCDIC** character set:

ASCII
Causes data displays to show the **ASCII** interpretation of the displayed storage. To visually indicate this, the text portion of each display line will be framed with **vertical bars (|)** instead of asterisks.

EBCDIC
Causes data displays to show the **EBCDIC** interpretation of the displayed storage. To visually indicate this, the text portion of each display line will be framed

with **asterisks (*).**

More information on EBCDIC and ASCII can be found at **HELP CHARACTERSETS**

WIDE
NARROW
These operands control whether data displays are to be wide or narrow:

WIDE
This causes data displays to show up to **32 bytes** of storage per line (not just 16). This option works best when your terminal's display is at least 136 columns wide. For more information, see HELP FULLSCREEN TERMINALS.

NARROW
This causes data displays to show only **16 bytes** of storage per line. This width is appropriate for terminals with 80 character wide display lines.

INSTRUCTIONS
DATA
NOBIAS
The **FORMAT** command generally attempts to format storage contents as instructions, but this bias can be influenced by many factors, such as whether or not a PSW points to the storage being formatted, as well as the attributes of maps, fields, and equates that label the storage being displayed.

As a part of the management of this process, the **FORMAT** command maintains a concept called the **formatting bias** which, for a given displayed line, retains information about how the preceding line was formatted and, therefore, contributes information about how the current displayed line should be formatted. In other words, if a display line was formatted as data, then the **FORMAT** command will be biased towards formatting the following display line also as data, unless new information (such as label attributes) suggests otherwise.

The **INSTRUCTION, DATA,** and **NOBIAS** operands set the initial formatting bias for the **FORMAT** command as follows:

INSTRUCTION
The **FORMAT** command's initial formatting bias will be to interpret storage contents as being machine instructions. This will be done even if other factors might suggest that the storage contents should be interpreted as data. This will **not** be done, however, if the initial opcode is invalid.

DATA
The command's formatting bias will be to interpret storage contents as being data for the **entire** display. This will be done even if other factors might suggest that the storage contents should be interpreted as machine instructions. Note, when **DATA** is explicitly given...
- **SOURCE** and **BOTH** are both ignored in favor of **OBJECT.**
- **DATA** is not just the initial bias. It is coerced for the entire display.

NOBIAS
The command's initial formatting bias will not be forced. Absent other factors, the storage contents will initially be interpreted as machine instructions, but

if other factors suggest otherwise, then those factors won't be ignored.

The **INSTRUCTION** and **NOBIAS** operands set only the **initial** formatting bias. Once the display gets going, the formatting bias for the second and subsequent display lines will be influenced only by the normal factors discussed above.

Storage Protect Key Display

The **FORMAT** command shows the storage protection key as a one-character hexadecimal field just to the right of the address field.
Following the storage protect key may be additional characters:
f - The 'f' character indicates that the storage is fetch protected.
s - The 's' character indicates that the storage is store protected. (there are additional comments about store-protected storage below).
r - The 'r' character indicates that the storage may be redacted. (this character can only be seen when processing a redacted dump in dump/XDC). A redacted area is indicated by the storage being all binary zero. Note that it is impossible to distinguish storage that has a value of binary 0 from storage that has been redacted.

The "Store Protected" indicator, **s,** is displayed whenever Z/XDC detects that the storage being displayed has one or another of the following special protections:

- **Low Address Protection** (LAP): This is a special kind of protection that the Operating System uses to specifically protect the first 512 bytes of each of the first two pages of both real and virtual storage. It prevents the storage from being altered by any program, regardless of that program's state or key. (LAP does not apply to data space storage.)

- **Store Protection:** This is a kind of protection that applies to the following areas of storage:
  - The Pageable Link Pack Area (PLPA)
  - The read-only areas of the System Nucleus (IEASYS0x)
  - Often (but not always) load modules that have the rentrant (RENT) or refreshable (REFR) attribute.
  - Any area of storage that has been set to "read-only" by the PGSER or IARVSERV macros.

  When a page of storage has been marked as being "read-only", that storage cannot be directly altered by any program, regardless of that program's state or key. (Note, "Store Protection" does not apply to real storage. Thus, it cannot protect a virtual storage page when that page is accessed via its real address.)

- **Access List Entry Protection:** When an address space or a data space is accessed via an ALET that "points to" an Access List Entry (ALE) having its "fetch only" flag on, then accesses via that ALE are store protected. Note, accesses to that same space via a different ALE might not be store protected.

Examples:

F MYPROG,4 ASCII
F MYPROG,LINES=4 ASCII
These commands both display four lines (probably four machine instructions) located at the start of **MYPROG.** If those four lines include storage that has to be displayed

in raw hex-text format, then the text portions of the display will show the ASCII interpretation of that storage.

F ,WID
This displays storage following the storage shown in the previous display. The default number of lines is generated. For data displays, The text portions (if any) of the display revert to showing the EBCDIC interpretation (assuming that **SET FORMAT EBCDIC** is in effect), and the data will be displayed in wide format (up to 8 words per line).

F +0,8
F +0 LINES=8
These commands both display the same storage shown in the previous display, but now 8 lines are generated. (The **+0** refers to the **Current Display Pointer.)** Also, the display will be either wide or narrow according to the current **SET FORMAT** setting.

F ,OFF BYTES=100 BOTH HEX
This displays the next 256 bytes (X'100' bytes) of storage. **OFF** causes each line to be prefixed by an offset rather than a virtual address. If a source level map is available, then **BOTH** causes the source statements from the map to be displayed interspersed among the disassembled machine instructions. **HEX** causes machine instruction displacement fields to be displayed as hexadecimal numbers (instead of decimal numbers).

DMAP XDCMAPS.CVT
USING CVT 10?
MAP 1000000
FORMAT CVT OBJECT POINTERS
The Communications Vector Table contains a very large number of pointers to other control blocks and service routine throughout z/OS. If you want to easily see what those are, adding **POINTERS** to the **FORMAT** command will do the trick. Notes:

- Adding **POINTERS** to the **FORMAT** command is not necessary if **SET FORMAT POINTERS** is in effect.

- The **MAP 1000000** command loads the binder map for the System Nucleus. This makes the **FORMAT CVT POINTERS** display more interesting since many of the objects pointed to by the CVT are located in the System Nucleus, and the binder map allows the pointers to be resolved down to the actual name of what's being pointed to.

More Information

Other commands that can be used to display storage are:

DISPLAY - Displays a contiguous chunk of storage in a hex-text (dump like) format.

- **SHOW** - Accepts a list of address expressions and formats a 1-line display for each expression given.
- **WHERE** - Formats an area of storage containing the current **retry** level PSW's resume execution address.
- **EWHERE** - Formats an area of storage containing the Current **error** level PSW's resume execution address.
- **FIND** - Scans storage for a given string of data.
- **LIST VARIABLES** - Lists XL C/C++ and Metal C variables currently in scope.
- **LIST VSTACK** - Lists XL C/C++ and Metal C stack frames that contain variables.

Help COmmands FReemain

This command cannot be used in Foreign Address Space Mode.

The **FREEMAIN** command releases a given area of virtual storage. In addition it deletes all fixed-location equates that **start** at or within the area being released. (Floating equates are not deleted regardless of whether or not their locations happen right now to resolve into the freed storage.)

In determining which equates to delete, the **FREEMAIN** command examines only an equate's starting addresses. The lengths of the equates are neither examined nor adjusted. Thus:

- Equates starting prior to but overlapping the area being freed are neither deleted nor adjusted.
- Equates starting within but extending beyond the area being freed **are** deleted.

Based on the given address, the **FREEMAIN** command determines for itself which subpool contains the storage to be freed. The command then attempts to free the given storage from that subpool.

The attempt will succeed or fail according to whether or not the user's program has the ability to GETMAIN/FREEMAIN from the subpool. This ability varies according to whether the user program is running authorized or non-authorized:

When running non-authorized, storage from the following subpools can be freed:

- Subpools 0-127 when owned by or shared to the current task.
- Subpools 131 and 132 when the access key of the target storage is permitted by the current task's current key-mask.

When running authorized, storage from the following subpools can be freed:

- Subpools 0-127 when owned by or shared to the current task.
- Any storage in subpools 131 and 132 regardless of access key.
- Any system subpools, i.e. any LSQA, CSA, and SQA storage.

Syntax:

```
FREEMAIN addressexpression length subpoolcategory
         equatename
```

Operands

addressexpression
This gives the address of the area to be freed. The FREEMAIN SVC will fail if this address is not doubleword aligned.

length
This is a hexadecimal value giving the length of storage to be freed. The FREEMAIN SVC rounds this value up to an 8-byte multiple. This operand is required **unless** the address operand is a **pure** equate name in which case the length operand can be omitted and the length of the equate is used in its place. (The command will fail, of course, if the equate is **not an area equate,** i.e. if its assigned length is zero.)

equatename
If the given address expression actually is an unmodified name of an equate symbol, then the length operand can be omitted, in which case the area to be freed and its length are determined by Z/XDC from the attributes of the given equate.

subpoolcategory
This operand can be used when Z/XDC is running authorized, and you wish to free storage from a System subpool. It confirms the category of storage to be freed: **LSQA, CSA** or **SQA.**

If the address expression or equate name you have given identifies storage located within LSQA, CSA, or SQA, then you must prove to Z/XDC that this is your intent by providing this operand. This requirement is intended to reduce the likelihood that such storage will be freed by mistake.

Note that attempts to free LSQA, CSA and SQA storage can succeed only when Z/XDC is running authorized. Such attempts will always be failed by the System when Z/XDC is running non-authorized.

The following is a list of the recognized **subpoolcategory** keywords:
- **LSQA** - Permits the freeing of LSQA and extended LSQA.
- **CSA** - Permits the freeing of CSA and extended CSA.
- **SQA** - Permits the freeing of SQA and extended SQA.

These keywords may not be abbreviated.

Examples:

FR +0 7
8 bytes of storage (not seven), starting at the location pointed to by the "Current Display Pointer" (see **HELP ADDRESSING IMPLICIT)** are freed from whatever private area subpool the 8 bytes are contained within. In addition, if any fixed-location equate names are defined starting at or within the 8 bytes, then those equate names will be deleted.

Note, the **FREEMAIN** command will fail if the owning subpool is not one within which the user's program is allowed to GETMAIN or FREEMAIN.

FR AREA#3
The storage represented by the **AREA#3** equate is freed. The equate itself is deleted along with any other fixed-location equates that may have been defined to start within the area.

Note, **AREA#n** equates are automatic equates created by Z/XDC's **GETMAIN** command to label both the location and length of the storage that you've requested. In this example, the **FREEMAIN** command is taking advantage of the fact that the length attribute of the **AREA#3** equate matches the length of storage that was obtained and is now being freed.

FR F00000 100 SQA
If Z/XDC is running authorized, and if location X'00F00000' falls within the System Queue Area (SQA), and if that storage currently is allocated, then this command frees it. (This command fails when **any** of the above listed conditions are not met.)

Help COmmands GEtmain

This command cannot be used in Foreign Address Space Mode.

The **GETMAIN** command allocates an area of virtual storage and tells you where it is. Once allocated, Z/XDC does not keep track of it; you must do that for yourself. To help you do this, though, Z/XDC does automatically create two equates that can be helpful. See **Automatic Equates** (below) for more information.

Syntax:

```
GETMAIN length subpoolnumber  RMODE=24/31  KEY=nn
               SUBPOOL=nnn    LOC=24/31       =hh
               SP=nnn         HIGH
                              LOW
```

The **length** operand is required. All others are optional.

Operands appearing within columns are mutually exclusive.

length
This must be a hexadecimal value giving the size of the area to be obtained. The STORAGE OBTAIN service automatically rounds this value up to an 8-byte multiple.

subpoolnumber
SUBPOOL=subpoolnumber
SP=subpoolnumber
Subpoolnumber must be a decimal value identifying the subpool from which the storage is to be obtained.

Notes:

- If Z/XDC is authorized, then the z/OS system normally interprets references to subpool 0 to mean subpool 252. However, the GETMAIN command anticipates this system "quirk", so if you ask for subpool 0, then you will get subpool 0 regardless of whether or not Z/XDC is running authorized.

- If you want subpool 252, then you will have to specify subpool 252 on the GETMAIN command. Example: "GETMAIN 1000 252"

- If you request subpool 240 or 250 (when Z/XDC is authorized), then you will get subpool 0 instead, since this is the z/OS defined meaning of "subpool 240" and "subpool 250" requests.

- If Z/XDC is running authorized, then the user may request the number of any legal subpool that z/OS supports. On the other hand, if Z/XDC is running non-authorized, then only subpools 0 through 127 and subpools 131 and 132 may be requested. This is a system restriction.

- When no subpool number is provided, **subpool 0** is used by default.

HIGH
RMODE=31
LOC=31
These operands all indicate that the requested storage is to be obtained from 31-bit memory.

LOW
RMODE=24
LOC=24
These operands all indicate that the requested storage is to be obtained from 24-bit memory.

Notes:

- When the RMODE-setting operands are all omitted, the requested storage is obtained from 24-bit or 31-bit memory according to the retry level PSW's addressing mode.

- The program's addressing mode can be displayed by the **LIST PSW FORMAT** command (and changed by the SET PSW command).

KEY=nn
KEY=hh
This operand requests that the obtained storage be assigned to a particular storage key. The key value can be given either as a decimal number of as a hex digit. For

example, the following all request that storage be assigned to key eleven.

```
KEY=11
KEY=0B
KEY=B
```

Notes:

- Specifying KEY= is effective only for **multi-key** subpools. It is ignored otherwise. Detailed documentation about this can be found in IBM's **MVS Programming: Authorized Assembler Services Guide.** You can find it online by Googling for its title.

- When running non-authorized:
  - The System will only accept key values permitted by your execution environment's key-mask. Typically, you would be limited to **KEY=8** or **=9.**
  - The multi-key subpools available to non-authorized programs are subpools **131** (fetch protected) and **132** (not fetch protected).
  - The storage for all other non-authorized subpools (i.e. subpools **0-127)** will normally be assigned to **key 8** regardless of whether or not you provide a KEY= operand.

Security

Prior to actually performing the STORAGE OBTAIN request, the GETMAIN command issues a call to system security to see if the user is permitted access to the specified subpool. the GETMAIN command will then either proceed or abort according to whether the access is permitted or denied by security. See HELP SECURITY for more information.

Automatic Equates

When the **GETMAIN** command allocates an area of virtual storage, two automatic equates are created to help you keep track of it:

AREA#n
These equates uniquely label all storage areas obtained by the **GETMAIN** command. Each time the command is issued, **n** is incremented, so prior **AREA#n** are not replaced. (Prior **AREA#n** equates can, however, be deleted by the **FREEMAIN** command.)

AREA
This equate also is created to label the obtained storage. If a prior **AREA** exists, then it is replaced, so this equate winds up always labeling the most recently obtained storage.

Note, the **AREA** equate provides a predictable name by which command scripts can refer to storage obtained by the **GETMAIN** command. See HELP COMMANDS READ for information about reading command scripts.

Examples:

GE 7 H
8 bytes of storage are allocated from subpool 0 (regardless of whether or not Z/XDC is authorized). The storage is obtained from "above the line" (31-bit storage). The address of the allocated storage is displayed. An **AREA#n** equate is created to point to it. An **AREA** equate also pointing to it also is created. If an **AREA** equate already existed, then that prior definition is automatically deleted.

GE 20,78
32 bytes (decimal) of storage are allocated from subpool 78 (decimal). The storage is obtained from either above or below the line according to the current addressing mode of the program being debugged. an **AREA#n** and an **AREA** equate are (re)created to point to it.

Help COmmands GO

The **GO, GOT** and **GOX** commands all cause user program execution to resume under the control of the retry level RB. Newer RBs (including the error level RB if different from the retry level) are terminated. See HELP EXECUTIONLEVELS for more information about this. (In the case that Z/XDC has been entered as a Trap Handler, execution will resume at the same RB level as when Z/XDC was entered.)

Also, the **TCBCMP** field is zeroed.

The **TCBCMP** field is 4 bytes long and includes the **TCBCMPF** and **TCBCMPC** subfields. Normally, the TCBCMPC subfield contains the most recent ABEND code, and the TCBCMPF subfield contains dump control flags and other flags related to that ABEND.

But GO processing zeros these fields because when you issue a GO/GOT/GOX or TRACE command, you are indicating that your program's execution is to be resumed. (In the case of tracing, that resumption will be brief.) Therefore, the ABEND (s0C1 in the case of **0C1-type** tracing or trapping) is now "repaired" (supposedly). Consequently, all ABEND descriptions and controls contained within the TCBCMPx fields are now obsolete. Therefore, it would be misleading to programming if that information were allowed to remain.

Note, as soon as the next ABEND occurs (s0C1 in the case of **0C1-type** tracing or trapping), the System will store new ABEND information into the **TCBCMPx** fields prior to passing control to Z/XDC again.

On the other hand, if **SET TRACE TRAP2** is in effect, then breakpoints will cause the Z/XDC Trap Handler to be driven, so there will be no ABEND, so there will be no change to the **TCBCMPx** fields.

The **GO** command (and friends) can be issued only when Z/XDC is **not** running in Foreign Address Space Mode **(FASM** mode). For more information about **Foreign** vs. **Local** Address Space Mode, see HELP VIRTMEM XDCACCESS and its subtopics.

GO vs GOT vs GOX Differ Only in TSO (Not in the Batch)

If you are debugging a program that is running within TSO (i.e. you are not

debugging a background program via cdf/XDC), and if Z/XDC's Fullscreen Support is turned on, then:

- **GO** causes the display screen to be cleared prior to resumption. This is desirable if you anticipate that your program will want to write a display and then read a response prior to the next time Z/XDC receives control.

- **GOT** causes the user program to be resumed without clearing the display screen. This is desirable if you expect that your program will not attempt to issue any displays prior to the next time Z/XDC receives control. ("GOT" means "GO for tracing".)

- **GOX** causes Z/XDC to attempt to restore whatever information was being displayed at the terminal prior to when Z/XDC received control.

  But owing to the very poor documentation of the hodge-podge of interactions between TSO, session manager and ISPF, this restoration attempt will generally not be very satisfactory. See HELP XDCSRVER CDF TSO for a better approach for dealing with terminal screen display conflicts.

These distinctions between GO, GOT and GOX occur only when debugging a program that is running within TSO. If you are debugging a program using **cdf/XDC,** all distinctions **disappear,** and the GO, GOT and GOX commands behave identically.

This is true even if you are connected to that batch job debugging session via **option 3** of the **Z/XDC Startup Panel** in ISPF.

For more information about batch job debugging, See HELP XDCSRVER CDF.

Syntax:

GO	**addressexpression**	**REMOVEXDC**	**RELEASECAPS**	**FORCE**
GOT	**NOWHERE**	**NOWHERE**	**NOWHERE**	**NOWHERE**
GOX	**omitted**	**KEEPXDC**	**KEEPXDC**	**omitted**
		omitted	**omitted**	

Notes:

- All operands are optional.

- Operands appearing within columns are mutually exclusive.

- When multiple operands are given, they may be given in any order.

- The **NOWHERE** operand is mutually exclusive with **all** other operands.

addressexpression (or omitted)
This gives the desired resume address. To learn about address expressions, see HELP ADDRESSING.

If the addressexpression operand is omitted (and NOWHERE also is absent), then the address pointed to by the **retry level PSW[E]** is used by default. In other words, the program generally will resume where it was interrupted. For more information, see HELP EXECUTIONLEVELS.

Tip: If the addressexpression you want to use happens to be indistinguishable from one of the GO command's keyword operands, then append a **+0** to make it distinguishable. Example:

- **EQUATE KEEP wherever** [Creates an equate named KEEP.]

- **GO KEEP+0** [The KEEP equate I just created is not distinguishable from the KEEP abbreviation of the KEEPXDC operand, but adding the **+0** makes it clear that I'm using KEEP here as part of an address expression, not as a keyword.]

REMOVEXDC

Under certain circumstances, this operand can be used to remove the current instance of Z/XDC from your program's recovery environment. (It does **not** remove all instances of Z/XDC.)

Generally, this operand is intended primarily to remove those instances of Z/XDC that have been established **dynamically** via the **HOOK** command.

REMOVEXDC addresses a problem that arises when a user program attempts to cancel its own recovery routine without knowing that you, during debugging, have used a hook to insert Z/XDC into the environment. When you are done debugging and wish to let the program go on its merry way, if:

(1) - The program eventually issues an **ESTAE[X] 0** to cancel its own ESTAE[X] (or a **SETFRR D** to delete its own FRR), and if

(2) - The hooked-instance of Z/XDC remains on the SCB queue or FRR stack,

Then the user program's "ESTAE[X] 0" or "SETFRR D" will cancel the **wrong** recovery routine, and the recovery routine that the program had thought it had canceled will remain in effect. This, of course, can lead to arbitrary failures. For more information, see **HELP HOOKS UNDOING.**

REMOVEXDC does not remove **all** instances of Z/XDC, and it can remove only **certain** instances of Z/XDC. Here are the rules:

- REMOVEXDC can remove only the **current** instance of Z/XDC. If other instances of Z/XDC are pending in the recovery stack, those remain in effect.

- REMOVEXDC can remove Z/XDC only if it is the environment's **newest** recovery routine. It cannot be used if the current instance of Z/XDC is running under any older STAE Control Block (SCB).

- REMOVEXDC can remove Z/XDC only if it is running as an **ESTAE,** an **ESTAEX,** or an **FRR.** It cannot be used if Z/XDC is running as an ESTAI, as an ARR or as any other type of recovery routine.

- REMOVEXDC can be used only if Z/XDC has been entered **directly** from the System (i.e. from the Recovery/Termination Manager [RTM]). It cannot be used if Z/XDC was **BASR'd** to from a user written ESTAE[X] routine or **jumped** to from a user written FRR.

Note, when Z/XDC is running as an ESTAE-type recovery routine, the **LIST ESTAES** command can be used to view detailed information about the recovery environment.

The instances of Z/XDC that are created by HOOK commands and scripts meet all of the above criteria. But it is also quite possible that user program created instances can meet these criteria as well. If they do, then they will be removed. This may or may not be a bad thing depending upon the specific facts of the situation. So if you are in an instance of Z/XDC that was **not** created by a hook, then use this **REMOVEXDC** operand only with the greatest of care and understanding.

If the current instance of Z/XDC is not removable, then the **GO REMOVEXDC** command is failed with an error message.

RELEASECAPS
This operand can be used to end a debugging session without also ending the program being debugged. Instead, the execution of the program is resumed, But all Z/XDC related processes are terminated, all Z/XDC objects are purged, and all Z/XDC related resources are released. These include:

- All maps (Binder maps, csect maps, source maps, dsect maps),
- All equates (both automatic and normal, but not built-in),
- All breakpoints (deferred and otherwise),
- And in particular, **all licensing permits (CAPs) are released.**

Concurrent Access Permits (CAPS) are a limited resource. Generally, once a debugging session has been started within a program, that program would retain an exclusive hold on the session's CAPs for the life of the job. But this is problematic for smaller customers who wanted to use the **HOOK** command (for example) to kick off a brief, ad-hoc debugging session in a long-running job.

But when the programmer is finished poking around, if he ends the debugging session with a **GO RELEASECAPS** command, c/XDC will cleanup everything (including releasing the CAPs) and go away.

Subsequently, if Z/XDC receives control again, a new debugging session will be started.

KEEPXDC (or omitted)
This is the default. If neither **REMOVEXDC** nor **RELEASECAPS** is specified, then all current instances of Z/XDC will remain, and the current recovery environment will remain unchanged.

FORCE
This operand is relevant only during tracing when TRAP2 type breakpoints are being used. (See **HELP BREAKPOINTS TYPES** for information about breakpoint types.)

During TRAP2 type tracing, situations can occur that can cause the System's Trap Save Area to be corrupted. (See **HELP BREAKPOINTS TYPES CORRUPTION** for a discussion.) When Z/XDC detects that that has happened, it is reluctant to allow you to resume program execution.

However, it is possible for you to use the **ZAP** and **SET PSW** commands to fix the corrupted registers and PSW. So if you do that, then you can use this **FORCE** operand on your next **TRACE** or **GO** command to cause Z/XDC to allow you to resume program execution in spite of the corruption.

If a Trap Save Area corruption has not been detected, this operand is ignored.

NOWHERE
This is a strange bird. Basically, it causes Z/XDC to resume the user's program in such a way that control immediately bounces back to Z/XDC without allowing the program to do anything at all.

It does, however, allow the System's Recovery/Termination Manager (RTM) the opportunity to reschedule Z/XDC, and that can have several interesting consequences.

For details about when to use **GO NOWHERE,** why you would want to use it, and all the consequences of using it, see HELP COMMANDS GO NOWHERE.

The **NOWHERE** operand is incompatible with all operands, so it must always be given alone.

Examples:

GO
GOT
GOX
All of these commands cause the user program to resume execution at whatever address is pointed to by the retry level PSW[E].

If your debugging session is running within TSO (i.e. you are **not** debugging a background JOB or STC via cdf/XDC), then you generally get better results if you used **GOT** instead of GO.

- When you expect control to return to Z/XDC, using **GOT** is preferable to using GO.
- However, if you expect your program to write its own displays to the screen before control returns to Z/XDC, then using **GO** would be better.
- On the other hand, if you **are** debugging a background JOB or STC via cdf/XDC, then all distinctions between **GO, GOT** and **GOX** disappear, so it doesn't matter which you use.

Z PSW,R10?;GOT

The **ZAP** command sets the retry level PSW to point to the address pointed to by R10.

The **GOT** command then causes user program execution to resume at that address.

The display screen is not cleared prior to user program resumption.

GOT R10?
This is equivalent to the preceding example.

F R10?;GOT +0
This is similar to the preceding examples:

- The **FORMAT** command displays the storage pointed to by R10.
- Also, it sets the **Current Display Pointer** (@CDP) to that address.
- The **GOT** command then causes user program execution to resume at **+0** past the location pointed to by the Current Display Pointer.

GO REMOVEXDC
GO REMOVEX PSW?
GO PSW? REMOVE
These three commands all produce the identical result:

- The user program is resumed at the location pointed to by the Retry Level PSW.
- And the current instance of Z/XDC is removed from the Recovery Environment.

Generally, you would use **REMOVEXDC** at the end of a debugging session that had been initiated by the **HOOK command** or the **HOOK script.**

Help COmmands GO Nowhere

The **GO NOWHERE** command causes the user program to resume execution in such a way that control immediately bounces back to Z/XDC without the user program being allowed to execute any instructions at all. So... why bother?

Well... it does give the System’s Recovery/Termination Manager (RTM) an opportunity to regain control and reschedule Z/XDC. So while the user program has not been able to do anything, the RTM has been able to do **a lot!** And that can have some very useful results.

Perhaps the most important use for the **GO NOWHERE** command is to complete the process of switching your debugging session from being non-authorized to **being authorized...**

Completing a SET AUTH Command Action

Z/XDC does not have an internal way to increase its authority level. It cannot simply say, "I want to be authorized now." Instead, the process requires an external action from some other process that is already authorized. (Without this requirement, there would be an integrity exposure.) An authorized debugging session running in another address space serves this purpose quite nicely.

So to start the process, you must start up a second debugging session that is authorized and that is running in another address space. One easy way to do this is:

- Start up a **2nd TSO** session,
- Navigate to **ISPF's** Command Shell panel **(=6),**
- And issue **XDCCALLA IEFBR14.**

If System Security permits, this will start an authorized debugging session.

Then use the **SET ASID jobname** command to target the address space in which your non-authorized debugging session is running.

Then use **LIST TASKS, LIST RBS** and **LIST ESTAES** commands to find the STAE Control Block (SCB) that describes the debugging session you want to make authorized.

Finally, use the **SET AUTH SCB#n** command to start the authorization process. That command will change some flags in your debugging session's SCB that will cause RTM to make that session authorized the next time RTM runs Z/XDC. See HELP COMMANDS SET AUTH for more details about this.

This is where **GO NOWHERE** comes into play. It causes RTM to reschedule Z/XDC without changing anything affecting the program being debugged.

Making Z/XDC Authorized Does NOT Make the User Program Authorized

It is important to understand that this conferring of authorization upon Z/XDC does **not** in any way affect the user program. It will continue to run non-authorized and its **MODESET** macros (if it had any [which it probably wouldn't]) would continue to fail with s047 ABENDs.

Other Effects of GO NOWHERE

While completing an externally initiated SET AUTH action is the most important use of GO NOWHERE, it is also important to know that it has other effects that you might need to be aware of.

Register and PSW Changes are Hardened

For one thing, if you had used the **ZAP** command to change any registers, those changes have now been propagated to the actual registers that your program sees. The main consequence of this is that the changes will now show up in the **Error Level registers.**

Similarly, if you had used the **SET PSW** or **SET PSWE** command to change PSW attributes, those changes are also hardened, so they will also now show up in the **Error Level PSW[E].**

The Error Level Environment is Lost

If the **Error Level** execution environment had been **different** from the Retry Level, that difference is lost:

- The Retry Level Environment is propagated into the Error level.
- The **Request Blocks queue** is purged back to the Retry Level RB. All newer RBs are lost.
- The **Linkage Stack** is purged back to the Retry Level Linkage Stack Entry. All newer LSEs are lost.
- All Error Level **Registers** and **PSWs** are lost. All displays of Error Level registers and other objects will now show information that is identical to the Retry Level data.
- The Error Level execution location is lost. The **EWHERE** command will now show essentially the same location as the **WHERE** command. (There will remain a 2-byte difference in the resume address arising from adjustments made by Z/XDC to the Retry Level PSW. See HELP COMMANDS EWHERE fore more information.)

Usually losing the Error Level environment is no big deal, but if you had been studying it, then you had better be finished before issuing **GO NOWHERE!**

Accordingly, if you do issue **GO NOWHERE,** and the command detects that the Error Level and Retry Level Environments are **different,** then Z/XDC will issue query message **DBC543Q** to be sure that you know the consequence of what you are doing.

Help COmmands GOT

The **GO, GOT** and **GOX** commands all cause user program execution to resume under the control of the retry level RB. Newer RBs (including the error level RB if different from the retry level) are terminated. See HELP EXECUTIONLEVELS for more information about this. (In the case that Z/XDC has been entered as a Trap Handler, execution will resume at the same RB level as when Z/XDC was entered.)

Also, the **TCBCMP** field is zeroed.

The **TCBCMP** field is 4 bytes long and includes the **TCBCMPF** and **TCBCMPC** subfields. Normally, the TCBCMPC subfield contains the most recent ABEND code, and the TCBCMPF subfield contains dump control flags and other flags related to that ABEND.

But GO processing zeros these fields because when you issue a GO/GOT/GOX or TRACE command, you are indicating that your program’s execution is to be resumed. (In the case of tracing, that resumption will be brief.) Therefore, the ABEND (s0C1 in the case of **0C1-type** tracing or trapping) is now "repaired" (supposedly). Consequently, all ABEND descriptions and controls contained within the TCBCMPx fields are now obsolete. Therefore, it would be misleading to programming if that information were allowed to remain.

Note, as soon as the next ABEND occurs (s0C1 in the case of **0C1-type** tracing or trapping), the System will store new ABEND information into the **TCBCMPx** fields

prior to passing control to Z/XDC again.

On the other hand, if **SET TRACE TRAP2** is in effect, then breakpoints will cause the Z/XDC Trap Handler to be driven, so there will be no ABEND, so there will be no change to the **TCBCMPx** fields.

The **GO** command (and friends) can be issued only when Z/XDC is **not** running in Foreign Address Space Mode **(FASM** mode). For more information about **Foreign** vs. **Local** Address Space Mode, see HELP VIRTMEM XDCACCESS and its subtopics.

GO vs GOT vs GOX Differ Only in TSO (Not in the Batch)

If you are debugging a program that is running within TSO (i.e. you are not debugging a background program via cdf/XDC), and if Z/XDC's Fullscreen Support is turned on, then:

- **GO** causes the display screen to be cleared prior to resumption. This is desirable if you anticipate that your program will want to write a display and then read a response prior to the next time Z/XDC receives control.

- **GOT** causes the user program to be resumed without clearing the display screen. This is desirable if you expect that your program will not attempt to issue any displays prior to the next time Z/XDC receives control. ("GOT" means "GO for tracing".)

- **GOX** causes Z/XDC to attempt to restore whatever information was being displayed at the terminal prior to when Z/XDC received control.

  But owing to the very poor documentation of the hodge-podge of interactions between TSO, session manager and ISPF, this restoration attempt will generally not be very satisfactory. See HELP XDCSRVER CDF TSO for a better approach for dealing with terminal screen display conflicts.

These distinctions between GO, GOT and GOX occur only when debugging a program that is running within TSO. If you are debugging a program using **cdf/XDC,** all distinctions **disappear,** and the GO, GOT and GOX commands behave identically.

This is true even if you are connected to that batch job debugging session via **option 3** of the **Z/XDC Startup Panel** in ISPF.

For more information about batch job debugging, See HELP XDCSRVER CDF.

Syntax:

GO	**addressexpression**	**REMOVEXDC**	**RELEASECAPS**	**FORCE**
GOT	**NOWHERE**	**NOWHERE**	**NOWHERE**	**NOWHERE**
GOX	**omitted**	**KEEPXDC**	**KEEPXDC**	**omitted**
		omitted	**omitted**	

Notes:

- All operands are optional.

- Operands appearing within columns are mutually exclusive.

- When multiple operands are given, they may be given in any order.

- The **NOWHERE** operand is mutually exclusive with **all** other operands.

addressexpression (or omitted)
This gives the desired resume address. To learn about address expressions, see HELP ADDRESSING.

If the addressexpression operand is omitted (and NOWHERE also is absent), then the address pointed to by the **retry level PSW[E]** is used by default. In other words, the program generally will resume where it was interrupted. For more information, see HELP EXECUTIONLEVELS.

Tip: If the addressexpression you want to use happens to be indistinguishable from one of the GO command's keyword operands, then append a **+0** to make it distinguishable. Example:

- **EQUATE KEEP wherever** [Creates an equate named KEEP.]

- **GO KEEP+0** [The KEEP equate I just created is not distinguishable from the KEEP abbreviation of the KEEPXDC operand, but adding the **+0** makes it clear that I'm using KEEP here as part of an address expression, not as a keyword.]

REMOVEXDC
Under certain circumstances, this operand can be used to remove the current instance of Z/XDC from your program's recovery environment. (It does **not** remove all instances of Z/XDC.)

Generally, this operand is intended primarily to remove those instances of Z/XDC that have been established **dynamically** via the **HOOK** command.

REMOVEXDC addresses a problem that arises when a user program attempts to cancel its own recovery routine without knowing that you, during debugging, have used a hook to insert Z/XDC into the environment. When you are done debugging and wish to let the program go on its merry way, if:
(1) - The program eventually issues an **ESTAE[X] 0** to cancel its own ESTAE[X] (or a **SETFRR D** to delete its own FRR), and if
(2) - The hooked-instance of Z/XDC remains on the SCB queue or FRR stack,
Then the user program's "ESTAE[X] 0" or "SETFRR D" will cancel the **wrong** recovery routine, and the recovery routine that the program had thought it had canceled will remain in effect. This, of course, can lead to arbitrary failures. For more information, see **HELP HOOKS UNDOING.**

REMOVEXDC does not remove **all** instances of Z/XDC, and it can remove only **certain** instances of Z/XDC. Here are the rules:

- REMOVEXDC can remove only the **current** instance of Z/XDC. If other instances of

Z/XDC are pending in the recovery stack, those remain in effect.

- REMOVEXDC can remove Z/XDC only if it is the environment's **newest** recovery routine. It cannot be used if the current instance of Z/XDC is running under any older STAE Control Block (SCB).

- REMOVEXDC can remove Z/XDC only if it is running as an **ESTAE,** an **ESTAEX,** or an **FRR.** It cannot be used if Z/XDC is running as an ESTAI, as an ARR or as any other type of recovery routine.

- REMOVEXDC can be used only if Z/XDC has been entered **directly** from the System (i.e. from the Recovery/Termination Manager [RTM]). It cannot be used if Z/XDC was **BASR'd** to from a user written ESTAE[X] routine or **jumped** to from a user written FRR.

Note, when Z/XDC is running as an ESTAE-type recovery routine, the **LIST ESTAES** command can be used to view detailed information about the recovery environment.

The instances of Z/XDC that are created by HOOK commands and scripts meet all of the above criteria. But it is also quite possible that user program created instances can meet these criteria as well. If they do, then they will be removed. This may or may not be a bad thing depending upon the specific facts of the situation. So if you are in an instance of Z/XDC that was **not** created by a hook, then use this **REMOVEXDC** operand only with the greatest of care and understanding.

If the current instance of Z/XDC is not removable, then the **GO REMOVEXDC** command is failed with an error message.

RELEASECAPS
This operand can be used to end a debugging session without also ending the program being debugged. Instead, the execution of the program is resumed, But all Z/XDC related processes are terminated, all Z/XDC objects are purged, and all Z/XDC related resources are released. These include:

- All maps (Binder maps, csect maps, source maps, dsect maps),
- All equates (both automatic and normal, but not built-in),
- All breakpoints (deferred and otherwise),
- And in particular, **all licensing permits (CAPs) are released.**

Concurrent Access Permits (CAPS) are a limited resource. Generally, once a debugging session has been started within a program, that program would retain an exclusive hold on the session's CAPs for the life of the job. But this is problematic for smaller customers who wanted to use the **HOOK** command (for example) to kick off a brief, ad-hoc debugging session in a long-running job.

But when the programmer is finished poking around, if he ends the debugging session with a **GO RELEASECAPS** command, c/XDC will cleanup everything (including releasing the CAPs) and go away.

Subsequently, if Z/XDC receives control again, a new debugging session will be started.

KEEPXDC (or omitted)
This is the default. If neither **REMOVEXDC** nor **RELEASECAPS** is specified, then all current instances of Z/XDC will remain, and the current recovery environment will remain unchanged.

FORCE
This operand is relevant only during tracing when TRAP2 type breakpoints are being used. (See **HELP BREAKPOINTS TYPES** for information about breakpoint types.)

During TRAP2 type tracing, situations can occur that can cause the System's Trap Save Area to be corrupted. (See **HELP BREAKPOINTS TYPES CORRUPTION** for a discussion.) When Z/XDC detects that that has happened, it is reluctant to allow you to resume program execution.

However, it is possible for you to use the **ZAP** and **SET PSW** commands to fix the corrupted registers and PSW. So if you do that, then you can use this **FORCE** operand on your next **TRACE** or **GO** command to cause Z/XDC to allow you to resume program execution in spite of the corruption.

If a Trap Save Area corruption has not been detected, this operand is ignored.

NOWHERE
This is a strange bird. Basically, it causes Z/XDC to resume the user's program in such a way that control immediately bounces back to Z/XDC without allowing the program to do anything at all.

It does, however, allow the System's Recovery/Termination Manager (RTM) the opportunity to reschedule Z/XDC, and that can have several interesting consequences.

For details about when to use **GO NOWHERE,** why you would want to use it, and all the consequences of using it, see HELP COMMANDS GO NOWHERE.

The **NOWHERE** operand is incompatible with all operands, so it must always be given alone.

Examples:

GO
GOT
GOX
All of these commands cause the user program to resume execution at whatever address is pointed to by the retry level PSW[E].

If your debugging session is running within TSO (i.e. you are **not** debugging a background JOB or STC via cdf/XDC), then you generally get better results if you used **GOT** instead of GO.

- When you expect control to return to Z/XDC, using **GOT** is preferable to using GO.

- However, if you expect your program to write its own displays to the screen before control returns to Z/XDC, then using **GO** would be better.

- On the other hand, if you **are** debugging a background JOB or STC via cdf/XDC, then all distinctions between **GO, GOT** and **GOX** disappear, so it doesn't matter which you use.

Z PSW,R10?;GOT

The **ZAP** command sets the retry level PSW to point to the address pointed to by R10.

The **GOT** command then causes user program execution to resume at that address.

The display screen is not cleared prior to user program resumption.

GOT R10?
This is equivalent to the preceding example.

F R10?;GOT +0
This is similar to the preceding examples:

- The **FORMAT** command displays the storage pointed to by R10.
- Also, it sets the **Current Display Pointer** (@CDP) to that address.
- The **GOT** command then causes user program execution to resume at **+0** past the location pointed to by the Current Display Pointer.

GO REMOVEXDC
GO REMOVEX PSW?
GO PSW? REMOVE
These three commands all produce the identical result:

- The user program is resumed at the location pointed to by the Retry Level PSW.
- And the current instance of Z/XDC is removed from the Recovery Environment.

Generally, you would use **REMOVEXDC** at the end of a debugging session that had been initiated by the **HOOK command** or the **HOOK script.**

Help COmmands GOT Nowhere

The **GO NOWHERE** command causes the user program to resume execution in such a way that control immediately bounces back to Z/XDC without the user program being allowed to execute any instructions at all. So... why bother?

Well... it does give the System's Recovery/Termination Manager (RTM) an opportunity to regain control and reschedule Z/XDC. So while the user program has not been able

to do anything, the RTM has been able to do **a lot!** And that can have some very useful results.

Perhaps the most important use for the **GO NOWHERE** command is to complete the process of switching your debugging session from being non-authorized to **being authorized...**

Completing a SET AUTH Command Action

Z/XDC does not have an internal way to increase its authority level. It cannot simply say, "I want to be authorized now." Instead, the process requires an external action from some other process that is already authorized. (Without this requirement, there would be an integrity exposure.) An authorized debugging session running in another address space serves this purpose quite nicely.

So to start the process, you must start up a second debugging session that is authorized and that is running in another address space. One easy way to do this is:

- Start up a **2nd TSO** session,
- Navigate to **ISPF's** Command Shell panel **(=6),**
- And issue **XDCCALLA IEFBR14.**

If System Security permits, this will start an authorized debugging session.

Then use the **SET ASID jobname** command to target the address space in which your non-authorized debugging session is running.

Then use **LIST TASKS, LIST RBS** and **LIST ESTAES** commands to find the STAE Control Block (SCB) that describes the debugging session you want to make authorized.

Finally, use the **SET AUTH SCB#n** command to start the authorization process. That command will change some flags in your debugging session's SCB that will cause RTM to make that session authorized the next time RTM runs Z/XDC. See HELP COMMANDS SET AUTH for more details about this.

This is where **GO NOWHERE** comes into play. It causes RTM to reschedule Z/XDC without changing anything affecting the program being debugged.

Making Z/XDC Authorized Does NOT Make the User Program Authorized

It is important to understand that this conferring of authorization upon Z/XDC does **not** in any way affect the user program. It will continue to run non-authorized and its **MODESET** macros (if it had any [which it probably wouldn't]) would continue to fail with s047 ABENDs.

Other Effects of GO NOWHERE

While completing an externally initiated SET AUTH action is the most important use of GO NOWHERE, it is also important to know that it has other effects that you might need to be aware of.

Register and PSW Changes are Hardened

For one thing, if you had used the **ZAP** command to change any registers, those changes have now been propagated to the actual registers that your program sees. The main consequence of this is that the changes will now show up in the **Error Level registers.**

Similarly, if you had used the **SET PSW** or **SET PSWE** command to change PSW attributes, those changes are also hardened, so they will also now show up in the **Error Level PSW[E].**

The Error Level Environment is Lost

If the **Error Level** execution environment had been **different** from the Retry Level, that difference is lost:

- The Retry Level Environment is propagated into the Error level.
- The **Request Blocks queue** is purged back to the Retry Level RB. All newer RBs are lost.
- The **Linkage Stack** is purged back to the Retry Level Linkage Stack Entry. All newer LSEs are lost.
- All Error Level **Registers** and **PSWs** are lost. All displays of Error Level registers and other objects will now show information that is identical to the Retry Level data.
- The Error Level execution location is lost. The **EWHERE** command will now show essentially the same location as the **WHERE** command. (There will remain a 2-byte difference in the resume address arising from adjustments made by Z/XDC to the Retry Level PSW. See HELP COMMANDS EWHERE fore more information.)

Usually losing the Error Level environment is no big deal, but if you had been studying it, then you had better be finished before issuing **GO NOWHERE!**

Accordingly, if you do issue **GO NOWHERE,** and the command detects that the Error Level and Retry Level Environments are **different,** then Z/XDC will issue query message **DBC543Q** to be sure that you know the consequence of what you are doing.

Help COmmands GOX

The **GO, GOT** and **GOX** commands all cause user program execution to resume under the control of the retry level RB. Newer RBs (including the error level RB if different from the retry level) are terminated. See HELP EXECUTIONLEVELS for more information about this. (In the case that Z/XDC has been entered as a Trap Handler, execution will resume at the same RB level as when Z/XDC was entered.)

Also, the **TCBCMP** field is zeroed.

The **TCBCMP** field is 4 bytes long and includes the **TCBCMPF** and **TCBCMPC** subfields. Normally, the TCBCMPC subfield contains the most recent ABEND code, and the TCBCMPF subfield contains dump control flags and other flags related to that ABEND.

But GO processing zeros these fields because when you issue a GO/GOT/GOX or TRACE command, you are indicating that your program's execution is to be resumed. (In the case of tracing, that resumption will be brief.) Therefore, the ABEND (s0C1 in the case of **0C1-type** tracing or trapping) is now "repaired" (supposedly). Consequently, all ABEND descriptions and controls contained within the TCBCMPx fields are now obsolete. Therefore, it would be misleading to programming if that information were allowed to remain.

Note, as soon as the next ABEND occurs (s0C1 in the case of **0C1-type** tracing or trapping), the System will store new ABEND information into the **TCBCMPx** fields prior to passing control to Z/XDC again.

On the other hand, if **SET TRACE TRAP2** is in effect, then breakpoints will cause the Z/XDC Trap Handler to be driven, so there will be no ABEND, so there will be no change to the **TCBCMPx** fields.

The **GO** command (and friends) can be issued only when Z/XDC is **not** running in Foreign Address Space Mode **(FASM** mode). For more information about **Foreign** vs. **Local** Address Space Mode, see HELP VIRTMEM XDCACCESS and its subtopics.

GO vs GOT vs GOX Differ Only in TSO (Not in the Batch)

If you are debugging a program that is running within TSO (i.e. you are not debugging a background program via cdf/XDC), and if Z/XDC's Fullscreen Support is turned on, then:

- **GO** causes the display screen to be cleared prior to resumption. This is desirable if you anticipate that your program will want to write a display and then read a response prior to the next time Z/XDC receives control.

- **GOT** causes the user program to be resumed without clearing the display screen. This is desirable if you expect that your program will not attempt to issue any displays prior to the next time Z/XDC receives control. ("GOT" means "GO for tracing".)

- **GOX** causes Z/XDC to attempt to restore whatever information was being displayed at the terminal prior to when Z/XDC received control.

  But owing to the very poor documentation of the hodge-podge of interactions between TSO, session manager and ISPF, this restoration attempt will generally not be very satisfactory. See HELP XDCSRVER CDF TSO for a better approach for dealing with terminal screen display conflicts.

These distinctions between GO, GOT and GOX occur only when debugging a program that is running within TSO. If you are debugging a program using **cdf/XDC,** all distinctions **disappear,** and the GO, GOT and GOX commands behave identically.

This is true even if you are connected to that batch job debugging session via **option 3** of the **Z/XDC Startup Panel** in ISPF.

For more information about batch job debugging, See HELP XDCSRVER CDF.

Syntax:

GO	**addressexpression**	**REMOVEXDC**	**RELEASECAPS**	**FORCE**
GOT	**NOWHERE**	**NOWHERE**	**NOWHERE**	**NOWHERE**
GOX	**omitted**	**KEEPXDC**	**KEEPXDC**	**omitted**
		omitted	**omitted**	

Notes:

- All operands are optional.
- Operands appearing within columns are mutually exclusive.
- When multiple operands are given, they may be given in any order.
- The **NOWHERE** operand is mutually exclusive with **all** other operands.

addressexpression (or omitted)
This gives the desired resume address. To learn about address expressions, see HELP ADDRESSING.

If the addressexpression operand is omitted (and NOWHERE also is absent), then the address pointed to by the **retry level PSW[E]** is used by default. In other words, the program generally will resume where it was interrupted. For more information, see HELP EXECUTIONLEVELS.

Tip: If the addressexpression you want to use happens to be indistinguishable from one of the GO command's keyword operands, then append a **+0** to make it distinguishable. Example:

- **EQUATE KEEP wherever** [Creates an equate named KEEP.]
- **GO KEEP+0** [The KEEP equate I just created is not distinguishable from the KEEP abbreviation of the KEEPXDC operand, but adding the **+0** makes it clear that I'm using KEEP here as part of an address expression, not as a keyword.]

REMOVEXDC
Under certain circumstances, this operand can be used to remove the current instance of Z/XDC from your program's recovery environment. (It does **not** remove all instances of Z/XDC.)

Generally, this operand is intended primarily to remove those instances of Z/XDC that have been established **dynamically** via the **HOOK** command.

REMOVEXDC addresses a problem that arises when a user program attempts to cancel its

own recovery routine without knowing that you, during debugging, have used a hook to insert Z/XDC into the environment. When you are done debugging and wish to let the program go on its merry way, if:

(1) - The program eventually issues an **ESTAE[X] 0** to cancel its own ESTAE[X] (or a **SETFRR D** to delete its own FRR), and if

(2) - The hooked-instance of Z/XDC remains on the SCB queue or FRR stack,

Then the user program's "ESTAE[X] 0" or "SETFRR D" will cancel the **wrong** recovery routine, and the recovery routine that the program had thought it had canceled will remain in effect. This, of course, can lead to arbitrary failures. For more information, see **HELP HOOKS UNDOING.**

REMOVEXDC does not remove **all** instances of Z/XDC, and it can remove only **certain** instances of Z/XDC. Here are the rules:

- REMOVEXDC can remove only the **current** instance of Z/XDC. If other instances of Z/XDC are pending in the recovery stack, those remain in effect.

- REMOVEXDC can remove Z/XDC only if it is the environment's **newest** recovery routine. It cannot be used if the current instance of Z/XDC is running under any older STAE Control Block (SCB).

- REMOVEXDC can remove Z/XDC only if it is running as an **ESTAE,** an **ESTAEX,** or an **FRR.** It cannot be used if Z/XDC is running as an ESTAI, as an ARR or as any other type of recovery routine.

- REMOVEXDC can be used only if Z/XDC has been entered **directly** from the System (i.e. from the Recovery/Termination Manager [RTM]). It cannot be used if Z/XDC was **BASR'd** to from a user written ESTAE[X] routine or **jumped** to from a user written FRR.

Note, when Z/XDC is running as an ESTAE-type recovery routine, the **LIST ESTAES** command can be used to view detailed information about the recovery environment.

The instances of Z/XDC that are created by HOOK commands and scripts meet all of the above criteria. But it is also quite possible that user program created instances can meet these criteria as well. If they do, then they will be removed. This may or may not be a bad thing depending upon the specific facts of the situation. So if you are in an instance of Z/XDC that was **not** created by a hook, then use this **REMOVEXDC** operand only with the greatest of care and understanding.

If the current instance of Z/XDC is not removable, then the **GO REMOVEXDC** command is failed with an error message.

RELEASECAPS

This operand can be used to end a debugging session without also ending the program being debugged. Instead, the execution of the program is resumed, But all Z/XDC related processes are terminated, all Z/XDC objects are purged, and all Z/XDC related resources are released. These include:

- All maps (Binder maps, csect maps, source maps, dsect maps),
- All equates (both automatic and normal, but not built-in),
- All breakpoints (deferred and otherwise),
- And in particular, **all licensing permits (CAPs) are released.**

Concurrent Access Permits (CAPS) are a limited resource. Generally, once a debugging session has been started within a program, that program would retain an exclusive hold on the session's CAPs for the life of the job. But this is problematic for smaller customers who wanted to use the **HOOK** command (for example) to kick off a brief, ad-hoc debugging session in a long-running job.

But when the programmer is finished poking around, if he ends the debugging session with a **GO RELEASECAPS** command, c/XDC will cleanup everything (including releasing the CAPs) and go away.

Subsequently, if Z/XDC receives control again, a new debugging session will be started.

KEEPXDC (or omitted)
This is the default. If neither **REMOVEXDC** nor **RELEASECAPS** is specified, then all current instances of Z/XDC will remain, and the current recovery environment will remain unchanged.

FORCE
This operand is relevant only during tracing when TRAP2 type breakpoints are being used. (See **HELP BREAKPOINTS TYPES** for information about breakpoint types.)

During TRAP2 type tracing, situations can occur that can cause the System's Trap Save Area to be corrupted. (See **HELP BREAKPOINTS TYPES CORRUPTION** for a discussion.) When Z/XDC detects that that has happened, it is reluctant to allow you to resume program execution.

However, it is possible for you to use the **ZAP** and **SET PSW** commands to fix the corrupted registers and PSW. So if you do that, then you can use this **FORCE** operand on your next **TRACE** or **GO** command to cause Z/XDC to allow you to resume program execution in spite of the corruption.

If a Trap Save Area corruption has not been detected, this operand is ignored.

NOWHERE
This is a strange bird. Basically, it causes Z/XDC to resume the user's program in such a way that control immediately bounces back to Z/XDC without allowing the program to do anything at all.

It does, however, allow the System's Recovery/Termination Manager (RTM) the opportunity to reschedule Z/XDC, and that can have several interesting consequences.

For details about when to use **GO NOWHERE,** why you would want to use it, and all the consequences of using it, see HELP COMMANDS GO NOWHERE.

The **NOWHERE** operand is incompatible with all operands, so it must always be given alone.

Examples:

GO
GOT
GOX
All of these commands cause the user program to resume execution at whatever address is pointed to by the retry level PSW[E].

If your debugging session is running within TSO (i.e. you are **not** debugging a background JOB or STC via cdf/XDC), then you generally get better results if you used **GOT** instead of GO.

- When you expect control to return to Z/XDC, using **GOT** is preferable to using GO.
- However, if you expect your program to write its own displays to the screen before control returns to Z/XDC, then using **GO** would be better.
- On the other hand, if you **are** debugging a background JOB or STC via cdf/XDC, then all distinctions between **GO, GOT** and **GOX** disappear, so it doesn't matter which you use.

Z PSW,R10?;GOT

The **ZAP** command sets the retry level PSW to point to the address pointed to by R10.

The **GOT** command then causes user program execution to resume at that address.

The display screen is not cleared prior to user program resumption.

GOT R10?
This is equivalent to the preceding example.

F R10?;GOT +0
This is similar to the preceding examples:

- The **FORMAT** command displays the storage pointed to by R10.
- Also, it sets the **Current Display Pointer** (@CDP) to that address.
- The **GOT** command then causes user program execution to resume at **+0** past the location pointed to by the Current Display Pointer.

GO REMOVEXDC
GO REMOVEX PSW?
GO PSW? REMOVE
These three commands all produce the identical result:

- The user program is resumed at the location pointed to by the Retry Level PSW.
- And the current instance of Z/XDC is removed from the Recovery Environment.

Generally, you would use **REMOVEXDC** at the end of a debugging session that had been initiated by the **HOOK command** or the **HOOK script.**

Help COmmands GOX Nowhere

The **GO NOWHERE** command causes the user program to resume execution in such a way that control immediately bounces back to Z/XDC without the user program being allowed to execute any instructions at all. So... why bother?

Well... it does give the System's Recovery/Termination Manager (RTM) an opportunity to regain control and reschedule Z/XDC. So while the user program has not been able to do anything, the RTM has been able to do **a lot!** And that can have some very useful results.

Perhaps the most important use for the **GO NOWHERE** command is to complete the process of switching your debugging session from being non-authorized to **being authorized...**

Completing a SET AUTH Command Action

Z/XDC does not have an internal way to increase its authority level. It cannot simply say, "I want to be authorized now." Instead, the process requires an external action from some other process that is already authorized. (Without this requirement, there would be an integrity exposure.) An authorized debugging session running in another address space serves this purpose quite nicely.

So to start the process, you must start up a second debugging session that is authorized and that is running in another address space. One easy way to do this is:

- Start up a **2nd TSO** session,
- Navigate to **ISPF's** Command Shell panel **(=6),**
- And issue **XDCCALLA IEFBR14.**

If System Security permits, this will start an authorized debugging session.

Then use the **SET ASID jobname** command to target the address space in which your non-authorized debugging session is running.

Then use **LIST TASKS, LIST RBS** and **LIST ESTAES** commands to find the STAE Control Block (SCB) that describes the debugging session you want to make authorized.

Finally, use the **SET AUTH SCB#n** command to start the authorization process. That command will change some flags in your debugging session's SCB that will cause RTM to make that session authorized the next time RTM runs Z/XDC. See HELP COMMANDS SET AUTH for more details about this.

This is where **GO NOWHERE** comes into play. It causes RTM to reschedule Z/XDC without changing anything affecting the program being debugged.

Making Z/XDC Authorized Does NOT Make the User Program Authorized

It is important to understand that this conferring of authorization upon Z/XDC does **not** in any way affect the user program. It will continue to run non-authorized and its **MODESET** macros (if it had any [which it probably wouldn't]) would continue to fail with s047 ABENDs.

Other Effects of GO NOWHERE

While completing an externally initiated SET AUTH action is the most important use of GO NOWHERE, it is also important to know that it has other effects that you might need to be aware of.

Register and PSW Changes are Hardened

For one thing, if you had used the **ZAP** command to change any registers, those changes have now been propagated to the actual registers that your program sees. The main consequence of this is that the changes will now show up in the **Error Level registers.**

Similarly, if you had used the **SET PSW** or **SET PSWE** command to change PSW attributes, those changes are also hardened, so they will also now show up in the **Error Level PSW[E].**

The Error Level Environment is Lost

If the **Error Level** execution environment had been **different** from the Retry Level, that difference is lost:

- The Retry Level Environment is propagated into the Error level.
- The **Request Blocks queue** is purged back to the Retry Level RB. All newer RBs are lost.
- The **Linkage Stack** is purged back to the Retry Level Linkage Stack Entry. All newer LSEs are lost.
- All Error Level **Registers** and **PSWs** are lost. All displays of Error Level registers and other objects will now show information that is identical to the Retry Level data.
- The Error Level execution location is lost. The **EWHERE** command will now show essentially the same location as the **WHERE** command. (There will remain a 2-byte difference in the resume address arising from adjustments made by Z/XDC to the Retry Level PSW. See HELP COMMANDS EWHERE fore more information.)

Usually losing the Error Level environment is no big deal, but if you had been studying it, then you had better be finished before issuing **GO NOWHERE!**

Accordingly, if you do issue **GO NOWHERE,** and the command detects that the Error

Level and Retry Level Environments are **different,** then Z/XDC will issue query message **DBC543Q** to be sure that you know the consequence of what you are doing.

Help COmmands HDeferred

The **HDEFERRED** command is used to set deferred **hooks.**
The **ADEFERRED** command is used to set deferred **persistent** breakpoints.
The **TDEFERRED** command is used to set deferred **transient** breakpoints.

The HDEFERRED command is documented here. The **ADEFERRED** and **TDEFERRED** commands, although quite similar, **are documented separately** at HELP COMMANDS ADEFERRED.

A **deferred** hook is one that is to be set into a load module that has not yet been brought into storage. Eventually, when the module is loaded, the hook will be set at that time.

Note that at this time, deferred breakpoints (of any type) cannot be set in programs that reside in a hierarchical file system (i.e., loaded by the loadhfs and loadHFSextended USS system calls).

Modules are loaded into storage as a result of LINK(X), LOAD, ATTACH(X), and XCTL(X) macros or the Loadhfs and LoadHFSextended USS system calls executed by your program. When a target module is read into storage as a result of one of these macros or system calls, then a Z/XDC support routine receives control from the System. After making appropriate security checks, this routine "clones" (i.e. makes a copy of) the deferred hook definition in order to create an active hook in the target module. All the attributes of the deferred definition are copied and assigned to the active hook.

If the System decides to satisfy the LINK(X), LOAD, ATTACH(X), or XCTL(X) macro or the Loadhfs and LoadHFSextended USS system calls with a copy of the target module that **already resides** in storage, then **the deferred hook definition is ignored!** No new active hook is created. Deferred hooks affect **only new copies** of the target module that are read from a library on disk (or from System cache).

If a copy of the target module already resides in storage at the time that the HDEFERRED command is issued, then that copy remains unaffected by the command. In order to set a hook into such a copy, you must use the **HOOK** command.

A deferred hook definition remains in existence for the life of the debugging session or until it is removed via the **OFF** command. This means that if additional copies of the target module are loaded into storage by subsequent LINK(X), LOAD, ATTACH(X), or XCTL(X) macros or the Loadhfs and LoadHFSextended USS system calls then additional active hooks will be generated by the deferred hook definition until that definition is removed via an **OFF** command.

Deferred hooks are supported only for modules that are loaded into the Home Address Space’s **private** area. Deferred hooks are not set in modules that are loaded into common storage.

Deferred hooks also are not set when **both** of the following are true:
- The module is loaded by the System into **protected** storage.
- Z/XDC was running **non-authorized** when it created the hook definition.

Deferred hooks also are not set when **both** of the following are true:
- the module is loaded above the bar (Binder option RMODE=64)
- The deferred hook was not defined with a work register.

The general syntax of the HDEFERRED command is:

```
HDEFERRED addressexpressions STATE=SUPERVISOR  AMODE=24
                                     PROBLEM           31
                                     LOCKED            64
                                                       ANY
                                WORKREG=n         ODDASCM=NO
                                                          YES
```

addressexpressions
This must be a list of one or more address expressions separated from each other by commas or blanks. In this context, only certain types of address expressions are valid. Specifically, the following restrictions must be met:

- The **base** of the expression must be the name of a load module (primary or alias). It **may not be** a register, a PSW, an equate or dsect name, or any field name. The named module either may or may not already be loaded into storage. Note, however, that the hook will **not** be set into any currently loaded copies of the module. (You will have to use the HOOK command for that.) Deferred hook definitions are effective **only for future** loadings of the module.

- If a module map of the load module has previously been loaded via either the MAP or **DMAP** command, then the given load module may be qualified by a csect name (e.g. "HD SUBMIT.IKJEFF10"). For more information about using the **DMAP** command to load module maps, see HELP MAPS CSECTSASDSECTS.

- If the given load module name is **not** qualified by a csect name, then the name refers to the module's entry address **(NOT!** its first byte of storage, i.e. not its load point).

- If you wish to refer to a module's load point, then qualify the module's name with **.X#1** (e.g. **HD SUBMIT.X#1).** This is useful if the module's entry address is different from its load point.

- The address expression may be modified by offset values (e.g. **HD OFFLOAD+1DC").** Both positive and negative offsets are permitted.

- However, the expression may **not** contain indirect operators (% ? !) or built-in functions.

Addressexpressions is a required operand. All other operands may be omitted.

STATE=SUPERVISOR
STATE=PROBLEM
STATE=LOCKED
Use these operands when the execution state of the Hook Point is going to be different from the execution state of the current code from which Z/XDC has received control.

For example, if Z/XDC has received control from a running in supervisor state, and the Hook Point is in code that you expect to run in problem state, then you would use **STATE=PROBLEM** when setting the hook.

If you are placing a hook into code that you expect to run locked or disabled, then you would use **STATE=LOCKED.**

AMODE=24
AMODE=31
AMODE=64
AMODE=ANY
Using this operand influences whether the Hook Processing control blocks will be built in 24-bit or 31-bit storage. Absent this operand, the control blocks will be built in storage that is compatible with the location of the Hook Point.

WORKREG=n
This operand nominates a work register that can be used by the generated hook processing code that is specific to this HDEFERRED request. You need to provide a work register if (1) you wish to hook addresses that are above the bar or (2) you wish to hook code that may be executing in SECONDARY or HOME ASC-MODE (you also have ODDASCM=YES specified). The work register must be a register number from 1 to 15 (R1 to R15) and the nominated register will be set to zero.

ODDASCM=NO
ODDASCM=YES
This operand indicates that the hook needs to support code that is executing in the "unusual" ASC-MODE of SECONDARY or HOME. Failure to use ODDASCM=YES when the hook is executed by code that is in SECONDARY or HOME ASC-MODE will result in an s0D3 ABEND. Note that you must also specify a work register if you specify ODDASCM=YES.

Related commands:
- LIST BREAKPOINTS (shows deferred hook definitions)
- LIST HOOKS (shows active hooks)
- DELETE HOOKS (deletes active hook definitions only)
- OFF (deletes both active and deferred hook definitions)

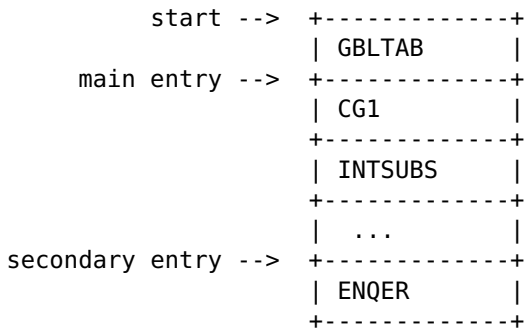
Examples:

Suppose that a module named CROSSGUN exists and has the following characteristics:

- It has not yet been loaded into storage.
- Its entry point is at a csect named GC1, and that csect is **not** first in the load module.
- The csect that is physically first in CROSSGUN is GBLTAB.
- The module has an alias named ENQER pointing to a csect of the same name.
- It has another csect named INTSUBS.

- However, a module map has **not** yet been loaded by either the MAP or DMAP command.

Schematically, the load module looks like this:



Then the following commands will have the following results.

HD CROSSGUN+3DC
When the CROSSGUN module is eventually loaded into storage via its primary name (CROSSGUN), a hook will be set at +X’3DC’ past the start of the GC1 csect.

HD ENQER
When the CROSSGUN module is eventually loaded into storage via its alias name (ENQER), a hook will be set at the start of the ENQER csect.

DM CROSSGUN.
HD CROSSGUN.ENQER
The DMAP command reads CROSSGUN’s module map as if it were a dsect. Once the map is available, Z/XDC can refer to it in order to learn where CROSSGUN’s csects are located. Later, when the CROSSGUN module is loaded into storage via its primary name, a hook will be set at the start of the ENQER csect.

HD CROSSGUN.X#1+2C
HD CROSSGUN.GBLTAB+2C
Both of these commands have the same result: When the CROSSGUN module is brought into storage via its primary name, a hook will be set at +X’2C’ past the start of the GBLTAB csect. The difference is that the second command requires the availability of the module’s module map ("DM CROSSGUN."), while the first command does not.

HD ENQER.X#1+2C
HD ENQER.GBLTAB+2C
If the CROSSGUN module is brought into storage via its alias name, the one or the other of these commands will be necessary for setting the hook at the GBLTAB+2C location.

HD CROSSGUN.INTSUBS
If a module map is not available, then this command will fail because Z/XDC will not know where in the load module to find the INTSUBS csect.

Related information can be found by typing HELP HOOKS.

Information about deferred breakpoints can be found at HELP COMMANDS ADEFERRED.

Help COmmands HElp

The **HELP** command displays topics discussing and describing how to use Z/XDC. There are over two thousand separate topics!

The topics are organized into a pyramid like hierarchy. Each topic in the hierarchy:
- Has a name.
- Is pointed to by a topic that is higher in the hierarchy.
- May point to additional topics that are lower in the hierarchy.

A given topic of information is identified by its name and its path. A "path" consists of the names of all topics located between the top of the pyramid and the desired topic. The first line displayed for each topic (the **header** line) shows that topic's path and name. For example, the name of the current topic is **HELP.** The name of its path is **HELP COMMANDS HELP.**

A topic's name can almost always be abbreviated. Only as much of a name need be given as is necessary to distinguish it from all other topics that are reachable via the same path (i.e. occupy the same node in the hierarchy). For example, the current topic can be reference via either **HELP COMMANDS HELP** or just **H CO HE.** A topic's header line shows, in uppercase, these minimum abbreviations.

The **LIST HELP** command can be used to display the organization of the HELP topic hierarchy. See **HELP COMMANDS LIST HELP** for complete information about this.

If a displayed line starts with a keyword (i.e. a name that is separated from the rest of the line by a hyphen), then more descriptive information about that name exists. To display this additional information, you should issue another **HELP** command using two operands. The first operand should be an asterisk (*); the second operand should be a unique abbreviation of the desired keyword. Below are examples of such keywords.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

ABSOLUTE - Locating HELP topics via absolute references.

RELATIVE - Locating HELP topics via relative references.
MIXING - Locating HELP topics via a mixture of absolute and relative references.

Examples:

To see my subtopic named **ABSOLUTE,** you can now type either **H CO HE A** (an absolute reference) or **H * A** (a mixed reference).

To view topics in "sequential" order (in depth-first sequential order, actually), type **H *N.**

To re-view the current topic, type **H *.**

For complete information concerning relative topic references, type **HELP COMMANDS HELP RELATIVE.**

Help COmmands HElp Absolute

An **absolute** reference to a HELP topic consists of both the name of the desired topic and the names of all topics that constitute a direct path from the top of the hierarchy to the desired topic.

Syntax:

HELP name1 name2 ...

Shortcut: H

names
The last name given must be the name of the desired topic. The preceding names identify a direct path from the top of the Built-in Help hierarchy to the desired topic.

The syntax rules for **names** are pretty loose:
- A name may contain nearly any character except commas, blanks and nulls.
- A name may contain an asterisk, but it may not start with one.
- A name may be a pure number, but if it is, it may not be a single digit.

Each of the given names can be abbreviated. Only as much of a name need be given as is necessary to distinguish it from all other topics that are reachable via the same path. Minimal abbreviations are shown both via the **LIST HELP** command and by a topic's header line.

Examples:

HELP COMMANDS HELP ABSOLUTE
Abbreviation - H CO HE A
This displays the current topic.

HELP COMMANDS LIST HELP
Abbreviation - H CO LIS HE
This displays information about the **LIST HELP** command.

H COM =JUMP
Abbreviation - H CO =
This displays information about using ISPF-style **=jump** commands from within Z/XDC.

H MAI 2021
This displays information about all software updates that have been applied to Z/XDC in 2021.

Help COmmands HElp Relative

The Z/XDC **HELP** command maintains a **Current Topic Pointer.** Every time a HELP topic is displayed, this pointer is set to point to that topic. Then the **HELP** command accepts keywords that locate other topics by their position **relative** to the current topic.

The **Current Topic Pointer** is also updated **during** command parse. As each command operand is resolved, the **Current Topic Pointer** is immediately updated to reference the resolved topic. Then if the next operand is one of these relative reference operands, its reference will be relative to the topic that the prior operand resolved to. See **HELP *EXAMPLES** for some examples. (Note, this itself is an example of a ***topicname** relative reference.)

Syntax:

```
HELP * ...
     *REPEAT ...
     *UP ...
     *DOWN ...
     *FORWARD ...
     *BACK ...
     *NEXT ...
     *PREVIOUS ...
     *topicname ...
```

Operands

*
***REPEAT** (or ***R)**
These are synonyms. They cause the current topic to be redisplayed.

***UP** (or ***U)** [F1]
This causes the topic directly above the current topic to be displayed.

***DOWN** (or ***D)** [F2]
This causes one of the lower topics directly below the current topic to be displayed. If no lower topics have yet been displayed, then the first lower one is displayed. If the current topic is displayed as a result of a preceding "*UP" command, then "*DOWN" redisplays that prior topic.

***FORWARD** (or ***F)** [F5]
***BACK** (or ***B)** [F4]
These display the next and previous topics that are accessible via the same path as the current topic (i.e. are at the same node or level in hierarchy).

***NEXT** (or ***N)** [F11]
***PREVIOUS** (or ***P)** [F10]
The Built-in Help topics are organized into a hierarchical or upside down tree-like structure. The **depth-first sequential** path through that upside down tree travels:
- Down (to daughter) where possible,
- Then over (to sister) until the next daughter occurs,
- Then up (to mother) when there are no more sisters.

The **next/previous** ordering follows this depth-first sequential path.

If you'd like to see what this sequential path looks like, the **LIST HELP** command displays the topics in their depth-first sequential order.

***topicname**
If the name given after the asterisk (*) is not one of the above keywords, then the command checks to see if it's the name of a subtopic of the current topic. If so, then command navigation continues to that topic.

However, if the given name is **not** the name of a subtopic of the current topic, then the command fails with a suitable error message.

...
These keywords can be use in combination with each other. As each keyword is examined, a new **current topic** is identified. The next keyword is then relative to this new topic.

Example: Type **"H *Next".** This will display the "next" topic, which is below this topic and named **EXAMPLES.**

Help COmmands HElp Relative Examples

Examples of topic references relative to **THIS** topic.

H * - Redisplays HELP COMMANDS HELP RELATIVE EXAMPLES.
H *U - Displays HELP COMMANDS HELP RELATIVE.
H *U *U *U - Displays HELP COMMANDS.
H *P - Displays HELP COMMANDS HELP RELATIVE. (Issue **LIST HELP *U *U 2** to see why.)
H *N - Displays HELP COMMANDS HELP MIXING.
H *U *F - Displays HELP COMMANDS HELP MIXING.
H *U *B - Displays HELP COMMANDS HELP ABSOLUTE.

H *F and **H *B** both fail here because this topic has no sisters. In other words there is no other topic that has the same mother as this topic. There is no other topic that is reachable by the same path as this topic. (The path to this topic is "HELP COMMANDS HELP RELATIVE".)

Suppose I had first issued **HELP COMMANDS SET.** Then the following commands will have the following results:

H *KEYS - Displays HELP COMMANDS SET KEYS (because KEYS is a subtopic of HELP COMMANDS SET).

H *KEYS SPECIAL - Displays HELP COMMANDS SET KEYS SPECIAL.
H *KEYS *SPECIAL - Also displays HELP COMMANDS SET KEYS SPECIAL.

H KEYS - But this command **fails** because KEYS is not a direct subtopic of the **HELP** main topic.

Note that using **name** and ***name** differ only when it is given as the first operand. When given as the second or subsequent operand, they function effectively the same.

Help COmmands HElp Mixing

Both absolute and relative references can be mixed on one HELP command. This works due to the fact that the HELP command sets the Current Topic Pointer each time it successfully parses and processes an operand. A following operand, therefore, functions relative to the current topic established by the preceding operand.

Examples:

HELP COMMANDS *N
This displays HELP COMMANDS SYNTAX. The SYNTAX topic is (according to the LIST HELP command) the "next" topic following the COMMANDS topic.

HELP COMMANDS *F
This displays the HELP SHORTCUTCMDS topic. This is the next topic following the COMMANDS topic and on the same level as the COMMANDS topic.

For the following examples, suppose that the initial "current" topic is HELP COMMANDS HELP.

H *U LIST
This displays the HELP COMMANDS LIST topic. The "*U" moves from HELP COMMANDS HELP up to HELP COMMANDS. The LIST topic then becomes accessible because it is one of the topics that is directly down from the HELP COMMANDS topic.

HELP R
This fails because there is no topic that is directly down from the top topic and whose abbreviation is "R".

Help COmmands HKeys

Z/XDC's **HKEYS** command functions similarly to the way in which the "KEYS" command does in ISPF: It displays a special panel (the "HELP Keys Panel") that shows Z/XDC's current PF key commands that are used during Built-in Help displays. The panel also shows certain options related to the HELP PF keys, and it allows you to change the PF key commands and options. For a description of the HELP Keys Panel, see HELP PROFILES MENU HKEYS.

Z/XDC's **HKEYS** command works regardless of whether or not ISPF is currently running in the Home Address Space.

Syntax:

HKEYS

This command should be given without operands. If operands are given, then this command becomes an alias of the **SET HKEYS** command, and the operands are processed accordingly. See HELP COMMANDS SET HKEYS for more information.

For general information about HELP PF keys, please see HELP FULLSCREEN PFKEYS HELPKEYS.

Help COmmands HOok

In order for Z/XDC to function correctly, it must be the newest recovery routine (ESTAE-type or FRR) for the environment of the code that you wish to debug. If Z/XDC is not the newest, then some other recovery routine will see ABENDs and breakpoints first, and that usually leads to failures that are hard to understand. For more information, see **HELP DEBUGGING ESTAES.**

The **HOOK** command is a very easy way to create a Z/XDC debugging session in an environment where it either is not already present or where it is present but other ESTAEs (or FRRs) have been created that are newer than Z/XDC.

The **HOOK** command can be used to create a Z/XDC debugging session either in any execution thread (including the current thread) running in the current (primary) address space or in any other address space in the System (security permitting).

Home Address Space

If you need to create or resume a Z/XDC debugging session in a piece of code that runs within an error recovery environment in which Z/XDC is not the newest recovery routine (ESTAE-type or FRR), then the **HOOK** command is a good way to make Z/XDC the newest.

Also, if you want to debug a Request Block driven exit routine (an attention exit, a timer exit, a DCB exit, etc.), then again, the **HOOK** command is a good way to do so. For more information, see **HELP DEBUGGING HOOK DYNAMIC HOMESPACE.**

Foreign Address Space

If you need to activate a Z/XDC debugging session in another address space, even one in which a Z/XDC interface has not already been created, then the **HOOK** command can do that too. For more information, see **HELP DEBUGGING HOOK DYNAMIC OTHERSPACES.**

What is a Hook?

A hook is a sequence of instructions that is dynamically zapped onto a target location. It jumps execution over to Hook Services prolog code.

When the **HOOK** command creates a hook, it saves the target’s original contents, and then zaps the instruction sequence into the target location.

Where can Hooks be Set?

In order to use the **HOOK** command, you need to have both **zap authority** (from security) and **store accessibility** to the location at which you want to place the hook.

In general, in order to use the **HOOK** command against a target in the Home Address Space, your program does not need to be running authorized. But then again it might need to be, depending upon whether or not that target code resides in protected storage.

If you are debugging a non-authorized program, and if you wish to set a hook into reentrant or refreshable code, then you might run into a problem because normally such programs are loaded into key 0 storage. You can get around this by using the keyword ddnames: **//xxxRENT8** and/or **//xxxREFR8** DD DUMMY's. See **HELP DDNAMES RENT9** and **HELP DDNAMES REFR8** for more information.

On the other hand, if you want to place a hook into another address space, then your program must be running authorized. For complete information, see **HELP HOOKS DYNAMIC SECURITY.**

Currently, hooks are not supported in 64-bit storage. They cannot be placed above the bar. (They do, of course, support code that runs with AMODE(64) set, but they must be placed below the bar.)

What Execution Environments are Supported by Hooks?

Hooks are designed to **create a debugging session** in arbitrary code without altering that code's registers or execution attributes in any way. It can be set into pretty much any kind of code there is. This includes:

- **Normal** code
- **Supervisor state** code
- **Any execution key**
- **Multitasking** code
- **SVC** routines
- **System exit** routines
- **PC** routines:
  - Stacking or basic
  - Space switching or not
  - Whatever
- **FRR** protected code
- **SRB** routines
- **Locked** or **disabled** code (with certain compromises)
- code executing in **PRIMARY** or **AR** ASC-MODE.
- code executing in **SECONDARY** or **HOME** ASC-MODE (with certain restrictions, See below)
- Pretty much anything else

If the code is executing above the bar (i.e. above 2G) then you must nominate a work register on the **HOOK** command (the **WORKREG=** operand).

If you wish to hook code that may be executing in SECONDARY or HOME ASC-MODE then you must indicate this by specifying ODDASCM=YES and you must also nominate a work register (the **WORKREG=** operand).

When the **HOOK** command creates a hook, it assigns a name of the form **HOOKnnnn.** This

name can be used to reference the hook in address expressions. Note, however, the following:

- The hook name is **not** an equate. It cannot be displayed by the **LIST EQUATES** command. It can only be displayed by the **LIST HOOKS** command.

- Even though the hook itself can survive past the end of the debugging session, its name cannot. A hook's name remains assigned to the hook only during the life of the current debugging.

- Subsequent debugging sessions will not be able to refer to or display the hook by its name, only by its address (which you must know and remember from the debugging session by which you set the hook). As far as other debugging sessions are concerned, the hook point is just unnamed.

Dynamic Hook, Static Hooks and Deferred Hooks

The **HOOK** command and **HDEFERRED** command create **dynamic** hooks. These are hooks that do not exist, of course, until the **HOOK** command (or **HDEFERRED** command) is issued to create them. See **HELP HOOKS DYNAMIC** for more information.

Z/XDC also supports **static** hooks. Those are hooks that can be created at assembly time by the **#XDCHOOK** macro. For more information about static hooks, see **HELP HOOKS STATIC.**

As just noted, Z/XDC supports **deferred hooks** that are dynamic **yet do not exist** until a targeted module is loaded into storage. A deferred hook is similar to a deferred breakpoint. For more information about deferred hooks, see **HELP HOOKS DEFERRED.**

Hooking Common Storage Routines and PC Routines

The **HOOK** command can (when running authorized and permitted by System Security) set hooks into programs that reside in common storage, including in the System Nucleus and in the PLPA! Obviously, you must be extraordinarily careful when doing this both because common storage routines can be executed by any address space in the System, and because common storage routines often are system critical. A misplaced hook can have such undesirable results as:

- Creating Z/XDC debugging sessions in another user's address space.
- Crashing major subsystems such as CICS, VTAM, JES2, IMS, DB/2 (to name just a few).
- Crashing the entire system in less time than it takes you to remove your finger from the ENTER key!
- Crashing the System hours, days, or even weeks later when everybody has moved on to other projects and no longer has any recollection of what they might have done.

The same is true for space switched PC routines located in private storage. Such code, like common storage routines, can be executed by any address space in the System. So again, you must be extra careful when setting such hooks.

For more information relating to debugging PLPA and Nucleus programs, see:

- **HELP SECURITY**
- **HELP DEBUGGING PLPA**

Preventing Debugging Sessions in the "Wrong" Address Space

Code located either in common storage or in space switched PC routines located in private storage can be executed from multiple address spaces simultaneously. This means that any hook set into such code can be executed by any address space! In other words, the first address space that reaches the hook **may well not be the aspace you want it to be!**

The problem is, the first address space that reaches a hook **must be the aspace that erases the hook!** You cannot control that. That's the way it **must** be in order to avoid corrupting the execution of a random address space. **But** whether or not a debugging session gets created... that is something you **can** control.

When you set a hook in common storage (or in a PC routine), you usually have it in mind that it is to be executed by some particular address space. But the reality is that unless you are very careful, the hook can be first executed by some address space other than the one you intended. This can lead to the undesirable result of a debugging session popping up at some random user's terminal who would then be totally mystified by what is going on. That's not good.

The **HOOK** command has operands **(ASID=** and **ASNAME=)** to help mitigate the consequences of a hook being first executed by a wrong address space. The hook will still always be erased. But these operands will prevent an unwanted debugging session from being created. Please see below for more information.

Predicting the Target Code's Execution Environment

Often, when you are issuing the **HOOK** command to set a hook, the target code will be running in an execution environment that is more or less similar to the environment in which Z/XDC itself is running. In these cases, you can just issue a simple **HOOK targetaddress** command and not bother with any other operands. But sometimes things get a bit more complicated.

Consider for example setting a hook into a different address space from the one in which Z/XDC is running. In this case, there is no relationship whatsoever between the execution environment from which Z/XDC received control and the target address space's execution environment.

The problem is, there really is no one-size-fits-all solution for hooks.
The **HOOK** command really does need to know the execution environment that will exist when a hook itself is reached by execution. Among other things, the hook control data areas need to be built in different subpools depending upon such things as...

- Whether the hook will be executed in supervisor state or problem state.
- Whether the hook will be executed while locked or unlocked.
- Whether the hook will be executed in AMODE 24, 31 or 64.
- And so forth.

These are things that the **HOOK** command cannot always predict correctly. Only you, the human, can know for sure.

The **HOOK** command does try to make predictions based on the execution environment of the program from which Z/XDC received control, but when such predictions turn out to be wrong, the **HOOK** command does have operands with which you can bring enlightenment

to its sometimes muddled choices.

Excessive Hooking? (Nope)

As noted above, a hook, when reached by execution, creates a debugging session. It does this by issuing an **ESTAEX** or **SETFRR** (as appropriate) to set up Z/XDC has the current execution environment's newest recovery routine.

However, the Hook Processing logic will try to avoid creating a new ESTAEX or FRR if it does not have to. So if execution reaches two or more hooks located within the same execution environment, extra ESTAEX's or FRR's will **not** be created.

Syntax:

```
HOOK addressexpression  ASID=aspacereference  ...
                        ASNAME=jobname

     STATE=SUPERVISOR  AMODE=24
           PROBLEM           31
           LOCKED            64
                             ANY

     WORKREG=n         ODDASCM=NO
                               YES
```

Shortcut: K

addressexpression
This identifies the **location** (address space and virtual address) where a hook is to be set. Six bytes, starting at this **Hook Point** are overlaid by the hook. A copy of those six bytes is saved so that they may be restored and executed when the hook is reached by program execution. For complete information about address expressions, see **HELP ADDRESSING.**

If the hook is set...
- Either in common storage,
- Or in private storage but within a space switched PC routine,

Then the hook will be associated with a specific address space according to the **ASID=** and **ASNAME=** operands discussed below.

The hook cannot be set into above the bar storage.

If the HOOK command executes successfully, then no messages are generated, and Z/XDC's current terminal display is not altered. On the other hand, if an error occurs, then a suitable error message is displayed.

ASID=aspacereference
ASNAME=jobname
You would use these operands when setting hooks either into common storage or into a

space switched PC routine located in private storage. They are filters that the Hook Processing logic (i.e. the Hook Support logic that is executed when a Hook Point is reached by execution...) that the Hook Processing logic uses to decide whether the **right** address space or a **wrong** address space has reached the Hook Point.

- **ASID=aspacereference**
  This identifies a specific, **currently existing** address space as being the **right** address space. **Aspacereference** can be:
  - A 4 digit hexadecimal number
  - A 1 to 8 character jobname
  - A keyword such as **HOME, PASID** etc.

  For details, see **HELP COMMANDS SYNTAX ASIDS.**

- **ASNAME=jobname**
  This identifies the **right** address space by name. The difference between this and **ASID=jobname** is:

  - With **ASID=jobname,** the target address space must already exist at the time the **HOOK** command is issued.

  - With **ASNAME=jobname,** the target address space can exist, but it doesn't need to. When the hook is created in common storage, it can be created before the target address space is even started.

Hooks are always immediately removed upon being reached by execution. For a discussion of the reasons why, see **HELP HOOKS DYNAMIC COMMONSTORAGE.**

So it is important that the **right** address space always be the first to reach a hook. In the event that a **wrong** address space has reached the hook, the hook will be erased, and the address space will be resumed so that it can execute the restored code and proceed as if nothing has happened. So another **HOOK** command will have to be issued to create a replacement hook.

The reason why hooks are always immediately erased upon being reached by execution is that is the only way an address space can be allowed to proceed uncorrupted. If for some reason something goes wrong such that the hook cannot be erased, then Hook Processing has no choice but to terminate the execution thread with an ABEND. For more information, see **HELP HOOKS DYNAMIC COMMONSTORAGE.**

It is recommended that these operands be used when creating a hook either in common storage or in a space switch PC routine, but it is not required. When these operands are not given, the target address space will be set to whatever address space Z/XDC's Foreign Address Space Mode (FASM mode) is set to. For more information, see **HELP VIRTMEM XDCACCESS FASM.**

STATE=SUPERVISOR
STATE=PROBLEM
STATE=LOCKED
Use these operands when the execution state of the Hook Point is going to be different from the execution state of the current code from which Z/XDC has received control.

For example, if Z/XDC has received control from an authorized program, and the Hook Point is in code that you expect to run in problem state, then you would use **STATE=PROBLEM** when setting the hook.

If you are placing a hook into code that you expect to run locked or disabled, then you would use **STATE=LOCKED.**

AMODE=24
AMODE=31
AMODE=64
AMODE=ANY
Using this operand influences whether the Hook Processing control blocks will be built in 24-bit or 31-bit storage. Absent this operand, the control blocks will be built in storage that is compatible with the location of the Hook Point.

WORKREG=n
This operand nominates a work register that can be used by the generated hook processing code that is specific to this **HOOK** request. You need to provide a work register if (1) you wish to hook addresses that are above the bar or (2) you wish to hook code that may be executing in SECONDARY or HOME ASC-MODE (you also have **ODDASCM=YES** specified). The work register must be a register number from 1 to 15 (R1 to R15) and the nominated register will be set to zero.

ODDASCM=NO
ODDASCM=YES
This operand indicates that the hook needs to support code that is executing in SECONDARY or HOME ASC-MODE. Failure to use **ODDASCM=YES** when the hook is executed by code that is in SECONDARY or HOME ASC-MODE will result in an S0D3 ABEND. Note that you must also specify a work register if you specify **ODDASCM=YES.**

Related commands:
- LIST HOOKS
- DELETE HOOKS
- K shortcut

Related topics:

HELP HOOKS: This is a comprehensive discussion of hooks and how to use them in a debugging session.

Examples:

```
MAP MYPROG.DISKIO
S Q MYPROG.DISKIO
HOOK .DCBEXIT
LIST HOOKS
FORMAT HOOK0001
```

The **MAP** command loads a module map of the MYPROG load module and a csect map of the DISKIO csect.

The **SET QUALIFIER** command makes MYPROG the default load module name and DISKIO the default csect name. (See **HELP ADDRESSING PARSERS ASM RESOLUTION** for more information about default names.)

The **HOOK** command sets a hook at the start of a Request Block driven DCB OPEN exit named DCBEXIT. (See **HELP EXECUTIONLEVELS** for more information about Request Blocks.)

The **LIST HOOKS** command displays information about the hook that has just been created.

The **FORMAT HOOK0001** command displays the hooked location with the hook itself properly formatted.

IEFACTRT is a customer written (usually) SMF exit routine that automatically receives control from the System at the end of every jobstep. It resides in common storage, usually in the PLPA. Suppose I've written some new code at label COMPUTE$, and I want to debug it.

Well, the first thing I need to do is bring up my own private sandbox system where there is very little other activity going on. This is to reduce the likelihood that the wrong address space will reach the hook (that I am about to set) ahead of me.

Then I can issue the following commands.

```
MAP IEFACTRT.SROUTINS
SET QUALIFIER IEFACTRT.SROUTINS
FORMAT .COMPUTE$
HOOK +0 ASNAME=TSODBC
```

The **MAP** command loads:

- A Binder map of the IEFACTRT load module.
- A csect map (if any) of the SROUTINS control section.

The **SET QUALIFIER** command makes IEFACTRT and SROUTINS the default load module name and csect name, respectively. (See **HELP ADDRESSING DOTPLUS.)**

The **FORMAT .COMPUTE$** command displays the code that I want to hook.

The **HOOK +0 ASNAME=TSODBC** command sets a hook at the location that I just displayed. (That's what the **+0** means.) The **ASNAME=TSODBC** operand means that a debugging session will be created only if the first address space to execute the hook is named TSODBC.

No matter what, when the first address space reaches the Hook Point, the hook will be erased. However, a debugging session will be started only if that address space is named TSODBC. In all other cases, warning messages **(DBC964E)** will be issued, and after the hook has been erased, that address space will be allowed to continue execution as if nothing had happened.

Warning: The following example should be attempted only by a System's Programmer who is responsible for JES2. In any case, attempting this example will fail for users who have not been suitable permitted by System Security. For more information about using Z/XDC with JES2 development, see **HELP DEBUGGING JES2.**

SET ASID JES2
MAP HASJES20 SYS1.LINKLIB
LIST TASKS
LIST RBS TCB#3
FORMAT RB#1+14?-2
HOOK +2
LIST HOOKS

The **SET ASID** command establishes Foreign Address Space Mode against JES2's address space.

The **MAP** command reads a module map of JES2's main load module from the library named SYS3.JES2.LINKLIB (a hypothetical name, of course).

The **LIST TASKS** command displays JES2's current subtask structure, and it also creates a series of equates named TCB#n that label the locations of each of the TCBs displayed. (See **HELP EQUATES BUILTIN** for more information about the **TCB#n** equates and the **RB#n** equates mentioned below.)

The **LIST RBS TCB#3** command displays the Request Blocks that are currently queued from JES2's jobstep task. (See **HELP EXECUTIONLEVELS** for information about Request Blocks.) Note, JES2, like most programs, spends a very large percentage of its time in a WAIT state. Consequently, this **LIST RBS** display almost always will show only one RB with the "Current Execution Location" pointing into the HASPNUC csect just past a BR R15 instruction that JES2 uses to branch-enter the System's WAIT service routine.

The **FORMAT RB#1+14?-2** command should display the above described BR R15 instruction.

The **HOOK +2** command sets a hook at JES2's return point from the WAIT service. This will cause a debugging session to be created almost immediately in JES2's address space in JES2's main task. You should be able to connect to that debugging session via cdf/XDC. (See **HELP XDCSRVER CDF** for more information.)

The **LIST HOOKS** command displays information about the hook that has just been created.

Help COmmands IPcs

(This command is only available under dump/XDC, but cannot be used in a batch environment)

What it Does

Z/XDC's **IPCS** command can be used to execute one or more arbitrary IPCS commands

and/or command procedures from within Z/XDC. The output from these commands is written to the current Window's scroll area as well as to the Z/XDC session log where it can be reviewed as needed.

Note that Z/XDC's **IPCS** command cannot be used when using IPCS in batch.

Almost any IPCS command can be executed; however, you should avoid commands that:

- Close or change the current dump dataset (because if this happens, dump/XDC will detect it and terminate the dump/XDC session),

- Change the allocation of IPCS's PRINT file (because dump/XDC itself monkeys with the PRINT file's allocation, so if you do it, it will mess that up).

How it Works

When you issue Z/XDC's **IPCS** command, whatever command(s) you enter will be prefixed by a prolog of preparation commands that do the following:
- Suspend IPCS terminal output,
- Allocates a temporary PRINT output dataset using a temporary ddname. (This is where the invoked IPCS command's output will be captured.)

Epilog commands will also be added that:
- Undo the changes made by the prolog commands.
- Extract the IPCS command's output from the temporary dataset,
- And write that output to Z/XDC's scroll area and session log.
- Delete the temporary dataset.

Syntax:

```
IPCS  command args
     'command args'
     'command args;command args;command args'
```

command
This is an IPCS command or command procedure name known to IPCS.

args
These are the arguments (if any) required by the given command. Their syntax is as defined by IPCS.

Notes:

- If a command string contains syntactically sensitive characters, then it must be enclosed within tics ('). Otherwise, the tics may be omitted.

- You may issue multiple IPCS commands as long as they are separated by semicolons

(;) and the entire string is tic enclosed.

- Many fields on the output lines from the IPCS command can be used by the point-and-shoot facility of Z/XDC. Note that some fields may contain 64-bit addresses (they will be 16 or 17 characters long) and in that case you should use **?!** as the point-and-shoot command (where **?** is the actual point-and-shoot command and **!** indicates that 64-bit addressing is to be used). Note that the **I** point-and-shoot command will issue a **LIST LOCATION** command (this would be particularly useful with the output from the **IPCS SYSTRACE** command).

Help COmmands ISpf

Z/XDC's **ISPF** command can be used to invoke ISPF from withing a Z/XDC debugging session. This works in both of the following cases:
- Z/XDC has been started from TSO's **READY** prompt.
- Z/XDC has been started from within ISPF.

When ISPF is not yet running, Z/XDC's **ISPF** command functions identically to issuing TSO's **ISPF** command from the **READY** prompt.

When ISPF is already running, Z/XDC's **ISPF** command reinvokes ISPF below Z/XDC. So...
- Z/XDC is running under ISPF.
- And ISPF is running again under Z/XDC.

Note, an alternative way to start ISPF without leaving your debugging session is to use Z/XDC's **TSO ISPF** command, but only sometimes...
- If Z/XDC has been started from TSO's **READY** prompt, then Z/XDC's **TSO ISPF** command will work, but then so will Z/XDC's **ISPF** command by itself.
- If Z/XDC was started within ISPF, then Z/XDC's **TSO ISPF** command will fail; however, Z/XDC's **ISPF** command, by itself, will succeed.

See **HELP COMMANDS TSO** for more information.

Syntax:

ISPF parameters

parameters
If you wish, you can specify any parameters that are valid for ISPF.

Before using the **ISPF** command, you can use the **SET ISR@PRIM** command to set the initial panel to be used by ISPF; however, **SET ISR@PRIM** will be effective **only when both** of the following are true:
- ISPF is invoked by Z/XDC's **ISPF** command (this command)
- And only when Z/XDC is already running within ISPF (i.e. not from the READY prompt).

See **HELP COMMANDS SET ISR@PRIM** for more information.

To return to Z/XDC from ISPF, **do not use =X.** Instead, use ISPF's **EXIT** command **(F3).**

Help COmmands Keys

Z/XDC's **KEYS** command functions in the same way it does in ISPF: It displays a special panel (the "Keys Panel") that shows Z/XDC's current PF key settings, 12 keys at a time, and allows you to change those settings. For a description of the Keys Panel, see HELP PROFILES MENU KEYS.

Z/XDC's **KEYS** command works regardless of whether or not ISPF is currently running.

Syntax:

KEYS

If the KEYS command is issued without operands from an Built-in Help display, then it causes the HELP PF keys to be displayed instead of the normal PF keys. In other words, it behaves like the HKEYS command.

This command should be given without operands. If operands are given, then this command becomes an alias of the **SET KEYS** command, and the operands are processed accordingly. See HELP COMMANDS SET KEYS for more information.

For general information about PF keys, please see HELP FULLSCREEN PFKEYS.

Help COmmands LEft

The **LEFT** command causes a display window to be scrolled leftwards. Columns to the left are brought into view.

The window that is scrolled is always the one that contains the cursor at the time that the command is issued.

shift-F10's factory default value is **LEFT -.** The factory default setting for F10 is **NOT** "LEFT". So be careful!

In the definition of shift-F10, the dash is important: It permits you to optionally place an operand on a window's command line and have that operand merged with the PF key's definition. If the dash were not included in the definition, then any potential operand placed on the command line would be ignored. For more information about this, see HELP FULLSCREEN PFKEYS.

Syntax:

LEFT CURSOR
FULL (Alias: **PAGE)**
DATA

HALF
MAX
nnn
omitted

CURSOR
If the cursor is within a window's data area, that window is scrolled leftwards so as to move the column containing the cursor to the right-hand edge of the window.

If the cursor is located on the window's command line, then LEFT CURSOR functions like LEFT FULL: The window is scrolled leftwards by its full width so as to bring data to the left of the window into view.

FULL
PAGE
The window containing the cursor is scrolled leftwards by its full width so as to bring data to the left of the window into view.

DATA
The window containing the cursor is scrolled leftwards by nearly its full width so as to bring the column that was at the window's left-hand edge over to its right-hand edge.

HALF
The window containing the cursor is scrolled leftwards by half its width so as to bring the column that was at the window's left-hand edge over to the middle of the display.

MAX
The window containing the cursor is scrolled leftwards so as to bring the leftmost column of data into view.

nnn
The window containing the cursor is scrolled leftwards by the number of columns specified.

omitted
The window containing the cursor is scrolled leftwards by its default amount. This default is established by the SET WINDOW HORIZONTAL command.

The default scroll amount can be displayed by the LIST WINDOW command.

Help COmmands LIBrarylists

For a Glossary of Terms, see HELP DEBUGGING C LIBRARYLISTS GLOSSARY.

Finding DWARF and C Source Files

When an XL C/C++ and Metal C program is compiled, one option is to produce debugging information in the form of a **DWARF data** side file. The method differs between XL C/C++ vs. Metal C, so see **HELP DEBUGGING C SETUP** for details.

Normally, the name of the DWARF file ends up being hardcoded into metadata in the resulting program object. Further, the names of the C source files end up being hardcoded into the DWARF data itself. Unfortunately, there are various scenarios wherein the name of the DWARF data file is **not** saved or the name, for one reason or another, has been changed since the program was compiled. This all is discussed in greater detail in HELP DEBUGGING C LIBRARYLISTS.

This **LIBRARYLISTS** command is one of the mechanisms provided to help cope with this "missing DWARF data" problem. The command organizes information by which a dsname or filename provided from the program object can be **corrected** to the actual name where the DWARF data or C source code can be found.

(Another, simpler mechanism is to use the **DWARFLIB=** and **SOURCELIB=** operands on the **MAP** command.)

Library Lists

In order to understand the purpose of the LIBRARYLISTS command, you need to understand **Library Lists Management** (which is described in great detail in the several topics starting with **HELP DEBUGGING C LIBRARYLISTS.** Reading that information will help considerably with understanding this command.)

Briefly, the **LIBRARYLISTS** command builds lists that c/XDC can search when it needs to find a DWARF data file or a C source code file during the C Source Image Map build process. When c/XDC needs to open a DWARF file, for example, it gets the file's **Original Name** from a hardcoding that the XL C/C++ compiler passed to the Binder and that the Binder saved in the program object. c/XDC then calls Library Lists Management to see if that Original Name needs to be **redirected** to a different dataset, library, file or folder. LLM then consults lists that you have created with the **LIBRARYLISTS** command and passes back to c/XDC one or more **Candidate Names** for c/XDC to examine to see if it contains the DWARF data (or C source code) that it needs.

As discussed in HELP DEBUGGING C LIBRARYLISTS LISTS, Library Lists Management deals with three kinds of lists. The LIBRARYLISTS command is used to build and manage two of them:

- A **Redirect List** consists of entries that contain **Match Patterns** connected to **Result Objects.** These Result Objects can be many things: dataset names, library names, HFS file names, folder names, //ddnames or names of **Candidates Lists.** See HELP DEBUGGING C LIBRARYLISTS LISTS REDIRECTLIST for details.

  Redirect Lists are built by the **LIBRARYLISTS REDIRECT** command.

- **Candidates Lists** are lists of Result Objects (see above). They are used when you want a Match Pattern to link to a collection of Candidate Names. These lists are described in HELP DEBUGGING C LIBRARYLISTS LISTS CANDIDATESLISTS.

  Candidates Lists are built by the **LIBRARYLISTS ADD** command.

- A **Results List** is a temporary List built by Library Lists Management for passing Candidate Names back to c/XDC. It is described fully in DEBUGGING C LIBRARYLISTS LISTS RESULTSLIST and need not be discussed further here.

List Families

There are two **Families** of Library Lists, one for DWARF files and one for CSOURCE files.

Each Family contains its own Redirect List which may link to one or more Candidates Lists. For more information, see HELP DEBUGGING C LIBRARYLISTS LISTS FAMILIES.

Nearly all LIBRARYLISTS commands specify which Family they are to deal with.

Families can be deleted as a whole by the LIBRARYLISTS RESET command.

Syntax

The **LIBRARYLISTS** command has numerous subcommands, each having its own requirements; but overall, they all follow a common syntax that is pretty similar to the rest of Z/XDC, but nonetheless has some differences, so please pay attention...

In particular, the command has its own way of recognizing case sensitive strings, which is particularly important if you want to provide an HFS path/file name.

For details, please see **HELP * SYNTAX.**

Library Lists Management

Managing a Library List requires a variety of actions such as:
- Creating/deleting/displaying lists,
- Creating/deleting/displaying list entries,
- Saving/restoring Library Lists Data
- Running test cases to make sure your lists do what you want them to do.

Accordingly, the **LIBRARYLISTS** command has a number of subcommands as follows. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

LIBRARYLISTS ADD {DWARF|CSOURCE} ...
LL
Adds one or more **Result Objects** to an existing or new **Candidates List.**

Result Objects are described in HELP DEBUGGING C LIBRARYLISTS LISTS CANDIDATESLISTS RESULTOBJECTS.

LIBRARYLISTS CHECKPOINT
LL CKPT

Creates a **Saved Instance** of the **Active Instance** of the Library Lists Data by copying it into buffer located within the in-storage instance of Z/XDC's Session Profile Data. See HELP DEBUGGING C LIBRARYLISTS LISTS SAVING for more information.

LIBRARYLISTS CHECKPOINT RESTORE
LL CKPT REST
Restores the **Active instance** of the Library Lists Data from a previously saved **Saved Instance.**

LIBRARYLISTS DEFAULT {DWARF|CSOURCE} ...
Names a particular **Candidates List** to be the "default" list. Then when another LIBRARYLISTS command omits any reference to Candidates Lists, the default list will be the list whose use is implied.

LIBRARYLISTS DELETE {DWARF|CSOURCE} ...
LL
Either deletes **Result Objects** from a Candidates List or deletes **entire Candidates Lists.**

LIBRARYLISTS DELETE {DWARF|CSOURCE} PATTERN=matchpattern
LL
Delete entries from a Family's **Redirect List.**

LIBRARYLISTS REDIRECT {DWARF|CSOURCE} ...
Adds one or more Pattern Match entries to a Family's Redirect List. See HELP DEBUGGING C LIBRARYLISTS LISTS REDIRECTLIST for more information.

LIBRARYLISTS RESET {DWARF|CSOURCE|ALL}
Deletes one or both **Families** of lists. Each Family's **Redirect List** is deleted, along with all **Candidates Lists** owned by the deleted Family.

LIBRARYLISTS SHOW [DWARF|CSOURCE] [ALL] [DETAILS]
Displays one or more Redirect Lists and Candidates Lists either in summary or in detail.

LIBRARYLISTS TEST MATCH ...
Shows the Match variables that result when a trial Original Name is tested against a trial Match Pattern.

LIBRARYLISTS TEST {DWARF|CSOURCE} TRY ...
For a given set of lists, this command shows the **Results List** that Library Lists

Management would return to c/XDC.

Help COmmands LIBrarylists SYntax

For a Glossary of Terms, see HELP DEBUGGING C LIBRARYLISTS GLOSSARY.

LIBRARYLISTS General Command Syntax

The LIBRARYLISTS command has several keywords and parameters. Here are a few rules.

LL is an **alias** for LIBRARYLISTS.

The word LIBRARYLISTS itself currently (as of Z/XDC z2.2) can be abbreviated to **LIB,** but that might change in the future should another Z/XDC command be created whose name also starts with LIB.

Specifying Keywords and Their Values

As is generally the case throughout the rest of Z/XDC, keyword operand names can be abbreviated down to the minimum length necessary to keep them unique amongst the names of all other keywords existing in the same context.

Sometimes shorter **aliases** are supported for long keywords when there is either a common or useful variation whose spelling differs from the start of the original keyword. **LL** being an alias of LIBRARYLISTS is an example of this.

When **multiple** operands are given, they are delimited from each other by **blanks.** Comma delimiters are not permitted. (This rule differs from the rest of Z/XDC where commas and blanks can be used interchangeably.)

Some keywords have values, some do not. When a keyword takes a value, an **=** will be needed to connect the keyword to its value. Example: **LISTNAME=value**

Some keywords accept multiple values, in which case the values are given within a parenthesized list and are delimited by commas. Example: **LISTNAME=(value1,value2,value3)**

When a keyword accepts **multiple** values the delimiters between the values **must be commas.** Blanks are not permitted. (This rule differs from the rest of Z/XDC where commas and blanks can be used interchangeably.)

Whenever a single value is given, enclosing parentheses **usually** are optional. This is true even for keywords that accept only single values. For Example: **LISTNAME=value1** and **LISTNAME=(value1)** mean the same thing. (This rule differs from the rest of Z/XDC where generally parentheses cannot be used with keywords that accept only single values.)

For single values, enclosing parentheses are **required** if the value being given **contains a blank!** (such as can happen with HFS filenames).

All keywords and parameters may appear in any order on the command line.

However, values within a list may be **position-sensitive.**

HFS Path/file Names and Other Strings

Unquoted values are generally upcased but there is an exception: If the value contains a forward slash (/), then it is taken to be a path/file name and, therefore, is left with the casing that you gave it.

Since only commas (and not blanks) are value delimiters, blanks are instead considered to be part of the value in which they appear. (This allows blanks to be contained within path/file names.)

Regarding values enclosed within single-quotes (’):

- Values enclosed within single-quotes (**’**) are **upcased.** (This includes **’path/file names’.** which becomes **PATH/FILE NAMES.**)
- If the value includes a single-quote, it needs to be doubled up. Example: **’Don’’t’** becomes **DON’T**
- The enclosing single-quotes are stripped away and embedded doubled single-quotes are singlized before a given value is used.
- Double-quotes (**"**) embedded within single-quoted strings must not be doubled up.

Regarding values enclosed within double-quotes ("):

- Values enclosed within double-quotes retain their casings.
- If the value includes a double-quote, it needs to be doubled up. Example: **"He said ""No!""""** becomes **He said "No!"**
- The enclosing double-quotes are stripped away and embedded doubled double-quotes are singlized before a given value is used.
- Single-quotes (**’**) embedded within double-quoted strings must not be doubled up. Example: **"He said ""No! Don’t!""""** becomes **He said "No! Don’t!".**

Support for double-quotes (**"**) is unique to the LIBRARYLISTS command. The rest of Z/XDC has no special support for double-quotes.

Match Patterns

Match Patterns are used in **Redirect List** entries. Library Lists Management searches a Redirect List by comparing a file’s **Original Name** from c/XDC to each Redirect List entry’s **Match Pattern.**

A Match Pattern may contain **Wildcard Characters** [**?** * **] allowing the Pattern to match a variety of Original Names. For more information about Match Patterns, see HELP DEBUGGING C LIBRARYLISTS LISTS REDIRECTLIST WILDCARDS.

Result Objects

A Result Object is a **Candidate filename** or **library Name** but prior to the resolution of **match variables.**

A Result Object may indicate the name of a file, folder, library or dataset, **or** it may be a **//ddname** or the name of **another** Candidates List (to be included in the resolution process).

Result Objects are given in **Redirect Lists** and in **Candidate Lists.** They can be any of the following:

- The name of a classic z/OS sequential file
- (Including a specific PDS member name),
- The name of a library (PDS or PDSE),
- The name of an HFS file (including its path),
- The name of an HFS folder,
- The //ddname of a current allocation of one or more datasets, files, folders and/or libraries,
- The name of another Candidates List (to be included by reference).

Unless the Result Object is a //ddname or the name of a Candidates List, it **may** contain **Match Variables** which are replaced by those parts of the Original Name that were matched by Wildcards from the Match Pattern. For more information, see HELP DEBUGGING C LIBRARYLISTS LISTS REDIRECTLIST MATCHVARIABLES.

The syntax of Result Objects is discussed in further detail in HELP DEBUGGING C LIBRARYLISTS LISTS CANDIDATESLISTS RESULTOBJECTS.

Candidate Names and Result Lists

Once all wildcards have been resolved, the result is a **Candidate Name** that is added to the **Results List** that is passed back to c/XDC for processing.

Help COmmands LIBrarylists Add

For a Glossary of Terms, see HELP DEBUGGING C LIBRARYLISTS GLOSSARY.

The **LIBRARYLISTS ADD** command adds one or more **Result Objects** to an existing or new **Candidates List.**

Syntax:

```
LIBRARYLISTS ADD DWARF    LISTNAME=listname  ENTRY=(value,value,...)
LL               CSOURCE  LNAME=listname     ENTRIES=(value,value,...)
                          omitted
```

Rules:

- Operands shown in the same column are mutually exclusive.
- Keyword names can be truncated to the minimum number of characters needed to keep them unique. (But most keywords have a minimum length of 3).
- Operands may be given in any order.
- Parenthesized values **must** be separated by **commas** (not blanks).
- For a discussion of additional Syntax Rules that are unique to the LIBRARYLISTS commands, see HELP COMMANDS LIBRARYLISTS SYNTAX.

Operand Descriptions

DWARF
CSOURCE
This identifies the **Family** to which the Candidates List will or does belong.

LISTNAME=listname
LNAME=listname
omitted
This identifies the name of the **Candidates List** to be updated or created.

A **listname:**
- May be up to 8 characters long,
- May contain any mix of alphabetic and numeric characters,
- May start with a digit,
- May be entirely digits,
- May not contain any special characters,
- Not even the national characters (@ # and $).

If this operand is omitted, then the default Candidates List is targeted.

ENTRY=value
ENTRY=(value,value,...)
ENTRIES=(value,value,...)
This is a list of one or more **Result Objects** to be added to the **Candidates List.**

A **Result Object** can be any of the following:
- The name of a classic z/OS sequential file
- (Including a specific PDS member name),
- The name of a library (PDS or PDSE),

- The name of an HFS file (including its path),
- The name of an HFS folder,
- The ddname of a current allocation of one or more datasets, files, folders and/or libraries,
- The name of another Candidates List (to be included by reference).

For detailed syntax information regarding Result Objects, HELP DEBUGGING C LIBRARYLISTS LISTS CANDIDATESLISTS RESULTOBJECTS.

When multiple Result Objects are given, they **must** be separated by **commas** (not blanks).

When a single Result Object is given, but its name includes a blank (such as a path/filename might), then that name **must** be enclosed within parentheses.

The values given typically either create a new Candidates List or add new Result Objects to an existing Candidates List, but for **existing Lists,** they may also reference **existing list entries** either by an exact name match or by a **relative position number** (example: **#3**).

All Result Objects given will be moved to the top of the Candidates List's search order, in the order given.

The Result Objects do not have to actually exist at LIBRARYLISTS command time. Any that still do not exist at **MAP** command time will be ignored.

Examples

Suppose I have an existing Candidates List that look likes this:

```
    LL SHOW DWARF LISTNAME=MYLIST
[or LL SHO  D     LN=MYLIST       ]

    LIST MYLIST
      #1 MVS  MY.DWARF3
      #2 MVS  MY.DWARF2
      #3 HFS  /u/rob/my dwarf9
```

Now suppose I want to:
- Rearrange the list into sorted order,
- Add MY.DWARF1 to the front of the list,
- Insert MY.DWARF4 following MY.DWARF3.

The following **ADD** command will do the trick:

```
    LL ADD DWARF LISTNAME=MYLIST ENTRIES=(MY.DWARF1,#2,#1,MY.DWARF4)
[or LL ADD D     LN=MYLIST       ENT=(MY.DWARF1,#2,#1,MY.DWARF4)     ]

    LIST MYLIST
      #1 MVS  MY.DWARF1
      #2 MVS  MY.DWARF2
      #3 MVS  MY.DWARF3
      #4 MVS  MY.DWARF4
      #5 HFS  /u/rob/my dwarf9
```

If I forget the commas (which I did BTW), this happens:

```
    LL ADD DWARF LISTNAME=MYLIST ENTRIES=(MY.DWARF1 #2 #1 MY.DWARF4)
[or LL ADD D     LN=MYLIST       ENT=(MY.DWARF1 #2 #1 MY.DWARF4)     ]

    LIST MYLIST
      #1 MVS  MY.DWARF1 #2 #1 MY.DWARF4
      #2 MVS  MY.DWARF3
      #3 MVS  MY.DWARF2
      #4 HFS  /u/rob/my dwarf9

[sigh]
```

Help COmmands LIBrarylists Checkpoint

For a Glossary of Terms, see HELP DEBUGGING C LIBRARYLISTS GLOSSARY.

The **LIBRARYLISTS CHECKPOINT** command copies the **Active Instance** of the Library Lists Data into a buffer located in the in-memory instance of Session Profile Data. This is called the **Saved Instance** of the Library Lists Data. For more information, see HELP DEBUGGING C LIBRARYLISTS LISTS SAVING.

This Saved Instance has **not** been hardened to disk. In order to do that, you will need to use the **PROFILE SAVE** command. (Creates the **Stored Instance).**

While the size of the Active Instance is unlimited, The size of the Saved Instance is not. It cannot be larger than **8,000 bytes.** If you attempt to checkpoint an Active Instance that has become too large, the command will issue an error message and abort. See HELP DEBUGGING C LIBRARYLISTS LISTS SAVING for more information, including a suggested workaround.

One of the response messages to LIBRARYLISTS CHECKPOINT reports the amount of the Saved Instance storage that is needed to hold the Library Lists Data.

Note, use the **LIBRARYLISTS CHECKPOINT RESTORE** command to restore the Saved Instance back to the Active Instance.

Syntax:

```
    LIBRARYLISTS CHECKPOINT   omitted
                 CKPT         RESTORE
```

Rules:

- Operands shown in the same column are mutually exclusive.

- Keyword names can be truncated to the minimum number of characters needed to keep them unique. (But most keywords have a minimum length of 3).

Operand Descriptions

omitted
RESTORE
This indicates whether a save or restore operation is to be done.

Examples

LL CHECKPOINT
[or LL CHE]
[or LL CKPT]
[or LL CKP]

Library Lists checkpointed to PROFILE area - 1% used.
Saved at 10/01/2016 12:19:43
Issue PROFILE SAVE to harden to disk.

LL CKPT RESTORE
[or LL CKPT REST]

Library Lists restored from checkpoint at 10/01/2016 12:19:43

[Output from an internally issued LL SHOW command]

Help COmmands LIBrarylists Checkpoint Restore

For a Glossary of Terms, see HELP DEBUGGING C LIBRARYLISTS GLOSSARY.

To restore the Active Instance of Library Lists Data just use **LIBRARYLISTS CHECKPOINT RESTORE.** This copies the Saved Instance into the Active Instance. For further information, see HELP COMMANDS LIBRARYLISTS CHECKPOINT.

See HELP DEBUGGING C LIBRARYLISTS LISTS SAVING for more information about the various Instances of Library Lists Data.

Help COmmands LIBrarylists DEFault

For a Glossary of Terms, see HELP DEBUGGING C LIBRARYLISTS GLOSSARY.

The **LIBRARYLISTS DEFAULT** command determines which Candidates List will be the **default** list for a given Family. Each Family can have its own default Candidates List.

When a default Candidates List is established, then:

- If the **LISTNAMES=** operand is omitted from the **LIBRARYLISTS ADD** command, then the default Candidates List is added to.
- If both the **LISTNAMES=** operand and the **ALL** operand are omitted from the **LIBRARYLISTS SHOW** command, then the default Candidates List is displayed.
- The **USE=DEFAULT** operand can be used on the **LIBRARYLISTS REDIRECT** command. This means that a Match Pattern can be connected to whatever Candidates List happens to be the current default.

Syntax:

```
LIBRARYLISTS DEFAULT DWARF      LISTNAME=listname
LL                   CSOURCE    LNAME=listname
                                NONE
```

Rules:

- Operands shown in the same column are mutually exclusive.
- Keyword names can be truncated to the minimum number of characters needed to keep them unique. (But most keywords have a minimum length of 3).
- Operands may be given in any order.

Operand Descriptions

DWARF
CSOURCE
This identifies the **Family** to which the Candidates List will or does belong.

LISTNAME=listname
LNAME=listname
NONE
This identifies the name of the **Candidates List** to be made the default.

If **NONE** is specified, then Library Lists Management operates without a default Candidates List.

The specified **listnames** must preexist. You can use the **LIBRARYLISTS SHOW family ALL** command to see what Candidates Lists currently exist.

Examples

Suppose I have existing Candidates Lists that look likes this:

```
    LL SHOW DWARF ALL
[or LL SHO  D     ALL ]

    Library List DWARF
      #1  REDIRECT: '?**(*)' -> LIST(DBCSLIST)

      LIST MYLIST
      LIST DBCSLIST
```

Now suppose I want to:
- Make MYLIST the **default** Candidates List
- Change the **?**(*)** Redirect List entry to use the default list.

I will need two commands to do this.

First, let's make MYLIST the default list:

```
    LL DEFAULT DWARF LNAME=MYLIST
[or LL DEF     D     LN=MYLIST    ]

    Library List DWARF
      #1  REDIRECT: '?**(*)' -> LIST(DBCSLIST)

      LIST MYLIST    DEFAULTED
```

Now, let's make the **?**(*)** Redirect List entry use it:

```
    LL REDIRECT DWARF PATTERN=#1 USE=DEFAULT
[or LL RED      D     PAT=#1     USE=DEFAULT ]

    Pattern '#1' is now DEFAULT
    Library List DWARF
      #1  REDIRECT: '?**(*)' -> DEFAULT LIST(MYLIST)
```

Help COmmands LIBrarylists DELete

For a Glossary of Terms, see HELP DEBUGGING C LIBRARYLISTS GLOSSARY.

The **LIBRARYLISTS DELETE** command can be used:

- To delete **Result Objects** from a **Candidates List,**

- To delete entire **Candidates Lists,**
- To delete **Pattern Match** entries from a **Redirect List.**

Syntax:

Deleting Result Objects from a Candidates List:

```
    LIBRARYLISTS DELETE DWARF    LISTNAME=listname  ENTRY=(value,...)
    LL                  CSOURCE  LNAME=listname     ENTRIES=(value,...)
```

Deleting entire Candidates Lists:

```
    LIBRARYLISTS DELETE DWARF    LISTNAMES=(listname,listname,...)
    LL                  CSOURCE  LNAMES=(listname,listname,...)
```

Deleting Pattern Match entries from a Redirect List:

```
    LIBRARYLISTS DELETE DWARF    PATTERNS=(pattern,pattern,...)
    LL                  CSOURCE
```

Rules:

- Operands shown in the same column are mutually exclusive.
- **PATTERNS=** is mutually exclusive both with **LISTNAMES=** and with **ENTRIES=**
- Keyword names can be truncated to the minimum number of characters needed to keep them unique. (But most keywords have a minimum length of 3).
- Operands may be given in any order.
- Parenthesized values **must** be separated by **commas** (not blanks).

- For a discussion of additional Syntax Rules that are unique to the LIBRARYLISTS commands, see HELP COMMANDS LIBRARYLISTS SYNTAX.

Operand Descriptions

DWARF
CSOURCE

This identifies the **Family** from which you wish to delete lists or entries.

PATTERNS=(pattern,pattern,...)

Use of this operand indicates that you wish to delete Match Patterns from a Family's Redirect List.

The **patterns** may be either:

- A Match Pattern's **relative position** in the Redirect List (example: **#3**),

- Or an **exact match** to the Match Pattern itself.

When multiple patterns are given, they **must** be separated by **commas** (not blanks).

When PATTERNS= is given, both LISTNAMES= and ENTRIES= **must be omitted.**

Note, using the PATTERNS= operand on the LIBRARYLISTS DELETE command is identical to using **USE=NONE** on the LIBRARYLISTS REDIRECT command.

LISTNAMES=(listname,listname,...)
LNAMES=(listname,listname,...)
Use of this operand indicates that you wish to delete either one or more **Result Objects** from a **Candidates List** or one or more **entire Candidates Lists.**

The **Listnames** operand identifies either a single **Candidates List** to be updated, or one or more **Candidates Lists** to be deleted.

All given **listnames** must preexist. You can use the **LIBRARYLISTS SHOW family ALL** command to see what Candidates Lists currently exist.

When **LISTNAMES=** is present:
- The **ENTRIES=** operand is optional,
- The **PATTERNS=** operand must be omitted.

When **ENTRIES=** is given:
- The intent is to delete Result Objects from a Candidates List,
- The **LISTNAMES=** operand may not name more that one currently existing Candidates List.

When **ENTRIES=** is omitted:
- The intent is to delete one or more entire Candidates Lists,
- The **LISTNAMES=** operand may name multiple Candidates Lists.

ENTRY=value
ENTRY=(value,value,...)
ENTRIES=(value,value,...)
This is a list of one or more **Result Objects** to be deleted from a currently existing **Candidates List.**

The **values** may be either:
- A Result Object's **relative position** in the Candidates List (example: **#3**),
- Or an **exact match** to the Result Object itself.

You can use the **LIBRARYLISTS SHOW family ALL DETAILS** command to see what Result Objects are defined in all Candidates Lists. The command also shows what their relative positions are in the lists.

When multiple Result Objects are given, they **must** be separated by **commas** (not blanks).

When **ENTRIES=** is given:
- The intent is to delete Result Objects from a Candidates List,
- The **LISTNAMES=** operand may not name more that one Candidates List.

When **ENTRIES=** is omitted:
- The intent is to delete one or more entire Candidates Lists,
- The **LISTNAMES=** operand may name multiple Candidates Lists.

Examples

Suppose I have existing Candidates Lists and a Redirect List that look likes this:

```
    LL SHOW DWARF ALL DETAILS
[or LL SHO  D     ALL DET     ]

    Library List DWARF
      #1  REDIRECT: '?**(*)' -> LIST(DBCSLIST)
      #2  REDIRECT: '**(*)'
                USE: MVS(hlq.XDCV31.XDCDWARF)
      #3  REDIRECT: '*.MYDWF2' -> DEFAULT LIST(unknown)

      LIST MYLIST
        #1 MVS  MY.DWARF3
        #2 MVS  MY.DWARF2
        #3 HFS  /u/rob/my dwarf

      LIST DBCSLIST
        #1 MVS  DBC.DWARFLIB
```

Notice the following:
- The **DWARF** Family is being displayed.
- Its **Redirect List** contains three **Pattern Match** entries.
- There are two Candidates Lists defined.

Now suppose I want to:
- Delete the 2nd Result Object from the MYLIST Candidates List,
- Delete the DBCSLIST Candidates List entirely,
- Delete the 2nd and 3rd Redirect List entries.

I will need three commands to do this.

First, let's delete the 2nd Result Object from the MYLIST Candidates List:

```
    LL DELETE DWARF LNAME=MYLIST ENTRY=#2
[or LL DEL    D     LN=MYLIST    ENT=#2   ]

    Library List DWARF
      #1  REDIRECT: '?**(*)' -> LIST(DBCSLIST)
      #2  REDIRECT: '**(*)'
                USE: MVS(hlq.XDCV31.XDCDWARF)
      #3  REDIRECT: '*.MYDWF2' -> DEFAULT LIST(unknown)

      LIST MYLIST
        #1 MVS  MY.DWARF3
        #2 HFS  /u/rob/my dwarf
```

```
      LIST DBCSLIST
        #1 MVS  DBC.DWARFLIB
        #2 HFS  /u/dbc/dwarflib
```

Next, let's delete the DBCSLIST entirely:

```
    LL DELETE DWARF LNAME=DBCSLIST
[or LL DEL    D     LN=DBCSLIST    ]

    1 of 1 list deleted
```

```
    LL SHOW DWARF ALL DETAILS
[or LL SHO  D     ALL DET     ]

    Library List DWARF
      #1  REDIRECT: '?**(*)' -> LIST(DBCSLIST)
      #2  REDIRECT: '**(*)'
                USE: MVS(hlq.XDCV31.XDCDWARF)
      #3  REDIRECT: '*.MYDWF2' -> DEFAULT LIST(unknown)

      LIST MYLIST
        #1 MVS  MY.DWARF3
        #2 HFS  /u/rob/my dwarf
```

Finally, let's delete the two Redirect List Match Patterns we don't like anymore:

```
    LL DELETE DWARF PATTERNS=(#3,**(*))
[or LL DELETE DWARF PAT=(#3,**(*))      ]

    Pattern '#3' is now removed
    Pattern '**(*)' is now removed
    Library List DWARF
      #1  REDIRECT: '?**(*)' -> LIST(DBCSLIST)
```

Help COmmands LIBrarylists REDirect

For a Glossary of Terms, see HELP DEBUGGING C LIBRARYLISTS GLOSSARY.

A **Redirect List** is a list that consists of one or more entries that connect **Match Patterns** to named **Candidates Lists** of files, folders, datasets, libraries and //ddnames that c/XDC is to search for the needed DWARF data or C source file.

There is one Redirect List per Family (of lists).

Each entry in a Family's Redirect List contains two fields:

- The first is a **Match Pattern** which may contain one or wildcard characters (? *

and **) intermixed amongst constant characters. A Match Pattern is intended to match one or more Original Names from c/XDC. (Example: **MY.LIBRARY(*)** .) For more information about wildcard characters, see HELP DEBUGGING C LIBRARYLISTS LISTS REDIRECTLIST WILDCARDS.

- The second field is one or more **Result Objects.** These objects may be the names of datasets, files, folders, libraries, //ddnames and **named Candidates Lists** to be further searched or returned. For detailed descriptions of Result Objects, see HELP DEBUGGING C LIBRARYLISTS LISTS CANDIDATESLISTS RESULTOBJECTS.

When an **Original Name** is received from c/XDC, that name is used to search the appropriate Family's Redirect List. For each matched entry, one or more Candidate Names is added to the **Results List.**

Syntax:

LIBRARYLISTS REDIRECT	**DWARF**	**PATTERNS=(pattern,...)**	**USE=resultobject**
LL	**CSOURCE**		**USE=DEFAULT**
			USE=NONE
			omitted

Rules:

- Operands shown in the same column are mutually exclusive.
- Keyword names can be truncated to the minimum number of characters needed to keep them unique. (But most keywords have a minimum length of 3).
- Operands may be given in any order.
- For a discussion of additional Syntax Rules that are unique to the LIBRARYLISTS commands, see HELP COMMANDS LIBRARYLISTS SYNTAX.

Operand Descriptions

DWARF
CSOURCE
This identifies the **Family** to which the Candidates List will or does belong.

PATTERNS=(pattern,pattern,...)
This identifies one or more new or existing Redirect List entries that you want to assign the same Result Object to.
The **patterns** may be either:
- An existing Match Pattern's **relative position** in the Redirect List (example: **#3**),
- Or an **exact match** to an existing Match Pattern,
- Or a **new** Match Pattern that you wish to add to the list.

When multiple patterns are given, they **must** be separated by **commas** (not blanks).

USE=resultobject

The **USE=** operand connects the Match Pattern to one or more **Result Objects.** For detailed information about what Result Objects are and the syntax of coding them, see HELP DEBUGGING C LIBRARYLISTS LISTS CANDIDATESLISTS RESULTOBJECTS. But for now, let me give just a brief overview.

Ultimately, a Result Object leads to one or more **Candidate Names.** These are names of datasets, files, libraries of folders where c/XDC can look to find DWARF data or C source code.

Library List Management uses Candidate Names to build **Result Lists** which eventually are passed back to c/XDC for his consideration. He will search each Candidate in the Results List looking for the DWARF data or C source code that he needs for building Source Image Maps.

There are two kinds of Result Objects, **Individual** objects and **Collective** objects:

- **Individual Result Objects** are strings that are specific Candidate Names, except that the strings may contain **Match Variables** that would need to be resolved before the name can be added to the Results List.

- **Collective Result Objects** are either //ddnames or the names of Candidate Lists:

  - A **//ddname** must be the name of an existing allocation to a concatenation of one or more datasets, files, folders and libraries, each of which will be added to the Results List.

  - A **Candidates List** is a list of Result Objects which is to be further examined. The Result Objects in a Candidates List may themselves be either Individual or Collective. Individual Result Objects are resolved and added to the Results List. Collective Result Objects are further resolved.

For detailed information about the syntax of Result Objects, see HELP DEBUGGING C LIBRARYLISTS LISTS CANDIDATESLISTS RESULTOBJECTS.

USE=DEFAULT

This connects the Match Pattern to the current **default** Candidates List. The default list is set by the **LIBRARYLISTS DEFAULT** command.

USE=NONE

This **deletes** the Match Pattern from the Redirect List. This action is identical to that produced by the **LIBRARYLISTS DELETE** command.

[USE= is omitted]

Omit the USE= operand when all you want to do is reorder the Results List.

Examples

Suppose I already have a couple of Candidate Lists and an empty Redirect List as follows:

```
    LL SHOW DWARF ALL DETAILS
[or LL SHO  D     ALL DET     ]

    Library List DWARF
      Redirection is OFF

      LIST MYLIST    DEFAULTED
        #1 MVS  MY.DWARF3
        #2 HFS  /u/rob/my dwarf

      LIST DBCSLIST
        #1 HFS  /u/dbc/dwarflib
        #2 MVS  DBC.DWARFLIB
```

First, I want to create a Match Pattern that will redirect all Original Names that specify a PDS[E] member name to a member of the same name in a library named MY.DWARFLIB.

```
    LL REDIRECT DWARF PATTERNS=**(*)  USE=MY.DWARFLIB(&2)
[or LL RED      D     PAT=**(*)       USE=MY.DWARFLIB(&2)  ]

    Pattern '**(*)' is now redirected to MVS(MY.DWARFLIB(&2))
    Library List DWARF
      #1  REDIRECT: '**(*)'
                USE: MVS(MY.DWARFLIB(&2))
```

Notice the **&2** Match Variable. It is replaced by whatever was matched by the lone * occurring within "(*)".

Then I want all other Original Names [i.e. those that are not of the form pdsname(membername)] to be redirected to the Candidates identified by the current **default** list.

```
    LL REDIRECT DWARF PATTERNS=**  USE=DEFAULT
[or LL RED      D     PAT=**       USE=DEFAULT  ]

    Library List DWARF
      #1  REDIRECT: '**' -> DEFAULT LIST(MYLIST)

      #2  REDIRECT: '**(*)'
                USE: MVS(MY.DWARFLIB(&2))
```

Ok, the Redirect List entry I wanted got created just fine, but they're in the wrong order: Original Names that are pdsname(membername), will match both patterns, but with this order, the Results List will have the Candidate Names from the default list occurring ahead of MY.DWARFLIB(membername). I want it to be the other way

around. So...

```
    LL REDIRECT DWARF PATTERNS=(#2,#1)
[or LL RED      D     PAT=(#2,#1)       ]

    Library List DWARF
      #1  REDIRECT: '**(*)'
                USE: MVS(MY.DWARFLIB(&2))

      #2  REDIRECT: '**' -> DEFAULT LIST(MYLIST)
```

Help COmmands LIBrarylists RESet

For a Glossary of Terms, see HELP DEBUGGING C LIBRARYLISTS GLOSSARY.

The **LIBRARYLISTS RESET** command purges the Redirect List and all Candidates Lists from either one or all **Families.**

Syntax:

```
    LIBRARYLISTS RESET DWARF
    LL                 CSOURCE
                       ALL
```

Rules:

- Operands shown in the same column are mutually exclusive.
- Keyword names can be truncated to the minimum number of characters needed to keep them unique. (But most keywords have a minimum length of 3).
- Operands may be given in any order.

Operand Descriptions

DWARF
CSOURCE
ALL

This identifies the **Family** to be purged. Either the named Family or all Families will be purged. The Family's Redirect List and all of its Candidates Lists will be deleted from the Library Lists.

Examples

Suppose the Library Lists contain two families and several Candidates Lists as follows:

```
    LL SHOW
[or LL SHO  ]

    Library List CSOURCE
      #1  REDIRECT: 'DBCOLE.**.SOURCLIB(*)' -> LIST(CSOURCE)

      LIST CSOURCE

    Library List DWARF
      #1  REDIRECT: 'DBCOLE.**.DWARFLIB(*)' -> LIST(DBCSLIST)

      LIST DBCSLIST
      LIST XDCSTUFF
```

If I want to delete everything pertaining to the DWARF family, this will do the trick:

```
    LL RESET DWARF
(or LL RESET D     )
    Library List DWARF RESET
```

```
    LL SHOW
[or LL SHO  ]

    Library List CSOURCE
      #1  REDIRECT: 'DBCOLE.**.SOURCLIB(*)' -> LIST(CSOURCE)

    Library List DWARF
      Redirection is OFF
```

Help COmmands LIBrarylists SHow

For a Glossary of Terms, see HELP DEBUGGING C LIBRARYLISTS GLOSSARY.

The **LIBRARYLISTS SHOW** command displays:

- What **Families** are defined,

- Each Family's **Redirect List** and all the Match Patterns within,

- What **Candidates Lists** are present and to which Families they belong,

- Optionally, the **Result Objects** contained within each Candidates List.

Syntax:

```
    LIBRARYLISTS SHOW DWARF    LISTNAMES=(name,name,...)  ALL      DETAILS
    LL               CSOURCE   LNAMES=(name,name,...)     omitted  SUMMARY
```

omitted **omitted**

Rules:

- Operands shown in the same column are mutually exclusive.
- Keyword names can be truncated to the minimum number of characters needed to keep them unique. (But most keywords have a minimum length of 3).
- Operands may be given in any order.
- Parenthesized values **must** be separated by **commas** (not blanks).
- For a discussion of additional Syntax Rules that are unique to the LIBRARYLISTS commands, see HELP COMMANDS LIBRARYLISTS SYNTAX.

Operand Descriptions

LL SHOW [All operands omitted]

The lists from **all Families** are displayed.

The **Redirect Lists** are shown in detail.

The **Candidates Lists** are shown in summary (i.e. only their names are shown.)

The default lists (if any) are indicated.

LL SHOW DETAILS [All other operands are omitted]

This shows everything that is possible to show with the **LIBRARYLISTS SHOW** command. It shows the same display as above except that the Candidates List entries **(Result Objects)** are also displayed.

DWARF
CSOURCE
This narrows the display to just the indicated **Family.**

DETAILS
SUMMARY
omitted
For any display, when **DETAILS** is given, The Candidates List entries **(Result Objects)** are also displayed.

When **SUMMARY** is given (or when both are omitted) The **Candidate List** names are shown, but their contents are not.

This does not affect the display of **Redirect Lists.** Their entries are always displayed.

ALL
omitted
This operand can be given **only** when a **Family** name is given. When given, all of the given Family's Candidates Lists are displayed.

When omitted, Only the Family's **default** Candidates List is shown.

Examples

Suppose that I have two **Families** of lists defined each with various **Candidates Lists...**

The following command shows:
- The **Redirect Lists** and the **Candidates Lists** for all **Families.**
- The **Redirect Lists** are shown in detail.
- The **Candidates Lists** are shown in summary.

```
    LL SHOW
[or LL SHO  ]

    Library List CSOURCE
      #1  REDIRECT: 'DBCOLE.**.SOURCLIB(*)' -> LIST(CSOURCE)

      LIST CSOURCE

    Library List DWARF
      #1  REDIRECT: 'DBCOLE.**.DWARFLIB(*)' -> LIST(DBCSLIST)

      LIST DBCSLIST    DEFAULTED
      LIST XDCSTUFF
```

The following command shows everything that the **LIBRARYLISTS SHOW** command is capable of showing. It shows:
- The **Redirect Lists** and the **Candidates Lists** for all **Families.**
- The **Redirect Lists** are shown in detail.
- The **Candidates Lists** also are shown in detail.

```
    LL SHOW DETAILS
[or LL SHO  DET     ]

    Library List CSOURCE
      #1  REDIRECT: 'DBCOLE.**.SOURCLIB(*)' -> LIST(CSOURCE)

      LIST CSOURCE
        #1 HFS  /u/cee/sourclib
        #2 MVS  CEE.SOURCLIB
```

```
    Library List DWARF
      #1  REDIRECT: 'DBCOLE.**.DWARFLIB(*)' -> LIST(DBCSLIST)

      LIST DBCSLIST    DEFAULTED
        #1 HFS  /u/dbc/dwarflib
        #2 MVS  DBC.DWARFLIB
        #3 DDN  //DWARFLIB
        #4 LIST XDCSTUFF

      LIST XDCSTUFF
        #1 HFS  /u/xdc/dwarflib
        #2 MVS  XDC.DWARFLIB
        #3 DDN  //DWARFXDC
```

The following command shows everything that the **LIBRARYLISTS SHOW** command is capable of showing about just the **DWARF** Family. It shows:

- The **Redirect Lists** and the **Candidates Lists** for the **DWARF** Family.
- The **Redirect Lists** is shown in detail.
- The **Candidates Lists** also are shown in detail.

```
    LL SHOW DWARF ALL DETAILS
[or LL SHO  D     ALL DET     ]

    Library List DWARF
      #1  REDIRECT: 'DBCOLE.**.DWARFLIB(*)' -> LIST(DBCSLIST)

      LIST DBCSLIST    DEFAULTED
        #1 HFS  /u/dbc/dwarflib
        #2 MVS  DBC.DWARFLIB
        #3 DDN  //DWARFLIB
        #4 LIST XDCSTUFF

      LIST XDCSTUFF
        #1 HFS  /u/xdc/dwarflib
        #2 MVS  XDC.DWARFLIB
        #3 DDN  //DWARFXDC
```

The following command shows information about the **DWARF** Family, but not quite everything. It shows:

- The **Redirect Lists** and the **Candidates Lists** for the **DWARF** Family.
- The **Redirect Lists** is shown in detail.
- But the **Candidates Lists** are shown only in summary.

```
    LL SHOW DWARF ALL
[or LL SHO  D     ALL ]

    Library List DWARF
      #1  REDIRECT: 'DBCOLE.**.DWARFLIB(*)' -> LIST(DBCSLIST)

      LIST DBCSLIST    DEFAULTED
      LIST XDCSTUFF
```

Help COmmands LIBrarylists Test

For a Glossary of Terms, see HELP DEBUGGING C LIBRARYLISTS GLOSSARY.

The **LIBRARYLISTS TEST** commands can be used to help you form proper **Match Patterns** and **Results Lists.** There are two variations...

The **LIBRARYLISTS TEST MATCH** command allows you to investigate **Match Variables.** With it, you can test a given **Original Name** against a given **Match Pattern** and see the exact Match Variables that result. For details, see HELP * MATCH.

The **LIBRARYLISTS TEST TRY** command allows you to investigate **Result Lists.** After you've built suitable **Redirect Lists** and **Candidates Lists,** You can use this command to see the Results List that Library Lists Management would pass back to c/XDC for a given **Original Name.** For details, see HELP * TRY.

Help COmmands LIBrarylists Test Match

For a Glossary of Terms, see HELP DEBUGGING C LIBRARYLISTS GLOSSARY.

The **LIBRARYLISTS TEST MATCH** command allows you to investigate **Match Variables.** With it, you can test a given **Original Name** (file name) against a given **Match Pattern** and see:

- Whether or not The Name matches the Pattern
- And if so, then the exact Match Variable **values** that result.

The **LIBRARYLISTS TEST MATCH** command **does not reference** any existing **Library Lists,** so you can use it to test file names against Match Patterns even before you begin to construct your lists.

Syntax:

LIBRARYLISTS TEST MATCH PATTERN=pattern FILENAME=originalname
LL

Rules:

- Keyword names can be truncated to the minimum number of characters needed to

keep them unique. (But most keywords have a minimum length of 3).

- Operands may be given in any order.

Operand Descriptions

FILENAME=originalname

This is a trial dataset name (or other string) that you would like to test against the given Match Pattern. Typically, the string would be either:
- A classic z/OS dataset name, library name or pdsname(member),
- Or an HFS pathname/filename or pathname/foldername.

PATTERN=pattern

This is a test Match Pattern against which you wish to test the Original Name. Typically, it would contain a mix of constant characters and wildcard strings. See HELP DEBUGGING C LIBRARYLISTS LISTS REDIRECTLIST WILDCARDS for detailed information.

Examples

```
    LL TEST MATCH ...
    PATTERN=*.A???B**C?*DDD(*) ...
    FILE=DBCOLE.AUSEBNOT.MYDEPT.MYPROJ.MCCCCDDD(MEMBER)
```

Or more compactly:

```
    LL TES MAT PAT=*.A???B**C?*DDD(*) ...
    F=DBCOLE.AUSEBNOT.MYDEPT.MYPROJ.MCCCCDDD(MEMBER)
```

Either way, the result is this:

```
   Testing DBCOLE.AUSEBNOT.MYDEPT.MYPROJ.MCCCCDDD(MEMBER)
   Against pattern *.A???B**C?*DDD(*)
   Pattern result is MATCH

   &1="DBCOLE"
   &2="USE"
   &3="NOT.MYDEPT.MYPROJ.MCC"
   &4="C"
   &5="MEMBER"
   &6=""
   &7=""
   &8=""
   &9=""
```

Help COmmands LIBrarylists Test Try

For a Glossary of Terms, see HELP DEBUGGING C LIBRARYLISTS GLOSSARY.

The **LIBRARYLISTS TEST TRY** command allows you to investigate **Results Lists.** With it, you can test a given **Original Name** (file name) against a given **Family of Lists** and see the list of datasets, files, folders and libraries (i.e. the **Results List)** that Library Lists Management is going to suggest c/XDC search to find the DWARF data or C source files that it needs for building Source Image maps.

In order to use this **LIBRARYLISTS TEST TRY** command, you will first have to build a fully functioning set of lists that includes:
- A Redirect List of Match Patterns,
- Along with a set of Candidate Lists.

Then you can use **LIBRARYLISTS TEST TRY** with a trial **Original Name** to see what sort of **Result List** gets built.

Syntax:

```
    LIBRARYLISTS TEST TRY DWARF    FILENAME=originalname
    LL                    CSOURCE
```

Rules:

- Keyword names can be truncated to the minimum number of characters needed to keep them unique. (But most keywords have a minimum length of 3).

- Operands may be given in any order.

Operand Descriptions

DWARF
CSOURCE
This identifies the **Family** whose lists are to be tested.

FILENAME=originalname
This is a trial dataset name (or other string) that you would like to test against the given Family of Lists. Typically, the string would be either:
- A classic z/OS dataset name, library name or pdsname(member),
- Or an HFS pathname/filename or pathname/foldername.

Examples

Suppose that the following Families of Lists exist:

```
    LL SHOW DETAILS
[or LL SHO  DET     ]

    Library List CSOURCE
```

```
      #1  REDIRECT: 'DBCOLE.**.SOURCLIB(*)' -> LIST(CSOURCE)

      LIST CSOURCE
        #1 HFS  /u/cee/sourclib/&2..txt
        #2 MVS  CEE.SOURCELIB(&2)

    Library List DWARF
      #1  REDIRECT: 'DBCOLE.**.DWARFLIB(*)' -> LIST(DBCSLIST)

      LIST DBCSLIST    DEFAULTED
        #1 HFS  /u/dbc/dwarflib
        #2 MVS  DBC.DWARFLIB
        #3 DDN  //DWARFLIB
        #4 LIST XDCSTUFF

      LIST XDCSTUFF
        #1 HFS  /u/xdc/dwarflib
        #2 MVS  XDC.DWARFLIB
```

Then the following commands would produce the following results:

```
    LL TEST TRY CSOURCE FILE=DBCOLE.A.B.C.SOURCLIB(MYPGM)
[or LL TES  TRY C       FIL=DBCOLE.A.B.C.SOURCLIB(MYPGM)  ]

    Trying with Filename=DBCOLE.A.B.C.SOURCLIB(MYPGM)
    Redirect 1:/u/cee/sourclib/MYPGM.txt
    Redirect 2:CEE.SOURCELIB(MYPGM)
```

Help COmmands LICense

The **LICENSE** command is one of two methods by which Z/XDC's Licensing Control Data Display/Update panel can be displayed. (The other occurs automatically when Z/XDC detects incorrect licensing information.) Any user may issue the LICENSE command; however, only those users that (a) have write access to the library containing the xxxLCNSE load module, and (b) have received valid licensing information from ColeSoft will be able to change the License Control Data.

Syntax:

```
    LICENSE license-module-name
            omitted
```

license-module-name
This is an optional operand that allows the user to specify the name of the license control load module that Z/XDC will attempt to zap if valid license data changes are accepted by Z/XDC.

If this operand is omitted, then the default target load module is **xxxLCNSE,** where **xxx** matches Z/XDC's current name (usually **XDC).** The target load module must be located in a library (other than a PLPA library) that is within the

current load module search order (a task library, a STEPLIB, a JOBLIB, or a linklist library). The library **must** be a **PDS.** (It cannot be a PDSE.) The user must have write access to the library containing the module.

Z/XDC performs a validity check to make sure that the specified load module does, in fact, contain License Control Data. The LICENSE command is rejected if it does not.

The Licensing Control Data panel allows the user to attempt to make changes to the displayed information. However, all licensing information is guarded by a control key that also is displayed. Any changes to the licensing information will also require a corresponding change to the control key. Z/XDC will reject all invalid attempts to change the licensing information. Only ColeSoft can provide control keys that correspond to specific licensing data.

If acceptable changes are made on the Licensing Control Data panel, then Z/XDC will save those changes to disk for use by all future Z/XDC debugging sessions. When the License Control panel is displayed via the LICENSE command, any changes made will **not** affect the current debugging session, only future ones.

Z/XDC saves the License Control information by zapping the target load module. Therefore, the user making the changes must have write access to the library containing that module.

If the Licensing Control Data has been successfully changed, and if the xxxLCNSE module resides in a linklist library, then an LLA refresh will have to be done. A System Operator will have to issue a "MODIFY LLA,REFRESH" command from an Operator's Console.

Some of the displayed information is not License Control Data and so can be changed by any appropriate user. Specifically, the "Expiration Warning (Days)" field can be changed by any user having write access to the library containing the target load module.

Subtopics Index

For more information, select the following topics. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

PANEL - Describes the management of the Licensing Control Data Display/Update panel and the commands that it accepts.

DATA - Describes the fields displayed by the Licensing Control Data Display/Update panel.

For information about **Licensed Features** and **CAPs,** see HELP SUPPORT FEATURESANDCAPS.

Help COmmands LICense Panel

Z/XDC's Licensing Control Data Display/Update panel is displayed under two circumstances:

- Automatically at the start of a debugging session if invalid, incorrect, or

expired licensing information is detected.
- Upon request when the user issues the LICENSE command.

A user may change License Control Data by overtyping the displayed information with correct data and then issuing an **END** command (F3). If correct License Control Data has been given, then that data will be permanently saved. However, correct License Control Data can be obtained only from ColeSoft.

If you are having problems with your licensing data, then please see HELP SUPPORT CONTACTUS for information about contacting ColeSoft.

A user may cancel attempted License Control changes by issuing a **CANCEL** or **QUIT** command. F4 may also be used.

The **ENTER** key can be pressed at any time to cause Z/XDC to validity check attempted changes. Valid changes will not be written, however, until an **END** command is issued. Invalid changes will never be written under any circumstances.

END Command Processing

When the **END** command is issued, what happens next depends upon whether the user has made valid changes, invalid changes, or no changes:

- If valid changes have been made, then Z/XDC attempts to save the changed information to disk. Z/XDC saves the information by zapping a target load module usually named **xxxLCNSE** (where **xxx** matches Z/XDC's current name, usually **XDC).** Therefore, the user making the changes must have write access to the library containing the target load module. (Note that the name of the target load module can be overridden by an operand of the **LICENSE** command.)

- If invalid changes have been made, then Z/XDC rejects the **END** command and continues to display the Licensing Control panel. The user must either make valid changes or use the CANCEL command to terminate the panel and abandon the change attempt.

- If the user has made no changes, then Z/XDC checks to see if the panel was entered via the **LICENSE** command vs. due to incorrect licensing information. In the former case, the Licensing Control panel is terminated and the target load module is not zapped. in the latter case, the **END** command is rejected and the Licensing Control panel remains displayed. The user must either make valid changes or use the **CANCEL** command to abort the panel and, therefore, the debugging session.

If valid changes are made to the License Control Data and the **END** command has been accepted, then whether or not Z/XDC immediately acts upon those changes depends upon why the License panel was displayed:

- If the panel was displayed via a **LICENSE** command, then the current debugging session is continued under the control of the old licensing data. The new changes only affect all future debugging sessions.

- If the panel was displayed because incorrect licensing information was detected, then the changes are acted upon immediately and the current debugging session continues or fails according to whether the changed licensing information supports the current CPU.

If the xxxLCNSE load module is located in a linklist library, then a System Operator will have to issue an **F LLA,REFRESH** command in order to make the license changes available to subsequent debugging sessions.

CANCEL Command Processing

When the **CANCEL** command is issued (or the **QUIT** command or **F4),** Z/XDC aborts the Licensing Control panel and discards all attempted changes. Whether the debugging session continues or aborts depends upon why the panel had been displayed:

- If the panel was displayed via a **LICENSE** command, then the current debugging session is continued under the control of the old licensing data.

- If the panel was displayed because incorrect licensing information was detected, then the debugging session aborts.

When Z/XDC aborts a debugging session, the ABEND for which Z/XDC was entered in the first place is percolated to older ESTAE, ESTAI, or ESTAEX routines. In other words, ABEND processing continues as if Z/XDC had never been called in the first place.

For specific information about the fields appearing on the Licensing panel, type HELP COMMANDS LICENSE DATA.

Help COmmands LICense Data

Most of the fields appearing on the Licensing Control Data Display/Update panel are guarded by a control key that also is displayed. Any changes made to those fields require a corresponding change to the control key; otherwise, the change will be rejected. Only ColeSoft (and the C.I.A.) can provide control keys that match specific licensing information. For assistance, please contact us. **HELP SUPPORT CONTACTUS** explains how.

The License Control Data panel contains the following information fields (from top to bottom).

Z/XDC RELEASE v3.1
This is a display only field that shows to which release of Z/XDC the License Control Data applies.

Current CPU
This is a display only field that shows the 4 digit machine type and the lo-order 5 digits of the serial number of the CPU on which Z/XDC currently is running. This field is displayed because oftentimes problems with License Control Data occur because the customer has provided ColeSoft with incorrect CPU identification information. In order for Z/XDC to operate on the current computer, one of the entries in the **TYPE** and **SER#** fields will have to match the **Current CPU** information displayed here.

Current SYSPLEX
This is a display only field that shows the 8 character SYSPLEX name within which Z/XDC is currently running.

License Dsname
This is a display only field that shows the name of the load module and library that Z/XDC will zap if valid license data changes are made by you and are accepted by Z/XDC. The module name will be either **xxxLCNSE** (where **xxx** matches Z/XDC's current name, usually **XDC),** or it will be the name of a load module given when you issued the **LICENSE** command.

Customer ID
This is unique number or name by which we identify our customers. This value is different for each customer. This value is guarded by the Control Key.

License Start Date
This specifies the date before which Z/XDC will refuse to operate. This value is guarded by the Control Key.

base/XDC Quota
auth/XDC Quota
asm/XDC Quota
c/XDC Quota
dump/XDC Quota
These are display only fields that show the total **CAP Quotas** being being licensed for each Feature. They are computed from information given in the **CAP** and **FLAG** fields.

Control Key
This is an 8-digit hex number whose value is based on an encryption of most of the rest of the fields on the panel. Fields "guarded" by this key cannot be changed without making a corresponding change to this key. Only ColeSoft can provide control keys that are correct for a given set of guarded values. If you are having problems with the control key, or if you need to change one of the guarded values, then please contact us for assistance. See **HELP SUPPORT CONTACTUS** for information about how to do so.

License End Date
This specifies the date after which Z/XDC will cease to operate. This value is guarded by the Control Key. For trial customers, the expiration date will typically be up to a month or so into the future. For licensed customers, this will typically be up to a year into the future.

Expiration Warning nnn (Days)
This specifies the number of days prior to license expiration that a warning message is issued to all users of Z/XDC. This value is **not** guarded by the Control Key. Any user having write access to the library containing the License Control Data load module may change this value.

The factory default warning period is **37** days.

CAP TYPE SER# FLAG and **SYSPLEX**
A specific copy of Z/XDC can be licensed to run on up to 92 computers connected together into up to 92 SYSPLEXs. This table allows you to define the licensed computers and the licensed SYSPLEXs. The values given here are guarded by the Control Key.

CAP
This is a SYSPLEX-specific suballocation of the total CAP quota being licensed. The Features to which this allocation applies are indicated by the FLAG values. Concurrent usage of the affected Features within the SYSPLEX will be limited by the value specified here.

TYPE SER#
These specify the 4-digit machine type of a permitted CPU (e.q. 3090, 9021, etc.) and the lo-order 5 digits of the serial number of a permitted CPU. The digit **F** is a wildcard permitting any value in the corresponding position of a CPU's identification number. Thus, multi-engine computers and partitioned computers are easily supported.

FLAG
This is a set of flags that permit or prohibit the use of Z/XDC and its various Licensed Features.

SYSPLEX
This is the name of a SYSPLEX to which a CAP quota suballocation applies.

Help COmmands LISt

The **LIST** command is used to display any of a variety of different objects.

Syntax:

LIST subcommand,operands
L

Shortcut: L [in specific situations]

subcommand
This selects the particular subcommand to be executed. It may be any of the following:

ACCESSLISTS - Shows the details of any access list (PASN-AL or DU-AL) for any task or SRB routine running in any address space, System wide.

accessregisters - About displaying **access registers.**

AL - Alias of ACCESSLISTS.

ALES - Alias of ACCESSLISTS.

AMODE - Displays the current default addressing mode.

AR# - Displays an individual retry level **access registers.**

AREGS - Displays the whole set of retry level access registers.

ASID - Displays the ASID of an address space identified by jobname or by other information.

AUTOMAP - Displays Z/XDC's current AUTOMAP settings.

BANG - Displays the current meaning of the "!" indirect operator.

BEA	- Displays the retry-level Breaking Event Address (BEA) or an arbitrary BEA.
BELL	- Displays the status of the terminal bell.
BFR#	- Displays in Binary Floating Point format the entire 8 bytes of an individual **floating point register.**
BPTS	- Alias of BREAKPOINTS.
BRANCHES	- Displays a list of recent user program branching instructions at which Z/XDC has received control. Also displays the most recent **BEA** (Breaking Event Address).
BREAKPOINTS	- Lists all currently defined breakpoints.
CACHE	- Displays information about files of ADATA cached in 31-bit storage.
CANDET	- Shows whether or not Z/XDC will intercept CANCEL-type and DETACH-type ABENDs.
CAPS	- Alias of LICENSE.
CDF	- Displays the current characteristics of cdf/XDC.
COLORS	- Displays the colors that Z/XDC will use to build fullscreen images.
COLOURS	- Alias of COLORS.
CONSOLES	- Displays information about some or all available MCS and EMCS System Operator Consoles.
controlregisters	- About displaying **control registers.**
CR#	- Displays the lo-order half (4 bytes) of an individual retry level **control register.**
CREGS	- Displays the lo-order halves (4 bytes) of the whole set of retry level **control registers.**
CRH#	- Displays the hi-order half (4 bytes) of an individual retry level **control register.**
CRHREGS	- Displays the hi-order halves (4 bytes) of the whole set of retry level **control registers.**
CRW#	- Displays the entire 8 bytes of an individual retry level **control register.**
CRWREGS	- Displays the entire 8 bytes of the whole set of retry level **control registers.**
CURRENT	- Reports whether or not there is a Current Address Space (CAS) or Current eXecution Unit (CXU) and if so details about it.
CWD	- displays the USS Current Working Directory (CWD).
CXDC	- Reports on the current status of the c/XDC Licensed Feature.
CXU	- Reports the current execution unit (aspace, TCB, RB, SRB and such)
DBC640Q	- Displays the status of the cdf/XDC pending signon message.
DDNTRANS	- Displays the current list of ddname translation rules.
DFR#	- Displays in Decimal Floating Point format the entire 8 bytes of an individual **floating point register.**
DSECTLIBS	- Displays the current list of dsect lookup libraries.
DSPACES	- Displays information about some or all dataspaces owned by any accessible address space.
DSTG	- Displays storage information from the current dump being processed by dump/XDC
DUB	- Displays the SET DUB settings.
DUMP	- Displays the status of the SET DUMP settings.
DXDC	- Displays information about the current dump being processed by dump/XDC

EAR# - Displays an individual error level **access registers.**
EAREGS - Displays the whole set of error level **access registers.**
EBEA - Displays the error-level Breaking Event Address (BEA).
ENQ - Displays information about system and user ENQs. Also detects and reports ENQ contention.
EPSW - Displays the error level PSW in the old **8-byte** format.
EPSWE - Displays the error level PSW in the full **16-byte** format.
EQUATES - Lists all currently defined equate names.
ER# - Displays the lo-order half (4 bytes) of an individual error level **general register.**
EREGS - Displays the lo-order halves (4 bytes) of the whole set of error level **general registers.**
ERH# - Displays the hi-order half (4 bytes) of an individual error level **general register.**
ERHREGS - Displays the hi-order halves (4 bytes) of the whole set of error level **general registers.**
ERW# - Displays the entire 8 bytes of an individual error level **general register.**
ERWREGS - Displays the entire 8 bytes of the whole set of error level **general registers.**
ESTAES - Displays the STAE Control Block (SCB) queue for any task in any accessible address space.
EXITS - Displays the status of the SET EXITS command's control over Z/XDC's Miscellaneous Exits Interface.
FEATURES - Displays various hardware and software features detected by Z/XDC.
FIXED - Displays a given storage location in fixed-point format.
FLC - Displays the current Function Leader Character.
FLOAT - Displays a given storage location in floating point format.
floatingpointregi - About displaying **floating point registers.**
FORMAT - Displays the current state of the "SET FORMAT" command.
FPC - Displays the floating point control register.
FR# - Displays the entire 8 bytes of an individual **floating point register** in HFP, BFP and DFP formats.
FREGS - Displays the entire 8 bytes of the whole set of **floating point registers.**
generalregisters - About displaying **general registers.**
GSR# - Displays a specific guarded storage register.
GSREGS - Displays the guarded storage registers.
HELP - Shows the hierarchy of information that is displayable by the HELP command.
HFR# - Displays in Hexadecimal Floating Point format the entire 8 bytes of an individual **floating point register.**
HILIGHT - Displays the extended hilighting attributes that Z/XDC will use to build fullscreen images.
HKEYS - Displays the status of the "HELP-mode" PF keys.
HOOKS - Displays descriptions of all dynamic hooks created by and known to the current debugging session.
ILC - Displays the status of the **Z** shortcut's Instruction Length Check.
ILIT - Displays the names and values of IPCS literal equates.
INTENSITY - Displays the field intensities that Z/XDC will use to build fullscreen images.

ISPF - Displays the current characteristics of Z/XDC's ISPF-fullscreen interface.

ISR@PRIM - Displays the name of the Primary Options Menu that will be used if Z/XDC's **ISPF** command is used to invoke ISPF recursively under Z/XDC.

KEYS - Displays the assignments of PF key sets to function key ranges.

LICENSE - Displays information about currently active debugging sessions and the licenses they hold.

LKEDMAP - Displays the Binder map of a load module or program object.

LOCATION - Displays the location(s) that Z/XDC has determined for a given address.

LOCKS - Displays information about locks held by the program being debugged.

LOG - Displays the current status and characteristics of an external log file for Z/XDC commands and displays.

LOGONID - Alias of USERID.

LSES - Alias of LSTACK.

LSTACK - Displays Linkage Stack Entries (LSEs) for any task in any accessible address space.

MAINTENANCE - Alias of XDC.

MAPLIBS - Displays information both about the currently active MAPLIB list and about all saved MAPLIB lists.

MAPS - Lists all currently defined load module, csect, and dsect maps.

MEMORYOBJECTS - For any accessible address space, this command displays its above-the-bar **MEMLIMIT** as well as descriptions of all of its memory objects.

MMEFDSNS - Displays the current list of load module (and program object) information libraries.

MOBJECTS - Alias for MEMORYOBJECTS,

MSGS - Redisplays a subset of Z/XDC's opening salvo messages.

NOTES - Displays the current list of noted addresses as built by the "NOTE" command.

NT - Displays z/OS name/token entries visible from a task, address space, or the system.

OPERANDS - Displays the current storage operand values for a machine instruction.

OPTIMIZATION - Displays the status of fullscreen image optimization.

PARSEORDER - Displays both which and the order by which language-specific parsers are called when parsing the addressexpression operands of certain commands.

PC# - Displays a list of available program call numbers (PC#s) available to an address space.

PET - Displays information about a PET (Pause Element Token).

PFKEYS - Alias of KEYS.

PGMS - Shows information about various load module queues or about specific load modules.

PID - displays the USS Process ID.

PRIMARYSIZE - Displays the status of the primary vs. secondary screen dimension choice.

PRINT - Displays information about the XDCPRINT output file (for printing Built-in Help manuals).

PROFILES - Displays information about any or all Z/XDC profiles found in any or all available profile libraries.

PSW - Displays (in the old **8-byte** format) either the retry level PSW,

or the resume PSW associated with any given RB in any accessible address space.

- **PSWE** - Displays (in the full **16-byte** format) either the retry level PSW, or the resume PSW associated with any given RB in any accessible address space.
- **QUALIFIER** - Shows the current default load module and csect names used by address expressions and the DMAP command.
- **R#** - Displays the lo-order half (4 bytes) of an individual retry level **general register.**
- **RBS** - Shows the structure of RBs queued from a given TCB in any accessible address space.
- **READ** - Displays all options pertaining to the READ command.
- **READECHO** - Deprecated. Please use **LIST READ** instead.
- **REFRPROT** - Displays both the System-wide REFRPROT setting and the task-level overrides for any task in any accessible address space.
- **registers** - About displaying **registers.**
- **REGS** - Displays the lo-order halves (4 bytes) of the whole set of retry level **general registers.**
- **REXX** - Displays information about the currently active rexx/XDC Interface (if any).
- **RH#** - Displays the hi-order half (4 bytes) of an individual retry level **general register.**
- **RHREGS** - Displays the hi-order halves (4 bytes) of the whole set of retry level **general registers.**
- **RSA** - Displays a specified register save area chain.
- **RW#** - Displays the entire 8 bytes of an individual retry level **general register.**
- **RWREGS** - Displays the entire 8 bytes of the whole set of retry level **general registers.**
- **SCBS** - Alias of ESTAES.
- **SCREEN** - Alias of WINDOW.
- **SEARCHORDER** - Displays the search order for a task.
- **SECONDARYSIZE** - Displays the status of the primary vs. secondary screen dimension choice.
- **SECURITY** - Displays status and settings related to Z/XDC’s system security interface.
- **SESSIONS** - Alias of LICENSE.
- **SIGNONWAIT** - Displays the current period of time that cdf/XDC will wait for a terminal logon.
- **SIZEOF** - Displays the storage size consumed by one or more variables, arrays, structures, etc. defined in a High Level Language (HLL).
- **SRBS** - Alias of SSRBS.
- **SSCT** - Displays the SubSystem Control Table chain.
- **SSRBS** - Displays all suspended SRB routines found in a specified address space.
- **STATISTICS** - Displays various Z/XDC processing statistics.
- **STATS** - Alias of STATISTICS.
- **STEP** - Displays c/XDC’s current STEP command settings.
- **STGLAYOUT** - Displays the storage layout.
- **SUBPOOLS** - Shows the organization of storage subpools within any address space. Also can be used to diagnose REGION= limit problems.
- **TASKS** - Shows the subtask structure that currently exists in any

accessible address space.

- **TCBS** - Alias of TASKS.
- **TERMINAL** - Displays the current settings of various fullscreen terminal attributes: color, hilight, bell, screen sizes.
- **TFS** - Displays the current settings of the various fullscreen interface attributes: TSO, ISPF, cdf/XDC.
- **TIMEOUT** - Displays the current multi-tasking timeout period.
- **TIOT** - Displays the dataset allocations for any task in any accessible address space.
- **TPUT** - Reports what is being used to paint the fullscreen displays at your 3270 terminal.
- **TRACE** - Displays current trace related settings.
- **USERID** - Displays the TSO userid to be notified by cdf/XDC when a background program is awaiting connection to a programmer for a debugging session.
- **USS** - Displays the SET USS settings.
- **VARIABLES** - Displays one or more variables, arrays, structures, etc. defined in a High Level Language (HLL).
- **VARS** - Alias of the LIST VARIABLES command.
- **VDISPLAY** - Displays the current settings that control how the output of the LIST VARIABLES command appears on the display.
- **vectorregisters** - About displaying **vector registers.**
- **VR#** - Displays the entire 16 bytes of an individual **vector register in various formats.**
- **VREGS** - Displays the entire set of 32 **vector registers.**
- **VSETTINGS** - Displays various settings and limits pertaining to Z/XDC’s HLL variables support.
- **VSTACK** - Displays the stack of variable pools (also known as Language Environment Stack Frames) for the currently running HLL program.
- **WINDOW** - Displays the current characteristics of the selected window at the user’s terminal.
- **XDC** - Displays Z/XDC’s release, assembly date, maintenance level, ColeSoft contact information, and Feature licensing information.
- **XMS** - Displays the cross memory characteristics of the current error level and retry level environments.
- **ZAP** - Displays current zap related settings.

Specific information can be selected directly . Type an **H** at the left above to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

Help COmmands LISt ACCESSLists

The **LIST ACCESSLISTS** command produces a detailed display of any one or more access lists **(PASN-ALs** and/or **DU-ALs)** associated with any tasks, SRB routines and/or address spaces, System wide.

For each selected Access List, all of the Access List Entries **(ALEs)** that make up that list are displayed. For each ALE, the following is displayed:

- The **type** of space (address space or dataspace) pointed to by the ALE
- The **name** of the address space or dataspace (if any) pointed to by the ALE

- The **ALET** for accessing the space via this ALE
- For dataspaces:
  - The name and ASID of the address space that **owns** the dataspace
  - The **scope** of the dataspace (All, COMMON or PRIVATE)
  - The dataspace's **STOKEN**
  - For inaccessible dataspaces, the **ABEND code** by which access is blocked.
  - The dataspace's **attribute** flags:
    I -Invalid
    FO -Fetch Only
    PVT -Private

In addition, the **LIST ACCESSLISTS** command (re)creates several series of automatic equates:

ALE#n - These label, in real storage, the Access List Entries that constitute the Access List being displayed.

ASTE#n - These label, in real storage, the ASN Second Table Entries that are pointed to by the displayed ALEs and that define and anchor the address spaces and dataspaces being linked to. (Note, the **LIST DSPACES** command also [re]creates ASTE#n equates.)

#dspacename - For those dataspaces that are accessible to Z/XDC (ie, that can be displayed by Z/XDC commands), these equates label those spaces. So for example, if you would like to display location **X'0034DE03'** in the dataspace named **0000XYZZ,** then you can do so simply by issuing **FORMAT #0000XYZZ+34DE03.**

When the **LIST ACCESSLISTS** command is used to display multiple Access Lists, only those automatic equates are created that are associated with the last Access List displayed.

Syntax:

LIST ACCESSLISTS accesslistrefs ...
omitted

Aliases:

LIST ALES
LIST AL

Operands:

accesslistrefs
One or more Access List References may be given. Each reference identifies an Access List to be displayed. Each reference must be one of the following:

- **tcbaddress:** The address of any **TCB** in the System: The **DU-AL** Access List for the referenced task is displayed. Note, the **TCB#n** automatic equates created by the **LIST TASKS** command are perfectly good ways to specify TCB addresses.

- **ssrbaddress:** The address of any **SSRB** or **SSRX** in the System: The **DU-AL** Access List for the referenced SRB routine is displayed. Note, the **SSRB#n** and **SSRX#n** automatic equates created by the **LIST SSRBS** command are perfectly good ways to specify SSRB and SSRX addresses.

- **aspaceref:** The **PASN-AL** Access List for the referenced address space is displayed. Valid address space references include:
  - The address of the aspace's **ASCB**
  - The aspace's **ASID**
  - The aspace's **jobname**
  - A keyword such as **HOME, SASID** etc.

  For more information, see HELP COMMANDS SYNTAX ASIDS.

When multiple **accesslistref** operands are given, multiple Access Lists are displayed. However, only the automatic equates associated with the **last** displayed Access List are generated.

omitted
When the **LIST ACCESSLISTS** command is given **without** operands, two or three access lists are displayed:
- The **DU-AL** for the task or SRB routine being debugged
- The **DU-AL** for Z/XDC's home TCB (if different from the above)
- The home address space's **PASN-AL**

Help COmmands LISt ACCESSRegisters

Access registers are used for referencing storage in data spaces and (sometimes) in other address spaces. Information about setting up data spaces can be found in **MVS Extended Addressability Guide** (GC28-1769 for OS/390, SA22-7614 for z/OS).

There are 16 access registers, numbered from 0 to 15. For both OS/390 and z/OS, access registers are 4 bytes wide.

Z/XDC maintains two sets of access registers: an error level set and a retry level set. The error level registers are those that existed at the time that the ABEND or the breakpoint occurred that caused Z/XDC to receive control. The retry level registers are those that will exist should Z/XDC be used to cause the user's program to resume execution. For more information, see HELP EXECUTIONLEVELS.

Z/XDC does not limit you to displaying just **your** program's access registers. Every program has its own set of access registers, so (security and authorization permitting) Z/XDC allows you to display the access registers associated with **any** Request Block (RB), queued from **any** Task Control Block (TCB), located in **any** address space in the entire system! But please note the following:
- The access registers that are **associated** with a given Request Block are those that **belong to** the program that is running under the control of that Request Block.
- The access registers that are associated with a given Request Block are actually stored in the XSB that is anchored from the Request Block that is **next newer** than the designated Request Block. (Z/XDC automatically finds them.)
- The registers that are associated with a task's newest Request Block are actually stored in that task's STCB.

So the access registers that are displayed are those that are associated with (not

stored within) the given RB. These are the registers that belong to the program running under said RB.

Z/XDC also allows you to display the access registers belonging to a suspended SRB routine located in any address space in the system.

Subtopics Index

Z/XDC allows you either to display individual access registers or to display entire register sets. For specific information, type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

INDIVIDUAL - Displaying individual access registers.
REGISTERSET - Displaying an entire access register set.

Help COmmands LISt ACCESSRegisters Individual

Z/XDC allows you to display an individual access register:

- Belonging to the current program's retry level registers.
- Belonging to the current program's error level registers.
- Belonging to any program running under any RB queued from any TCB located in any accessible address space.
- Belonging to any suspended SRB routine located in any accessible address space.

The access register is displayed in a hex-text format. Its contents (an ALET) also are interpreted to identify the space that the register designates.

Syntax:

LIST ARn
nAR

Either of these commands can be used to display an individual access register from the current program's **retry level** register set. "n" is a decimal number ranging from 0 to 15. It can appear either before or after the letters AR.

LIST EARn
nEAR

These commands can be used to display an individual access register from the current program's **error level** register set.

LIST ARn rbaddress
nAR ssrbaddress

If an address expression is given, then it must resolve to either of the following:

- The address of a **Request Block** that is currently queued from any TCB located in any accessible address space anywhere in the system. (Note, the **RB#n** automatic equates created by the LIST RBS command are perfectly good address expressions to use for this rbaddress operand.)

- The address of an **SSRB** or **SSRX** for any SRB routine currently suspended in any accessible address space anywhere in the system. (Note, the **SSRB#n** and **SSRX#n** automatic equates created by the LIST SSRBS command are perfectly good address expressions to use for this ssrbaddress operand.)

(Note, specifying an address is not supported for error level registers.)

When an RB address is given, Z/XDC will display the access register that belongs to the program that is currently running under that RB. (This is the register that is stored in the next newer RB's XSB, or in the STCB in the case of the newest RB.) Note that the **LIST RBS** command, as a side effect, creates a series of equates named **RB#n** to label the locations of the Request Blocks that it displays. These equates make handy dandy address expressions for use with the various **LIST register** commands.

When an SSRB or SSRX address is given, Z/XDC will display the access register that belongs to the SRB routine that is currently suspended under that SSRB and SSRX. Note that the **LIST SSRBS** command, as a side effect, creates a series of equates named **SSRB#n** and **SSRX#n** to label the locations of the SSRBs and SSRXs from which it builds its displays. These equates can be used as the ssrbaddress address expression on the **LIST ARn** and other **LIST register** commands.

Examples:

LIST AR0
Lists the contents of retry level access register 0.

LIST 3EAR
Lists the contents of error level access register 3.

LIST TASKS JES2
LIST RBS TCB#3
LIST AR6 RB#1
If Z/XDC is running authorized, and if System Security permits the access, then the **LIST TASKS JES2** command displays the subtask structure for JES2's address space. It also creates a series of **TCB#n** equates labeling the locations of all TCBs within the JES2 address space.

The **LIST RBS** command then makes use of the **TCB#3** equate to display the Request Blocks currently queued to JES2's jobstep task. It also creates a series of **RB#n** equates labeling the locations of all such RBs.

Finally, the **LIST AR6 RB#1** command displays the contents of access register 6 that belongs to the program (HASJES20) running under the control of the jobstep task's oldest Request Block.

LIST SSRBS 1
LIST 7AR SSRB#1

LIST 7AR SSRX#1

If Z/XDC is running authorized, and if System Security permits the access, then the **LIST SSRBS 1** command displays a list of all SSRBs suspended in the Master Scheduler's address space. It also creates a series of **SSRB#n** and **SSRX#n** equates labeling the locations of all the SSRBs and SSRXs that were found.

Then the **LIST 7AR SSRB#1** command displays the contents of access register 7 belonging to the SRB routine running under the control of that SSRB.

The **LIST 7AR SSRX#1** command displays the exact same information. To display registers for a given SRB routine, it doesn't matter whether you provide the address of the SSRB or the SSRX. They both are associated with the same SRB routine, so they both result in the same register being displayed.

Help COmmands LISt ACCESSRegisters Registerset

The **LIST AREGS** and **LIST EAREGS** commands allow you to display the entire set of access registers:

- Belonging to the current program's retry level environment.
- Belonging to the current program's error level environment.
- Belonging to any program running under any RB queued from any TCB located in any accessible address space.
- Belonging to any suspended SRB routine located in any accessible address space.

The access registers are displayed in a hex-text format.

When the retry level set of access registers is displayed, those registers will be hilighted whose values have changed since the last time Z/XDC received control from the user's program.

Syntax #1:

```
LIST AREGS WIDE   EBCDIC
           NARROW ASCII
```

This form of the command (other forms are described below) can be used to display the access registers from the current program's retry level environment.

The **WIDE** and **NARROW** operands are optional. They control whether the resulting display will show 8 words of data per line **(WIDE)** or just 4 words **(NARROW).** If omitted, then the current SET FORMAT command setting is used. This setting can be displayed by the LIST FORMAT command and saved into your session profile by the PROFILE SAVE command. For more information, see:

HELP COMMANDS LIST FORMAT
HELP COMMANDS SET FORMAT
HELP COMMANDS PROFILE

Wide displays are suitable if your terminal display is set to 136 columns or wider. Narrow displays are suitable when only 80 columns are displayed. See HELP FULLSCREEN TERMINALS for more information.

The **EBCDIC** and **ASCII** operands also are optional. They control, for the character

portion of the display, whether the register contents are interpreted as EBCDIC characters or ASCII. If omitted, then the current SET FORMAT command setting is used.

Syntax #2:

```
LIST AREGS rbaddress WIDE   EBCDIC
           ssrbaddress NARROW ASCII
```

If an address expression is given, then it must resolve to either of the following:

- The address of a **Request Block** that is currently queued from any TCB located in any accessible address space anywhere in the system. (Note, the **RB#n** automatic equates created by the LIST RBS command are perfectly good address expressions to use for this rbaddress operand.)

- The address of an **SSRB** or **SSRX** for any SRB routine currently suspended in any accessible address space anywhere in the system. (Note, the **SSRB#n** and **SSRX#n** automatic equates created by the LIST SSRBS command are perfectly good address expressions to use for this ssrbaddress operand.)

(Note, specifying an address is not supported for error level registers.)

When an RB address is given, Z/XDC will display the access registers that belong to the program that is currently running under that RB. (These are the registers that are stored in the next newer RB's XSB, or in the STCB in the case of the newest RB.) Note that the **LIST RBS** command, as a side effect, creates a series of equates named **RB#n** to label the locations of the Request Blocks that it displays. These equates make handy dandy address expressions for use with the **LIST AREGS** command.

When an SSRB or SSRX address is given, Z/XDC will display the access registers that belong to the SRB routine that is currently suspended under that SSRB and SSRX. Note that the **LIST SSRBS** command, as a side effect, creates a series of equates named **SSRB#n** and **SSRX#n** to label the locations of the SSRBs and SSRXs from which it builds its displays. These equates can be used as the ssrbaddress address expression on the **LIST AREGS** command.

Syntax #3:

```
LIST EAREGS WIDE   EBCDIC
            NARROW ASCII
```

This form of the command can be used to display the access registers from the current program's error level environment.

Automatic Equates

The **LIST AREGS** command, when given with an **rbaddress** or **ssrbaddress** operand (see **Syntax #2** above), (re)generates a **#AREGS** equate that labels the system control

block field in which the access register data was found. For more information, see HELP EQUATES BUILTIN AUTOMATIC.

Notice: the **#AREGS** equate is **static!** It does not float. The location that it targets can change only when another **LIST AREGS** command is issued. In other words, the equate can become obsolete without warning simply due to the ongoing execution of the program whose access registers are being displayed.

Examples:

LIST AREGS
Lists the contents of entire set of retry level access registers. The display will be either 4 words wide or 8 words wide according to the current WIDE/NARROW setting of the SET FORMAT command.

LIST EAREGS WIDE
Lists the contents of entire set of error level access registers. The display will be 8 words wide.

LIST TASKS JES2
LIST RBS TCB#3
LIST AREGS RB#1
If Z/XDC is running authorized, and if System Security permits the access, then the **LIST TASKS JES2** command displays the subtask structure for JES2's address space. It also creates a series of **TCB#n** equates labeling the locations of all TCBs within the JES2 address space.

The **LIST RBS** command then makes use of the **TCB#3** equate to display the Request Blocks currently queued to JES2's jobstep task. It also creates a series of **RB#n** equates labeling the locations of all such RBs.

Finally, the **LIST AREGS RB#1** command displays the contents of the entire set of access registers that belong to the program (HASJES20) running under the control of the jobstep task's oldest Request Block.

Because an **rbaddress** operand was given, the following automatic equate is (re)generated:

- **#AREGS** labels the field in which the access register data was found. That will be one of the following:
  - If **RB#1** is **TCB#3's** newest Request Block, then **#AREGS** will label the **STCBARS** field in the STCB pointed to by TCB#3.
  - If **RB#1** is **not TCB#3's** newest Request Block, then **#AREGS** will label the **XSBARS** field in the XSB pointed to by the next newer RB.

Warning! The location of the **#AREGS** equate will **not** change with changes in the target program's environment. It can be changed only by reissuing the **LIST AREGS RB#1** command.

LIST SSRBS 1
LIST AREGS SSRB#1 [or **SSRX#1]**

If Z/XDC is running authorized, and if System Security permits the access, then the **LIST SSRBS 1** command displays a list of all SSRBs suspended in the Master Scheduler's address space. It also creates a series of **SSRB#n** and **SSRX#n** equates labeling the locations of all the SSRBs and SSRXs that were found.

Then the **LIST AREGS SSRB#1** command (or **LIST AREGS SSRX#1** command) displays all access registers belonging to the SRB routine running under the control of that SSRB and SSRX.

Because an **ssrbaddress** operand was given, the following automatic equate is (re)generated:

- **#AREGS** labels the field in which the access register data was found.

Warning! The location of the **#AREGS** equate will **not** change with changes in the target program's environment. It can be changed only by reissuing the **LIST AREGS RB#1** command.

Help COmmands LISt AL

This command is an alias of the **LIST ACCESSLISTS** command. See HELP COMMANDS LIST ACCESSLISTS for more information.

Help COmmands LISt ALEs

This command is an alias of the **LIST ACCESSLISTS** command. See HELP COMMANDS LIST ACCESSLISTS for more information.

Help COmmands LISt AMode

The **LIST AMODE** command displays the retry level PSW's current addressing mode setting, 24-bit, 31-bit, or 64-bit (z/OS only). This setting is used by Z/XDC for several purposes:

- For resuming user program execution (via GO or TRACE).
- For resolving the "!" operator in address expressions (when SET BANG AMODE is in effect).
- For determining the width of an address value to be stored via the "address-data" form of the ZAP command.
- For determining whether Z/XDC's GETMAIN command is to obtain storage from 24-bit or 31-bit storage.

Syntax:

LIST AMODE

The current addressing mode can be changed via the "SET PSW" command.

Help COmmands LISt AR#

The **LIST ARn** command displays an individual retry level access register. For more information, see HELP COMMANDS LIST ACCESSREGISTERS.

Help COmmands LISt AREgs

The **LIST AREGS** command displays the whole set of retry level access registers. For more information, see HELP COMMANDS LIST ACCESSREGISTERS.

Help COmmands LISt ASID

The **LIST ASID** command can be used to identify the ASIDs of one or more address spaces identified by jobname (or other information) and vice versa.

The display produced by the LIST ASID command shows the following information:
- **ASID:** The address space’s id number.
- **JOBNAME:** The address space’s name (job name, system task name, or TSO userid).
- **SUBSYSTEM:** The name of the subsystem that owns the address space (JES2, JES3, or MSTR usually).
- **JOBNUMBER:** The JES2/JES3 job type (JOB, TSU, or STC) and number, if applicable.
- **USS?:** An indication that the address space was created to execute USS processes.

Syntax:

LIST ASID aspaceref
omitted

aspaceref
This identifies the one or more address spaces for which information is to be displayed. Briefly, this can be any of the following:
- A jobname. It may also contain * wildcard character .
- A keyword indicating the home space or the retry level or error level, primary, secondary, or instruction execution address space. Examples: HOME, PASID, ESASID, IASID, etc.
- The keyword REAL, indicating real storage.
- An ASID number.
- An address expression pointing to an ASID number.
- A register containing an ASID number.

- The name of an equate or dsect that has been assigned to represent storage in an address space. (The assigned address space is used.)

For detailed information, see HELP COMMANDS SYNTAX ASIDS.

omitted
When no operand is given, information about Z/XDC's current target address space (either the home space or as established by the SET ASID command) will be displayed.

Examples:

LIST ASID INIT
Information will be displayed about all address spaces named INIT.

**LIST ASID XXXX*
Information will be displayed about all address spaces whose jobname matches the wildcard pattern **XXX*.**

LIST ASID 1
Information about the Master Scheduler address space will be displayed.

LIST ASID EIASID
Information about the error level, instruction execution, address space will be displayed.

LIST ASID PASID
Information about the retry level primary address space will be displayed.

LIST ASID .ASCBASID
Assuming that the ASCB dsect map has been loaded and assigned to represent an Address Space Control Block of interest, the address space that that ASCB describes will be displayed. (In this case, the command's operand is resolved as an address expression, and the contents of the two bytes to which that expression points is what is used.)

LIST ASID CR3
Information about the retry level secondary address space will be displayed.

LIST ASID ECR3
Information about the error level secondary address space will be displayed.

LIST ASID R5

The two low order bytes of register R5 are interpreted as an ASID, so information will be displayed about the address space having that ASID.

Help COmmands LISt AUtomap

Auto mapping is the automatic loading of load module, csect and source image maps at the point where they are first needed.

When AUTOMAPing is set **ON,** then whenever Z/XDC receives control, it checks both the error level PSW and the retry level PSW to see what load modules and csects they point into. If the mapping of those modules/csects has not yet been attempted, then Z/XDC will automatically map them now (or at least try to).

For more information about maps, see HELP MAPS.

AUTOMAPing controls:

- Are established by the **SET AUTOMAP** command,
- Are displayed by the **LIST AUTOMAP** command (described here),
- And are saved in your session profile by the **PROFILE SAVE** command.

They also can be displayed, set and saved by the **Profile Menuing System.**

Syntax:

LIST AUTOMAP

This command accepts no operands.

For detailed information about the AUTOMAP settings, see HELP COMMANDS SET AUTOMAP.

Help COmmands LISt BAng

Notice: This command is deprecated! It may be dropped without notice from a future release of Z/XDC. Z/XDC has standardized on the meaning of the ! operator being "64-bit indirect". Support for its older meaning (AMODE-sensitive indirect) may be dropped in a future release.

In address expressions, there are three characters that Z/XDC interprets as being "indirect operators". They are the percent sign (%), the question mark (?), and the exclamation point (! - a.k.a. "bang"). The percent sign loads a 24-bit wide pointer, and the question mark loads a 31-bit wide pointer, but the bang character can have either of two meanings:

- **AMODE:** The bang is amode sensitive. This means that it loads a pointer whose width matches the retry level PSW's current addressing mode: 24-bit, 31-bit, or (for z/OS systems) 64-bit.
- **64BIT:** The bang always loads a 64-bit wide pointer (z/OS only).

For more information about indirect operators, see HELP ADDRESSING SYNTAX BETWEENTERMS INDIRECT.

The **LIST BANG** command displays the "!" character's current meaning.

Syntax:

LIST BANG

This command accepts no operands.

The BANG setting can be saved in your session profile. It can be changed by the SET BANG command. It also can be displayed and changed by Z/XDC's "Profile Menuing System". For more information, please see:

HELP COMMANDS SET BANG
HELP PROFILES MENU

The initial factory default setting is **64BIT.**

You might notice that Z/XDC has something of a problem regarding indirect operators: There are at least four distinct useful indirect operations (more, actually), but only three characters (% ? !) are available to represent them. So to pick up the slack, there is a built-in function that can be used to perform a variety of indirect operations. Its name is **~INDIRECT(...).** See HELP FUNCTIONS INDIRECT for more information.

Help COmmands LISt BEA

The **LIST BEA** command displays the Breaking Event Address (BEA). It can be used to display any of the following BEAs:

- The current program's retry level BEA.
- The BEA belonging to any program running under any RB queued from any TCB located in any accessible address space.
- The BEA belonging to any suspended SRB routine located in any accessible address space.

The **BEA** is a feature on newer IBM mainframes and more recent levels of z/OS. It records the location of the most recent branching instruction (or any other instruction that changes the execution flow) executed prior to an interrupt. It is a great help for debugging wild branches to unreasonable locations.

In order to have the **BEA** available to the debugging session, the following conditions must be met:

- You must be running on a mainframe that has BEA support implemented. Mainframes starting with the z890 have this facility.
- You must be running with a level of z/OS that exploits the BEA hardware support. z/OS 1.7 and later have this exploitation.
- If you are displaying the retry level BEA and the error level and retry level are the same, the SDWARC4 section of the SDWA must also be present. (see the section in this help titled **Caution: SDWARC4 May Be Required** for more information.)

@BEA is a built-in equate that labels the BEA location. **@BEA** can be used in address expressions.

Syntax:

```
LIST BEA  rbaddress
          ssrbaddress
          omitted
```

Operands:

rbaddress
This must be the address of any Request Block, queued from any TCB, located in any accessible address space in the system. When the RB is found, the BEA contained within the XSB pointed to by the RB is displayed.

Note, after a **LIST RBS** command is issued, any of the **RB#n** equates that it generates can be used as an rbaddress operand for this **LIST BEA** command.

ssrbaddress
This must be the address of any SSRB or SSRX located in any accessible address space in the system. When the SSRB or SSRX is found, the BEA contained within the XSB pointed to by the SSRB is displayed.

Note, after a **LIST SSRBS** command is issued, any of the **SSRB#n** equates or **SSRX#n** equates that it generates can be used as an ssrbaddress operand for this **LIST BEA** command.

omitted
When no address operand is given, then the **retry level BEA** is displayed. (See HELP EXECUTIONLEVELS for information about retry level vs. error level environments.)

Automatic Equates

When an **rbaddress** or **ssrbaddress** operand is given on the **LIST BEA** command, the following automatic equate is (re)generated:

#BEA
This labels the location pointed by the BEA.

This equate is **not** generated when the **LIST BEA** command is given without operands. For more information, see HELP EQUATES BUILTIN AUTOMATIC.

Notice: This automatic equate is **static!** It does not float. The locations that it targets can change only when another **LIST BEA address** command is issued. In other words, the equate can become obsolete without warning simply due to the ongoing execution of the program whose BEA is being displayed.

Related Commands:

- Use the **LIST EBEA** command to display the **error level** BEA.

Caution: SDWARC4 May Be Required

If displaying the retry BEA (i.e. no operands were used in the **LIST BEA** command) and the retry level and error level are the same, the following is relevant...

Note that the BEA is in the SDWARC4 extension of the SDWA, and this extension only exists if the SDWA is above the 16m line (i.e. in 31 bit storage)

If the SDWARC4 extension does not exist, you will receive an error message informing you of this reason.

You can ensure that the SDWARC4 extension exists by:

- Using the ESTAEX or ATTACHX Assembler macros.
- Specifying SDWALOC31=YES on your ATTACH or ESTAE or FESTAE or SETFRR Assembler macros.

Caution: Aspace information not recorded

The hardware support for the **BEA** only records the address. It does not record the address space in which that address is located. Consequently, under certain conditions the location displayed by this **LIST BEA** command might be in the wrong address space.

This situation can occur when the last prior flow-of-execution changing event (branching instruction, PC instruction, whatever) occurred in one address space while the interrupt that caused the BEA to be saved occurred in a different address space.

The best example of this occurs when a trap is placed at the start of a space switching PC routine. When Z/XDC receives control due to the trap, the execution address space will be that in which the PC routine resides. But the BEA location will be at the PC instruction located in the calling address space.

This issue can also arise with the **L PSW** and **L PSWE** commands.

Examples:

L BEA
This displays the retry level BEA.

L BEA,.TCBRBP?
Presumedly, the dsect that maps the TCB has been assigned to a suitable location. This command then displays the BEA for the newest RB chained from that TCB.

L RBS
L BEA,RB#1
This displays the BEA for the oldest RB chained from the current TCB. The **LIST RBS** command generated the **RB#1** equate to represent the location of the task's oldest RB.

Because an **rbaddress** operand was given, the following automatic equate is (re)generated:

- The **#BEA** equate labels the location pointed to by the BEA.

Warning! The location of this equate will **not** change as the program running under **RB#1** executes. It can be changed only by reissuing the **LIST BEA RB#1** command.

L SSRBS 1
L BEA SSRB#1 [or **SSRX#1]**
The **LIST SSRBS 1** command displays all SRBs that are currently suspended in the Master Scheduler address space. It also creates a series of **SSRB#n** and **SSRX#n** equates labeling the locations of all the SSRBs and SSRXs that were found.

The **LIST BEA SSRB#1** command then displays the BEA for the first of those SRBs.

Because an **ssrbaddress** operand was given, the following automatic equate is (re)generated:

- The **#BEA** equate labels the location pointed to by the BEA.

Warning! The location of this equate will **not** change as the program running under **SSRB#1** executes. It can be changed only by reissuing the **LIST BEA SSRB#1** command.

L EBEA
This displays the error level BEA.

Help COmmands LISt BELl

The **LIST BELL** command displays a report of the circumstances under which Z/XDC will sound your display terminal's "alarm" (or whatever sound you may have set in place of the alarm).

Syntax:

LIST BELL

This command accepts no operands.

The BELL setting can be saved in your session profile. It can be changed by the SET BELL command. It also can be displayed and changed by Z/XDC's "Profile Menuing

System". For more information, please see:
    HELP COMMANDS SET BELL
    HELP PROFILES MENU

Help COmmands LISt BFr#

The **LIST BFRn** command displays the entire 8 bytes of an individual floating point register in the following formats:
  - Raw hex-EBCDIC or hex-ASCII
  - Binary floating point (BFP), short and long

For more information, see HELP COMMANDS LIST FLOATINGPOINTREGISTERS.

Help COmmands LISt BPts

This command is an alias of the **LIST BREAKPOINTS** command. See HELP COMMANDS LIST BREAKPOINTS for more information.

Help COmmands LISt BRAnches

The **LIST BRANCHES** command displays a list of recent user-program machine instructions...
  (a) That have been executed by the user program,
  (b) That are branch-type instructions, and
  (c) (Big caveat here!) That Z/XDC knows about.

Every time Z/XDC receives control, it checks to see whether or not the user program's current retry level instruction is a branch type instruction (including SVC instructions and PC instructions). If so, then an entry is added to an internal **Branch History Table** to record this event. The **LIST BRANCHES** instruction then displays information from that table.

Not every branch that the user program executes will be recorded in the Branch History Table. Only those branches at which Z/XDC receives control will be recorded. But Z/XDC receives control **more often** than you might realize. Examples:

- When you issue a **T BNC** command, Z/XDC receives control for **all** encountered branches, not just the successful branches. (This is necessary so that Z/XDC can determine whether or not a particular branch will or will not be successful.) These **hidden branches** will be added to the Branch History Table.

- During conditional trapping or tracing, all calls to Z/XDC for which the relevant conditional expressions all resolve **FALSE** will be hidden. But for those

hidden calls that occur at branching instructions, the branch still will be tabled.

- In particular, the **TRACE...(FALSE)** and **TRAP...(FALSE)** commands can be used creatively to add entries to the Branch History Table.

By the way, an excellent way to accurately populate the Branch History Table is to use a **T BNC (nnn)** command (where **nnn** is a countdown number). This will cause Z/XDC to follow your program's execution for up to **nnn** successful branches. For each branch that your program executes (successful or otherwise), an entry will be added to the Branch History Table.

A reasonable countdown might be **T BNC (1000).**
A reasonable countdown might also be **T BNC (5000).**
Your mileage will vary.

An important characteristic of **T BNC** traces is that execution is not followed into subroutines. You must keep this in mind if unexpected things happen during a **T BNC (nnn)** trace. For a thorough discussion of all the ins and outs, see **HELP COMMANDS TRACE BNC.**

Syntax:

LIST	**BRANCHES**	**addressexpression**	**count**
	BR	**omitted**	**omitted**

Notes and Defaults:

The addressexpression and count operands are both optional. When both operands are given, they may be given in any order.

When two plain decimal numbers are given, the first such is resolved as a count, the second as an address expression.

When a given count is out of range (1 thru 4096) or has other syntactical issues, it is reparsed as an address expression. Example: In **L BR 5000** the **5000** will be parsed as an address expression because it is greater than 4096.

When both the count and address expression operands are omitted, the default count will be **100.**

When the address expression operand is given and only the count operand is omitted, the address filtering is applied against the entire table.

When both the count and address expression operands are given, the address filtering is limited to only those table entries that are allowed by the count.

Operands:

addressexpression

This is a filtering operand. When given, only those Branch History Table entries are displayed whose branch-FROM address or branch-TO address matches the location (including address space) given by this operand. For the syntax of address expressions, see **HELP COMMANDS SYNTAX ADDRESSEXPRESSIONS.**

count
This operand must be a plain decimal number ranging from 1 to 4096. When given, it limits the display to showing not more than the given number of Branch History Table entries.

Examples:

LIST BR
This produces a display of the newest **100** entries currently contained in the Branch History Table.

LIST BRANCHES .TOPLOOP
If the Branch History Table contains entries:
- For branch instructions that jump **to** TOPLOOP,
- Or for jumps **away from** TOPLOOP (if TOPLOOP happens to label a branch type instruction),

Then only those table entries are displayed.

The entire table is scanned in search of .TOPLOOP entries.

LIST BRANCHES .TOPLOOP 100
LIST BRANCHES 100 .TOPLOOP
These two commands both produce the same display as the preceding except that the search for TOPLOOP related entries is limited to the 100 newest table entries.

Help COmmands LISt BREakpoints

The **LIST BREAKPOINTS** (or **LIST BPTS)** command displays the following information about all currently defined breakpoints:

- Its name. This is a composite of the breakpoint’s type (AT, TRAP, or TRACE) and its family identifier.

- Its location. This is shown as both a raw address and relative to the nearest relevant load module, map, or equate (if any).

- Its associated conditional expression, if any. If there is such an expression, then a count of the total number of times the breakpoint has been reached (regardless of whether the hit was accepted or bypassed) is also displayed.

- Its associated automatic commands, if any.

- For deferred breakpoint definitions, the number of times the breakpoint has been cloned, and the number of cloning errors that have occurred are shown.

In addition, LIST BREAKPOINTS (or LIST BPTS) also shows:
- The current default conditional expression (if any).
- The current default automatic commands string (if any).
- Whether execution location displays will roll or scroll during tracing. (See HELP COMMANDS SET TRACE ROLL.)

Syntax:

```
LIST BREAKPOINTS
     BPTS
     B
```

This command accepts no operands.

Help COmmands LISt CAChe

Because of its shear massiveness, when Z/XDC reads ADATA, it caches an extraction of the data into storage just in case it might be needed again. A cached ADATA file can run to several megabytes of 31-bit storage, so its possible that the presence of the cached data could cause a storage shortage. The **LIST CACHE** command can be used to display information about what ADATA files are cached and how much storage they take.

Syntax:

```
LIST CACHE dsname         CSECTS
           dsname(member) DSECTS
           #nnn           COMMON
           omitted        PRIVATE
                          ALLSECTIONS
                          NOSECTIONS
                          omitted
```

Default: **LIST CACHE ALLSECTIONS**

Notes:
- Any combination of operands may be given in any order.
- Operands from the first column allow you to limit the display to only those cache files owned by particular MAPLIB datasets.
- Operands from the second column allow you to limit the display to only particular ESD types.

dsname
dsname(member)
These operands identify by name which cache files to display. If the cache file is

associated with a sequential dataset, then the **dsname** form has to be used. Similarly, if the cache file is associated with a member of a partitioned dataset, then the **dsname(member)** form has to be used.

#nnn
"nnn" must be a decimal number. This operand permits you to display information about a particular cache file by referencing its sequence number as shown by a preceding LIST CACHE display.

omitted
Information for all cached files is displayed.

ALLSECTIONS
This is the default. For all cached ADATA files to be displayed, the reports will show the names of all csects, dsects, common blocks, and private areas described by the cached ADATA.

CSECTS
DSECTS
COMMON
PRIVATE
These operands limit the displayed reports to showing information about particular section types: csects or dsects or common blocks or private areas.

NOSECTIONS
This suppresses the lists of section names in the displayed reports. Only the names of the MAPLIBs that own the cache files are displayed.

Examples:

SET MAPLIBS hlq.XDCV31.XDCADATA

DMAP XDCMAPS.PSA
LIST CACHE ALLSECTIONS

The first command makes the ADATA version of Z/XDC's XDCMAPS module available to the DMAP command. (See HELP MAPS XDCMAPS ADATA for more information.)

The second command (DMAP XDCMAPS.PSA) reads the ADATA for XDCMAPS in order to find and build a dsect map for the PSA. In addition, the command causes a copy of all of XDCMAPS' ADATA to be cached into 31-bit storage.

The third command (LIST CACHE ALLSECTIONS) produces a display that (among other things) shows the names of all of the dsects contained in the XDCMAPS module.

Help COmmands LISt CANdet

The **LIST CANDET** command displays a message that shows whether or not Z/XDC will intercept and debug "termination" ABENDs; i.e. ABENDs that are a result of cancellation (x22 ABENDs) or subtask DETACH (x3E ABENDs) or other impossible-to-continue ABENDs (such as s00C-04).

When the System calls a recovery routine to handle an ABEND, if the System has set **SDWACLUP=1,** then the System considers the ABEND to be impossible to recover

from. Frequently, debugging such ABENDs is pointless, since the cause of the ABEND is entirely external to the program (Operator Cancel, for example). However,if you want to use Z/XDC to poke around anyway, you can use **SET CANDET DEBUG** to allow it. See HELP COMMANDS SET CANDET for more information.

Syntax:

LIST CANDET

This command does not accept operands.

This setting can be saved in your session profile. It can be changed by the "SET CANDET" command. It also can be displayed and changed by Z/XDC's "Profile Menuing System". For more information, please see:

HELP COMMANDS SET CANDET
HELP PROFILES MENU

Help COmmands LISt CAPs

This command is an alias of the **LIST LICENSE** command. See **HELP COMMANDS LIST LICENSE** for more information.

Help COmmands LISt CDf

The **LIST CDF** command displays the following cdf/XDC environment settings:

- Whether or not cdf/XDC services are being used by this debugging session.
- Whether or not the **DBC640Q** WTOR will be displayed for pending CDF signons.
- The period of time that cdf/XDC may wait for a terminal to signon to the debugging session.

Syntax:

LIST CDF

This command accepts no operands.

For more information, see:

HELP XDCSRVER CDF
HELP COMMANDS SET DBC640Q
HELP COMMANDS SET SIGNONWAIT

Help COmmands LISt COLORs

Screen images generated by Z/XDC contain up to four different kinds of fields:

(A) - **Low** intensity **input** fields for displaying data in secondary input areas and in overwritable fields (such as storage displays).

(B) - **High** intensity **input** fields for primary input areas such as command lines.

(C) - **Low** intensity **protected** fields for displaying nonoverwritable information.

(D) - **High** intensity **protected** fields for displaying title and header lines, warning messages, etc.

For those terminals that have the necessary hardware support, display colors and extended hilighting can be set by the following commands:

- SET COLORS
- SET HILIGHT
- SET INTENSITY

The **LIST COLORS** command displays the current values for the field colors.

Syntax:

LIST COLORS
COLOURS

This command accepts no operands.

The field colors can also be displayed by the LIST TERMINAL command. In addition, they can be both displayed and set by Z/XDC's Profile Menuing System.

For more information, see:

HELP COMMANDS SET COLORS
HELP COMMANDS LIST TERMINAL
HELP COMMANDS PROFILE
HELP PROFILES MENU

Note: Even if you have a color terminal, your color choices can be ignored under the following circumstances:

- Your VTAM Systems Programmer has not properly defined your terminal as supporting "Extended Attribute Bytes" (done via a suitable PSERVIC= value on a MODEENT macro in a VTAM mode table definition).
- You are using a PC workstation program to emulate a terminal, but you have not properly configured your emulation to support "color orders" from VTAM.

See HELP FULLSCREEN TERMINALS for more information and for suggestions about remedying these problems.

Help COmmands LISt COLOUrs

This command is an alias of the **LIST COLORS** command. See HELP COMMANDS LIST COLORS for more information.

Help COmmands LISt CONSoles

The **LIST CONSOLES** command produces displays of and information about the System Operator Consoles that are currently available in the System.

Broadly speaking, the Operating System supports three categories of consoles: MCS consoles, SMCS consoles and EMCS consoles. These categories are distinguished by the underlying software technology that creates them. If you would like more detailed information about this, see **HELP COMMANDS LIST CONSOLES MCS.**

Note, the LIST CONSOLES command can display information only about MCS and EMCS consoles, not SMCS consoles. If you need a display of SMCS consoles, then you will have to use the Operating System's DISPLAY CONSOLES command.

Syntax:

LIST CONSOLES	**ACTIVE**	**MCS**	**CURRENT**	**names**
	INACTIVE	**EMCS**	**omitted**	**numbers**
	ALLCONS	**omitted**		**omitted**
	omitted			

Rules:

- All operands are optional. When all operands are omitted, then only active consoles are displayed.
- Operands appearing above in the same column are mutually exclusive.
- Any number of console names may be given.
- Any number of console id#'s may be given.
- Keyword operands that are excessively abbreviated will be parsed as console names instead of as keywords.

Operands:

ACT
ACTIVE
[ACTIVE, INACTIVE and ALLCONS all omitted]
The report will list only active consoles. Notes:
- This is the default when ACTIVE, INACTIVE and ALLCONS are all omitted.

- The minimum abbreviation of ACTIVE is ACT.

INACT
INACTIVE
The report will list only inactive consoles. The minimum abbreviation of INACTIVE is INACT.

ALLCONS
The report will list both active and inactive consoles. ALLCONS cannot be abbreviated.

MCS

Only MCS consoles are eligible for inclusion in the report. (See **HELP * MCS** for information about console types.) MCS cannot be abbreviated.

EMCS

Only EMCS consoles are eligible for inclusion in the report. (See **HELP * MCS** for information about console types.) terminals. EMCS cannot be abbreviated.

CURR
CURRENT (or omitted)
If Z/XDC is currently conducting the debugging session via a particular Operator Console, then that console will be included in the report. The minimum abbreviation of CURRENT is CURR.

Console names
These are the names of particular consoles to be included in the report. The syntax of these names is the normal 8-character name syntax use throughout classic z/OS. If the given names match against **any name** associated with a console, that console will be included in the report. **"Any name"** means the following names will be checked:
- Command system name
- Console name
- Job name
- System name
- Terminal name
- User key

Console id numbers
These are the hexadecimal ID numbers of the consoles to be included in the report.

If you prefer to provide a decimal number, just trail the number with an **n** .

Ambiguous numbers
If an given id# starts with a Hex digit A thru F, then syntactically the number could also be a console name. In these cases, the given number is first searched for as a name before being searched for as an ID number.

To avoid the ambiguity, just prepend the number with a leading zero.

Examples:

LIST CONSOLES
All active MCS and EMCS consoles that are defined in the current system are displayed. (SMCS consoles are not displayed.)

In addition, if Z/XDC's user interface is currently using a console, then that console, is flagged with an asterisk (*).

LIST CONSOLES CURRENT
If Z/XDC's user interface is currently using a console, then that console, and just that console, is displayed.

LIST CONSOLES A
LIST CONSOLES 0A
LIST CONSOLES 10n
These commands each display the console whose 4-byte console id# is X'0000000A'.

Note, this may be unlikely, but if there were a console that had a name string of CL8'A', then the first of the above three commands would cause that console to be displayed as well.

LIST CONSOLES 2000001
LIST CONSOLES 16777218n
These commands both display the console whose 4-byte console id# is X'02000001'.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

MCS - Discusses the differences between MCS, SMCS and EMCS consoles.

Help COmmands LISt CONSoles Mcs

Broadly speaking, the Operating System supports three categories of consoles: MCS consoles, SMCS consoles and EMCS consoles. These categories are distinguished by the underlying software technology that creates them:

- **MCS** stands for "Multiple Console Support". MCS consoles historically have been hardware terminals, located in or near the Data Center's machine room, and used by System Operators for responding to and controlling the Operating System. They are locally attached devices. They are connected to the system via hardware that cannot be used for telecommunications purposes.

- **SMCS consoles** are "SNA MCS" consoles. They are consoles that are connected to the system via a telecommunications protocol (SNA or TCP/IP). Consequently, these consoles can be located either locally or remotely, anywhere at all. (Note, in this case, "MCS" stands for "Master Console Support". Funny how acronyms sometimes wander over time.)

- **EMCS consoles** are "Extended MCS" consoles. These are pseudo consoles created dynamically by authorized programs to provide an on-the-fly system command interface for whatever reason the creating program deems appropriate. Both IBM's **SDSF** and Triangle System's **IOF** are examples of programs that create EMCS consoles. Both of these programs use EMCS consoles to allow their users to issue system commands and receive their responses.

Z/XDC's LIST CONSOLES command can display MCS and EMCS consoles only. Information about SMCS consoles cannot be displayed by Z/XDC. (You can, instead, display this information via the Operating System's DISPLAY CONSOLES command.)

Help COmmands LISt CONTrolregisters

Control registers are used for controlling a large number of fundamental characteristics of the computer's hardware. They control such things as which page tables are to be used for virtual address translation, whether or not the current program is running in cross memory mode and (if so) what other address spaces are being accessed in that mode, and a myriad of other highly specific characteristics. Complete documentation can be found in IBM's various **Principles of Operation** manuals (SA22-7201 in OS/390 and SA22-7832 in z/OS).

There are 16 control registers, numbered from 0 to 15. For OS/390 they are 4 bytes wide. For z/OS they are 8 bytes wide.

Unlike for general registers and access registers, z/OS does not maintain a unique set of control registers for each individual running program. Instead, only one set is kept per execution thread (task and SRB routine). Therefore, there can be no distinction made between an "error level" set and a "retry level set" of control registers. There is only one set for Z/XDC to maintain per execution thread.

Z/XDC does not limit you to displaying just **your** program's control registers. Every task (and SRB) has its own set of control registers, so (security and authorization permitting) Z/XDC allows you to display the control registers associated with **any** Task Control Block (TCB), located in **any** address space in the entire system! But Please note the following:

- The control registers that are associated with a given execution thread (that is

not the current execution thread being debugged) are reconstructed (as much as possible) from information stored in various control blocks that are related to the thread.
- But not all control register data can be reconstructed. Consequently, register fields that cannot be figured out will be displayed dashed out.

Z/XDC also allows you to display the control registers associated with a suspended SRB routine located in any address space in the system.

In some cases, the contents of all or part of a control register are not actually knowable. Retry level register CR12 is a case in point. This register is supposed to be the system's trace table cursor. Well, the system is constantly writing entries to its trace table, and since by definition the retry level environment is located in the future, there is no way to actually predict what the retry level CR12 really will be at user program retry time.

Accordingly, when the contents of all or part of a control register cannot be known or otherwise figured out, the unknowable field will be displayed as dashes.

Subtopics Index

Z/XDC allows you either to display individual control registers or to display entire register sets. For specific information, type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

INDIVIDUAL - Displaying individual control registers.
REGISTERSET - Displaying an entire control register set.

Help COmmands LISt CONTrolregisters Individual

Z/XDC allows you to display an individual control register:
- Belonging to the current program.
- Belonging to any program running under any RB queued from any TCB located in any accessible address space.
- Belonging to any suspended SRB routine located in any accessible address space.

Please note that for control registers, floating-point registers and vector registers, z/OS does not maintain a distinction between the error level and retry level environments.

The control register is displayed in a hex-text format. Its contents also are interpreted and displayed in highly formatted and thoroughly annotated detail.

If any part of a control register is, for one reason or another, unknowable, then that part of the display is dashed out.

In z/OS, Z/XDC provides support for three different views of the control registers. You can display either their high halves, their low halves, or their entireties. Z/XDC defines a different name for the control registers depending upon which view you want to see. The high halves are named CRHn, the low halves are named CRn, and the entireties are named CRWn.

Syntax:

LIST CRn
nCR

In OS/390, either of these commands displays the entirety (all 4 bytes) of a **retry level** control register. In z/OS, these commands display only the low half (the 5th thru 8th bytes) of a retry level control register.

n is a decimal number ranging from 0 to 15. It can appear either before or after the letters CR.

LIST CRHn
nCRH

In z/OS, these commands display the high half of a retry level control register. (These commands are not permitted in OS/390.)

LIST CRWn
nCRW

In z/OS, these commands display the entirety (all 8 bytes) of a retry level control register. (These commands are not permitted in OS/390.)

LIST ECRn
nECR
ECRHn
nECRH
ECRWn
nECRW

Since z/OS does not create a distinction between error level and retry level control registers, these commands are merely synonyms for the corresponding non-error level versions.

LIST CRn **rbaddress**
nCR **ssrbaddress**
CRHn
nCRH
CRWn
nCRW

If an address expression is given, then it must resolve to either of the following:

- The address of a **Request Block** that is currently queued from any TCB located in any accessible address space anywhere in the system. (Note, the **RB#n** automatic equates created by the LIST RBS command are perfectly good address expressions to use for this rbaddress operand.)

- The address of an **SSRB** or **SSRX** for any SRB routine currently suspended in any accessible address space anywhere in the system. (Note, the **SSRB#n** and **SSRX#n** automatic equates created by the LIST SSRBS command are perfectly good address expressions to use for this ssrbaddress operand.)

When an RB address is given, Z/XDC will attempt to display the control registers that belong to the program that is currently running under that RB. Note that the **LIST RBS** command, as a side effect, creates a series of equates

named **RB#n** to label the locations of the Request Blocks that it displays. These equates make handy dandy address expressions for use with the various **LIST controlregister** commands.

When an SSRB or SSRX address is given, Z/XDC will display the control register that belongs to the SRB routine that is currently suspended under that SSRB and SSRX. Note that the **LIST SSRBS** command, as a side effect, creates a series of equates named **SSRB#n** and **SSRX#n** to label the locations of the SSRBs and SSRXs from which it builds its displays. These equates can be used as the ssrbaddress address expression on the **LIST controlregister** and other **LIST register** commands.

Examples:

LIST CR0
Lists and formats the contents of retry level control register 0. In OS/390, the entire CR0 is displayed. In z/OS, only the low half is displayed.

LIST 3CRW
In z/OS, this displays the entire contents (all 8 bytes) of error level control register 3.

LIST TASKS JES2
LIST RBS TCB#3
LIST CR4 RB#1
If Z/XDC is running authorized, and if System Security permits the access, then the **LIST TASKS JES2** command displays the subtask structure for JES2's address space. It also creates a series of **TCB#n** equates labeling the locations of all TCBs within the JES2 address space.

The **LIST RBS** command then makes use of the **TCB#3** equate to display the Request Blocks currently queued to JES2's jobstep task. It also creates a series of **RB#n** equates labeling the locations of all such RBs.

Finally, the **LIST CR4 RB#1** command displays the contents of control register 4 that is **associated** with the program (HASJES20) running under the control of the jobstep task's oldest Request Block. In OS/390, the entire CR4 is displayed. In z/OS, only the low half is displayed.

LIST SSRBS 1
LIST 3CR SSRB#1 [or **SSRX#1]**
If Z/XDC is running authorized, and if System Security permits the access, then the **LIST SSRBS 1** command displays a list of all SSRBs suspended in the Master Scheduler's address space. It also creates a series of **SSRB#n** and **SSRX#n** equates labeling the locations of all the SSRBs and SSRXs that were found.

Then the **LIST 3CR SSRB#1** command (or **LIST 3CR SSRX#1** command) displays the contents of control register 3 belonging to the SRB routine running under the control of that SSRB and SSRX.

Help COmmands LISt CONTrolregisters Registerset

The various **LIST CREGS** commands allow you to display the entire set of control registers:

- Belonging to the current program's retry level environment.
- Belonging to the current program's error level environment.
- Belonging to any program running under any RB queued from any TCB located in any accessible address space.
- Belonging to any suspended SRB routine located in any accessible address space.

The control registers are displayed in a hex-text format. (To display a register formatted by functionality, use the **LIST CRn** command [and friends] to display the registers individually.)

If any part of a control register is, for one reason or another, unknowable, then that part of the display is dashed out.

In z/OS, Z/XDC provides support for three different views of the control registers. You can display either their high halves, their low halves, or their entireties. Z/XDC defines a different name for the control registers depending upon which view you want to see. The high halves are named **CRHREGS,** the low halves are named **CREGS** and the entireties are named **CRWREGS.** (In OS/390, only the name **CREGS** is permitted.)

Syntax #1:

```
LIST CREGS   WIDE   EBCDIC
     CRHREGS NARROW ASCII
     CRWREGS
```

These forms of the command (other forms are described below) can be used to display the control registers from the current program's retry level environment.

In **OS/390,** LIST CREGS displays the entireties of all 16 control registers, and the LIST CRHREGS and LIST CRWREGS commands are not permitted.

In **z/OS:**

- LIST CREGS displays the low halves of the control registers.
- LIST CRHREGS displays the high halves of the control registers.
- LIST CRWREGS displays the entireties of the control registers.

The **WIDE** and **NARROW** operands are optional. They control whether the resulting display will show 8 words of data per line **(WIDE)** or just 4 words **(NARROW).** If omitted, then the current current **SET FORMAT** command setting is used. This setting can be displayed by the **LIST FORMAT** command and saved into your session profile by the PROFILE SAVE command. For more information, see:

```
HELP COMMANDS LIST FORMAT
HELP COMMANDS SET FORMAT
HELP COMMANDS PROFILE
```

Wide displays are suitable if your terminal display is set to 136 columns or

wider. Narrow displays are suitable when only 80 columns are displayed. See HELP FULLSCREEN TERMINALS for more information.

The **EBCDIC** and **ASCII** operands also are optional. They control, for the character portion of the display, whether the register contents are interpreted as EBCDIC characters or ASCII. If omitted, then the current SET FORMAT command setting is used.

Syntax #2:

```
LIST CREGS    rbaddress WIDE    EBCDIC
     CRHREGS ssrbaddress NARROW  ASCII
     CRWREGS
```

If an address expression is given, then it must resolve to either of the following:

- The address of a **Request Block** that is currently queued from any TCB located in any accessible address space anywhere in the system. (Note, the **RB#n** automatic equates created by the LIST RBS command are perfectly good address expressions to use for this rbaddress operand.)

- The address of an **SSRB** or **SSRX** for any SRB routine currently suspended in any accessible address space anywhere in the system. (Note, the **SSRB#n** and **SSRX#n** automatic equates created by the LIST SSRBS command are perfectly good address expressions to use for this ssrbaddress operand.)

(Note, specifying an address is not supported for error level registers.)

When an RB address is given, Z/XDC will display the control registers that belong to the program that is currently running under that RB. Note that the **LIST RBS** command, as a side effect, creates a series of equates named **RB#n** to label the locations of the Request Blocks that it displays. These equates make handy dandy address expressions for use with the **LIST CREGS** (and friends) commands.

When an SSRB or SSRX address is given, Z/XDC will display the control registers that belong to the SRB routine that is currently suspended under that SSRB and SSRX. Note that the **LIST SSRBS** command, as a side effect, creates a series of equates named **SSRB#n** and **SSRX#n** to label the locations of the SSRBs and SSRXs from which it builds its displays. These equates can be used as the ssrbaddress address expression on the **LIST controlregister** command.

Syntax #3:

```
LIST ECREGS   WIDE   EBCDIC
     ECRHREGS NARROW ASCII
     ECRWREGS
```

These forms of the command can be used to display the control registers from the current program's error level environment.

Examples:

LIST CREGS
Lists the contents of the entire set of retry level control registers. In OS/390, the entirety of each register is displayed. In z/OS, only the low half of each register is displayed.

LIST ECRWREGS WIDE
Lists the contents of entire set of error level control registers. In z/OS, the entire 8 bytes of each register is displayed, and four registers (32 bytes worth) are displayed per line instead of just two. In OS/390, this command fails.

LIST TASKS JES2
LIST RBS TCB#3
LIST CRHREGS RB#1
If Z/XDC is running authorized, and if System Security permits the access, then the **LIST TASKS JES2** command displays the subtask structure for JES2's address space. It also creates a series of **TCB#n** equates labeling the locations of all TCBs within the JES2 address space.

The **LIST RBS** command then makes use of the **TCB#3** equate to display the Request Blocks currently queued to JES2's jobstep task. It also creates a series of **RB#n** equates labeling the locations of all such RBs.

Finally, the **LIST CRHREGS RB#1** command displays the contents of the hi-order halves of all control registers that belong to the program (HASJES20) running under the control of the jobstep task's oldest Request Block.

LIST SSRBS 1
LIST CRWREGS SSRB#1 [or **SSRX#1]**
If Z/XDC is running authorized, and if System Security permits the access, then the **LIST SSRBS 1** command displays a list of all SSRBs suspended in the Master Scheduler's address space. It also creates a series of **SSRB#n** and **SSRX#n** equates labeling the locations of all the SSRBs and SSRXs that were found.

Then the **LIST CRWREGS SSRB#1** command (or **LIST CRWREGS SSRX#1** command) displays 64-bit views of all control registers belonging to the SRB routine running under the control of that SSRB and SSRX.

Help COmmands LISt CR#

The **LIST CRn** command displays the lo-order half (4 bytes) of an individual retry level control register. For more information, see HELP COMMANDS LIST CONTROLREGISTERS.

Help COmmands LISt CREgs

The **LIST CREGS** command displays the lo-order halves (4 bytes) of the whole set of retry level control registers. For more information, see HELP COMMANDS LIST CONTROLREGISTERS.

Help COmmands LISt CRH#

The **LIST CRHn** command displays the hi-order half (4 bytes) of an individual retry level control register. For more information, see HELP COMMANDS LIST CONTROLREGISTERS.

Help COmmands LISt CRHRegs

The **LIST CRHREGS** command displays the hi-order halves (4 bytes) of the whole set of retry level control registers. For more information, see HELP COMMANDS LIST CONTROLREGISTERS.

Help COmmands LISt CRW#

The **LIST CRWn** command displays all 8 bytes of an individual retry level control register. For more information, see HELP COMMANDS LIST CONTROLREGISTERS.

Help COmmands LISt CRWRegs

The **LIST CRWREGS** command displays the entire 8 bytes of the whole set of retry level control registers. For more information, see HELP COMMANDS LIST CONTROLREGISTERS.

Help COmmands LISt CUrrent

The **LIST CURRENT** command displays information about the currently designated "current execution thread" (CXU):

- For live program debugging, this will be the task-mode or SRB-mode execution thread that ABENDed (or reached a breakpoint) causing control to pass to Z/XDC.
- For dump debugging, this initially will be the execution thread (if determinable) for which the dump was created.
- For dumps, the user can issue the **SET CURRENT** command to designate any thread he

wants as being "current".

Syntax:

LIST CURRENT
LIST CXU

This command does not accept operands.

Help COmmands LISt CWd

The **LIST CWD** command displays the USS Current Working Directory (CWD).

There are circumstances where the CWD is not available:
- If debugging an SRB
- If debugging a dump and there is no CXU or it is an SRB
- If the current task is not a USS process, and has not been 'dubbed', and 'dubbing' is not allowed.

In these cases a message will be displayed indicating the reason that the CWD is not available

Syntax:

LIST CWD [DUB={PROFILE|NO|YES|ASK}]

Options

DUB={PROFILE|NO|YES|ASK}
This option controls whether an undubbed task will be dubbed. If this option is omitted, the default used is 'DUB=PROFILE'. which means that the profile setting of the DUB option will be used.

The values that can be specified are:
- PROFILE - the DUB setting on the profile is used.
- NO - dubbing is not allowed.
- YES - dubbing is allowed.
- ASK - a question will be asked, and the answer to that question will determine if dubbing is allowed.

Help COmmands LISt CXDc

The **LIST CXDC** command displays the current usability of the c/XDC Licensed feature. It reports whether or not the Feature currently is usable, and if not, then why not.

Syntax:

LIST CXDC

This command does not accept operands.

Help COmmands LISt CXU

The **LIST CXU** command is an alias of the **LIST CURRENT** command. Please refer to that command for details and syntax.

Help COmmands LISt DBc640q

When Z/XDC receives control in a background job or system task, it needs to wait for a user to connect to the debugging session that Z/XDC wants to create. (See **HELP XDCSRVER CDF** for information about debugging background programs.)

While awaiting the user signon, Z/XDC issues a WTOR to the System Consoles giving the user some choices over what he may want to do instead of signing on. This message is **DBC640Q.** See **HELP MESSAGES DBC640** for details about what those choices are.

The **LIST DBC640Q** command shows whether displaying the **DBC640Q** message is turned on or off.

Syntax:

LIST DBC640Q

This command accepts no operands.

Notes

This setting is displayed by the **LIST CDF** command.
It also is displayed by the **LIST TFS** command.
This setting can be changed by the **SET DBC640Q** command.
This setting can also be changed by the **Profile Menuing System.**

This setting is saved in your **session profile.**

Help COmmands LISt DDntrans

The **LIST DDNTRANS** command can be used to display the list of currently defined **ddname translation** tokens.

DDNTRANS tokens are used to translate task library ddnames found in a dump (such as STEPLIB) to **LIBn** ddnames found in Module Mapping Extraction Files. For information about **MMEF** files, see **HELP DUMPS MAPPINGDATA MMEFFILES.**

For information on creating DDNTRANS tokens, and details about what they do, see **HELP COMMANDS DEFINE DDNTRANS.**

Syntax:

LIST DDNTRANS

This command does not accept operands.

Help COmmands LISt DFr#

The **LIST DFRn** command displays the entire 8 bytes of an individual floating point register in the following formats:

- Raw hex-EBCDIC or hex-ASCII
- Decimal floating point (DFP), short and long

For more information, see HELP COMMANDS LIST FLOATINGPOINTREGISTERS.

Help COmmands LISt DSEctlibs

The **LIST DSECTLIBS** command can be used to display the list of currently active DSECTLIB tokens identifying libraries to be searched by the DMAP command when looking for mapping data for building DSECT maps.

DSECTLIB libraries contain either ADATA side files or load modules (which may contain SYM data and/or ADATA) to be used by the DMAP command for building dsect maps of control blocks and data areas found in dumps created on foreign systems. When DSECTLIB tokens exist, these libraries are searched instead of the local address space’s task libraries, PLPA libraries and linklist libraries. For more information on creating DSECTLIB tokens and details about how they work, see **HELP**

COMMANDS DEFINE DSECTLIB.

Syntax:

LIST DSECTLIBS

This command does not accept operands.

Help COmmands LISt DSPaces

This command works significantly differently under dump/XDC. See the notes below...
The **LIST DSPACES** command displays a list of some or all dataspaces owned by a specified (and accessible) address space. The dataspaces are displayed in alphabetic order by dataspace name. A wildcard mask can be given to limit the display to only selected dataspaces.

For each dataspace, the following information is displayed:
- The dataspace's name
- The dataspace's type (dataspace vs. subspace vs. real storage)
- Whether the data space is public or private
- The number of authority table entries defined for the dataspace
- Up to the first 32 authority table entries
- A sequence# relating each dataspace to an automatically generated **ASTE#n** equate. (See below.)

In addition, the **LIST DSPACES** command (re)creates a series of equates **(ASTE#n)** that label, in real storage, the **ASN Second Table Entries** that define the dataspaces and anchor their page tables. If a prior set of **ASTE#n** equates exists, then they are deleted prior to the creation of the new equates.

(Note, the **LIST ACCESSLISTS** command also [re]creates **ASTE#n** equates.)

LIST DSPACES under dump/XDC.
Under dump/XDC, this command may be used to obtain a list of the dataspaces (and their owning ASIDs) actually present on the dump.

The ALL operand may be used for the ASID to indicate that all ASIDs are to be reported. In this case, the dataspaces are ordered by name within ASID.

There is no:
- Type (dataspace, subspace, etc)
- Scope (private or public)
- Authority Table information
- Stoken

(this information is not available to dump/XDC).

In addition, no ASTE#n entries are created (any old ones will be left alone).

The list of dataspaces produced by dump/XDC has the following information:
- The owning ASID
- the dataspace name. If the name has non-displayable characters or embedded blanks, the name is also displayed in hex.
- an indication that the dataspace has been **constructed** from several dataspaces. In some cases, IPCS joins several dataspaces together when creating a dump to make one **super** dataspace that may be longer than the z/OS limit on dataspace size of 2G. (Note that z/Architecture does not have a 2G limit on dataspace size; Only z/OS has).
- An ALET that can be used to access the dataspace regardless of the CXU or CAS.
- The lowest address available on the dump.
- The highest address available on the dump.

In addition, 3 equates are set up for each dataspace provided that:
- The dataspace name has only displayable characters in it.
- The dataspace name has no embedded blanks in it.
- The ALET for access is not zero.

None of the generated equates have the owning ASID in the equate name. If several listed dataspaces owned by different ASIDs have the same name, only the equates for the last one will be present.

Also, if a dataspace name happens to end in the strings used as a suffix for the equate names, only the equates for the last one will be present.

Equates are only generated for listed dataspaces. If the list has been filtered by ASID and/or dataspace name only the listed dataspaces will have equates generated. Existing equates will **not** be deleted. If an existing equate has the same name as one of the new equates, the definition will be replaced.

The 3 equates are generated as each dataspace is listed.

The first equate labels the lowest address of dataspace storage actually on the dump.
This equate has the name **#<dspname>1ST**

The second equate labels the highest address of dataspace storage actually on the dump.
This equate has the name **#<dspname>LAST**

The third equate can be used in many storage expressions to access the dataspace as if the **SET ACCESSTO DATASPACE ...** command was issued.
This equate has the name **#<dspname>**

various storage-display shortcut commands can be used on the list produced by this command to display the lowest dataspace storage on the dump. (for example, the **D** shortcut command)

Syntax:

LIST DSPACES aspaceref NAME=mask

Rules:
- Both operands are optional.
- When both operands are given, they may be given in either order.

Operands:

aspaceref
This identifies the address space whose dataspaces are to be displayed. Briefly, this operand can be any of the following:
- A jobname.
- A keyword indicating the home space or the retry level or error level, primary, secondary, or instruction execution address space. Examples: HOME, PASID, ESASID, IASID, etc.
- Under dump/XDC only, the keyword **ALL** may also be used to select all ASIDs.
- An ASID number.
- An address expression pointing to an ASID number.
- A register containing an ASID number.
- The name of an equate or dsect that has been assigned to represent storage in an address space. (The assigned address space is used.)

For complete and detailed information, see **HELP COMMANDS SYNTAX ASIDS.**

When this operand is omitted, the dataspaces owned by the current target address space (either the home space or as established by the **SET ASID** command) will be displayed.

NAME=mask

Use this operand to limit the display to only selected dataspaces. **Mask** may contain **?** and * wildcard characters so that more than one dataspace can be selected for display. For detailed syntax, see **HELP COMMANDS SYNTAX PATTERNMATCH.**

The **mask** is not case sensitive.

Examples:

LIST DSPACES
All dataspaces owned by the current address space are displayed. Also, **ASTE#n** equates are automatically (re)generated for all **ASN Second Table Entries** that relate to those dataspaces.

LIST DSPACES JES2
All dataspaces owned by the JES2 address space are displayed. Also, **ASTE#n** equates are generated. (All prior **ASTE#n** equates, if any, are deleted.)

LIST DSPACES 1
All dataspaces owned by the Master Scheduler address space are displayed.

**LIST DSPACES 1 NAME=I*
All Master Scheduler owned dataspaces whose names start with the letter **I** are displayed. Also, **ASTE#n** equates are automatically (re)generated for all **ASN Second Table Entries** that relate to **all** Master Scheduler owned dataspaces (regardless of whether or not they were selected for display).

Help COmmands LISt DSTg

The **LIST DSTG** command can be used to display storage information from the dump currently being processed by dump/XDC.

Note that this command is only valid when dump/XDC is active.

Syntax:

LIST DSTG options ...
omitted

options is an optional list of keywords or Address Space IDs (ASIDs) that select the sections that will be displayed.

WAIT - Processing will wait, if required for dataset information to be available.

NOWAIT - Processing will not wait for dataset information to be available. If the information is not available yet, an error message is displayed.

COMMON - Information about available common storage ranges is displayed. All address spaces (but not data spaces) have the common storage ranges. This is a 'selection' option.

DSPACES - If an address space 'owns' data spaces, and at least one storage range has been dumped for a specific data space, this operand will cause the output to show that data space name. If no address space selection operands are present on the command, the 'DSOWN' operand is assumed to be specified.

NOSTORAGE - Causes the suppression of detailed information about storage ranges on the dump. Note that address space storage ranges are never shown for the 'DSOWN' operand.

ALL - Information about all address spaces on the dump is displayed. Note that the ASID displayed will be the lowest ASID on the dump, and the highest ASID displayed will be the highest ASID on the dump. This is a 'selection' option. Also, this is an 'address space selection' option.

SELECTED - Information about the address spaces selected by the dump/XDC address space selection option (or default) is is displayed. This is a 'selection' option. Also, this is an 'address space selection'

option.

EXTRA - Information about the address spaces that are not 'selected' but are the CXU's home or primary or secondary address space is displayed. This is a 'selection' option. Also, this is an 'address space selection' option.

SX - A combination of the 'SELECTED' and 'EXTRA' options. This is a 'selection' option. Also, this is an 'address space selection' option.

DSOWN - Information about address spaces that own 1 or more data spaces (on the dump) is displayed. Note that regardless of the use or otherwise of the 'NOSTORAGE' option no address space storage ranges are displayed. However, the display of owned data space storage ranges are under the control of the 'NOSTORAGE' operand. This is a 'selection' option. Also, this is an 'address space selection' option.

asid - information about the specific ASID is provided. A single ASID can be a unique job or STC name or TSO user name (any job name or started task name or TSO user name that is present on the dump can be used) or a hex or decimal value. Note that if a job (etc) name is the same as a keyword name, the keyword takes precedence. If this causes a problem, use the hex or decimal ASID instead. This is a 'selection' option. Also, this is an 'address space selection' option.

asid-asid - information about all ASIDs in the indicated ASID range is displayed. Note that the range is altered to start at the lowest ASID actually on the dump, and to end at the highest ASID actually on the dump. The ASID values used to indicate the start and end of a range can only be a hex or decimal number. This is a 'selection' option. Also, this is an 'address space selection' option.

If no 'selection' options are specified on the command (this includes the case where no options at all are specified on the command), the **COMMON** and **SX** options are assumed.

If no 'address space selection' option is specified on the command, but the 'DSPACES' keyword is specified, the 'DSOWN' address space selection option is assumed.

Examples:

LIST DSTG
common storage address ranges are displayed. Also, selected or extra address space information is displayed (including storage ranges).

LIST DSTG COMMON SX DSPACES NOSTORAGE
The number of common storage address ranges are displayed. Also, selected or extra address space information is displayed. If an address space 'owns' dataspaces information about them is displayed. Neither address space or data space information

will show storage ranges.

Help COmmands LISt DUB

The **LIST DUB** command displays the current setting of the **DUB** value.

Syntax:

LIST DUB

This command accepts no operands.

The DUB setting can be saved in your session profile. It can be changed by the SET DUB command. It also can be displayed and changed by Z/XDC's "Profile Menuing System". For more information, please see:

HELP COMMANDS SET DUB
HELP PROFILES MENU

Help COmmands LISt DUMp

The **LIST DUMP** command displays the current **SET DUMP** command settings. For more information, see HELP COMMANDS SET DUMP.

Syntax:

LIST DUMP

This command does not accept operands.

Help COmmands LISt DXdc

The **LIST DXDC** command can be used to display information about the dump currently being processed by dump/XDC.

Note that this command is only valid when dump/XDC is active.

Syntax:

LIST DXDC options ...

omitted

options is an optional list of keywords that select the sections that will be displayed.

ALL - All sections will be displayed. [This is the default when no section selection options are given.]
SUMMARY - Selected sections will be displayed.

ALLAS - If the address spaces section is displayed, forces the display of all address spaces available on the dump.
ALLCPU - If the CPUs section is displayed, forces the display of all defined CPUs.
ONLCPU - If the CPUs section is displayed, only online CPUs are displayed. (this is the default).
SELAS - If the address spaces section is displayed, only 'selected' address spaces (based on the setting of the AS... IPCS dump/XDC option). (also - any 'extra' address spaces are shown. 'extra' means that the address space is one of the CXU's home/primary/secondary address spaces and is not 'selected') (this is the default).

[NO]DI - The **Dump Information Section** will or will not be displayed.
[NO]OS - The **Operating System Section** will or will not be displayed.
[NO]CU - The **Current Execution Section** will or will not be displayed.
[NO]ASIDS - The **ASIDs Section** will or will not be displayed.
[NO]CPUS - The **CPUs Section** will or will not be displayed.
[NO]DATASET - The **Dump Dataset Section** will or will not be displayed.
[NO]ID - The **ID Section** will or will not be displayed.
[NO]PARMLIST - The **SDUMP Plist Section** will (if available) or will not be displayed.
[NO]SLIP - The **SLIP Section** (if available) will or will not be displayed.

Each section displays a category of information as documented below.

When **SUMMARY** is requested, the following sections are displayed:
- Dump Information Section
- Operating System Section
- Current Execution Section

dump/XDC options

Some of the information displayed may be affected by the dump/XDC options in effect. These options are set, in IPCS, before dump/XDC is started. The options that can be set, and how to set them, is described in **HELP DUMPS STARTINGDUMPXDC PROCESSINGOPTIONS**

Section Descriptions

LIST DXDC DI
The **Dump Information Section** displays general information about the dump. The following information is displayed:
- The dump title
- The dump DSN

- The original dump DSN
- The dump's IPCS type
- The dump's XDC type
- Whether or not the dump was taken on dump/XDC's current host system.

LIST DXDC OS

The **Operating System Section** displays information about the operating system for the dumped system. The following information is displayed:

- The Operating System name and version
- The system name [from CVTSNAME]
- The system's clone name [from ECVTCLON, distinguishes systems within a sysplex]
- The sysplex name
- the IPL local STCK date/time, the IPL local date, and the IPL local time

LIST DXDC CU

The **Current Execution Section** displays information about the execution aspace **(CAS)** or thread **(CXU)** that has been designated as "currently executing". See **HELP DUMPS EXECUTIONTHREAD** for a discussion of the **CAS** and **CXU.**

The following information is displayed:

- The type of currentness (none, CAS, or CXU).
- If the CAS or CXU has been chosen by the user, then for most dump types, a warning is displayed reporting this (just not for Stand-alone dumps).

  Using the **CU** shortcut on this line will reset things back to the choice made by dump/XDC.

- If there is a CAS set, the ASID (in hex) and the jobname are shown.
- If there is a CXU set, the CXU type (a task, SSRB, or SRB mode) will be shown.
- Depending on the CXU type, the TCB, and/or RB and/or SSRB addresses are displayed. The lines showing the addresses of these control blocks can be selected using the **D** and **F** shortcuts.
- If there is a current execution unit, the home ASID (and jobname), primary ASID, and secondary ASID are shown. Using the **L** shortcut in this line will cause a **LIST XMS** command to be issued.
- If there is a current execution unit, and it is in ABEND processing, then details about the ABEND are reported.
- If there is a current execution unit, and it is holding or waiting on any locks, then a warning message is shown. Using the **L** shortcut on this line will issue a **LIST LOCKS** command so that details can be seen.

LIST DXDC ASID

The **ASIDs Section** displays information about address spaces that dump/XDC allows you

to reference. (Yes, there may be multiple address spaces in the dump). The information reported will include:

- The ASID
- The job name
- The ASCB address
- The ASXB address
- An indication as to whether this address space is current or not (heading is **C?)**
- An indication as to whether this address space was 'selected' during dump/XDC IPCS processing (heading is **S?)**
- An indication as to whether this address space is 'extra' because it is the CXU's home or primary or secondary address space, and the address space was not 'selected' (heading is **X?)**

Note that if an address space is current (either has been nominated as the current address space or is the home address space of an execution unit), the line is hilighted. Also, the **CU** shortcut command can be used on any ASID line to set the current address space to the nominated one.

Note that which entries are on the address space list depends on the presence or absence of the **ALLAS** or **SELAS** operands.

Note that, in all cases, the number of address spaces shown on the section heading line is the number of address spaces available on the dump.

Note that making an address current or referencing an address space that, on the dump, does not have all the relevant control blocks present may result in some Z/XDC commands failing, indicating that required control blocks are missing.

LIST DXDC CPU

The **CPUs Section** displays information about CPUs that were available on the dumped system. The information displayed includes:

- The CPU ID, in hex
- The CPU's PCCA address
- The CPU's LCCA address
- The virtual address of the CPU's PSA (heading is **V@'PSA)**
- An indication as to whether the CPU is online (heading is **O)**
- An indication as to whether the CPU is a ZIIP (heading is **Z)**
- An indication as to whether the CPU is currently executing (heading is **X)**
- An indication as to whether the CPU is currently in SRB mode (heading is **S)**
- If executing, the ASID that it is executing in
- If executing, the TCB address if executing in task mode

Notes:

- Every CPU has its own, unique PSA. A particular PSA is mapped to location zero only when its owning CPU is actively executing and only when referenced from that particular CPU. But each PSA instance can be accessed via its unique virtual address regardless of whether or not its owning CPU is running or waiting.

- If a CPU is current, the line is hilighted. (by 'current' that means that there is a CXU, it is executing, and is executing on that CPU).

- If the dump type allows it, the **CU** shortcut command can be used on any executing CPU line to set the current execution unit to whatever thread the nominated CPU

is executing.

Note that which entries are on the CPU list depends on the presence or absence of the **ALLCPU** or **ONLCPU** operands.

Note that, in all cases, the number of CPUs shown on the section heading line is the value from CVTMAXMP (+1 because CPU address 0 is one of the CPU addresses) in the dump.

LIST DXDC DATASET

The **Dump Dataset Section** displays the following information about the dump dataset:

- Its dsname
- Its DCB attributes
- Its size in both physical blocks and logical records
- The length of time it took to load the dump
- The number of address spaces in the dump
- The number of common storage ranges on the dump
- The total number of storage ranges on the dump
- The number of data spaces in the dump. (This line will be reactive to an **L** shortcut.)

Notes:

- Some of the above reported information is not available via IPCS APIs, so dump/XDC has to read the dump dataset directly to obtain this information. It does so via an asynchronous subtask. If the Dataset Section report is requested before this subtask completes:
  - In some cases the reported information may not be complete.
  - In other cases, dump/XDC may wait for the subtask to complete before emitting the report (in which case, messages **DBC795I** and **DBC796I** will be issued to keep you informed).

- The following information cannot be obtained via IPCS APIs and require dump/XDC to read the dump directly:
  - The number of unique dataspaces actually in the dump,
  - Virtual storage ranges present in the dump. (This information is used to make the **FIND** command run a LOT faster!)

LIST DXDC ID

The **ID Section** displays dump identification information. This includes:

- The dump CPU id
- The dump timestamp, both in raw hex (STCK format), and formatted for human readability (with adjustments for leap seconds and timezone).

LIST DXDC PARMLIST

The **SDUMP Plist Section** is displayed only when both requested and available. For Operator Command dumps and for SDUMPs, this section shows the parameter list received by SDUMP from its caller. The plist is shown both interpreted and in raw hex.

LIST DXDC SLIP
For SLIP type dumps, the **SLIP Section** displays the following:
- The SLIP command text.
- For certain SLIP types, additional information is displayed:
  - For **PER SLIPs,** hardware PER information.
  - For **COMP SLIPs,** the ABEND code and and reason code (if any).
  - For **MSGID SLIPs,** the message text.
  - For **ERROR SLIPs,** currently no extra information is displayed.

Examples:

LIST DXDC SUMMARY
Summary information about the current dump will be displayed.

LIST DXDC
LIST DXDC ALL
All information about the current dump will be displayed. If the dump dataset read task has not completed, a message will indicate that the information is not yet available.

LIST DXDC ALL DATASET
All information about the current dump will be displayed. In addition, if the dump dataset read task has not completed, it will be waited for.

Help COmmands LISt EAR#

The **LIST EARn** command displays an individual error level access register. For more information, see **HELP COMMANDS LIST ACCESSREGISTERS.**

Help COmmands LISt EAREgs

The **LIST EAREGS** command displays the whole set of error level access registers. For more information, see HELP COMMANDS LIST ACCESSREGISTERS.

Help COmmands LISt EBea

The **LIST EBEA** command displays the error level breaking Event Address (BEA).

The **BEA** is a feature on newer IBM mainframes and more recent levels of z/OS. It records the location of the most recent branching instruction (or any other instruction that changes the execution flow) executed prior to an interrupt. It is a great help for debugging wild branches to unreasonable locations.

In order to have the **EBEA** available to the debugging session, the following conditions must be met:
- You must be running on a mainframe that has BEA support implemented. Mainframes starting with the z890 have this facility.
- You must be running with a level of z/OS that exploits the BEA hardware support. z/OS 1.7 and later have this exploitation.
- The error-level BEA is in the SDWA in the SDWARC4 section. This SDWA section must be present. (see the section in this help titled **Caution: SDWARC4 Required** for more information.)

@EBEA is a built-in equate that labels the EBEA location. **@EBEA** can be used in address expressions. **But note that the** @EBEA **equate only exists when the error level and retry level are different.**

Syntax:

LIST EBEA

This command has no operands.

The error level BEA is extracted from the error level information which is in the SDWA. (see HELP EXECUTIONLEVELS for information about retry level vs. error level environments.)

Related Commands:

- Use the **LIST BEA** command to display other BEA values.

Caution: SDWARC4 Required

Note that the BEA is in the SDWARC4 extension of the SDWA, and this extension only exists if the SDWA is above the 16m line (i.e. in 31 bit storage)

If the SDWARC4 extension does not exist, you will receive an error message informing you of this reason.

You can ensure that the SDWARC4 extension exists by:
- Using the ESTAEX or ATTACHX Assembler macros.
- Specifying SDWALOC31=YES on your ATTACH or ESTAE or FESTAE or SETFRR Assembler macros.

Caution: Aspace information not recorded

The hardware support for the **BEA** only records the address. It does not record the address space in which that address is located. Consequently, under certain conditions the location displayed by this **LIST EBEA** command might be in the wrong address space.

This situation can occur when the last prior flow-of-execution changing event (branching instruction, PC instruction, whatever) occurred in one address space while the interrupt that caused the BEA to be saved occurred in a different address space.

The best example of this occurs when a trap is placed at the start of a space switching PC routine. When Z/XDC receives control due to the trap, the execution address space will be that in which the PC routine resides. But the BEA location will be at the PC instruction located in the calling address space.

This issue can also arise with the **L EPSW** and **L EPSWE** commands.

Examples:

L EBEA
This displays the error level BEA.

Help COmmands LISt ENq

The LIST ENQ command can be used to display a list of system and user ENQs and also to display contention among ENQs. (Contention is defined as the situation where one task has access to a resource that prevents another task from getting access, thereby causing the latter task to go into a wait.)

Note that this command cannot be used under dump/XDC. Instead, the IPCS command, **VERBEXIT GRSTRACE** can be used.

This command produces output in **DETAILED** or **SUMMARY** mode.

In **DETAILED** mode, the following information is shown:

- **SYSID:** The system identifier of a task requesting an ENQ.
- **ASID:** The address space id number of a task requesting an ENQ.
- **JOBNAME:** The name of the job requesting an ENQ.
- **TYPE:** The type of the ENQ request, either shared **(S)** or exclusive **(E).**
- **QNAME:** The QNAME of the ENQ resource requested. This is, in general, the category of the resource, e.g. SYSDSN for data set access.
- **RANK:** The position of a task in the queue of requests.
- **RNAME:** The RNAME of the ENQ resource requested. This, in combination with the QNAME, defines the full name of the resource. The displayed RNAME is preceded by its length (in decimal).

In **SUMMARY** mode, the following information is shown:

- **#OWNERS:** The number of tasks which have ownership of the resource. For SHARED requests, more than one task can own the resource.
- **#WAITS-E:** The number of tasks which are waiting for EXCLUSIVE ownership of the resource.
- **#WAITS-S:** The number of tasks which are waiting for SHARED ownership of the resource.
- **QNAME:** The QNAME of the ENQ resource requested.
- **RNAME:** The RNAME of the ENQ resource requested.

Syntax:

```
LIST ENQ  qname  rname  JOBNAME=jobname  SYSNAME=sysid  ...
                        ASID=aspaceref

          ... SCOPE=ANY      CONTENTION  SUMMARY
                    STEP                 DETAILED
                    SYSPLEX
                    SYSTEM
                    SYSTEMS
```

WARNING: All strings are case-sensitive! Example: **list enq sys* *** will usually fail to produce any display at all. Instead, you would have to issue:
list enq SYS* *

All operands omitted
When no operands are given, information about all instances of ENQ contention on the local system is returned. Note, omitting all operands is equivalent to:
LIST ENQ * * CONTENTION.

qname
rname
When any operands are given, then these two operands are required. They are positional operands specifying the QNAME and RNAME for the ENQs for which information is to be displayed.

These operands are character strings which may designate either exact names or, via wildcards, ranges of names. Also, they may be either quoted or unquoted using tic marks (’). If a string contains colons (:), blanks ( ) or semicolons (;), then it must be quoted, and embedded quotes must be doubled up (’’).

All typeable characters are permitted. All characters are treated literally except wildcard characters: **?** matches any single character, and * matches any string of zero, one or more characters. (But there is a way to enter wildcard characters literally.) For more details, see **HELP COMMANDS SYNTAX PATTERNMATCH.**

ASID=aspaceref
This operand filters the report to include only those ENQ-requesting tasks located

within the specified address space. Briefly, the given value can be any of the following:

- A jobname.
- A keyword indicating the home space or the retry level or error level, primary, secondary, or instruction execution address space. Examples: HOME, PASID, ESASID, IASID, etc.
- An ASID number.
- An address expression pointing to an ASID number.
- A register containing an ASID number.
- The name of an equate or dsect that has been assigned to represent storage in an address space. (The assigned address space is used.)

For detailed information, see **HELP COMMANDS SYNTAX ASIDS.**

Note, This operand is mutually exclusive with the **JOBNAME=** operand.

JOBNAME=jobname
Like the ASID= operand, the JOBNAME= operand filters the report to include only those ENQ-requesting tasks located within the specified address spaces. The difference is, the JOBNAME= operand only accepts jobnames (not ASIDs or keywords).

For a **DETAILED** report, JOBNAME can make use of wildcard characters (* and ?) to produce a report that includes ENQs from multiple address spaces. For more details about wildcards, see **HELP COMMANDS SYNTAX PATTERNMATCH.**

Note: This operand is mutually exclusive with the **ASID=** operand.

SYSNAME=sysid
This operand filters the report to include only those ENQ-requesting tasks located on the specified system. When omitted, **all** systems are reported on.

The given value must specify the names of one or more systems connected in the current SYSPLEX. It can make use of wildcard characters (* and ?) to produce a report that includes ENQs from multiple systems. For complete details, see **HELP COMMANDS SYNTAX PATTERNMATCH.**

SCOPE=STEP
SCOPE=SYSTEM
SCOPE=SYSPLEX (or **=SYSTEMS)**
SCOPE=ANY
This operand filters the report according to each ENQ's scope:

- **STEP** - The ENQ's scope is limited to the address space within which the issuing task is running.
- **SYSTEM** - The ENQ's scope is limited to the system within which the issuing address space is running.
- **SYSPLEX** - The ENQ's scope extends across the entire SYSPLEX.
- **ANY** - The report is not filtered according to scope.

When omitted, SCOPE=ANY is used by default.

CONTENTION
When this operand is given, the report is limited to showing only those ENQs for which one or more requesting tasks are forced to wait for an owning task to release its ENQ on the resource.

The default output mode is DETAILED.

Note, issuing the LIST ENQ command without any operands at all is a shortcut way to

issue:
    LIST END * * CONTENTION.

SUMMARY
DETAILED
This specifies whether the produced report summarizes each ENQ or gives complete details. In a detailed report, when multiple tasks are requesting a resource, each of those tasks are listed individually. While in a summarized report, for each requested resource, only a count of the number of requesters is shown.

SUMMARY is the default output mode, except when the **CONTENTION** operand is specified. For CONTENTION, the default output mode is DETAILED.

Help COmmands LISt EPSW

The **LIST EPSW** command displays the error time PSW in its old **8-byte** format. Usually, this is the resolved PSW for the error level Request Block however, if the ABEND occurred within a type-1 or type-6 SVC, then this PSW is not found within any RB.

Related Commands:
- Use the **LIST EPSWE** command to display the error level PSW in its full **16-byte** format.
- Use the **LIST PSW** command to display the retry level PSW or other PSWs in their old **8-byte** format.
- Use the **LIST PSWE** command to display the retry level PSW or other PSWs in their full **16-byte** format.

Important! The PSW that is displayed is **not** necessarily the contents of either the **RBOPSW** field or the **XSBOPSW16** field (new in z/OS R1.13). Instead, it is a **resolved** PSW that is derived when RBOPSW and XSBOPSW16 **differ.** For more information, see HELP EXECUTIONLEVELS RBS RESUMEADDRESS.

Syntax:

```
    LIST EPSW FORMAT
              FMT
              omitted
```

FORMAT
FMT
This causes the PSW to be displayed with **all** of its flags and fields interpreted.

If **FORMAT** is **omitted,** then only the condition code, addressing mode and instruction address are interpreted.

When the **FORMAT** keyword is specified, the following additional information is displayed:
- The system mask is displayed and interpreted as follows:
  - **PER:** Program Event Recording
  - **DAT:** Dynamic Address Translation

    - **I/O:** I/O Interrupt status
    - **EXT:** External Interrupt status
- The Protection Key and Problem State flag are displayed.
- The Address Space Control mode (ASC-MODE) is displayed and interpreted as follows:
    - **PRIMARY**
    - **ACCESS REGISTER**
    - **SECONDARY**
    - **HOME**
- The current Condition Code is displayed and interpreted.
- The program mask is displayed and interpreted as follows:
    - **FIX:** Fixed-Point Overflow
    - **DEC:** Decimal Overflow
    - **EXP:** Exponent Underflow
    - **SIG:** Significance
- The addressing mode (24-bit, 31-bit or 64-bit) and Instruction Address are displayed and interpreted.

Examples:

L EPSW
This displays the **scrunched** (8-byte wide) version of the error level PSW. Example:

```
EPSW 078D1000 00069CFC (cc-LO) (24) - PRIVATE+068CFC
```

In the 8-byte format, the highest resume address that can be held is 2GIG-2 (i.e. below-the-bar). But if the error level address is actually above-the-bar, then the display is adjusted so that the true resume address is displayed anyway. Example:

```
EPSW  078D1001 [00000048_0000001E] (cc-LO) (64) - DBC810W NOT WITHIN ANY
IDENTIFIABLE MODULE OR ANY OTHER OBJECT
```

Note that the full 64-bit resume address is displayed even though it does not actually fit within the 8-byte wide format of the PSW! This violation of the 8-byte limit is indicated in the display by the open and close square brackets.

LIST EPSW,FORMAT
This produces a display of the error level PSW in an exploded format. Example:

```
     +---------------- PER(OFF)
     |+--------------- DAT(ON) I/O(ON) EXT(ON)
     ||+-------------- KEY(8)
     |||+------------- PROBLEM STATE
     ||||+------------ PRIMARY, CC=1 (MINUS, MIXED, LOW)
     |||||+----------- FIX(OFF) DEC(OFF) EXP(OFF) SIG(OFF)
     |||||| +-+------- AMODE(31)
     |||||| | |
EPSW 078D1000 80069CC2 - PRIVATE+068CC2
```

Help COmmands LISt EPSWE

The **LIST EPSWE** command displays the error time PSW in its full **16-byte** format. Usually, this is the resolved PSW for the error level Request Block however, if the ABEND occurred within a type-1 or type-6 SVC, then this PSW is not found within any RB.

Related Commands:

- Use the **LIST EPSW** command to display the error level PSW in its old **8-byte** format.
- Use the **LIST PSWE** command to display the retry level PSW or other PSWs in their full **16-byte** format.
- Use the **LIST PSW** command to display the retry level PSW or other PSWs in their old **8-byte** format.

Important! The PSW that is displayed is **not** necessarily the contents of either the **RBOPSW** field or the **XSBOPSW16** field (new in z/OS R1.13). Instead, it is a **resolved** PSW that is derived when RBOPSW and XSBOPSW16 **differ.** For more information, see HELP EXECUTIONLEVELS RBS RESUMEADDRESS.

Syntax:

```
LIST EPSWE FORMAT
           FMT
           omitted
```

FORMAT
FMT
This causes the PSW to be displayed with **all** of its flags and fields interpreted.

If **FORMAT** is **omitted,** then only the condition code, addressing mode and instruction address are interpreted.

When the **FORMAT** keyword is specified, the following additional information is displayed:

- The system mask is displayed and interpreted as follows:
  - **PER:** Program Event Recording
  - **DAT:** Dynamic Address Translation
  - **I/O:** I/O Interrupt status
  - **EXT:** External Interrupt status
- The Protection Key and Problem State flag are displayed.
- The Address Space Control mode (ASC-MODE) is displayed and interpreted as follows:
  - **PRIMARY**
  - **ACCESS REGISTER**
  - **SECONDARY**
  - **HOME**
- The current Condition Code is displayed and interpreted.
- The program mask is displayed and interpreted as follows:
  - **FIX:** Fixed-Point Overflow
  - **DEC:** Decimal Overflow
  - **EXP:** Exponent Underflow
  - **SIG:** Significance
- The addressing mode (24-bit, 31-bit or 64-bit) and Instruction Address are displayed and interpreted.

Examples:

L EPSWE
This displays the **16-byte** version of the error level PSW. Example:

```
    EPSWE 47851001 80000000 00000000_000858B0 (cc-LO) (64) - .DIE9106Z
```

LIST EPSWE,FORMAT
This produces a display of the error level PSW in an exploded format. Example:

```
      +----------------------- PER(ON)
      |+---------------------- DAT(ON) I/O(ON) EXT(ON)
      ||+--------------------- KEY(8)
      |||+-------------------- PROBLEM STATE
      ||||+------------------- ASC-MODE(PRIMARY), CC=1 (MINUS, MIXED, LOW)
      |||||+------------------ FIX(OFF) DEC(OFF) EXP(OFF) SIG(OFF)
      |||||| +-+-------------- AMODE(31)
      |||||| | |
EPSWE 47851000 80000000 00000000_000854D0 - XDCTESTS+0
```

Help COmmands LISt EQuates

The **LIST EQUATES** command produces a report showing comprehensive information about selected **equates.** The information displayed includes:

- The name of the equate.
- It's current location (if resolvable).
- The length (if any) of storage represented by the equate. Note, the length is shown both in hex and in scaled decimal.
- The location's defining address expression (for floating equates)
- The equate's formatting bias **(D)** or **(I),** if any.

The display can report either all defined equates or just a selection, depending upon the operands provided to the **LIST EQUATES** command.

The equates are displayed in an **alpha-numeric** sequence. Specifically, when an equate's name ends with digits, then:

- The equates are sorted first by **root** name (ie. that part of the name preceding the final digits),

- Then by the numeric value of the trailing digits,

- And finally by the entire name taken as a text string.

Notes:

- The third sort criteria comes into play only when the trailing digits of two

equates have identical numeric values. This can happen, for example, when one of the sequence numbers is given with leading zeros, while the other is not.

- This LIST EQUATES command is the **only** command that considers and make use of equate sequence numbers as binary values. All other commands always treat the entire equate name as a text string.

- It is never expected that two equates will exist that have identical names.

Additional information can be found by typing:

HELP EQUATES
HELP COMMANDS EQUATE

Syntax:

```
LIST EQUATES mask1 mask2 ...
             omitted
```

Operands:

mask1
mask2
...
Any number of selection masks may be given to select the particular equates to be reported. The masks:

- Are not case sensitive (because equate names are not case sensitive either).
- May contain **?** and ∗ wildcard characters. (For detailed syntax, see **HELP COMMANDS SYNTAX PATTERNMATCH.)**
- Are additive. When multiple masks are given, the display will report all equates that are selected by one or more masks.)

omitted
When no operands are given, **all** defined equates are reported.

Help COmmands LISt ER#

The **LIST ERn** command displays the lo-order half (4 bytes) of an individual error level general register. For more information, see HELP COMMANDS LIST GENERALREGISTERS.

Help COmmands LISt EREgs

The **LIST EREGS** command displays the lo-order halves (4 bytes) of the whole set of error level general registers. For more information, see HELP COMMANDS LIST GENERALREGISTERS.

Help COmmands LISt ERH#

The **LIST ERHn** command displays the hi-order half (4 bytes) of an individual error level general register. For more information, see HELP COMMANDS LIST GENERALREGISTERS.

Help COmmands LISt ERHRegs

The **LIST ERHREGS** command displays the hi-order halves (4 bytes) of the whole set of error level general registers. For more information, see HELP COMMANDS LIST GENERALREGISTERS.

Help COmmands LISt ERW#

The **LIST ERWn** command displays the entire 8 bytes of an individual error level general register. For more information, see HELP COMMANDS LIST GENERALREGISTERS.

Help COmmands LISt ERWRegs

The **LIST ERWREGS** command displays the entire 8 bytes of the whole set of error level general registers. For more information, see HELP COMMANDS LIST GENERALREGISTERS.

Help COmmands LISt ESTaes

The **LIST ESTAES** command can be used to display the STAE Control Blocks (SCBs) that are queued for any task in any accessible address space. The SCB is the z/OS system control block that defines the existence of an ABEND recovery routine (ESTAE, ESTAEX, ESTAI, ARR, etc.). In other words, when a program issues an ESTAE macro (for example), the System creates an SCB and records into it information from the ESTAE macro and the macro-time environment. It then queues the SCB to the task under which the ESTAE macro was issued. Accordingly, an understanding of information available from the SCB queue is very important to the understanding of:

- The processing of ABENDs by your program when they occur.
- The integration of Z/XDC into your ABEND recovery structure.

The display produced by the LIST ESTAES command presents SCB queue information in a quickly understandable form. The format and contents of SCBs are documented in the following IBM manuals:

MVS/XA: Debugging Handbook Data Areas vol. 5: R-S (LC28-1168)
MVS/ESA 3: Data Areas vol. 4: RAB-SRRA (LY28-1046)

```
MVS/ESA 4: Data Areas vol. 4: RCWJ-SRRA (LY28-1824)
MVS/ESA 5: Data Areas vol. 4: RD-SRRA (LY28-1860)
OS/390:    Data Areas vol. 4: RD-SRRA (SY28-1167)
z/OS:      Data Areas vol. 4: RD-SRRA (GA22-7584)
```

In addition the **LIST ESTAES** command numbers the SCBs from oldest to newest. These numbers are then referenced by various messages from Z/XDC.

When SCBs queued from the current task are displayed, the newest one listed will have the label **(ARTIFACT)** appended to it, indicating that it is an "artifact" of the debugging session. This particular SCB describes an ESTAEX that Z/XDC has issued internally, establishing an ABEND recovery routine for its own, internal protection. Whenever Z/XDC relinquishes execution (i.e. whenever the user issues a TRACE, GO or END command), this ESTAEX is purged. Therefore, this artifact SCB will never be seen by the user's program. Therefore, this artifact SCB can be disregarded.

When Z/XDC is executing as a TRAP handler, there will still be an SCB list, and an ESTAEX will still be established by Z/XDC.

Syntax:

```
LIST ESTAES tcbaddress
     SCBS   omitted
```

tcbaddress

This must be the address of a TCB located in any accessible address space. (See HELP VIRTMEM XDCACCESS for information about accessing storage located in other address spaces.) This operand identifies the task whose SCB queue is to be displayed. If this operand is omitted, then the "current" TCB located in the user program's Home Address Space may be used as the default.

Automatic Equates

The **LIST ESTAES** command (re)generates all of the **SCB#n** equates that label the displayed STAE Control Blocks. For more information, see HELP EQUATES BUILTIN AUTOMATIC.

Notice: the **SCB#n** equates are **static!** They do not float. The locations that they target can change only when another **LIST SCBS** command is issued. In other words, when the **LIST SCBS** command is used to display the STAE Control Block queue for any task other than the current task, the **SCB#n** equates can become obsolete without warning simply due to the ongoing execution of the task whose SCBs are being displayed.

Examples:

For a cool example of using **autocloned maps** to label entire SCB chains for any task, see the examples in HELP COMMANDS USING AUTOCLONING.

L SCB
If Z/XDC is not running in Foreign Address Space Mode, then the SCBs that are queued to the Home Address Space's current task are displayed. (The "current" task is the task under which Z/XDC is executing.) On the other hand, if Z/XDC **is** running in Foreign Address Space Mode, then there is no such thing as a "current" task, and so this command will fail.

L TASKS
L SCB TCB#5
The LIST TASKS command not only displays the address space's subtask tree, but also it automatically generates a series of numbered equates named "TCB#n" that label the displayed TCBs. The "LIST SCB TCB#5" command then displays the SCBs that are queued from TCB number 5.

L SCB 21C%
DM XDCMAPS.SCB
U SCB SCB#1
F SCB
This series of commands does the following:
- The **LIST SCB 21C%** command displays the SCBs that are queued to the current task (just like the "LIST SCB" command given without operands). The command also automatically generates a series of equates named "SCB#n" that label the SCBs from which the display was generated.
- The **DMAP XDCMAPS.SCB** command reads a dsect map for the fields contained within SCBs.
- The **USING SCB SCB#1** command assigns the SCB dsect map to format the storage containing the oldest SCB queued to the current task.
- The **FORMAT SCB** displays that SCB field by field.

You can refer to the appropriate "Data Areas" manual from IBM for detailed descriptions of the field contents.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP ∗NEXT (F11)** to proceed sequentially. Use **HELP ∗FORWARD (F5)** to skip.

REPORT - A description of the report produced by LIST ESTAES.

Help COmmands LISt ESTaes Report

Below is a typical report produced by the LIST ESTAES command:

```
                                   XDC    FLAGSn
SCB# TYPE   OWNER   STATUS         REL.    1, 2, 3  EXIT ADDRESS
1    ESTAI  TCB#7   PENDING              96,80,00  IKJEFT04+02C
2    ESTAI  TCB#8   PENDING              96,80,40  ISPMAIN.X#1+021370
3    ESTAI  TCB#9   PENDING              96,80,00  ISPSUBX+0E0
```

```
4    ESTAI  TCB#10  PENDING       v3.1  96,80,48  XDCCALL+0EC4
5    ESTAI  TCB#11  ACTIVE-RB#3   v3.1  96,90,40  V31CALL+0ECC
6    ESTAE  RB#3    PENDING             16,80,01  V31.X#1+01AF30 (ARTIFACT)
```

Each SCB defines an ABEND error recovery exit that might receive control upon the occurrence of an ABEND. This report lists the SCBs from oldest (SCB#1) to newest. If an ABEND occurs, then the System will pass control to the newest SCB whose status is "PENDING".

You might notice in the above example that SCB#6 is both "pending" and newer than the "active" SCB (SCB#5). From this fact, you can infer two things:

- At the time the ABEND occurred for which SCB#5 was activated, SCB#6 did not exist.
- SCB#6 was created when the exit routine pointed to by SCB#5 issued its own ESTAE macro in order to create its own ABEND protection environment.

In any case, the LIST ESTAES report provides the following information:

SCB#
This field assigns sequence numbers to the SCBs being displayed. The oldest SCB is assigned sequence number 1. Also, equates of the form "SCB#n" are automatically created to label the storage containing the SCBs.

TYPE
This field shows the type of ABEND recovery routine that the SCB represents. Possible values are:

```
    ESTAEX  FESTAE  ESTAE  STAE
    ESTAIX  ARR     ESTAI  STAI
```

Discussions of the various types of recovery routines supported by z/OS can be found in the following IBM manuals:

- OS/390 MVS Programming: Assembler Services Guide (GC28-1762)
- OS/390 MVS Programming: Authorized Assembler Services Guide (GC28-1763)
- z/OS MVS Programming: Assembler Services Guide (GC28-1762)
- z/OS MVS Programming: Authorized Assembler Services Guide (SA22-7608)

OWNER
This field identifies who created the recovery routine, i.e. who issued the ESTAE macro or the "ATTACH ESTAI=" macro. Note:

- ESTAI-type routines (STAI, ESTAI, ESTAIX) are created when a parent task attaches a subtask, so they are owned by the parent task. Also, z/OS automatically propagates ESTAI-type routines to all descendant tasks. This is why multiple ESTAI-type routines can exist in protection of any given task.
- ESTAE-type routines (STAE, ESTAE, ESTAEX, FESTAE) are created when a program issues an ESTAE-type macro. For these routines, the "owner" is the Request Block (RB) under which a program was executing at the time that it issued the ESTAE-type macro.

STATUS
This field can display the following values:

PENDING
This means that the SCB is not currently processing any ABEND. If a new ABEND occurs, then z/OS will pass control to the newest pending SCB. If an active SCB "percolates" the ABEND that it is processing, then the newest pending SCB that is older than the active SCB will then get its turn to process the ABEND. In the above example, these rules have the following effects:

- If the exit that is currently active as SCB#5 should itself ABEND, then

SCB#6 will receive control.
- On the other hand, if SCB#5 should percolate the ABEND that it is processing, then SCB#4 will receive control.
- Finally, if SCB#6 should "recover" the ABEND, then no more SCBs will receive control. Instead, the routine that had ABENDed will resume execution.

ACTIVE-RB#nnn

This means that the SCB is currently active and processing an ABEND. The SCB's ABEND exit routine is running under the control of RB#nnn. The LIST RBS command can be used to find more information about the ABEND exit routine's execution progress.

Sxxx-RB#nnn
Unnnn-RB#nnn

These mean that the SCB's ABEND exit routine had been active and processing an ABEND when the routine itself ABEND (i.e. a nested ABEND has occurred). "Sxxx" or "Unnnn" is the System or user ABEND code that the exit routine suffered. Use the LIST RBS command to find more information.

PERCOLATED
This means that the SCB's ABEND exit routine has finished executing and has decided to let the ABEND process proceed. As a result, z/OS has passed control to the next older pending SCB (which now, therefore, will be "active") to see what it wants to do about the ABEND.

DUMMY
This means that the SCB has been deactivated and so will not receive control for any ABEND.

XDC REL.
If the ABEND exit routine pointed to by the SCB is Z/XDC, then this displays which release of Z/XDC it is. If the ABEND exit might call Z/XDC, then this field will be set to "UNKN". Note, a user written ABEND exit routine can indicate that it has an interface to Z/XDC by preceding its entry point with the string "CALLSXDC".

FLAGSn
1, 2, 3
These three fields show the hexadecimal contents of three flag bytes from the SCB: SCBFLGS1, SCBFLGS2, and SCBFLGS3. The meanings of these flags are documented in the following IBM manuals:

```
MVS/XA:    Debugging Handbook Data Areas vol. 5: R-S (LC28-1168)
MVS/ESA 3: Data Areas vol. 4: RAB-SRRA (LY28-1046)
MVS/ESA 4: Data Areas vol. 4: RCWJ-SRRA (LY28-1824)
MVS/ESA 5: Data Areas vol. 4: RD-SRRA (LY28-1860)
OS/390:    Data Areas vol. 4: RD-SRRA (SY28-1167)
z/OS:      Data Areas vol. 4: RD-SRRA (GA22-7584)
```

EXIT ADDRESS
This displays the ABEND exit routine's entry address.

Help COmmands LISt EXits

The **LIST EXITS** command displays the current **SET EXITS** command settings. For more information, see HELP COMMANDS SET EXITS.

Syntax:

LIST EXITS

This command does not accept operands.

Help COmmands LISt FEatures

The **LIST FEATURES** command displays some hardware and software features used by Z/XDC. Some functions of Z/XDC depend on the presence of hardware or software features. For example, zapping store protected storage requires the Suppression-On-Protection Facility (SOPF) and the virtual services facility (IARVSERV).

This command organizes hardware and system features into two broad categories: **ESA/390** related and **z/Architecture** related. Command operands allow you to select which category(ies) of features are displayed.

Syntax:

```
    LIST FEATURES INSTALLED     ZOS
                  NOTINSTALLED  ZARCHITECTURE
                                omitted

                                OS390
                                ESA390
                                OARCHITECTURE

                                ALL
```

INSTALLED
NOTINSTALLED
Z/XDC can detect somewhere around 60 or so hardware and software features of the current operating system and the machine on which it is running. These two operands control whether Z/XDC displays the features that **were** found or that **were not** found. The default is **INSTALLED.**

ZOS
ZARCHITECTURE (alias of ZOS)
omitted
This operand restricts the display to only those features that are new since the introduction of z/Architecture hardware and the z/OS Operating System. (This is the default when system selection operands are omitted from the command.)

OS390
ESA390 (alias of OS390)
OARCHITECTURE (alias of OS390)
This operand restricts the display to only those features that were introduced in older versions of the hardware and Operating System.

ALL
This operand displays all system and hardware features regardless of when they were introduced.

Subtopics Index

Select the following topics for detailed information, regarding the features that Z/XDC detects. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

ZOS - Detailed descriptions of z/Architecture and z/OS features that can be displayed.
OS390 - Detailed descriptions of older features that can be displayed.

Help COmmands LISt FEatures Zos

The **LIST FEATURES ZOS** command can detect and display the following hardware and software features (and **bugs)** that are new since the introduction of z/Architecture and the z/OS Operating System:

ABOVE-BAR-X - Above the Bar Execution
z/OS R1.13 has limited support for executing programs located in 64-bit storage. Enabled programs (enabled for interrupts) may run above-the-bar, and page faults will be handled. But programs may not invoke any System Services. They may not issue SVCs, they may not call System owned PC routines, and they may not call I/O access methods, branch-entry routines, and the like.

ALRF - ASN and LX Reuse Facility (hardware)
This provides hardware support for the reuse of Address Space Numbers (ASNs) under circumstances where previously they could not be reused. It also provides for a restructuring of PC numbers to increase the number of PC routines possible. ALRF is documented in:
- z/OS: Principles of Operation (SA22-7832-03+)

ALRF-zOS - ASN and LX Reuse Facility support (in z/OS)
Starting with z/OS R1.6, several operating system control blocks and other structures were changed in order to support ARLF. The changes are not compatible with prior releases of the operating system. For example, when ALRF is present, all 8 bytes of control registers 3 and 4 are used, not just 4. So while in prior releases, z/OS needed to save only 4 bytes of CR3 and CR4, with the advent of ARLF, all 8 bytes of each register now need to be saved. So for the control blocks that have CR3/CR4 save areas, IBM generally created new 8-byte wide fields and abandoned the older 4-byte wide fields. Thus, programs that relied on the older fields now see nothing at all. (The XSB is an example of such a control block.) Note that these operating system control block changes, when present, will be present **regardless** of whether or not ALRF is actually installed in the hardware.

BEA - Breaking Event Address Support
BEA is a hardware feature that records the address of the branching, SVC, PC, LPSW or any other instruction that most recently changed the flow of execution prior to

an ABEND. It is a great help for debugging wild branches to unreasonable locations.

BIGPSW - 128-Bit PSW Support
z/OS R1.13 has support for saving and restoring 128-bit wide PSWs. New fields have been added to the TCBE, the XRB, the SDWA etc. so that z/OS can save and restore status when 64-bit execution is interrupted.

DFP - Decimal Floating Point
DFPHP - High Performance Decimal Floating Point
The Decimal Floating Point feature consists of a large number of machine instructions that interpret floating point data in a new decimal floating point format and that perform floating point arithmetic on that data. "High Performance" DFP is a version of DFP that performs considerably more efficiently.

DFP and high performance DFP are documented in Principles of Operation (SA22-7832).

ECTG - Extract CPU Time instruction
The ECTG instruction was introduced with the **z9 Processor.** It provides a very fast (and easy) way for a problem state program to keep track of the amount of CPU time consumed by a task.

ESTAILMT - ESTAI Retry levels Limitation
This is a system bug! Normally, when an ABEND error recovery routine (ESTAE etc.) itself ABENDs, and this second ABEND causes an ESTAI routine to receive control, if the ESTAI routine successfully recovers from the ABEND, it can cause the ABENDed ESTAE routine to resume execution. However, briefly in z/OS 1.2, z/OS 1.3, and z/OS 1.4, the Recovery/Termination Manager (RTM) was changed such that ESTAIs could **not** recover ABENDs that occurred within recovery routines. Eventually, this problem was fixed, by the following PTFs:
- z/OS 1.2: UA01639
- z/OS 1.3: UA01640
- z/OS 1.4: UA01641

This "feature" is actually a bug. When this "feature" is present, then when Z/XDC is running as an ESTAI, you will not be able to debug ABENDs that occur within ESTAE(X) routines. This problem can be fixed by APPLY'ing one of the above PTFs from IBM. For more documentation, see the applicable PTF.

EX-IMM - Extended Immediate instructions
The extended immediate machine instructions are available. These consist of several instructions that load, add, compare, etc. a two-byte immediate field from the instruction itself into a register. These instructions are documented in:
- z/OS: Principles of Operation (SA22-7832-04+)

GCOMMON - 64-bit Common Area
Support for an above-the-bar **Common Area** (a GCSA, if you will) is present.

GSYSAREA - 64-bit System Area
Support for an above-the-bar **System Area** is present.

HFP-MA/S - Hexadecimal Floating Point, MULTIPLY+ADD/SUBTRACT instructions
This provides multiply-and-add and multiply-and-subtract instructions for improving the performance of common hexadecimal floating point operations. This facility is available on z990 (and newer) systems. It is documented in:
- z/OS: Principles of Operation (SA22-7832-02+)

HFP-UNNX - Hexadecimal Floating Point, Unnormalized extension
Hexadecimal floating point ADD and SUBTRACT instructions are available that produce

unnormalized results. For more information, see:
- z/OS: Principles of Operation (SA22-7832-02+)

IDBPT - Improved Deferred Breakpoint Support (CSW)
IDBPT - Improved Deferred Breakpoint Support (IBM)
This is a change to IBM's Contents Supervisor logic that enhances deferred breakpoint support by causing targeted reentrant load modules to be loaded into a storage subpool selected according to normal MVS rules (instead of according to TSO-TEST rules). Starting in z/OS R1.6, this change is supported by IBM. Prior to z/OS R1.6, this change was implemented by Z/XDC's internal logic.

IDTE - IDTE instruction, basic (purge TLB only)
IDTE-ASCE-S - IDTE instruction, clearing by ASCE at the Segment Table level
IDTE-ASCE-R - IDTE instruction, clearing by ASCE at the Region Table level
The Invalidate DAT Table Entry instruction may be installed in your system's hardware at any of three different levels of capability. This instruction is available on z990 (and newer) systems. The IDTE instruction is documented in:
- z/OS: Principles of Operation (SA22-7832-02+)

JAVAWORK - JAVA Work Area
Support for a **JAVA Work Area** is present. (Note, the JAVA Work Area starts at the 2G line, thus replacing [and going beyond] the old DEADZONE.)

LDF - Long Displacement Facility
LDF-HI - Long Displacement Facility, high performance version
This implements a 20-bit, signed displacement field in 69 existing and 44 new machine instructions. This facility is available on z990 (and newer) systems. LDF is documented in:
- z/OS: Principles of Operation (SA22-7832-02+)

LOSTLOCKSI - Lost Locks Information available in SDWA even for ESTAE-type recovery
Starting in z/OS R1.12, The Recovery Termination Manager started recording information in the SDWA about locks that were held (and are now lost) at the time of error. Previously, this was done only for FRR-type recovery routines. Beginning in z/OS R1.12, this is now done for ESTAE-type recovery routines as well.

The SDWA fields involved are SDWAHLHI, SDWASUPR, SDWASPN and SDWACLSE.

LPDEL - LPDE LENGTH: nnn BYTES
When IBM implemented RMODE64 support (in z/OS R2.3), the size of **LPDEs** (Link Pack Directory Entries) had to be increased. This message shows its size, in decimal, in the current System. (Prior to R2.3, it was 40 bytes. Starting with R2.3, it was increased to 48 bytes.)

LPTEA - Load Page Table Entry Address
This is a machine instruction for finding the address of a page table entry for a given page of storage. This facility is available on z9 (and newer) systems. LPTEA is documented in:
- z/OS: Principles of Operation (SA22-7832-04+)

MIXCPSWD - Mixed-Case Password Support in RACF (status)
This support permits logon passwords to include a mixture of uppercase and lowercase alphabetics. The support was introduced by IBM in R1.7 of z/OS.

MSA - Message Security Assist
This consists of a series of machine instructions for encrypting and decrypting data using any of several encryption algorithms. This facility is available on z990 (and

newer) systems. MSA is documented in:
- z/OS: Principles of Operation (SA22-7832-02+)

REFRPROT - Load REFR modules into key 0 storage (status)
Starting with z/OS R1.9, system support is available for loading modules, having the **refreshable** attribute (REFR), into fetchable key 0 storage (instead of fetch protected user key storage). This feature can be turned on or left off system wide according to PARMLIB settings. For more information, see:
- z/OS MVS Initialization and Tuning Reference (SA22-7592, -15 and newer)

RMODE64 - Above the Bar load modules
Starting with z/OS R2.3, System support became available for loading load modules into storage located Above The Bar.

SSRX - SSRB/SSRX Splitup
Starting in z/OS R1.13 the SSRB has been split into two smaller control blocks:
- A 64-bit storage resident **SSRX**
- A 31-bit storage resident **SSRB** remnant

The 31-bit resident SSRB had been getting quite big, so this split-off provides 31-bit Virtual Storage Constraint Relief (VSCR). Note, most of the fields in the remnant SSRB have been rearranged, so an SSRB map produced by an **IHASSRB** macro from a pre-R1.13 release of z/OS is **entirely invalid** for R1.13 and going forward!

STCKF - Store Clock Fast
This is a machine instruction for storing the current value of the system clock without insuring unique values stored by two consecutive Store Clocks. This facility is available on z9 (and newer) systems. STCKF is documented in:
- z/OS: Principles of Operation (SA22-7832-04+)

STFLE - Store Facilities List Extended
This is a machine instruction for storing the current hardware's list of facilities into as much storage as would be needed to hold all the information. In addition, the STFLE instruction is a problem state instruction. (The older STFL was a supervisor state instruction.) This facility is available on z9 (and newer) systems. STFLE is documented in:
- z/OS: Principles of Operation (SA22-7832-04+)

XTR2 - Extended Translate Facility 2
This is a set of extended translation machine instructions useful for manipulating double-byte data, ASCII characters, decimal data, and Unicode data. This facility is documented in:
- z/OS: Principles of Operation (SA22-7832)

XTR3 - Extended Translate Facility 3
This is a set of instructions that perform operations on Unicode and Unicode-transformation-format (UTF) characters. It also includes a right-to-left TRANSLATE AND TEST instruction. This facility is documented in:
- z/OS: Principles of Operation (SA22-7832-03+)

z/ARCH - 64-bit Hardware
z/ARCH - 64-bit Hardware Installed but Not In Use
z/ARCH - 64-bit Hardware Installed and In Use
The current hardware has the machine instructions and general architecture to support 64-bit addressing. z/Architecture is documented in:
- z/OS: Principles of Operation (SA22-7832)

Help COmmands LISt FEatures Os390

The **LIST FEATURES OS390** command can detect and display the following hardware and software features that are older than z/Architecture and z/OS:

BFP - Binary Floating Point
When BFP is installed, the number of floating point registers is increased from 4 to 16, and IEEE's binary floating point format is recognized (in addition to the hexadecimal floating point format that IBM has supported for decades). BFP is documented in:
- OS/390: Principles of Operation (SA22-7201)
- z/OS: Principles of Operation (SA22-7832)

BFP - Binary Floating Point (emulated)
Support for binary floating point numbers is only simulated by the Operating System. It is not actually recognized by the hardware.

BSA - Branch and Set Authority Instruction
The BSA (branch and set authority) machine instruction can be used by a supervisor state routine to call a problem state subroutine and by the problem state subroutine to return control to the supervisor state routine. It is documented in:
- OS/390: Principles of Operation (SA22-7201)
- z/OS: Principles of Operation (SA22-7832)

BSG - Subspace Facility
When a large subsystem needs to create additional address spaces, those additional spaces can be designated as a subspace group owned by the originating address space. The BSG instruction then facilitates branching amongst the various subroutines within the subgroup. The BSG instruction and subspace groups in general are documented in:
- OS/390: Principles of Operation (SA22-7201)
- z/OS: Principles of Operation (SA22-7832)

CADS - SCOPE=COMMON Data Spaces
These are data spaces that can be shared across address spaces. They are documented in:
- OS/390: "MVS Programming: Extended Addressability Guide" (GC28-1769)
- z/OS: "MVS Programming: Extended Addressability Guide" (SA22-7614)

CKSM - Checksum Facility
The CKSM (checksum) machine instruction is documented in:
- OS/390: Principles of Operation (SA22-7201)
- z/OS: Principles of Operation (SA22-7832)

CSRL16J - Load-16-and-Jump Callable Service
The "Load 16 and Jump" callable service (CSRL16J) provides a way in which one routine can call another with all 16 general registers set to arbitrary values without needing to use any register as a base for the jump. This service was added to the operating system at MVS/ESA 4.3. It is documented in:
- OS/390: "MVS Programming: Assembler Services Reference" (GC28-1910)
- z/OS: "MVS Programming: Assembler Services Reference, Volume 1 (ABEND-HSPSERV)" (SA22-7606)

CSTR - C String Facility
This is a set of machine instructions designed to facilitate the C programming language. The instructions are:
- CLST: Compare logical string
- MVST: Move string
- SRST: Search string

They are documented in:
- OS/390: Principles of Operation (SA22-7201)
- z/OS: Principles of Operation (SA22-7832)

CUSE - Compare Until String Equal Instruction
CUSE is a machine instruction for finding one substring within another. It is documented in:
- OS/390: Principles of Operation (SA22-7201)
- z/OS: Principles of Operation (SA22-7832)

DLPA - Dynamic Link Pack Area
This service provides a formal way to add (or delete) modules to the System's Link Pack Area (functionally similar to the PLPA) without requiring an IPL. This service is used by the SETPROG Operator command and the CSVDYLPA macro. It was added to the Operating System at OS/390 R2.4.

IARVSERV - Virtual Services
The IARVSERV macro permits programs to define private areas of storage to be shared in two or more address spaces. This service was added at MVS/ESA 5.2.2. It is documented in:
- OS/390: "MVS Programming: Assembler Services Reference" (GC28-1910)
- z/OS: "MVS Programming: Assembler Services Reference, Volume 2 (IARR2V-XCTLX)" (SA22-7607)

IEAMSCHD - Partial Function SRB Scheduling
IEAMSCHD - Full Function SRB Scheduling
The IEAMSCHD macro provides a simplified way (as compared to the SCHEDULE macro) to schedule SRB routines. This service was added at MVS/ESA 5.2.2 and then improved in MVS/ESA 5.3. It is documented in:
- OS/390: "MVS Programming: Authorized Assembler Services Reference, Volume 2 (ENFREQ-IXGWRITE)" (GC28-1765)
- z/OS: "MVS Programming: Authorized Assembler Services Reference, Volume 2 (ENFREQ-IXGWRITE)" (SA22-7610)

IEANTASM - Name Token Services
This service provides a method by which arbitrary information can be communicated across address spaces. This service was added to the operating system at MVS/ESA 4.3. It is documented in:
- OS/390: "MVS Programming: Assembler Services Guide (GC28-1762)
- z/OS: "MVS Programming: Assembler Services Guide (SA22-7605)

ISGLMASM - Latch Manager
This service provides an efficient serialization mechanism available to authorized programs. This service was added to the operating system at MVS/ESA 4.3. It is documented in:
- OS/390: "MVS Programming: Authorized Assembler Services Guide" (GC28-1763)
- z/OS: "MVS Programming: Authorized Assembler Services Guide" (SA22-7608)

KEY9 - Key 9 Storage is Unprotected
When this feature is installed, key 9 storage is not store protected. This means that any program, regardless of authorization, and regardless of execution key, can

store into key 9 storage. This feature is known as "Storage Protection Override Control", and it is documented in:
- OS/390: Principles of Operation (SA22-7201)
- z/OS: Principles of Operation (SA22-7832)

MVPG - Move Page Facility
The MVPG machine instruction (move page) is used by the Operating System primarily to move pages of data into and out of expanded storage. It is documented in:
- OS/390: Principles of Operation (SA22-7201)
- z/OS: Principles of Operation (SA22-7832)

N3 - z/ARCH Instructions Retrofitted to ESA/390
The "N3" machine instructions are a group of instructions that were introduced with z/Architecture but that do not have anything to do with 64-bit addressing. So these instructions have been retrofitted to some ESA/Architecture computers. These instructions are documented in Appendix B of:
- OS/390: Principles of Operation (SA22-7201)
- z/OS: Principles of Operation (SA22-7832)

PGSER - Store Protection
This service allows a program to designate private area pages as "store protected". This means that attempts to store into them will fail regardless of whether or not the storing program’s execution key matches the protected page’s storage key. This service was added to the operating system at MVS/ESA 4.3. It is documented in:
- OS/390: "MVS Programming: Assembler Services Reference" (GC28-1910)
- z/OS: "MVS Programming: Assembler Services Reference, Volume 2 (IARR2V-XCTLX)" (SA22-7607)

PLO - Perform Locked Operation
The Perform Locked Operation machine instruction is documented in:
- OS/390: Principles of Operation (SA22-7201)
- z/OS: Principles of Operation (SA22-7832)

PRCC - Condition Code Set From State Entry
On some machines, the PR machine instruction (program return) sets the condition code to the value that was in the PSW that is being loaded from the linkage stack state entry that the PR instruction is popping. But on other computers, PR sets the condition code to a random value. This PRCC feature indicates that PR sets the condition code according to the PSW that is being loaded. The PR instruction is documented in:
- OS/390: Principles of Operation (SA22-7201)
- z/OS: Principles of Operation (SA22-7832)

RESMGR - Resource Manager
The RESMGR macro allows an authorized program to add or delete a Resource Manager to the System. A Resource Manager is an exit routine that is given control whenever a task or an address space ends. This service was added to the operating system at MVS/ESA 4.3. It is documented in:
- OS/390: "MVS Programming: Authorized Assembler Services Reference, Volume 3 (LLACOPY-SDUMPX)" (GC28-1766)
- z/OS: "MVS Programming: Authorized Assembler Services Reference, Volume 3 (LLACOPY-SDUMPX)" (SA22-7611)

RI - Relative and Immediate Instructions
These are machine instructions that address storage relative to their own locations in storage (instead of by means of a base register). Relative and immediate instructions are documented in:

- OS/390: Principles of Operation (SA22-7201)
- z/OS: Principles of Operation (SA22-7832)

RP - Resume Program Instruction
The RP (resume program) machine instruction provides a method for one routine to branch to another with all 16 general registers set to arbitrary values without needing to use any register as a base for the jump. RP is documented in:
- OS/390: Principles of Operation (SA22-7201)
- z/OS: Principles of Operation (SA22-7832)

SDWALSLV - Linkage Stack Level Preservation

When a recovery routine (ESTAE, ESTAI, etc.) causes an ABENDed program to resume execution, the SDWALSLV field allows the recovery routine to affect which linkage stack entries are preserved and which are purged. without this facility, the linkage stack is purged of all entries that were created after the ESTAE(X) macro (or whatever) was issued that created the recovery routine. The SDWALSLV allows a recovery routine to preserve all linkage stack entries that existed in the retry level environment at the time of error. The SDWALSLV field is documented in:
- OS/390: "MVS Programming: Assembler Services Guide" (GC28-1762)
- z/OS: "MVS Programming: Assembler Services Guide" (SA22-7605)

SOPF - Suppression on Protection Facility
This is a technical change to the way in which the hardware handles protection interrupts. It causes additional information to be stored so that the Operating System can more accurately understand what is going on when a machine instruction attempts to store into a protected page of storage. This change was necessary for the proper implementation of the FORK service in MVS’s Unix System Services. Suppression on Protection is documented in:
- OS/390: Principles of Operation (SA22-7201)
- z/OS: Principles of Operation (SA22-7832)

STORGXMS - STORAGE Macro: Full Cross Memory Service
The STORAGE macro can be used in Cross Memory routines (PC routines). This service was added to the operating system at MVS/ESA 4.3. The STORAGE macro is documented in:
- OS/390: "MVS Programming: Assembler Services Reference" (GC28-1910)
- z/OS: "MVS Programming: Assembler Services Reference, Volume 2 (IARR2V-XCTLX)" (SA22-7607)

TIMEDEC - TIME DEC,LINKAGE=SYSTEM Service
This support permits the TIME service to be called via a PC instruction instead of an SVC instruction. This permits the TIME service to be called from PC routines. This service was added to the operating system at MVS/ESA 4.3. The TIME macro is documented in:
- OS/390: "MVS Programming: Assembler Services Reference" (GC28-1910)
- z/OS: "MVS Programming: Assembler Services Reference, Volume 2 (IARR2V-XCTLX)" (SA22-7607)

TRAP2/4 - TRAP2 and TRAP4 Instructions
These are machine instructions that can be used by a debugging routine to set breakpoints into a program that is being debugged. They cause control to be transferred to a handler routine with as little environment state change as possible. These instructions are documented in:
- OS/390: Principles of Operation (SA22-7201)
- z/OS: Principles of Operation (SA22-7832)

z/ARCH - 64-bit Hardware

z/ARCH - 64-bit Hardware Installed but Not In Use
z/ARCH - 64-bit Hardware Installed and In Use
The current hardware has the machine instructions and general architecture to support 64-bit addressing. z/Architecture is documented in:
- z/OS: Principles of Operation (SA22-7832)

Help COmmands LISt FIxed

The **LIST FIXED** command displays four bytes of virtual storage in various fixed point formats. The area being displayed need not be aligned. The formats displayed are:
- Raw hex-text (dump like)
- 1-byte unsigned fixed point decimal
- 2-byte signed fixed point decimal
- 3-byte unsigned fixed point decimal
- 4-byte signed fixed point decimal

Syntax:

```
LIST FIXED addressexpression WIDE    EBCDIC  ADDRESSES
                             NARROW  ASCII   OFFSETS
```

addressexpression
This gives the virtual storage location to be displayed. Each fixed point number displayed has this location as its left-most byte.

WIDE
NARROW
These operands control whether data displays are to be wide or narrow:

WIDE
This operand causes the resulting display to be wide. Basically, the text interpretation of the displayed storage is placed towards the right-hand edge of a wide display. This option works best when your terminal's display is at least 136 columns wide. For more information, see HELP FULLSCREEN TERMINALS.

NARROW
This operand causes the resulting display to be narrow. This width is appropriate for terminals with 80 character wide display lines.

ASCII
EBCDIC
These operands control whether the text portion of the display is to be interpreted using the **ASCII** or **EBCDIC** character set:

ASCII
Causes the test display to show the **ASCII** interpretation of the displayed storage. To visually indicate this, the text portion of each display line will be framed with **vertical bars (|)** instead of asterisks.

EBCDIC
Causes the test display to show the **EBCDIC** interpretation of the displayed storage. To visually indicate this, the text portion of each display line will

be framed with **asterisks (*).**

ADDRESSES
OFFSETS
These operands control whether the far left column of the display (the "address" column) will show storage addresses or offsets:

ADDRESSES
Causes the line of displayed data to be prefixed by that data's virtual address.

OFFSETS
Causes the line of displayed data to be prefixed by that data's offset from the start of an appropriate including object (load module, csect, dsect, or equate).

All of the above keywords serve to override the corresponding default values established via the SET FORMAT command.

Example:

DMAP XDCMAPS.CVT
USING CVT 10?
LIST FIXED .CVTRLSTG ADD
This series of commands does the following:
- The DMAP command loads from disk a dsect map for the CVT.
- Then the USING command assigns it to represent the proper location in storage.
- Then the **LIST FIXED** command displays the contents of the CVTRLSTG field (the size of real storage) as decimal values.
- The presence of the **ADD** operand causes the storage location of CVTRLSTG to be shown as an address (instead of an offset).
- The absence of a **WIDE** or **NARROW** operand causes the width of the display to be controlled by the current SET FORMAT setting.

Help COmmands LISt FLC

The **LIST FLC** command displays the current Function Leader Character. This is a character that sometimes must precede a function name appearing within an address expression. See HELP FUNCTIONS FLC for more information.

Syntax:

LIST FLC

This command does not accept operands.

The FLC character can be set via the SET FLC command.

The FLC character can be saved in your session profile, and so it can be both displayed and set via the Profile Menuing System. See HELP PROFILES MENU for more information.

Help COmmands LISt FLOAT

The **LIST FLOAT** command displays eight bytes of virtual storage in various floating point formats. The area being displayed need not be aligned. The formats displayed are:

- Raw hex-text (dump like)
- 4-byte floating point decimal (HFP)
- 8-byte floating point decimal (HFP)

Syntax:

```
LIST FLOAT addressexpression WIDE    EBCDIC  ADDRESSES
                             NARROW  ASCII   OFFSETS
```

addressexpression
This gives the virtual storage location to be displayed. The two floating point numbers displayed have this location as their left-most bytes.

WIDE
NARROW
These operands control whether data displays are to be wide or narrow:

WIDE
This operand causes the resulting display to be wide. Basically, the text interpretation of the displayed storage is placed towards the right-hand edge of a wide display. This option works best when your terminal's display is at least 136 columns wide. For more information, see HELP FULLSCREEN TERMINALS.

NARROW
This operand causes the resulting display to be narrow. This width is appropriate for terminals with 80 character wide display lines.

ASCII
EBCDIC
These operands control whether the text portion of the display is to be interpreted using the **ASCII** or **EBCDIC** character set:

ASCII
Causes the test display to show the **ASCII** interpretation of the displayed storage. To visually indicate this, the text portion of each display line will be framed with **vertical bars (|)** instead of asterisks.

EBCDIC
Causes the test display to show the **EBCDIC** interpretation of the displayed storage. To visually indicate this, the text portion of each display line will be framed with **asterisks (*).**

ADDRESSES
OFFSETS
These operands control whether the far left column of the display (the "address" column) will show storage addresses or offsets:

ADDRESSES
Causes the line of displayed data to be preceded by that data's virtual address.

OFFSETS
Causes the line of displayed data to be preceded by that data's offset from the start of an appropriate including object (load module, csect, dsect, or equate).

All of the above keywords serve to override the corresponding default values established via the SET FORMAT command.

Example:

LIST FLOAT,21C?-20 WIDE
This displays the floating point register FR0 save area located in the current TCB. The display will be wide. If you are on only an 80 character wide terminal, you will have to scroll rightwards to see it all.

Help COmmands LISt FLOATIngpointregisters

Floating point registers are used exclusively for numeric calculations. They are 8 bytes wide. There are either four or sixteen of them depending upon whether (16) or not (4) Advanced Floating Point (AFP) support is installed for IEEE Binary Floating Point (BFP) and/or for Decimal Floating Point (DFP).

When AFP is **not** installed, the hardware and Operating System will interpret all floating point numbers as being in hexadecimal floating point (HFP) format. When AFP is installed, floating point numbers will be understood to be either in HFP or BFP or DFP format, depending upon the machine instructions that are being used to manipulate them.

Floating point numbers are documented in z/OS: Principles of Operation (SA22-7832). The **LIST FEATURES** command can be used to display whether or not AFP is installed on the current system.

Notice! While support for the full set of 16 floating point registers always exists in z/OS, the newer "extended registers" (FR1, FR3, FR5 and FR7-FR15) remain undefined for any task or SRB routine until they are first referenced by code running under said task or SRB. Accordingly, attempts to display the extended registers will result in the data being dashed out. (Try it, you will see what I mean.)

Also notice: Unless you use its **ZAP** command to intentionally set data into an AFP register, Z/XDC will never do anything to activate AFP support in an execution thread in which AFP support has not yet been activated.

Unlike other registers, the Operating System does not provide any Request Block level or Linkage Stack level save areas for floating point registers. It only provides a task level save area. Consequently, there is no error level vs. retry level distinction to be made for floating point registers. Whatever values they contained at the time of error, those will be the values they will contain at retry time. (For more information about the error level and retry level environments, see

HELP EXECUTIONLEVELS.)

Consequently, unlike other registers, Z/XDC does not maintain two sets of floating point registers, only one. Collectively, they are named **FREGS.** Individually, they are named **FRn, BFRn, DFRn** or **HFRn.**

Note, the names **FRn, BFRn, DFRn** and **HFRn** all reference the exact same (and only) set of floating point registers. The only distinction is in how the data within the register is interpreted and displayed.

Z/XDC does not limit you to displaying just **your** program's floating point registers. Every task and every SRB has its own set of floating point registers, so (security and authorization permitting) Z/XDC allows you to display the floating point registers associated with **any** task (TCB) or **any** SRB located in **any** address space in the entire system!

Subtopics Index

Z/XDC allows you either to display individual floating point registers or to display an entire register set. For specific information, type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

INDIVIDUAL - Displaying individual floating point registers.
REGISTERSET - Displaying an entire floating point register set.

Help COmmands LISt FLOATIngpointregisters Individual

Z/XDC allows you to display an individual floating point register:
- Belonging to the current program.
- Belonging to any program running under any TCB located in any accessible address space.
- Belonging to any suspended SRB routine located in any accessible address space.

Please note that for control registers, floating-point registers and vector registers, z/OS does not maintain a distinction between the error level and retry level environments.

If programs (running under the task or SRB for which an extended floating point register is being displayed) have not yet used any extended floating point registers, then z/OS has not yet enabled the extended floating point registers for that task or SRB, and so the display will show that register's data dashed out. For more information, see HELP COMMANDS LIST FLOATINGPOINTREGISTERS.

Floating point registers are displayed in any or all of the following formats:
- hex-text (dump like) format
- Decimal floating point format (DFP)
- Hexadecimal floating point format (HFP)
- Binary floating point format (BFP)

Further, each of the floating point formats is displayed twice:
- short (4-byte) format
- long (8-byte) format

Syntax:

LIST FRn **tcbaddress**
BFRn **rbaddress**
DFRn **ssrbaddress**
HFRn **addressomitted**
nFR
nBFR
nDFR
nHFR

These commands display a floating point register. If the binary floating point feature (BFP) is installed, or if the decimal floating point feature (DFP) is installed, then **n** may be be any decimal number from 0 to 15. Otherwise, **n** may be only 0, 2, 4, or 6.

LIST FRn [address]
nFR
The register's data is displayed in hex-text format and then in **all three** floating point formats (BFP, DFP and HFP). For each format, the data is displayed in both short and long formats.

LIST BFRn [address]
nBFR
The register's data is displayed in hex-text format and then in **binary** floating point format (BFP). The data is displayed in both short and long formats.

LIST DFRn [address]
nDFR
The register's data is displayed in hex-text format and then in **decimal** floating point format (DFP). The data is displayed in both short and long formats.

LIST HFRn [address]
nHFR
The register's data is displayed in hex-text format and then in **hexadecimal** floating point format (HFP). The data is displayed in both short and long formats.

LIST fpregname **tcbaddress**
ssrbaddress

The specified floating point register is displayed that is owned either by the specified task or by the specified SRB routine.

Note, if the specified register is one of the extended registers (FR1, FR3, FR5 or FR7-FR15), and if no code running under the specified task has yet referenced any extended floating point register, then the display will be dashed out. (See HELP COMMANDS LIST FLOATINGPOINTREGISTERS for more information.)

Note, the **TCB#n** equates created by the **LIST TASKS** command can, of course, be used as the **tcbaddress** operand

Also note, the **SSRB#n** and **SSRX#n** equates created by the **LIST SSRB** command can be

used as the **ssrbaddress** operand.

LIST fpregname **rbaddress**
Request Block support for this command is provided only for convenience. Since z/OS saves floating point registers only at the task level, not the RB level, Z/XDC treats a given RB address as if it were the address of the TCB to which the RB is queued. In other words, for the various **LIST floatingpointregistername** commands, a given **rbaddress** operand is processed identically as if a **tcbaddress** operand had been given instead.

LIST fpregname **addressomitted**

When an address operand is omitted, the current program's floating point register will be displayed. The value shown will be what was the register's contents at the time of error and what will be its contents at program resumption time. (z/OS does not maintain a distinction between error level and retry level floating point registers.)

Examples:

LIST FR0
Lists and formats the contents of the current program's floating point register 0 in every possible format.

LIST DFR1
Lists and formats the contents of the current program's floating point register 1 in only the decimal floating point format.

LIST SSRBS 1
LIST 1FR SSRB#1 [or **SSRX#1]**

If Z/XDC is running authorized, and if System Security permits the access, then the **LIST SSRBS 1** command displays a list of all SSRBs suspended in the Master Scheduler's address space. It also creates a series of **SSRB#n** and **SSRX#n** equates labeling the locations of all the SSRBs and SSRXs that were found.

The **LIST 1FR SSRB#1** command (or **LIST 1FR SSRX#1** command) then lists and formats the contents of the SRB routine's floating point register 1. Note, since this particular SRB routine does not use or reference any floating point registers, the extended registers have not been created, and so their displayed data will be dashed out.

Help COmmands LISt FLOATIngpointregisters Registerset

The **LIST FREGS** command allows you to display the entire set of floating point registers:
- Belonging to the current program.

- Belonging to any task located in any accessible address space.
- Belonging to any suspended SRB routine located in any accessible address space.

If programs (running under the task or SRB for which the floating point registers are being displayed) have not yet referenced any extended floating point registers, then z/OS has not yet enabled the extended floating point registers for that task or SRB, and so the display will show those registers data dashed out. For more information, see HELP COMMANDS LIST FLOATINGPOINTREGISTERS.

Note, the **LIST FREGS** command displays the floating point registers **only** in a raw hex-EBCDIC (or hex-ASCII) format. If you want to see the numeric interpretations (HFP, BFP and DFP) of the registers, then display them individually. See HELP COMMANDS LIST FLOATINGPOINTREGISTERS INDIVIDUAL for more information.

Syntax:

```
LIST FREGS  WIDE    EBCDIC  tcbaddress
            NARROW  ASCII   rbaddress
                            ssrbaddress
                            addressomitted
```

If either the binary floating point feature (BFP) or the decimal floating point feature (DFP) are installed, then 16 registers will be displayed; otherwise, the display will show only four registers: FR0, FR2, FR4, and FR6.

LIST FREGS **WIDE**
NARROW

These are optional. They control whether the resulting display will show 8 words of data per line **(WIDE)** or just 4 words **(NARROW).** If omitted, then the current **SET FORMAT** command setting is used. This setting can be displayed by the **LIST FORMAT** command and saved into your session profile by the **PROFILE SAVE** command. For more information, see:

HELP COMMANDS LIST FORMAT
HELP COMMANDS SET FORMAT
HELP COMMANDS PROFILE

Wide displays are suitable if your terminal display is set to 136 columns or wider. Narrow displays are suitable when only 80 columns are displayed. See HELP FULLSCREEN TERMINALS for more information.

LIST FREGS **EBCDIC**
ASCII

These operands also are optional. They control, for the character portion of the display, whether the register contents are interpreted as EBCDIC characters or ASCII. If omitted, then the current **SET FORMAT** command setting is used.

LIST FREGS **tcbaddress**
ssrbaddress

The floating point register set is displayed that is owned either by the

specified task or by the specified SRB routine.

Note, if no code running under the specified task or SRB has yet referenced any extended floating point register, then the extended floating point registers will not yet have been enabled by z/OS, and so their displays will be dashed out. (See HELP COMMANDS LIST FLOATINGPOINTREGISTERS for more information.)

Note, the **TCB#n** equates created by the **LIST TASKS** command can, of course, be used as the **tcbaddress** operand

Also note, the **SSRB#n** and **SSRX#n** equates created by the **LIST SSRB** command can be used as the **ssrbaddress** operand.

LIST FREGS **rbaddress**
Request Block support for this command is provided only for convenience. Since z/OS saves floating point registers only at the task level, not the RB level, Z/XDC treats a given RB address as if it were the address of the TCB to which the RB is queued. In other words, for the **LIST FREGS** command, a given **rbaddress** operand is processed identically as if an **tcbaddress** operand had been given instead.

LIST FREGS **addressomitted**
When an address operand is omitted, the current program's floating point registers will be displayed. The values shown will be what were the registers contents at the time of error and what will be their contents at program resumption time. (z/OS does not maintain a distinction between error level and retry level floating point registers.)

Examples:

LIST FREGS
Lists the contents of the entire set of floating point registers owned by the current task or SRB routine. The display is either wide or narrow according to the current setting of the **SET FORMAT** command.

LIST TASKS JES2
LIST FREGS TCB#3 WIDE
If Z/XDC is running authorized, and if System Security permits the access, then the **LIST TASKS JES2** command displays the subtask structure for JES2's address space. It also creates a series of **TCB#n** equates labeling the locations of all TCBs within the JES2 address space.

Then the **LIST FREGS TCB#3 WIDE** command displays the contents of the entire set of floating point registers that belong to the program (HASJES20) running under the control of the jobstep task (TCB#3). The display is wide, ie. is suitable for a wide geometry terminal with display lines that are at least 136 columns wide. (See HELP FULLSCREEN TERMINALS GEOMETRIES for more information.)

LIST SSRBS 1
LIST FREGS SSRB#1 [or **SSRX#1]**
If Z/XDC is running authorized, and if System Security permits the access, then the **LIST SSRBS 1** command displays a list of all SSRBs suspended in the Master Scheduler's address space. It also creates a series of **SSRB#n** and **SSRX#n** equates labeling the locations of all the SSRBs and SSRXs that were found.

Then the **LIST FREGS SSRB#1** command (or **LIST FREGS SSRX#1** command) displays all floating point registers belonging to the SRB routine running under the control of that SSRB. Note, if this particular SRB routine does not use or reference any floating point registers, the extended registers have not been created, and so their displayed data will be dashed out.

Help COmmands LISt FOrmat

The **LIST FORMAT** command displays the current values of storage formatting related settings. These settings affect the way in which Z/XDC's storage formatting commands display storage. The storage formatting commands are **FORMAT, FIND, SHOW, WHERE** and **EWHERE.** (Other commands affected by at least some of the **SET FORMAT** settings are **LIST FIXED, LIST FLOAT** and the various register display commands.)

The settings displayed by the **LIST FORMAT** command are:

SOURCE vs. **OBJECT** vs. **BOTH**
When an area of storage is being displayed by the FORMAT, WHERE or EWHERE command, and an ADATA map has been assigned to that area, then this setting controls whether (SOURCE) or not (OBJECT) that storage will be displayed showing source image information from the ADATA map. If not (OBJECT), then the storage will still be disassembled under the guidance of the ADATA map, and the display will still be annotated by statement labels. If BOTH is set, then source images from the adata map will be interspersed with disassemblies from storage.

SHOWMCODE vs. **HIDEMCODE** vs. **CURRENTMCODE**
Again, when an area of storage is being displayed by the FORMAT, WHERE or EWHERE command, and an ADATA map has been assigned to that area, and the ADATA map is **not** a data area of control block mapping dsect map, then this setting controls whether or not the display will include code generated from macro expansions. For more information, see **HELP MAPS ADATA MACROS.**

Note, the intent of the **CURRENTMCODE** operand is to improve the display of code written using structured programming macros.

ADDRESSES vs. **OFFSETS**
In the displays produced by the DISPLAY, FORMAT, SHOW, and WHERE commands the display's left-hand column shows the storage location of the information being

displayed. This setting controls whether the location information displayed is an actual storage address (ADDRESS) or an offset into whatever object (if any) that includes the displayed location (OFFSET).

HEXADECIMAL vs. **DECIMAL**
A large number of machine instructions use base-displacement addressing to reference storage. When such instructions are disassembled and displayed by the FORMAT, SHOW, and WHERE commands, this setting controls whether the displacements that are displayed are shown in hex or decimal.

EBCDIC vs. **ASCII**
When storage or register contents are displayed, Z/XDC usually will show the displayed information in a hexadecimal-text format (i.e. dump like). This setting controls whether the text portion of such displays show the EBCDIC interpretation of the data or the ASCII interpretation.

NARROW vs. **WIDE**

When registers or storage is displayed in a hex-text format, this setting controls whether the displayed lines will show up to 4 words (16 bytes) of information per line (NARROW) or up to 8 words (32 bytes) of information (WIDE). NARROW is suitable for terminals that display only 80 characters per line. WIDE is best used when the terminal displays 136 characters or more per line.
See **HELP FULLSCREEN TERMINALS GEOMETRIES** for more information.

POINTERS vs. **NOPOINTERS**
When a formatted display is produced (via FORMAT, [E]WHERE and SHOW), if the display contains 3-byte, 4-byte and 8-byte wide data fields containing addresses that point to other locations, **[NO]POINTERS** control whether or not those other locations are resolved and displayed as comments.

When very complex structures of **floating maps and equates** are present, the **POINTERS** setting may result in slower displays. In that case, you may wish to turn this off using the **SET FORMAT NOPOINTERS** command.

For more information about floating maps and equates, see **HELP MAPS FLOATING.**

The SET LINES Setting
When Z/XDC is running in line mode (i.e. not in fullscreen mode), the SET LINES setting is the default display size used by the FORMAT, DISPLAY, and WHERE commands. For more information, see **HELP COMMANDS SET LINES.**

Syntax:

LIST FORMAT

This command does not accept operands.

All of the displayed settings (other than the **SET LINES** setting) can be set via the **SET FORMAT** command.

The **SET LINES** setting can be set via the **SET LINES** command.

All of the displayed settings can be saved in your session profile with the **PROFILE SAVE** command.

All of the displayed settings can be both displayed and set via the **Profile Menuing System.** (See **HELP PROFILES MENU FORMATTRACEFIND** for more information.)

All of the displayed settings can be **overridden** for **individual commands** via operands accepted by those commands.

Help COmmands LISt FPc

The Floating Point Control register is used to control some aspects of binary floating point (BFP) processing, decimal floating point (DFP) processing, and vector processing. It also contains information that indicates whether or not certain events have occurred. The register is part of the Advanced Floating Point (AFP) facility.

Notice! While support for the full set of 16 floating point registers and the floating point control register always exists in z/OS, the newer "extended registers" (FR1, FR3, FR5 and FR7-FR15) and the FPC register remain undefined for any task or SRB routine until they are first referenced by code running under said task or SRB. Accordingly, attempts to display the FPC register will result in the data being dashed out. (Try it, you will see what I mean.)

Also notice: Unless you use its **ZAP** command to intentionally set data into the FPC register (or an AFP floating point register), Z/XDC will never do anything to activate AFP support in an execution thread in which AFP support has not yet been activated.

Unlike other registers, the Operating System does not provide any Request Block level or Linkage Stack level saveareas for the FPC register, It only provides a task level savearea. Consequently, there is no error level vs. retry level distinction to be made for the FPC register. Whatever value it contained at the time of error, that will be the value it will contain at retry time. (For more information about the error level and retry level environments, see HELP EXECUTIONLEVELS.)

Consequently, unlike other registers, Z/XDC does not maintain two versions of the FPC register, only one.

Z/XDC does not limit you to displaying just **your** program’s floating point control register. Every task and every SRB has its own floating point control register so (security and authorization permitting) Z/XDC allows you to display the floating point control register associated with **any** task (TCB) or **any** SRB located in **any** address space in the entire system!

Help COmmands LISt FR#

The **LIST FRn** command displays the entire 8 bytes of an individual floating point register in various formats.

The display shows all permitted interpretations of the register's contents both with respect to floating point type and with respect to length. The data types include:
- Binary Floating Point (BFP)
- Decimal Floating Point (DFP)
- HexaDecimal Floating Point (HFP)

If Advanced Floating Point (AFP) support does not exist, then the LIST FRn command will accept references only to floating point registers 0, 2, 4 and 6. Otherwise, references to any register (0 thru 15) are accepted.

If AFP support does exist but has not yet been activated in the current execution thread (task or SRB), then the LIST FRn display will show the register contents zeroed out (not dashed out).

For more information, see HELP COMMANDS LIST FLOATINGPOINTREGISTERS.

Help COmmands LISt FREgs

The **LIST FREGS** command displays either 4 or 16 floating point registers depending upon whether (16) or not (4) Advanced Floating Point (AFP) exists on the current processor. The LIST FEATURES command can be used to see whether or not AFP is installed.

The registers are displayed in a hex-text format. If AFP is installed but its use has not yet been activated in the current execution thread (task or SRB), then the LIST FREGS display will show the AFP registers dashed out.

For more information, see HELP COMMANDS LIST FLOATINGPOINTREGISTERS.

Help COmmands LISt GEneralregisters

There are 16 general registers, numbered from 0 to 15. For OS/390 they are 4 bytes wide. For z/OS they are 8 bytes wide. General registers are documented in:
- OS/390: Principles of Operation (SA22-7201)
- z/OS: Principles of Operation (SA22-7832)

Z/XDC maintains two sets of general registers: an error level set and a retry level set. The error level registers are those that existed at the time that the ABEND or the breakpoint occurred that caused Z/XDC to receive control. The retry level registers are those general registers that will exist should Z/XDC be used to cause the user's program to resume execution. For more information, see HELP EXECUTIONLEVELS.

Z/XDC does not limit you to displaying just **your** program's general registers. Every program has its own set of general registers, so (security and authorization permitting) Z/XDC allows you to display the general registers associated

with **any** Request Block (RB), queued from **any** Task Control Block (TCB), located in **any** address space in the entire system! But Please note the following:

- The general registers that are **associated** with a given Request Block are those that **belong to** the program that is running under the control of that Request Block.
- The registers that are associated with a given Request Block are actually stored in the Request Block that is **next newer** than the designated Request Block. (Z/XDC automatically finds them.)
- The registers that are associated with a task's newest Request Block are actually stored in that task's TCB.

So the registers that are displayed are those that are associated with (not stored within) the given RB. These are the registers that belong to the program running under said RB.

Z/XDC also allows you to display the general registers belonging to a suspended SRB routine located in any address space in the system.

Subtopics Index

Z/XDC allows you either to display individual registers or to display entire register sets. For specific information, type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

INDIVIDUAL - Displaying individual general registers.

REGISTERSET - Displaying an entire general register set.

Help COmmands LISt GEneralregisters Individual

Z/XDC allows you to display an individual general register:

- Belonging to the current program's retry level registers.

- Belonging to the current program's error level registers.

- Belonging to any program running under any RB queued from any TCB located in any accessible address space.

- Belonging to any suspended SRB routine located in any accessible address space.

The register is displayed in a hex-text format. Its contents also are interpreted and displayed as decimal numbers of various widths.

In z/OS, Z/XDC provides support for three different views of the registers. You can display either their high halves, their low halves, or their entireties. Z/XDC defines a different name for the registers depending upon which view you want to see. The high halves are named RHn, the low halves are named Rn, and the entireties are named RWn.

Syntax:

LIST Rn
nR

In OS/390, either of these commands displays the entirety (all 4 bytes) of

a **retry level** general register. In z/OS, these commands display only the low half of a retry level register.

"n" is a decimal number ranging from 0 to 15. It can appear either before or after the letter R.

LIST RHn
nRH
In z/OS, these commands display the high half of a retry level register. (These commands are not permitted in OS/390.)

LIST RWn
nRW
In z/OS, these commands display the entirety (all 8 bytes) of a retry level register. (These commands are not permitted in OS/390.)

LIST ERn
nER

In OS/390, either of these commands displays the entirety (all 4 bytes) of an **error level** general register. In z/OS, these commands display only the low half of an error level register.

LIST ERHn
nERH
In z/OS, these commands display the high half of an error level register. (These commands are not permitted in OS/390.)

LIST erwn
nerw
In z/OS, these commands display the entirety (all 8 bytes) of an error level register. (These commands are not permitted in OS/390.)

LIST Rn rbaddress
nR ssrbaddress
RHn
nRH
RWn
nRW
If an address expression is given, then it must resolve to either of the following:

- The address of a **Request Block** that is currently queued from any TCB located in any accessible address space anywhere in the system. (Note, the **RB#n** automatic equates created by the LIST RBS command are perfectly good address expressions to use for this rbaddress operand.)

- The address of an **SSRB** or **SSRX** for any SRB routine currently suspended in any accessible address space anywhere in the system. (Note, the **SSRB#n** and **SSRX#n** automatic equates created by the LIST SSRBS command are perfectly good address expressions to use for this ssrbaddress operand.)

(Note, specifying an address is not supported for error level registers.)

When an RB address is given, Z/XDC will attempt to display the general register that belongs to the program that is currently running under that RB. (This is the register that is stored in the next newer RB, or in the TCB in the case of the newest RB.) Note that the **LIST RBS** command, as a side effect, creates a

series of equates named **RB#n** to label the locations of the Request Blocks that it displays. These equates make handy dandy address expressions for use with the various **LIST generalregister** commands.

When an SSRB or SSRX address is given, Z/XDC will display the general register that belongs to the SRB routine that is currently suspended under that SSRB and SSRX. Note that the **LIST SSRBS** command, as a side effect, creates a series of equates named **SSRB#n** and **SSRX#n** to label the locations of the SSRBs and SSRXs from which it builds its displays. These equates can be used as the ssrbaddress address expression on the **LIST generalregister** command.

Examples:

LIST R0
Lists and formats the contents of retry level general register 0. In OS/390, the entire R0 is displayed. In z/OS, only the low half is displayed.

LIST 3RW
In z/OS, this displays the entire contents (all 8 bytes) of error level register 3.

LIST TASKS JES2
LIST RBS TCB#3
LIST R4 RB#1
If Z/XDC is running authorized, and if System Security permits the access, then the **LIST TASKS JES2** command displays the subtask structure for JES2's address space. It also creates a series of **TCB#n** equates labeling the locations of all TCBs within the JES2 address space.

The **LIST RBS** command then makes use of the **TCB#3** equate to display the Request Blocks currently queued to JES2's jobstep task. It also creates a series of **RB#n** equates labeling the locations of all such RBs.

Finally, the **LIST R4 RB#1** command displays the contents of general register 4 that belongs to the program (HASJES20) running under the control of the jobstep task's oldest Request Block. In OS/390, the entire R4 is displayed. In z/OS, only the low half is displayed.

LIST SSRBS 1
LIST 7RW SSRB#1 [or **SSRX#1]**
If Z/XDC is running authorized, and if System Security permits the access, then the **LIST SSRBS 1** command displays a list of all SSRBs suspended in the Master Scheduler's address space. It also creates a series of **SSRB#n** and **SSRX#n** equates labeling the locations of all the SSRBs and SSRXs that were found.

Then the **LIST 7RW SSRB#1** command (or **LIST 7RW SSRX#1** command) displays the 64-bit wide view of the contents of general register 7 belonging to the SRB routine running under the control of that SSRB and SSRX.

Help COmmands LISt GEneralregisters Registerset

This topic discusses the various **LIST REGS** commands that allow you to display entire sets of general registers:

- Belonging to the current program's **retry level** environment.

- Belonging to the current program's **error level** environment.

- Belonging to **any** program running under any RB queued from any TCB located in **any** accessible address space.

- Belonging to any suspended **SRB** routine located in any accessible address space.

The general registers are displayed in a hex-text format.

Z/XDC provides support for three different views of the registers. You can display either their **high halves,** their **low halves,** or their **entireties.**

Z/XDC defines a different name for the registers depending upon which view you want to see:

- The high halves are named **RHREGS** (or **ERHREGS** for error level instances).
- The low halves are named **REGS** (or **EREGS).**
- And the entireties are named **RWREGS** (or **ERWREGS)**

There are three syntactic forms for each of these three commands. I will discuss them individually.

Syntax #1:

```
LIST REGS   addressomitted WIDE   EBCDIC
     RHREGS                NARROW ASCII
     RWREGS
```

This form of the commands (other forms are described below) can be used to display the general registers from the current program's **retry level** environment.

- **LIST REGS** displays only the low halves of the registers. (Note, if any of the high halves of the registers contain a value that is neither all **00s** nor all **FFs,** then a warning message is also displayed.)
- **LIST RHREGS** displays only the high halves of the registers.
- **LIST RWREGS** displays the entireties of the registers.

```
LIST REGS   [...] WIDE
     RHREGS       NARROW
     RWREGS
```

These operands are optional. They control whether the resulting display will show 8 words of data per line **(WIDE)** or just 4 words **(NARROW).** If omitted, then

the current **SET FORMAT** command setting is used. This setting can be displayed by the **LIST FORMAT** command and saved into your session profile by the **PROFILE SAVE** command. For more information, see:

HELP COMMANDS LIST FORMAT
HELP COMMANDS SET FORMAT
HELP COMMANDS PROFILE

Wide displays are suitable if your terminal display is set to 136 columns or wider. Narrow displays are suitable when only 80 columns are displayed. See **HELP FULLSCREEN TERMINALS** for more information.

LIST REGS [...] **EBCDIC**
RHREGS **ASCII**
RWREGS

These operands also are optional. They control, for the character portion of the display, whether the register contents are interpreted as EBCDIC characters or ASCII. If omitted, then the current **SET FORMAT** command setting is used.

More information on EBCDIC and ASCII can be found at **HELP CHARACTERSETS**

Note, for **LIST REGS,** a check is made to see if all of the **RHn** registers contain only 00s or FFs. If any RHn register contains anything else, a warning message is displayed to alert you to that fact.

Syntax #2:

LIST REGS **rbaddress** WIDE EBCDIC
RHREGS **ssrbaddress** NARROW ASCII
RWREGS

If an address expression is given, then it must resolve to either of the following:

- The address of a **Request Block** that is currently queued from any TCB located in any accessible address space anywhere in the system. (Note, the **RB#n** automatic equates that are created by the **LIST RBS** command are perfectly good address expressions to use for this rbaddress operand.)

- The address of an **SSRB** or **SSRX** for any SRB routine currently suspended in any accessible address space anywhere in the system. (Note, the **SSRB#n** and **SSRX#n** automatic equates that are created by the **LIST SSRBS** command are perfectly good address expressions to use for this ssrbaddress operand.)

(Note, specifying an address is not supported for error level registers.)

When an RB address is given, Z/XDC will display the general registers that belong to the program that is currently running under that RB. (These are the registers that are stored in the next newer RB, or in the TCB in the case of the newest RB.)

Note that the **LIST RBS** command, as a side effect, creates a series of equates named **RB#n** to label the locations of the Request Blocks that it displays. These

equates make handy dandy address expressions for use with this variation of the **LIST REGS** command (and friends).

When an SSRB or SSRX address is given, Z/XDC will display the general registers that belong to the SRB routine that is currently suspended under that SSRB and SSRX. Note that the **LIST SSRBS** command, as a side effect, creates a series of equates named **SSRB#n** and **SSRX#n** to label the locations of the SSRBs and SSRXs from which it builds its displays. These equates can be used as the ssrbaddress address expression on this variation of the **LIST REGS** command (and friends).

Syntax #3:

```
LIST EREGS   WIDE   EBCDIC
     ERHREGS NARROW ASCII
     ERWREGS
```

These forms of the commands can be used to display the general registers from the current program's **error level** environment. Notes:

- For **LIST EREGS,** a check is made to see if all of the **ERHn** registers contain only either all **00s** or all **FFs.** If any other data is found, then a warning message is displayed to alert you to that fact.

- An addressexpression operand is not permitted for these **E forms** of the several list regs commands.

Automatic Equates

For the **LIST REGS, RHREGS** and **RWREGS** commands, when an **rbaddress** or **ssrbaddress** operand is given (see **Syntax #2** above), the following automatic equates are (re)generated:

#REGS
#RHREGS
These label the system control block fields in which the general register data was found.

#Rn
#RHn
#RWn
These label the locations pointed by each of the three views of each of the sixteen general registers.

For more information, see **HELP EQUATES BUILTIN AUTOMATIC.** Notes:

- All of these automatic equates are **static!** They do not float. The locations that they target can change only when another **LIST REGS/RHREG/ RWREGS address** command is issued. In other words, the equates can become obsolete without warning simply due to the ongoing execution of the program whose registers are being displayed.

- When an rbaddress or ssrbaddress operand is **not** given:
  - Replacement automatic equates are **not** generated.

- **Existing** automatic equates are not disturbed.

Examples:

LIST REGS
Lists the contents of the entire set of retry level registers. Only the low half of each register is displayed. The display is either wide or narrow according to the current setting of the **SET FORMAT** command.

Since no **rbaddress** or **ssrbaddress** operand was given, the various **#R*** automatic equates remain unchanged.

If any high half (the **RHn** half) of any register is neither all **00s** nor all **FFs,** then a warning message is displayed.

LIST ERWREGS WIDE
Lists the contents of the entire set of error level registers. The entire 8 bytes of each register is displayed, and four registers (32 bytes worth) are displayed per line instead of just two.

LIST TASKS JES2

LIST RBS TCB#3
LIST REGS RB#1

If Z/XDC is running authorized, and if System Security permits the access, then the **LIST TASKS JES2** command displays the subtask structure for JES2's address space. It also creates a series of **TCB#n** equates labeling the locations of all TCBs within the JES2 address space.

The **LIST RBS** command then makes use of the **TCB#3** equate to display the Request Blocks currently queued to JES2's jobstep task. It also creates a series of **RB#n** equates labeling the locations of all such RBs.

Finally, the **LIST REGS RB#1** command displays the contents of the lo-order 32 bits of all general registers that belong to the program (HASJES20) running under the control of the jobstep task's oldest Request Block.

Because an **rbaddress** operand was given, the following automatic equates are (re)generated:

- **#REGS** labels the field in which the low halves of the general register data was found. That will be one of the following:
  - If **RB#1** is **TCB#3's** newest (i.e. only) Request Block, then **#REGS** will label **TCB#3's TCBGRS** field.
  - If **RB#1** is **not TCB#3's** newest Request Block, then **#REGS** will label the next newer RB's **RBGRSAVE** field.
- **#RHREGS** labels the field in which the high halves of the general register data was found. That will be one of the following:

  - If **RB#1** is **TCB#3's** newest Request Block, then **#RHREGS** will label the **STCBG64H** in the STCB pointed to by **TCB#3.**

  - If **RB#1** is **not TCB#3's** newest Request Block, then **#RHREGS** will label the **XSBG64H** in the XSB pointed to by the next newer RB.

- The sixteen **#Rn** equates label the locations pointed to by the **low** halves of the sixteen general registers owned by the program running under **RB#1.**

- The sixteen **#RHn** equates label the locations pointed to by the **high** halves of the sixteen general registers owned by the program running under **RB#1.**

- The sixteen **#RWn** equates label the locations pointed to by the **entireties** of the sixteen general registers owned by the program running under **RB#1.**

Warning! The locations of all of these equates will **not** change with changes in the register values. They can be changed only by reissuing the **LIST REGS RB#1** command.

LIST SSRBS 1
LIST RWREGS SSRB#1 [or **SSRX#1]**

If Z/XDC is running authorized, and if System Security permits the access, then the **LIST SSRBS 1** command displays a list of all SSRBs suspended in the Master Scheduler's address space. It also creates a series of **SSRB#n** and **SSRX#n** equates labeling the locations of all the SSRBs and SSRXs that were found.

Then the **LIST RWREGS SSRB#1** command (or **LIST RWREGS SSRX#1** command) displays 64-bit views of all general registers belonging to the SRB routine running under the control of that SSRB and SSRX.

Because an **ssrbaddress** operand was given, the following automatic equates are (re)generated:

- **#REGS** labels the field in which the low halves of the general register data was found.

- **#RHREGS** labels the field in which the high halves of the general register data was found.

- The sixteen **#Rn** equates label the locations pointed to by the **low** halves of the sixteen general registers owned by the program running under **SSRB#1.**

- The sixteen **#RHn** equates label the locations pointed to by the **high** halves of the sixteen general registers owned by the program running under **SSRB#1.**

- The sixteen **#RWn** equates label the locations pointed to by the **entireties** of the sixteen general registers owned by the program running under **SSRB#1.**

Warning! The locations of all of these equates will **not** change with changes in the register values. They can be changed only by reissuing the **LIST RWREGS SSRB#1** command.

Help COmmands LISt GSR#

The **LIST GSRn** command displays the entire 8 bytes of an individual guarded storage register.

If guarded storage support does not exist, normally the LIST GSRn command will fail. However, if you'd like to see what the display looks like anyway, you can add the **PRETEND** operand to the command, and Z/XDC will pretend that guarded storage registers exist after all.

If guarded storage support does exist but has not yet been activated in the current execution thread then the LIST GSRn display will show the register contents zeroed out (not dashed out).

For more information, see HELP COMMANDS LIST GSREGISTERS.

Help COmmands LISt GSREgisters

Guarded Storage registers are used in conjunction with some instructions (e.g. LGG and LGGFSG), to guard a range of storage so that references to the guarded storage can be controlled. (This facility is typically used to implement efficient garbage collection).

If the Guarded Storage (GS) facility is not installed, use of the guarded storage instructions will result in a program check.

Guarded Storage is documented in z/OS: Principles of Operation (SA22-7832) (-11 and later). The **LIST FEATURES** command can be used to display whether or not GS is installed on the current system.

Notice! While support for GS is in the later releases of z/OS, use of the GS facility is enabled for a task when it needs it. (Note that the GS facility is not available for use by SRB routines). The GS controls are stored in a control block that is only allocated when needed. When GS support has not been activated (or when debugging an SRB routine) the GS data will be dashed out. (Try it, you will see what I mean.)

Also notice: Unless you use its **ZAP** command to intentionally set data into a GS register, Z/XDC will never do anything to activate GS support in an execution thread in which GS support has not yet been activated.

Unlike other registers, the Operating System does not provide any Request Block level or Linkage Stack level saveareas for guarded storage registers. It only provides a task level savearea. Consequently, there is no error level vs. retry level distinction to be made for guarded storage registers. Whatever values they contained at the time of error, those will be the values they will contain at retry time. (For more information about the error level and retry level environments, see HELP EXECUTIONLEVELS.)

Consequently, unlike other registers, Z/XDC does not maintain two sets of guarded storage registers, only one. Collectively, they are named **GSREGS.** Individually, they are named **GSRn**

Z/XDC does not limit you to displaying just **your** program's guarded storage

registers. Every task has its own set of guarded storage registers, so (security and authorization permitting) Z/XDC allows you to display the guarded storage registers associated with **any** task (TCB) located in **any** address space in the entire system!

Subtopics Index

Z/XDC allows you either to display individual guarded storage registers or to display an entire register set. For specific information, type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

INDIVIDUAL - Displaying individual guarded storage registers.
REGISTERSET - Displaying an entire guarded storage register set.

Help COmmands LISt GSREgisters Individual

Z/XDC allows you to display an individual guarded storage register:
- Belonging to the current program.
- Belonging to any program running under any TCB located in any accessible address space.

Please note that for control registers, floating-point registers, vector registers, and guarded storage registers z/OS does not maintain a distinction between the error level and retry level environments.

If programs for which a guarded storage register is being displayed have not yet used any guarded storage instructions, then z/OS has not yet enabled the guarded storage facility for that task. and so the display will show that register's data dashed out. For more information, see HELP COMMANDS LIST GSREGISTERS.

If guarded storage support does exist but has not yet been activated in the current execution thread then the LIST GSRn display will show the register contents zeroed out (not dashed out).

Syntax:

LIST GSRn **tcbaddress** **PRETEND**
nGSR **rbaddress**
addressomitted

LIST GSRn [address]
nGSR
The register's data is displayed.

LIST gsregname **tcbaddress**
The specified vector register is displayed that is owned by the specified task

Note, if no code running under the specified task has yet used any guarded storage instructions then the display will be dashed out. (See HELP COMMANDS LIST GSREGISTERS for more information.)

Note, the **TCB#n** equates created by the **LIST TASKS** command can, of course, be used as the **tcbaddress** operand

LIST gsregname **rbaddress**
Request Block support for this command is provided only for convenience. Since z/OS saves guarded storage registers only at the task level, not the RB level, Z/XDC treats a given RB address as if it were the address of the TCB to which the RB is queued.

LIST gsregname **addressomitted**
When an address operand is omitted, the current program's guarded storage registers will be displayed. The value shown will be what was the register's contents at the time of error and what will be its contents at program resumption time. (z/OS does not maintain a distinction between error level and retry level guarded storage registers.)

LIST gsregname [...] **PRETEND**
Normally, when the LIST gsregname command is issued while running on a computer that does not support guarded storage, the command fails (with message DBC092E). However, if you want to see what the display looks like anyway, you can use this **PRETEND** operand, and Z/XDC pretends that the registers exist after all.

You also can use the PRETEND operand on the **ZAP** command to make changes to the guarded storage registers. But of course, whenever Z/XDC releases control back to your program (i.e. you issue a GO or TRACE command), all zapped changes will be lost.

The **Z** shortcut commands does not need a PRETEND operand.

Examples:

LIST GSR0
Lists and formats the contents of the current program's guarded storage register 0.

If guarded storage support has not yet been activated in the correct thread zeros will be displayed. in the current thread (Activation will not occur.)

Help COmmands LISt GSREgisters Registerset

The **LIST GSREGS** command allows you to display the entire set of guarded storage registers:

- Belonging to the current program.
- Belonging to any task located in any accessible address space.

If programs running under the task for which the guarded storage registers are being displayed have not yet issued any guarded storage instructions then z/OS has not yet enabled the guarded storage facility for that task, and so the display will show those registers data dashed out. For more information, see HELP COMMANDS LIST GSREGISTERS.

The **LIST GSREGS** command displays the vector registers **only** in a raw hex-EBCDIC (or hex-ASCII) format. If you want to see element by element numeric interpretations, then display them individually. See HELP COMMANDS LIST GSREGISTERS INDIVIDUAL for more information.

Syntax:

```
LIST GSREGS WIDE    EBCDIC  tcbaddress      PRETEND
            NARROW  ASCII   rbaddress
                            addressomitted
```

LIST GSREGS **WIDE**
NARROW

These are optional. They control whether the resulting display will show 8 words of data per line **(WIDE)** or just 4 words **(NARROW).** If omitted, then the current **SET FORMAT** command setting is used. This setting can be displayed by the **LIST FORMAT** command and saved into your session profile by the **PROFILE SAVE** command. For more information, see:

HELP COMMANDS LIST FORMAT
HELP COMMANDS SET FORMAT
HELP COMMANDS PROFILE

Wide displays are suitable if your terminal display is set to 136 columns or wider. Narrow displays are suitable when only 80 columns are displayed. See HELP FULLSCREEN TERMINALS for more information.

LIST GSREGS **EBCDIC**
ASCII

These operands also are optional. They control, for the character portion of the display, whether the register contents are interpreted as EBCDIC characters or ASCII. If omitted, then the current **SET FORMAT** command setting is used.

LIST VREGS **tcbaddress**
ssrbaddress

The guarded storage register set is displayed that is owned by the specified task.

Note, if no code running under the specified task has yet used any guarded storage instructions then the guarded storage facility will not yet have been enabled by z/OS, and so their displays will be dashed out. (See HELP COMMANDS

LIST GSREGISTERS for more information.)

Note, the **TCB#n** equates created by the **LIST TASKS** command can, of course, be used as the **tcbaddress** operand

LIST GSREGS **rbaddress**
Request Block support for this command is provided only for convenience. Since z/OS saves guarded storage registers only at the task level, not the RB level, Z/XDC treats a given RB address as if it were the address of the TCB to which the RB is queued.

LIST GSREGS **addressomitted**
When an address operand is omitted, the current program's guarded storage registers will be displayed. The values shown will be what were the registers contents at the time of error and what will be their contents at program resumption time. (z/OS does not maintain a distinction between error level and retry level guarded storage registers.)

LIST GSREGS [...] **PRETEND**
Normally, when the LIST GSREGS command is issued while running on a computer that does not support guarded storage, the command fails (with message DBC092E). However, if you want to see what the display looks like anyway, you can use this **PRETEND** operand, and Z/XDC pretends that the registers exist after all.

You also can use the PRETEND operand on the **ZAP** command to make changes to the guarded storage registers. But of course, whenever Z/XDC releases control back to your program (i.e. you issue a GO or TRACE command), all zapped changes will be lost.

The **Z** shortcut commands does not need a PRETEND operand.

Examples:

LIST GSREGS
Lists the contents of the entire set of vector registers owned by the current task. The display is either wide or narrow according to the current setting of the **SET FORMAT** command.

LIST TASKS JES2
LIST GSREGS TCB#3 WIDE
If Z/XDC is running authorized, and if System Security permits the access, then the **LIST TASKS JES2** command displays the subtask structure for JES2's address space. It also creates a series of **TCB#n** equates labeling the locations of all TCBs within the JES2 address space.

Then the **LIST GSREGS TCB#3 WIDE** command displays the contents of the entire set of guarded storage registers that belong to the program (HASJES20) running under the

control of the jobstep task (TCB#3). The display is wide, ie. is suitable for a wide geometry terminal with display lines that are at least 136 columns wide. (See HELP FULLSCREEN TERMINALS GEOMETRIES for more information.)

Help COmmands LISt HElp

The **LIST HELP** command displays local and global indexes for Z/XDC's Built-in Help database.

The Built-in Help database has a pyramidal or hierarchical structure. (See **HELP ABOUTHELP** for a more detailed description.) With the LIST HELP command, you can select any topic within the database and display the names of all Built-in Help topics located below that topic. In other words, the LIST HELP command shows an index of all topics that are reachable from a given topic.

The display produced by LIST HELP includes entry fields at the left where you can type **L** and **H** shortcuts for navigating within the Built-in Help. See below for more information.

HINT: When within a Built-in Help display, the **LIST HELP *UP** command (HELP's **F6)** is a good navigational tool to use. It shows you an index of the other Built-in Help topics that are in the neighborhood of the topic that you are currently viewing.

(To return to where you were in the HELP topics, just use **HELP *** .)

Syntax:

```
    LIST HELP name1 name2 ... depth
              -or-  -or-
              rref1 rref2 ...
```

Operands

names
These **topic names** are names of HELP topics. They, together with relative references, help define a path into the Built-in Help database. Each topic name given must be the name of a topic that resides directly below (in the Built-in Help hierarchy) the topic identified by the command's preceding operand.

rrefs
These are any of the following relative references.

***Up** - This refers to the topic that is next higher than the topic indicated by the preceding operand.
***Down** - This refers to one of the next lower topics.
***Forward** - This refers to the next topic (if any) that is at the same level.
***Back** - This refers to the previous topic that is at the same level.
***Next** - This refers to the logically next topic. It may be at the same level or

a different level.
***Previous** - This refers to the logically previous topic. It may be at the same or different level.
***Repeat** - This refers to the same topic as the current topic.
***** - This is a synonym for ***repeat.** It refers to the same topic as the current topic.
***topicname** - This refers to a topic that is directly below the current topic. **HELP *topicname** functions identically to **HELP * topicname.**

The topic references (both names and rrefs) can be **intermixed.** Altogether, they define a starting point for the LIST HELP display: The index of topics located at and below that starting point is displayed.

Please see **HELP COMMANDS HELP** for more complete information about HELP topic references.

depth
This must be a single decimal digit ranging between 1 and 9. Once **LIST HELP** has selected a starting point, this operand controls to what further depth the structure of Built-in Help topics is displayed. If omitted, a default depth of **1** is used.

Each line in **LIST HELP's** display conveys the following information:

- Each line gives the **name** of a HELP topic.
- The degree of **indentation** shows how **deep** the topic is in HELP's hierarchical structure.
- The portion of the name that is **upcased** shows the minimum abbreviation required for that name. Lower case letters are not significant. They are not needed to uniquely distinguish the name.
- If a name is trailed by a **plus sign (+),** then there are more topics available deeper into the hierarchy below that name.
- The most recently displayed topic is marked as **(current).**
- The **sequence** of displayed topic names defines the **next/previous** ordering that is referred to by some of the relative reference operands described above.

Shortcuts

When Z/XDC's Fullscreen Support is turned on, the display produced by LIST HELP includes entry fields at the left where you can type **L** ("List") and **H** ("HELP") shortcuts:

L - If a HELP topic name is displayed with a **"+"** appended on the right,then undisplayed topics exist below the displayed name. The **L** shortcut causes the undisplayed topic names to be displayed.

On the other hand, if **L** is placed on the display's **first** line, then the next **higher** index level (if any) is shown.

H - This causes the named HELP topic's **contents** to be displayed.

HINT: To view the names of **all** of the topics in the Built-in Help database, do the following:

- Issue the **LIST HELP 9** command.
- Then use Z/XDC's **UP/DOWN** scrolling commands to scroll through the resulting messages.
- Also, use the **SCANLOG** command to search the messages for a specific topic name. (SCANLOG works like ISPF's "FIND" command.)
- When you find the name of a topic that interests you, use the **H** shortcut to display that topic's content.

More Information

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

EXAMPLES - Examples of the LIST HELP command.

Help COmmands LISt HElp Examples

The following are examples for using the **LIST HELP** command:

L H 9
This shows a complete index of the Built-in Help database. (The display will contain somewhere around 1300 lines!) The given display depth ("9") is what permits the complete display. The database has no topic that is as deep or deeper than nine levels.

L H
This shows just the top level of HELP topics. (The default depth of 1 is used.) The plus signs appearing in the display indicate where undisplayed levels occur. If Z/XDC's Fullscreen Support is turned on, then **L** shortcuts can be used to show selected deeper levels.

L H COM B 2
This shows the structure of HELP topics that describe breakpoint commands. The structure is shown to a further depth of 2 levels.

L H *U *U 9
The HELP topic that you are currently reading is named "HELP COMMANDS LIST HELP EXAMPLES". While reading this topic, if you right now type "L H *U *U 9", then the topic named "HELP COMMANDS LIST" will be displayed. That is because that topic is located two levels directly above this topic.

Help COmmands LISt HFr#

The **LIST HFRn** command displays the entire 8 bytes of an individual floating point register in the following formats:

- Raw hex-EBCDIC or hex-ASCII
- Hexadecimal floating point (HFP), short and long

For more information, see HELP COMMANDS LIST FLOATINGPOINTREGISTERS.

Help COmmands LISt HILight

This command is available only when Z/XDC's Fullscreen Support is turned on.

Screen images generated by Z/XDC contain up to four different kinds of fields:

(A) - **Low** intensity **input** fields for displaying data in secondary input areas and in overwritable fields (such as storage displays).

(B) - **High** intensity **input** fields for primary input areas such as command lines.

(C) - **Low** intensity **protected** fields for displaying nonoverwritable information.

(D) - **High** intensity **protected** fields for displaying title and header lines, warning messages, etc.

For those terminals that have the necessary hardware support, display colors and extended hilighting can be set by the following commands:

- **SET COLORS**
- **SET HILIGHT**
- **SET INTENSITY**

The **LIST HILIGHT** command displays the current values for extended hilighting.

Syntax:

LIST HILIGHT

This command accepts no operands.

The extended hilight settings can also be displayed by the LIST TERMINAL command. In addition, they can be both displayed and set by Z/XDC's Profile Menuing System.

Note: Even if you have a terminal that supports extended hilighting, your choices can be ignored under the following circumstances:

- Your VTAM Systems Programmer has not properly defined your terminal as supporting "Extended Attribute Bytes" (done via a suitable PSERVIC= value on a MODEENT macro in a VTAM mode table definition).
- You are using a PC workstation program to emulate a terminal, but you have not properly configured your emulation to support "hilight orders" from VTAM.

See HELP FULLSCREEN TERMINALS for more information and for suggestions about

remedying these problems.

Hilight related settings can be saved in your session profile. They can be displayed and changed by various commands as well as by Z/XDC's "Profile Menuing System". For more information, please see:

HELP COMMANDS SET HILIGHT
HELP COMMANDS LIST TERMINAL
HELP COMMANDS PROFILE
HELP PROFILES MENU

Help COmmands LISt HKeys

The **LIST HKEYS** command displays the following, "HELP-mode" PF key attributes:

- Whether or not special PF key definitions are active during HELP topic displays.
- Whether or not the special PF key definitions will be displayed at the bottom of HELP topic displays.

Syntax:

LIST HKEYS

This command accepts no operands.

The HKEYS settings can be saved in your session profile. They can be changed by the HKEYS and SET HKEYS commands. They also can be displayed and changed by Z/XDC's "Profile Menuing System". For more information, please see:

HELP COMMANDS HKEYS
HELP COMMANDS SET HKEYS
HELP PROFILES MENU

Help COmmands LISt HOoks

The **LIST HOOKS** command can be used to display information about dynamic hooks created by the HOOK command issued during the current debugging session.

Once a dynamic hook is created, it exists independently of the debugging session by which it was created. In particular, the hook can survive even past the termination of the creating debugging session. Unfortunately, the LIST HOOK command cannot display hooks that were not created by the current Z/XDC debugging session. For complete information about hooks, see HELP HOOKS.

The LIST HOOKS command cannot display static hooks (hooks created by the #XDCHOOK macro) at all. See HELP HOOKS STATIC for more information.

Syntax:

LIST HOOKS

This command does not accept operands.

Related commands:

- HOOK
- DELETE HOOKS

Related topic:

- HELP HOOKS: A comprehensive discussion of hooks and how to use them in a debugging session.

Examples:

```
MAP MYPROG.DISKIO
S Q MYPROG.DISKIO
HOOK .DCBEXIT
LIST HOOKS
```

- The **MAP** command loads a module map of the MYPROG load module and a csect map of the DISKIO csect.
- The **SET QUALIFIER** command makes MYPROG the default load module name and DISKIO the default csect name. (See HELP ADDRESSING PARSERS ASM RESOLUTION for more information about default names.)
- The **HOOK** command sets a hook at the start of a Request Block driven DCB OPEN exit named DCBEXIT. (See HELP EXECUTIONLEVELS for more information about Request Blocks.)
- The **LIST HOOKS** command displays information about the hook that has just been created.

Help COmmands LISt ILC

The **LIST ILC** command displays the status of the **Z** shortcut's "Instruction Length Check." This check is used to help ensure that **Z** short commands do not inadvertently alter the length of an instruction. The check is performed on the behalf of instruction opcode zaps only.

Syntax:

LIST ILC

This command accepts no operands.

The ILC setting can be saved in your session profile. It can be changed by the SET ILC command. It also can be displayed and changed by Z/XDC's "Profile Menuing System". For more information, please see:

HELP COMMANDS SET ILC
HELP PROFILES MENU

Help COmmands LISt ILIt

The **LIST ILIT** command displays the name(s) and optionally value(s) of IPCS literal equates. This command can only be used under dump/XDC.

Syntax:

```
LIST ILIT  name      omitted
                     CHAR
                     HEX
                     X
                     BOTH
                     NOVALUE
```

Operands:

name
The IPCS literal name. Note that an IPCS literal name can be from 1 to 31 characters long, with the first being alphabetic or national, and the rest being alphanumeric or national.

omitted
The value will be displayed in character form. Unprintable characters will be displayed as periods.

CHAR
The value will be displayed in character form. Unprintable characters will be displayed as periods.

HEX
The value will be displayed in hexadecimal form. Each byte in the value will be displayed as 2 hexadecimal characters.

X
The value will be displayed in hexadecimal form. Each byte in the value will be displayed as 2 hexadecimal characters.

BOTH
The value will be displayed in both character and hexadecimal form.

In all cases, if the value is too long for a complete display, the displayed value will be followed by an ellipses (...).

Examples

LIST ILIT LOADMODULE BOTH
The IPCS literal equate called 'LOADMODULE' will be displayed in both character and hexadecimal form.

Help COmmands LISt INtensity

Screen images generated by Z/XDC contain up to four different kinds of fields:

(A) - **Low** intensity **input** fields for displaying data in secondary input areas and in overwritable fields (such as storage displays).

(B) - **High** intensity **input** fields for primary input areas such as command lines.

(C) - **Low** intensity **protected** fields for displaying nonoverwritable information.

(D) - **High** intensity **protected** fields for displaying title and header lines, warning messages, etc.

Display colors and extended hilighting can be set by the following commands:

- **SET COLORS**
- **SET HILIGHT**
- **SET INTENSITY**

The **LIST INTENSITY** command displays the current settings for field intensities.

However, intensity settings will be honored only for those fields whose colors are set to **"DEFAULT"** (hardware default, not factory default).

Syntax:

LIST INTENSITY

This command accepts no operands.

The field intensities can also be displayed by the **LIST TERMINAL** command.

In addition, they can be both displayed and set by Z/XDC's **Profile Menuing System.**

When intensity settings are honored on color terminals, they will be shown in colors as set by the terminal's **hardware** defaults. Those colors are settable via dialogs with your 3270 emulation software (PCOMM, Vista, Attachmate, etc.). So if, for example, your terminal's blue color is too dark, you can make it lighter with your emulator software's color palette dialog.

Intensity related settings can be saved in your session profile.

They can be displayed and changed by various commands as well as by Z/XDC's **Profile Menuing System.**

For more information, please see:

HELP COMMANDS SET INTENSITY
HELP COMMANDS LIST TERMINAL
HELP COMMANDS PROFILE
HELP PROFILES MENU

Help COmmands LISt ISPf

The **LIST ISPF** command displays the following ISPF environment settings:

- Whether or not the ISPF fullscreen communications interface can be used in the current environment.
- Whether or not the ISPF communications interface can use a preexisting function variable pool or it must create and destroy its own function variable pool every time the ENTER key is pressed.
- The name of an initial ISPF panel as established by the **SET ISR@PRIM** command.

Syntax:

LIST ISPF

This command accepts no operands.

Various ISPF related settings can be saved in your session profile. They can be displayed and changed by various commands as well as by Z/XDC's "Profile Menuing System". For more information, please see:

HELP COMMANDS SET ISPF
HELP COMMANDS SET ISR@PRIM
HELP USERINTERFACES
HELP PROFILES MENU

Help COmmands LISt ISR@prim

The **LIST ISR@PRIM** command displays the name of the ISPF panel that ISPF will use as a Primary Options Menu if Z/XDC's **ISPF** command is used to (re)invoke ISPF recursively.

Syntax:

LIST ISR@PRIM

This command accepts no operands.

The ISR@PRIM setting can be saved in your session profile. It can be changed by the **SET ISR@PRIM** command. It also can be displayed and changed by Z/XDC's **Profile Menuing System.** For more information, please see:

HELP COMMANDS PROFILE SAVE
HELP COMMANDS PROFILE OMITTED
HELP PROFILES MENU
HELP COMMANDS SET ISR@PRIM

Help COmmands LISt Keys

The **LIST KEYS** command displays the assignments of PF key sets to function key ranges. Z/XDC organizes its PF key definitions into up to 3 sets of twelve definitions each. Z/XDC then allows you to assign these sets to whatever ranges of function keys are available at your terminal. For more information, please see **HELP FULLSCREEN PFKEYS PFKEYSETS.**

Syntax:

LIST KEYS
PFKEYS

This command accepts no operands.

PF key set assignments can be made using either the **SET KEYS** command or the Profile Menuing System (the **PROFILE** command). For more information, see:

HELP FULLSCREEN PFKEYS
HELP PROFILES
HELP COMMANDS KEYS
HELP COMMANDS SET KEYS
HELP COMMANDS PROFILE

Help COmmands LISt LIcense

The **LIST LICENSE** command reports both the status of all Licensable Features and a list of currently running Z/XDC debugging sessions and the features they hold.

For more information about **Licensed Features,** see **HELP SUPPORT FEATURESANDCAPS.**

See also **HELP COMMANDS LICENSE.**

The Features report shows:
- Which Features have been licensed and which have not.

  - For the Licensed features:
    - What their **CAP Quotas** are.
    - What their **current usage** is across the entire sysplex.
    - What their **High Water Mark** usages have been since the last IPL.

For each currently active debugging session, the display shows:
  - The **jobname** or **username** for the address space in which the session is running.
  - The name of the **system** on which that job/user is running.
  - The **ASN** (Address Space Number) for the address space in which that job/user is running.
  - Exactly which **Licensed Features** that debugging session is using.

Also shown are the current Z/XDC **customer ID#** and the name of the current **sysplex.**

Syntax:

```
LIST LICENSE
     CAPS      [alias]
     SESSIONS  [alias]
```

This command accepts no operands.

About High Water Marks

- Customers license use of Z/XDC and its various priced features by the **Concurrent Access Permit** or **CAP** for short. There is a CAP type for Z/XDC sessions **(base/XDC)** and CAP types for a handful of Licensed Features **(auth/XDC, c/XDC, dump/XDC,** etc.).

- **base/XDC** CAPs are checked and incremented when a debugging session starts. All other CAP types are checked and incremented upon first use of the associated Licensed Feature.

- CAP usage is measured by counting certain **ENQs** having **SCOPE=SYSTEMS** (plural!). This means that **current** CAP usage is measured across an **entire sysplex.**

- When a CAP type reaches a new high level, its value is stored in Colesoft's **Customer Anchor Table** as a **High Water Mark.**

- There is a separate HWM for each CAP type.

- When a new HWM is set, that information is **not propagated** to other members of the sysplex. Thus, HWM values may disagree from one member of a sysplex to the next.

High Water Marks Are Ephemeral

Z/XDC records High Water Mark values only in memory (in Colesoft's Customer Anchor Table). HWMs are not saved to disk, to spool or to the Internet.

Consequently, HWM information is lost whenever a system is reIPL'd.

Consequently, when **LIST CAPS** is issued shortly after a system is started, the listed HWMs won't be particularly useful.

Determining a Useful High Water Mark on Long-Running Systems

Taking all of the above into consideration, to learn what the highest usage is within a sysplex:

- Do not use HWM values early in the life of a sysplex member. Wait until the system has been running for a while before issuing **LIST CAPS.**
- Start a debugging session on each member of the sysplex. (Note, doing this will increment at least the base/XDC CAP by one.)
- Issue **LIST CAPS** on each sysplex member. The highest HWM seen for each CAP type will be the useful High Water Mark for that CAP type. All lesser values from other members can be disregarded.

Determining a Useful High Water Mark on Ephemeral Systems

The short answer is, you can't, at least not a "useful" one.

An ephemeral system is a z/OS that is spun up for short term use or personal use. Such a system will be "owned" and managed by an individual or a small workgroup. Such a system is brought up and taken down generally at the owner's discretion.

An ephemeral system might be created:
- As a VM guest
- As an instance in The Cloud
- As an image on a zPDT
- As a whatever

As a result, it is not feasible to determine an ephemeral system's Z/XDC usage over the long term.

CAP Limits Violations

Z/XDC's ability to enforce CAP limit violations is limited. Z/XDC does not have "phone home" code, so the scope of usage knowledge cannot reach beyond a sysplex.

Consequently, it is not possible for Z/XDC to know how many Z/XDC instances are truly in use across multiple sysplexes, especially when a customer uses ephemeral systems heavily.

However, our **Customer Licensing Agreements** contain language that states that concurrent usage limits **apply to all systems** regardless of Z/XDC's enforcement technology and regardless of whether the systems involved are long-running or ephemeral.

So for example, if:

- You have 50 current users across your long-running sysplexes,
- And also 100 ephemeral systems spun up, with 70 of those currently using z/XDC,
- And 200 z/OS's spun up in the cloud with 40 of those currently using z/XDC,

Then per the license, you would need 160 base CAPs in order not to be in violation.

The fact that our current technology cannot enforce this does not change the fact that a violation exists.

Accordingly, we at Colesoft have to rely on the honesty of our customer. So far, we have not been disappointed.

Assessing How Many CAPs You Really Need

Well, for large customers, it's difficult.

You could use Automation to issue **LIST CAPS** on your long-running systems, capture the reports, and filter and aggregate them, subject to the considerations discussed above.

But when it comes to your ephemeral systems, it's probably not possible to develop useful High Water Marks for those:

- For starters, when you run your Automation tool, there is no way to identify ephemeral systems that are not currently running.
- For that matter, how would you identify systems that are running?
- And even if you could identify such systems, it would be hit or miss as to whether enough time has elapsed since it was IPL'd for the HWM data to be meaningful.

Help COmmands LISt LKedmap

The **LIST LKEDMAP** command shows the internal structure of a load module or program object. All extents and major and minor entry names are shown. In addition, if a Binder time symbols map is available, then all classes, csects, and other ESD names also are shown.

If the module is an overlay structure, then the segment numbers are shown and **only** currently loaded segments are shown. Segments not in storage are not shown.

If the module is a program object, then classes and their attributes are shown.

Csects that have been mapped are indicated. Also, if one of the displayed csects is the current **default csect** (see **HELP COMMANDS SET QUALIFIER),** then that csect is suitably indicated.

Display related shortcut commands **(D F W),** map related shortcut commands **(M O)** and breakpoint related shortcut commands **(A K T X)** can be issued against the display lines produced by the **LIST LKED** command.

Syntax:

```
LIST LKEDMAP addressexpression ADDRESSES EXTENTS|SEGMENTS CFRIENDLY
                               OFFSETS   CLASSES          omitted
                               omitted   FULL
                                         SECTIONS|CSECTS|omitted
```

Rules:

- The addressexpression operand is required and must be given first.
- All other operands are optional. If given, they may be given in any order.
- Operands appearing above within the same column are mutually exclusive.

addressexpression
This must resolve to a virtual storage address that is located somewhere within a load module or program object. The module/object's structure is displayed.

Although this operand can be **any** address expression (PSW? R15! .SCBEXIT? etc.), the most common address expressions used are:
- modulename
- modulename.esdsymbol

ADDRESSES
This causes the starting addresses of all displayed symbols to be shown as 8-digit (or 16-digit) virtual addresses. If omitted, then the default, as established by the SET FORMAT command, is used.

OFFSETS
This causes the starting addresses of all displayed symbols to be shown as offsets relative to the start of the class or storage extent in which they occur. If omitted, then the default, as established by the SET FORMAT command, is used.

EXTENTS
SEGMENTS
These are aliases of each other. When given, they restrict the display to showing the storage extent(s) in which the load module or program object resides. Alias names (as defined by CDEs and/or LPDEs) also are shown. Information about classes, csects, and external symbols is not shown.

CLASSES
When given, this restricts the display to showing the classes within a program object. (Classes do not exist within load modules.) Extent information and alias

names also are shown. Information about csects and external symbols is not shown.

In order for the CLASSES operand to be effective, a module map needs to have been loaded via a prior MAP command.

When classes are displayed, the LIST LKEDMAP also displays the class attributes (the bind flags and the load flags). For specific information, please see **HELP COMMANDS LIST LKEDMAP CLASSES.**

SECTIONS
CSECTS
omitted
These are aliases of each other. When given (or when all filtering operands are omitted), they restrict the display to showing the control sections within a program object or load module. Extent information, alias names, and classes also are shown. Information about external symbols is not shown.

In order for the SECTIONS/CSECTS operand to be effective, a module map needs to have been loaded via a prior MAP command.

When control sections are displayed, the LIST LKEDMAP also displays the section's type (csect, common, private. part, etc). For specific information, please see **HELP COMMANDS LIST LKEDMAP SECTIONS.**

FULL
When this operand is given, all available information about a load module or program object is displayed. This includes:
- The storage extents within which the module/object resides.
- All aliases (as determined by a CDE/LPDE scan) within the module/object.
- All classes within the program object.
- All control sections within the module/object.
- All external symbols within the module/object.

CFRIENDLY
When this operand is given, the report is filtered to remove all C language **boilerplate.** All that remains are lines that describe user created code sections (csects) and external function names. All lines are removed that describe:
C support csects
Binder created csects
Data areas
etc.

This operand is intended to make it easier for customers to see those code sections that contain user written C language statements, and therefore, are sections that the customer might want to map.

CFRIENDLY can be used in combination with all other operands, but it is most useful when used with **FULL** and **SECTIONS** (or their aliases, of course).

Examples:

M FFFFFF
L LK FFFFFF CSECTS ADDRESSES
The MAP command loads the external symbol map for the current System Nucleus (usually IEANUC01). The LIST LKEDMAP command displays the contents of that map (in this case, a rather lengthy one). The display is limited to csects. (External symbols are not displayed.) The start of each csect is displayed as an 8-digit virtual address.

L LK PSW? OFFSETS
The structure of the currently executing load module is shown. The addresses of each displayed class, csect, and/or external symbol is shown as an offset relative to the start of the storage extent or class within which the name occurs.

L LK,IGC0001I
The OPEN SVC's load module and all of its alias names are shown.

MAP IGC0001I
L LK,IGC0001I
Now, the internal csects are also shown. (Its ENTRY names are not shown.)

MAP IGC0001I
L LK,IGC0001I FULL
Now, everything is shown.

Help COmmands LISt LKedmap Classes

When the **LIST LKEDMAP** command displays a program object's classes, it also displays certain of that classes attributes: its "bind flags" and its "load flags". These flags are provided by the Binder's API in the "Binder Names List" buffer, in the fields named Vn_BNL_BIND_FLAGS and Vn_BNL_LOAD_FLAGS, respectively. The Binder Names List buffer is documented in the following manuals:
- OS/390: DFSMS/MVS Program Management (SC26-4916)
- z/OS: MVS Program Management: User's Guide and Reference (SA22-7643)

Z/XDC displays the bind flags and load flags in hex with interpretation for those flag bits that might have relevance. Briefly, the following bind/load flags are defined:

Bind flags:
X'80' - **(INTERNAL)** This class is generated by the Binder.
X'40' - **(NODATA)** No data is present in this class.
X'20' - (not interpreted) This class contains varying length records.
X'10' - (not interpreted) Only descriptive data is present (no loadable text).

X’08’ - **(INITIALDATA)** This class has initial data.
X’04’ - (not interpreted) This class has a fill character.
X’01’ - **(INVALID)** This class is invalid.

Load flags:
X’80’ - **(READONLY)** This class does not need to be loaded into writable storage.
X’40’ - **(NOLOAD)** This class is not to be loaded into storage.
X’20’ - **(DEFERRED)** The loading of this class into storage is deferred.
X’10’ - **(SHARABLE)** This class is sharable.
X’08’ - **(MOVEABLE)** This class contains moveable data (no AD-CONS).

Help COmmands LISt LKedmap Sections

When the **LIST LKEDMAP** command displays a load module’s or program object’s control sections, it also displays the following to show the section’s type:

(SD) - SECTION DEFINITION: This is a control section (csect).
(PC) - PRIVATE CODE: This is an unnamed csect.
(CM) - COMMON BLOCK: This is a data area whose layout can be known and referenced from code in any control section.
(ST) - SEGMENT TABLE: This is used in the management of overlay structured load modules.
(ET) - ENTRY TABLE: This is used in the management of overlay structured load modules.
(PD) - PART DEFINITION: This a named portion of a merge class.
(UK) - UNRECOGNIZED: An unrecognized section type has been encountered.

Help COmmands LISt LOCAtion

The **LIST LOCATION** command can be used to display where Z/XDC thinks a location in storage is located.

The list of locations displayed by the **LIST LOCATION** command can change depending on the maps currently loaded and the equates currently defined.

If the location is not known by Z/XDC, the original address is displayed.

Syntax:

Z/XDC:

LIST LOCATION addressexpression
LO

Operands:

addressexpression

the address that is to have its location determined.

Notes

The output of the **LIST LOCATION** command may change for a given location.

This could happen if the information about the location that Z/XDC has changes. For example, a location may be within a load module, and that is the only information that Z/XDC has - thus the **LIST LOCATION** command can only report the load module name and the offset. However, if that load module is then mapped successfully, the **LIST LOCATION** command would now be able to provide the offset within a specific CSECT within the load module.

- The **LIST LOCATION** command is available as the **LL** shortcut command.
- It also is available as the **I** point-and-shoot command.

Examples

LIST LOCATION 0
Information about location 0 (in z/OS, the PSA) is displayed.

LIST LOCATION 12345678
If location **12345678** is within something known by Z/XDC, that information is displayed. If that location is not within anything known by Z/XDC, a warning message is shown instead.

Help COmmands LISt LOCKs

The **LIST LOCKS** command behaves differently depending on the environment in which it is invoked.

Absent dump/XDC, Z/XDC displays the names of all locks (if any) held by the program being debugged at the time that Z/XDC received control.

Under dump/XDC (i.e. when debugging a dump):

- **LIST LOCKS** will show the locks held by or requested by the **current execution unit** (CXU - See **HELP DUMPS EXECUTIONTHREAD** for more information.)
- **LIST LOCKS SYSTEM** will show all the locks held by the system.

In the case of just Z/XDC (i.e. when debugging in live storage), the **LIST LOCKS** command will:

- List those locks that will be reacquired if program execution is resumed,
- Display a message pointing out the fact that no locks are currently held.
- Additionally, if System Security (due to a **LOSTLOCKS** rule) will prohibit program execution from being resumed (TRACE and GO/GOT/GOX command), then the command will display message **DBC932** alerting you to that fact.

NOTICE! when debugging a program running live in storage (i.e. when not using dump/XDC), whenever Z/XDC receives control, if the program being debugged held

locks, those locks are always released. For the reasons behind this, and the resulting consequences and considerations, see **HELP DEBUGGING LOSTLOCKS.**

Syntax:

Z/XDC:

LIST LOCKS
LOC

dump/XDC:

LIST LOCKS
LIST LOCKS SYSTEM
LOC

Operands:

SYSTEM
This operand has meaning only when debugging dumps (i.e. only when using dump/XDC):
- When absent, the scope of the command's report is limited to locks held by the current execution unit (CXU).
- When present, all locks are reported that were held by the system from which the dump was taken.

Help COmmands LISt LOG

The **LIST LOG** command displays the following log related settings:
- The currently active **Latent Commands String** (if any),
- Whether or not the Latent Commands displays are to be included in the Scroll Area,
- The approximate number of records that can be retained in the Primary Window's Scroll Area,
- Whether Primary Window activity is **automatically** logged or must be **manually** logged,
- Whether logging will be sent to **SYSOUT** or written to a **DASD** file.

For **SYSOUT,** this command shows:
- The SYSOUT class to be used for a log written to a SYSOUT spool file,
- The RJE or local destination assigned to a log written to a SYSOUT spool file,
- Whether or not the SYSOUT spool class will be "held".

For **DASD,** this command shows:
- The dataset name to be used for a log written to a disk dataset,
- The volume serial number to be used for locating or allocating a log dataset on a disk,
- The unitname to be used when allocating a log dataset on a disk,
- Whether a preexisting log file will be overwritten or appended to.

Syntax:

LIST LOG

This command accepts no operands.

For more information, see HELP COMMANDS SET LOG.

Help COmmands LISt LOGOnid

This command is an alias of the **LIST USERID** command. See HELP COMMANDS LIST USERID for more information.

Help COmmands LISt LSEs

This command is an alias of the **LIST LSTACK** command. See HELP COMMANDS LIST LSTACK for more information.

Help COmmands LISt LSTack

The **LIST LSTACK** command can be used to display individual Linkage Stack "state" Entries **(LSEs)** currently existing in the linkage stack for any task or SRB routine running in any accessible address space. This command can also be used to display a summary list of all state entries residing in the linkage stack of any particular task or SRB. (Header and trailer LSEs are not displayed by the **LIST LSTACK** command.)

When LSEs are displayed for the task in which Z/XDC is running, the newest one listed will have the label **(ARTIFACT)** appended to it, indicating that it is an "artifact" of the debugging session. This particular LSE was created by a BAKR issued whenever Z/XDC receives control. It will be purged by a PR that will be issued whenever Z/XDC relinquishes execution (i.e. whenever the user issues a **TRACE, GO** or **END** command). Therefore, this artifact LSE will never be seen by the user's program. Therefore, this artifact LSE can be disregarded.

Syntax:

```
LIST LSTACK tcbaddress   lseaddress
     LSES   ssrbaddress  rbaddress
            omitted      scbaddress
                         nnn
                         omitted
```

- Either or both operands may be omitted.

- If both operands are given, they may be given in either order.

tcbaddress
This must be the address of a TCB located in any accessible address space. This operand identifies the task whose linkage stack or linkage stack entry is to be displayed. (Note, the **TCB#n** automatic equates created by the LIST TASKS command are perfectly good address expressions to use for this tcbaddress operand.)

ssrbaddress
This must be the address of an **SSRB** or **SSRX** located in any accessible address space. This operand identifies the suspended SRB routine whose linkage stack or linkage stack entry is to be displayed. (Note, the **SSRB#n** and **SSRX#n** automatic equates created by the LIST SSRBS command are perfectly good address expressions to use for this ssrbaddress operand.)

If both the **tcbaddress** and **ssrbaddress** operands are omitted, and if no specific linkage stack is implied by any other operand, then the **current** linkage stack is displayed by default. This will be the linkage stack that is owned either by the current task or by the current SRB routine, according to what is currently being debugged.

lseaddress
This identifies, by address, the specific linkage stack entry to be displayed. The entry is displayed in detail. The address may be any of the following:
- The address of the start of the data for a state type (PC or branch) stack entry. The addressed state entry is displayed.
- The address of the descriptor section of a state type stack entry. The state entry containing that descriptor section is displayed.
- The address of the descriptor section of a header stack entry. In this case the state entry that is pointed to by the header entry's back-pointer field is displayed. (Note, the z/Series hardware considers a pointer to a header entry to be **exactly** equivalent to a pointer to the next older state entry.)

Notes:
- If the stack's first header entry is given (or implied), then the message that is displayed is:
  THE GIVEN ADDRESS (whatever) IS OR IS ASSOCIATED WITH THE START OF THE TCB#n's LINKAGE STACK
- The **lseaddress** operand may **not** be the address of any trailer entry in a linkage stack.
- When the requested LSE is a PC state entry, the PC number displayed is just the numeric portion of the PC number, which is the numeric value of the PC number with any flag bits removed. Generally, the numeric portion of the PC number is the same as the value used with the PC instruction. However when ALRF is enabled and bit 44 of the PC number is on, the numeric portion is formed by discarding bit 44 and shifting all bits from the left 1 bit to the right.
- When the requested LSE is a PC state entry, the PC number is displayed as LFX-LSX-EX when ALRF is enabled, but as LFX-EX when ALRF is disabled.

The linkage stack entry to be displayed may be located on any linkage stack owned by any task or SRB routine located either in the current address space or in any other accessible address space.

When an **lseaddress** operand is given and both the **tcbaddress** and **ssrbaddress** operands are omitted, then all tasks and SRB routines in the address space are searched until the given linkage stack entry is found.

rbaddress
This must be the address of any Request Block queued to any TCB in the current address space. The state type linkage stack entry (LSE) that is pointed to by (or otherwise indicated by) that RB's XSB is displayed.

When both an **rbaddress** operand and a **tcbaddress** operand are given, then the RB search is limited to the given TCB.

scbaddress
This must be the address of a STAE Control Block that is queued to any TCB in the current address space. The state linkage stack entry that is pointed to by (or otherwise indicated by) the SCB's SCBX is displayed.

When both an **scbaddress** operand and a **tcbaddress** operand are given, then the linkage stack entry pointed to by the SCB's SCBX must be in that TCB's linkage stack.

Note, only SCBs that represent ARRs, STAEs, ESTAEs, ESTAEXs, STAIs, and ESTAIs contain useful pointers to linkage stack entries. SCBs that represent FESTAEs do not.

nnn
This must be a decimal number that specifies, by sequence, the state type LSE to be displayed. In locating the desired LSE, Z/XDC counts the oldest state entry as number 1, and it does not count header and trailer entries.

When an **nnn** operand is given and both the **tcbaddress** and **ssrbaddress** operands are omitted, then the current program's linkage stack is searched. If Z/XDC is debugging an SRB routine, then that would be the linkage stack owned by the SRB routine. Otherwise, it would be the current task's linkage stack.

lseaddress, rbaddress, scbaddress, and **nnn** all omitted
If no operand is given that selects a specific stack entry, then Z/XDC instead generates a display showing a summary of all state type stack entries that currently exist in the specified or default linkage stack. A one-line message is issued for each LSE found.

In addition, **LSE#n** equates are (re)generated to label the starting addresses of each **state** type LSE that is found. Also, an **LSE#0** equate is generated to label the linkage stack's first header entry. These equates can be used in subsequent **LIST LSTACK** (and other) commands to reference specific stack entries.

When all operands are omitted, then the default linkage stack is displayed. That's the linkage stack that is owned by the task or SRB mode routine that is currently being debugged.

For more information about linkage stacks, please refer to one of the following IBM manuals:
- **IBM ESA/370 Principles of Operation** (SA22-7200)
  Chapter 5: "Program Execution"
  Section: "Linkage-Stack Introduction".
- **ESA/390 Principles of Operation** (SA22-7201)
  Chapter 5: "Program Execution"
  Section: "Linkage-Stack Introduction".
- **z/Architecture Principles of Operation** (SA22-7832)
  Chapter 5: "Program Execution"
  Section: "Linkage-Stack Introduction".

Automatic Equates

When the **LIST LSTACK** command is used to display a summary list of LSEs, it also automatically generates a series of numbered equates named **LSE#n** to label the LSEs being displayed. **LSE#1** labels the oldest state LSE, and **LSE#0** labels the linkage stack's first header LSE. (Other header LSEs, if any, are not labeled.)
If **LSE#n** equates already exist from a previously issued **LIST LSTACK** command, then those equates are deleted before the new **LSE#n** equates are created. For more information, please see:

- HELP EQUATES
- HELP EQUATES BUILTIN AUTOMATIC
- HELP COMMANDS EQUATE

Since an **LSE#0** equate is also generated to label the linkage stack's first header entry, even the display of empty stacks will cause at least one equate to be generated. (Equates are **not** generated for 2nd and subsequent header entries. They also are not generated for trailer entries.)

Examples:

LIST LSTACK
LIST LSTACK 21C%
These two commands are nearly equivalent:

- When Z/XDC is being used to debug a task mode program, both commands display a summary list of all state type Linkage Stack Entries (LSEs) that are queued to the current task.

- When Z/XDC is being used to debug an SRB mode routine:
  - The first command displays a summary list of all LSEs that are in use by the SRB routine.
  - The second command fails because location X'0000021C', the PSATOLD field, is zero'd when SRB routines are running.

LIST TASKS JES2
LIST LSTACK TCB#3
The **LIST TASKS JES2** command, produces a display of all TCBs located in JES2's address space. in addition, the command creates a series of equates **(TCB#n)** labeling the starting addresses of each of the displayed TCBs.

Then the **LIST LSTACK TCB#3** command displays a summary of all state type linkage stack entries that are queued to the TCB (in JES2's address space) labeled by the **TCB#3** equate.

In addition, the **LIST LSTACK** command creates a series of **LSE#n** equates to label the start of each LSE displayed. (All prior **LSE#n** equates, if any, are discarded.)

LIST SSRBS 1

LIST LSTACK SSRB#1 [or **SSRX#1]**
If Z/XDC is running authorized, and if System Security permits the access, then the **LIST SSRBS 1** command displays a list of all SSRBs suspended in the Master Scheduler's address space. It also creates a series of **SSRB#n** and **SSRX#n** equates labeling the locations of all the SSRBs and SSRXs that were found.

Then the **LIST LSTACK SSRB#1** command (or **LIST LSTACK SSRX#1** command) displays a summary list of all state type entries found on that SRB's linkage stack. Also, the command (re)creates a set of **LSE#n** equates to label the starting addresses of all the state entries displayed.

DMAP XDCMAPS.PSA
DMAP .TCB
USING PSA 0
USING TCB .PSATOLD?
LIST LSTACK .TCBJSTCB%
The **DMAP** and **USING** commands for the PSA load and place a dsect map of the Prefixed Storage Area onto low storage.

The **DMAP** and **USING** commands for the TCB load and place a dsect map of a Task Control Block onto the current program's TCB.

Finally, the **LIST LSTACK .TCBJSTCB** command displays a summary of all state type linkage stack entries that are queued to the jobstep task.

In addition, the **LIST LSTACK** command creates a series of **LSE#n** equates to label the start of each LSE displayed. (All prior **LSE#n** equates, if any, are discarded.)

LIST LSTACK 3
LIST LSTACK LSE#2
LIST LSTACK LSE#2 TCB#7
LIST LSTACK LSE#2 SSRB#2
LIST LSTACK LSE#2 SSRX#2
LIST LSTACK RB#1
LIST LSTACK RB#1-20?+74?
LIST LSTACK TCB.TCBSTAB%%%%
LIST LSTACK SCB#4
These commands all produce detailed displays of a specific state type entry found in the linkage stack for a task, as follows:

- **LIST LSTACK 3:** This displays the third state entry in the linkage stack for the current task or SRB routine.

- **LIST LSTACK LSE#2:** This displays the state entry that is labeled by the equate **LSE#2** (a linkage stack's second entry). Please note that this command cannot be issued successfully prior to the creation of the **LSE#2** equate via a prior **LIST LSTACK** command.

- **LIST LSTACK LSE#2 TCB#7:** This displays the state entry that is labeled by the the **LSE#2** equate. This state entry **must** exist on the linkage stack that is owned by the TCB labeled by the **TCB#7** equate.

- **LIST LSTACK LSE#2 SSRB#2**

- **LIST LSTACK LSE#2 SSRX#2:** These two commands both do the same thing. They display the state entry that is labeled by the the **LSE#2** equate. (Presumedly, this state entry exists on the linkage stack that is owned by the second SRB routine listed by the most recently issued **LIST SSRBS** command.)

- **LIST LSTACK RB#1:** This displays the state entry that is pointed to by the XSB that is linked to the oldest Request Block for the task for which a **LIST RBS** command was most recently issued.

- **LIST LSTACK RB#1-20?+74?:** This produces the same display as **LIST LSTACK RB#1.** The given address expression **(RB#2-20?+74?)** resolves to the address of the linkage stack entry that is pointed to by the XSB that is pointed to by the **RB#2** equate.

- **LIST LSTACK .TCBSTAB%%%%:** This displays the state type stack entry that is associated with the 4th **newest** SCB existing for the TCB that is mapped by a TCB dsect.

- **LIST LSTACK SCB#4:** This displays the state type stack entry that is associated with the 4th **oldest** SCB existing for the task whose STAE Control Block queue was most recently displayed by a **LIST SCBS** command.

Note, none of these commands (re)create **LSE#n** equates. That's because they only display individual Linkage Stack Entries. Only **LIST LSTACK** commands that produce summary lists of an entire Linkage Stack produce **LSE#n** equates.

LIST LSTACK RB#2 TCB#3
LIST LSTACK TCB#3 RB#2
These two commands are equivalent. They both are intended to display the LSE that is associated with the 2nd oldest Request Block that is chained from TCB number 3. Note, this command will fail if the **RB#2** equate labels a Request Block that is chained to a different TCB than TCB#3. To be safe, you might use the following command sequence:

```
LIST TASK
LIST RBS TCB#3
LIST LSTACK RB#2 TCB#3
```

This will ensure that the correct **TCB#n** and **RB#n** equates are created.

Help COmmands LISt MAIntenance

This command is an alias of the **LIST XDC** command. See HELP COMMANDS LIST XDC for more information.

Help COmmands LISt MAPLibs

"MAPLIBs" is Z/XDC's name for those sequential and partitioned datasets in which the MAP and DMAP commands can find ADATA for building source image maps of programs and control blocks. The MAPLIB datasets may be any of the following:

- SYSADATA output created by any mainframe Assembler that supports producing IBM-compatible ADATA. As of this writing, there are three such assemblers that I know of:
  - IBM's High Level Assembler (www-01.ibm.com/software/awdtools/hlasm). Use PARM=ADATA.
  - Tachyon Software's Cross Assembler and z/Assembler (www.tachyonsoft.com/txaover.html).
  - Dignus' Systems/ASM (www.dignus.com).

- A GOFF file containing embedded ADATA. (If using IBM's High Level Assembler, then specify "PARM='GOFF(ADATA)'".)

- A file containing streamed ADATA produced by a PC based mainframe Assembler and then binary-uploaded to the mainframe into a RECFM=FB file of arbitrary LRECL.

A "MAPLIB list" is a list of MAPLIB files. In Z/XDC multiple MAPLIB lists can be defined, but at most only one MAPLIB list can be active. For more information, see HELP MAPS ADATA MAPLIBS.

MAPLIB lists can be created and managed by the SET MAPLIBS command. They can be displayed by the LIST MAPLIBS command. They can be purged by the DELETE MAPLIBS command.

The **LIST MAPLIBS** command can be used to display information about any or all MAPLIB lists.

Also, the **LIST MAPLIBS ACTIVE** command automatically reactivates the usability of MAPLIB sequence numbers for possible use by the DELETE MAPLIBS command. See HELP COMMANDS DELETE MAPLIBS for more information.

Syntax:

```
LIST MAPLIBS ACTIVE
             LISTS
             name
             ALL
```

Default: **ALL**

ALL is mutually exclusive with all other operands.

All other operands may be given in any combination and order.

ACTIVE
This displays the libraries and datasets that constitute the active MAPLIBS list.

LIST
This displays the names of the saved (and therefore inactive) MAPLIBS lists.

ALL
This displays both the active MAPLIBS list as well as both the names and the contents of all saved MAPLIBS lists.

name

This displays the contents of the named MAPLIBS list.

Help COmmands LISt MAPS

The **LIST MAPS** command displays the names and locations of all currently defined module maps, csect maps, and dsect maps. The dsect maps are shown in their search order. If a csect map is flagged INACTIVE, then it is in an overlay module in a segment that currently is not in storage.

LIST MAPS also shows:

- The map's type: **LKED, CSECT** or **DSECT**
- The map's data type: **ESD, SYM, ADATA** or **DWARF**
- Whether a map is **global** across all address spaces or is **local** to a particular address space
- Whether a map is case-sensitive **(c)** or not
- Whether a map represents a Privately Loaded module **(p)** or not
- Whether a particular csect map as been designated as the default map **(q)** by the **SET QUALIFIER** command
- For dsects, whether the map has been assigned a fixed base, a floating base, or remains with no base assigned
- For floating dsects, what the basing address expression is

Syntax:

LIST MAPS

This command does not accept operands.

Help COmmands LISt MEmoryobjects

The **LIST MOBJECTS** command displays information about various above-the-bar memory objects:

- located in the above-the-bar common area.
- located in the above-the-bar sharable area.
- located in any accessible address space.
- shared into any accessible address space from the above-the-bar sharable area.

If memory objects located in an accessible address space are reported, this command also reports statistics about the memory objects in that address space, and the address space's above-the-bar **MEMLIMIT** and how much of that limit has been used by memory objects.

Syntax:

LIST MEMORYOBJECTS **[operands]**
MOBJECTS

```
operands are:

   [ ACCESSIBLE ] [ COMMON ] [ PRIVATE ] [ SHARED ] [ ELIGIBLEFORSHARE | ES |
   SE ]

   [ NOCHECK | NC | NO | READ | READONLY | RO | WRITE ]

   [ EQUATES | NOEQUATES ]

   [ XDCDETERMINED | RETRY | ERROR | KEY=n ]

   [ SUMMARY | DETAIL ]

   [ aspaceref ]
```

Operands

ACCESSIBLE
COMMON
PRIVATE
SHARED
ELIGIBLEFORSHARE
ES
SE
These operands are used to select the type(s) of memory object(s) to be displayed. Any combination of types may be specified.
The following types may be specified:
- **ACCESSIBLE** - information about all memory objects that are accessible to the aspace will be displayed. (this is the default if no types are specified).
- **PRIVATE** - information about all memory objects that have been defined in the aspace will be displayed.
- **SHARED** - information about all memory objects that have been shared into the aspace will be displayed.
- **COMMON** - information about all memory objects that are common to all aspaces will be displayed.
- **ELIGIBLEFORSHARE** - information about all memory objects that are eligible for sharing to an aspace will be displayed. (Note that **ES** and **SE** are aliases of this type).

NOCHECK | NO | READ | READONLY | RO | WRITE
These operands (only one is allowed) can be used to indicate the level of access to the memory object this is required. Unless NOCHECK is specified, only memory objects that can be accessed are displayed.
The following access level keywords are recognized:
- **NOCHECK** - no access checks will be done.
- **NC** - an alias of **NOCHECK**
- **NO** - no access to this memory object is allowed.
- **READ** - the memory object can be at least read. (this is the default if this operand is omitted).
- **READONLY** - the memory object can be read (only). (i.e. not written to)
- **RO** - an alias of **READONLY**
- **WRITE** - the memory object can be written to (and read).

EQUATES | NOEQUATES

These operands (only one is allowed) can be used to control the generation of equates by this command. Note that if equates are generated, any previously-existing equates with the same prefixes as those that will be generated are deleted.
The following equates control keywords are recognized:
- **EQUATES** - equates will be generated. (this is the default if this operand is omitted).
- **NOEQUATES** - equates will not be generated.

XDCDETERMINED | RETRY | ERROR | KEY=n
These operands (only one is allowed) can be used to control the PSW key used to determine the accessibility of the memory object. The key can be derived from a PSW or specified explicitly.
The following PSW sources can be specified:
- **XDCDETERMINED** - the PSW key will be taken from the retry or error PSW depending on whether the error and retry levels are the same (retry PSW used) or different (error PSW used). (this is the default if all of the access-key-related operands are omitted).
- **RETRY** - the PSW key will be taken from the retry PSW.
- **ERROR** - the PSW key will be taken from the error PSW.

Alternatively, a specific key can be specified using the **KEY=n** operand. In this case, a **n** must be a number from 0 to F (hex) (15 decimal).

SUMMARY | DETAIL
These operands (only one is allowed) can be used to control the level of displayed information.
SUMMARY means that a single line of information is displayed for each selected memory object. (this is the default if these operands are omitted).
DETAIL means that several lines of information is displayed for each selected memory object.

aspaceref
This identifies the address space for which private and shared memory objects are to be displayed.
(If this operand is omitted, the current aspace is used).
Briefly, this operand can be any of the following:
- A jobname.
- A keyword indicating the home space or the retry level or error level, primary, secondary, or instruction execution address space. Examples: HOME PASID ESASID, IASID, etc.
- An ASID number.
- An address expression pointing to an ASID number.
- A register containing an ASID number.
- The name of an equate or dsect that has been assigned to represent storage in an address space. (The assigned address space is used.)

For detailed information, see HELP COMMANDS SYNTAX ASIDS.

Notes

If there is no current execution unit (CXU), which can happen if dump/XDC is in use, processing as if **NOCHECK** is in effect.

Share-Eligible memory objects can not be accessed by a specific address space until they are made accessible by the **IARV64 REQUEST=SHAREMEMOBJ,...** Assembler macro.

Keywords can be abbreviated as long as they remain unique.

If a particular operand is specified more than once, the first instance is treated as that operand, and the second instance is treated as an aspaceref (usually a jobname but in some cases a hexadecimal ASID).
(Note that in the case of memory object types, (for example COMMON or PRIVATE) each keyword is considered individually, but in all other cases, any keyword in a group (for example SUMMARY or DETAIL) is considered duplicated if any prior keyword in that group has been seen.)

Operands can be entered in any order.

Examples

LIST MOBJECTS JES2 KEY=0
Memory objects accessible by the JES2 address space when executing code is in key 0 are displayed.

LIST MEMORYOBJECTS 1
Memory objects accessible by the Master Scheduler’s address space are displayed.

LIST MEMORYOBJECTS C NC
all common memory objects are displayed.

LIST MEMORYOBJECTS SE
all share-eligible memory objects are displayed.

LIST MEMORYOBJECTS EIASID P
Memory objects defined within the error level, instruction execution address space are displayed.

LIST MEMORYOBJECTS PASID
Memory objects accessible from the retry level, primary address space are displayed.

LIST MEMORYOBJECTS .ASCBASID KEY=8
Assuming that the ASCB dsect map has been loaded and assigned to represent an Address Space Control Block of interest, then the memory objects accessible from the address space that the ASCB describes and that can be at least read by code using a PSW key of 8 are displayed. (In this case, the command’s operand is resolved as an address expression, and the contents of the two bytes to which that expression points is what is used.)

LIST MEMORYOBJECTS CR3
Memory objects accessible from the retry level, secondary address space are displayed.

LIST MEMORYOBJECTS ECR3 PSW=ERROR
Memory objects accessible from the error level, secondary address space are displayed.

LIST MEMORYOBJECTS R5 NC
The two low order bytes of register R5 are interpreted as an ASID, so the memory objects visible to that address space are displayed.

Help COmmands LISt MMefdsns

The **LIST MMEFDSNS** command can be used to display the list of currently defined Module Mapping Extraction Files.

MMEF files contain ESD data (mainly) and other mapping data (if any, and if requested) extracted from load libraries located on foreign systems. It is expected that this mapping data extraction file will accompany a dump created on the same foreign system as the extraction file. For more information about **MMEF** files, see **HELP DUMPS MAPPINGDATA MMEFFILES.**

For information on creating MMEFDSN tokens, and details about what they do, see **HELP COMMANDS DEFINE MMEFDSN.**

Syntax:

```
LIST MMEFDSNS SUMMARY
              DATASETS
              DDNAMES
              omitted
```

Operands:

SUMMARY
omitted
This produces a summary report of the contents of the MMEF extraction file.

DATASET
In addition to the SUMMARY report, a list is produced showing the names or all load

libraries whose mapping data extractions are included in the file.

DDNAMES
In addition to the DATASETS report, The extracted libraries are shown grouped according to the **LIBn** ddname to which they were concatenated.

Help COmmands LISt MObjects

This command is an alias of the **LIST MEMORYOBJECTS** command. See HELP COMMANDS LIST MEMORYOBJECTS for more information.

Help COmmands LISt MSgs

The **LIST MSGS** command redisplays the set of status messages that are generated (but not always displayed) whenever Z/XDC is called. These messages contain the following information.

- Z/XDC’s version number.
- Whether or not Z/XDC is running authorized.
- Under which RB and TCB Z/XDC is running.
- Whether the current error time and retry level environments are the same or different.
- A warning if the high halves of the error level and retry level 64-bit registers are not available (occurs in z/OS systems when SDWALOC31=NO is in effect).
- Which ESTAE or ESTAI Z/XDC is running under.
- The reason why Z/XDC was called. This might be:
  - An ABEND code
  - A breakpoint identification
  - A #DIE message
- A warning (when engaged in **FRR mode** debugging) when locks have been lost.
- A warning when the xxxSRVER job is down.

Note, when the error level and retry level environments are different, an additional warning message is displayed to emphasize the importance of being aware of and understanding that condition. (See HELP EXECUTIONLEVELS for more information.)

Syntax:

LIST MSGS

Help COmmands LISt NOtes

The **LIST NOTES** command displays a list of annotated addresses. The entries in this list are created by the NOTE command.

The text of the notes can be changed by overtyping. Individual notes can be removed from the list via the **P** shortcut.

Syntax:

LIST NOTES

This command accepts no operands.

For more information, see:
HELP COMMANDS NOTE
HELP FULLSCREEN NOTES

Help COmmands LISt NT

The **LIST NT** command displays z/OS name token entries that are visible from a task, address space, or the system. These entries are the name token pairs maintained by IBM name token services.

Token names can contain any byte value. In the normal display, non-printable name bytes are cleaned for display. Use the **NAMEHEX** operand when the exact name bytes must be inspected.

The display produced by the LIST NT command shows the following information:
- **ASID:** The target address space id. System-scope rows show 0000.
- **TCB#:** The task number used by Z/XDC, when a task-scope row is being shown.
- **PROGRAM:** The task or address-space owner name.
- **NAME:** The token name, cleaned for display.
- **NAMEHEX:** The token name bytes in hexadecimal, when NAMEHEX is specified.
- **TOKEN:** The 16-byte token value in hexadecimal.

Syntax:

LIST NT scope ASID=aspaceref TCB=tcbref NAMEHEX
omitted JOBNAME=jobname

scope
This selects which name/token scope is to be displayed. It may be one of the following:
- **TASK** displays task-scope name/token entries.
- **AS** displays address-space-scope name/token entries.
- **SYSTEM** displays system-scope name/token entries.

- **ALL** displays task-scope, address-space-scope, and system-scope entries.

When the scope is omitted, TASK is assumed.

ASID=aspaceref
JOBNAME=jobname
These operands select the target address space for TASK, AS, and ALL scope. JOBNAME= is accepted as a user-facing alias for ASID=jobname.

The aspaceref follows the normal Z/XDC address-space reference rules: job names, ASID numbers, ASID keywords, registers, equates, dsects, and address expressions may be used where they resolve to an address space. REAL storage is not valid for LIST NT.

For detailed information, see HELP COMMANDS SYNTAX ASIDS.

TCB=tcbref
This operand selects a task for TASK or ALL scope. It is not valid with AS or SYSTEM scope. The tcbref may be a TCB#n ordinal or address expression that resolves to a TCB.

A TCB#n ordinal is resolved within the target ASID named on the same command. It does not require automatic equates from a prior LIST TASKS command.

For TASK and ALL scope, the tcbref may also be given without the TCB= keyword.

NAMEHEX
This operand changes the NAME column to NAMEHEX and displays each token name as hexadecimal bytes. The TOKEN column is always displayed in hexadecimal.

Rules:

- The scope operands TASK, AS, SYSTEM, and ALL are mutually exclusive.
- ASID= and JOBNAME= are mutually exclusive.
- TCB= is valid only with TASK or ALL.
- SYSTEM does not accept ASID=, JOBNAME=, or TCB=.
- When ALL is used without TCB=, the task-scope section scans the TCBs in the selected address space. When TCB= is used, the task-scope section is limited to that task.

Examples:

LIST NT
Displays task-scope name/token entries for the current target address space and current task.

LIST NT NAMEHEX
Displays the same task-scope entries, but shows token names as hexadecimal bytes.

LIST NT ASID=0039 TCB=TCB#4
Displays task-scope name/token entries for TCB#4 in ASID 0039.

LIST NT ALL ASID=0039 TCB=TCB#4
Displays the task-scope entries for TCB#4 in ASID 0039, then the address-space-scope entries for ASID 0039, and finally the system-scope entries.

LIST NT AS JOBNAME=JCNT001
Displays address-space-scope name/token entries for the address space named JCNT001.

LIST NT SYSTEM
Displays system-scope name/token entries.

Related topics:
- HELP COMMANDS LIST ASID
- HELP COMMANDS LIST TASKS

Help COmmands LISt OPErands

This command displays the apparent current values in the storage operands of a designated machine instruction.

The "designated" instruction may be:
- That which is pointed to by the retry level PSW[E] and so is about to be executed next. (The **before** values are shown.)
- That which was about to be executed the **prior** time that Z/XDC received control. (Those operands' **current** values are shown.)
- That which is located at any given address. (The resulting display can easily be **entirely wrong!)**

Caution! The accuracy of the displayed values is dependent upon whether or not the registers and data fields referenced by a target machine instruction are the same now as they were (or will be) when said instruction was (or will be) executed. In particular, it does not make much sense to use this LIST OPERANDS command against an instruction or statement located in a different program than what is currently executing. This is mainly because the LIST OPERANDS command can only use a register's current value when resolving a reference to an instruction's operands. There is no way for Z/XDC to know what those registers or variables will contain when execution actually reaches said instruction.

So the LIST OPERANDS command is useful mainly to show:
- The current values in the operands that are about to be used or changed by the current (and not yet executed) machine instruction.
- The **current** values of the operands of the machine instruction that was about to be executed the **prior** time that Z/XDC received control.

Note, this command displays only **storage** operands. The contents of register operands are not displayed. (The **LIST RWREGS** command [for example] can be used to display that information.)

Syntax:

LIST OPERANDS machineinstructionaddress
PRIOR
CURRENT
omitted

Shortcut: LO

machineinstructionaddress
This is the address of any machine instruction. This causes the values to be displayed for each of the instruction's operands.

WARNING! Z/XDC uses **current** retry level register values when resolving the instruction's operand addresses. Consequently, if the execution context of the target machine instruction differs from your program's current execution context, then the resulting display will likely be entirely wrong! So beware.

CURRENT
omitted
This causes the operands to be displayed for the current machine instruction that is about to be executed at the retry level execution address. This shows their values **before** the instruction is executed.

This display is guaranteed to be accurate.

PRIOR
This causes the operands to be displayed for the instruction that was current the previous time that Z/XDC received control. It shows those operands' **current** values. This might be as they were set by said previous instruction, but only if those values were not changed by subsequent instructions that may have been executed between then and now. For more information, see the comments below.

For single instruction tracing (as is done by the **TRACE I** command), this display is highly likely to show correct information. However, for other kinds of tracing, as well as for trapping in general, multiple machine instructions will have been executed between now and the prior time that Z/XDC received control. Consequently, this display will not necessarily show the contents of that prior instruction's operands as that instruction left them. It only shows information based upon **current** retry level register values.

This Bears Repeating!

The more remote from current execution an instruction is, the less likely the resulting display will show correct information.

Example:

L OPERANDS;;L OPERANDS PRIOR

This is a good command string to set up in a Watch Window. Together, the two commands will produce two displays:

- The first will show the **before values** of the machine instruction that is about to be executed.
- The second will (likely) show the **after values** of the machine instruction that was executed the previous time that Z/XDC received control.

So in this example, when single stepping through your code **(TRACE I),** you will see both the before and after values for each machine instruction.

Help COmmands LISt OPTimization

The **LIST OPTIMIZATION** command displays the situations under which Z/XDC's Fullscreen Support will attempt to optimize the writing of fullscreen displays. This optimization involves guessing (essentially) which fields of the display have changed (as compared to the previous display) and which fields have not. then only the changed fields (instead of the entire display) are sent to the terminal. This can substantially reduce the time it takes for the terminal to receive each new display.

Syntax:

LIST OPTIMIZATION

This command accepts no operands.

The OPTIMIZATION setting can be saved in your session profile. It can be changed by the SET OPTIMIZATION command. It also can be displayed and changed by Z/XDC's "Profile Menuing System". For more information, please see:

HELP COMMANDS SET OPTIMIZATION
HELP PROFILES MENU

Help COmmands LISt PARseorder

This command is available only for customers who have Licensed Features for debugging one or more of the soon to be supported **High Level Languages.** For more information, see HELP SUPPORT FEATURESANDCAPS.

The syntax rules for variables differ from one language to the next. Consequently, one parser cannot be used to parse a variable for all languages.

Accordingly, Z/XDC provides a unique parser for each supported language type, and it allows you to define which parsers will be used to resolve a given variable name. This definition is called the **Parse Order,** and this **LIST PARSEORDER** command displays this definition.

More specifically, this **LIST PARSEORDER** command displays:
- Which Parsers (ASM or CEE [or future]) are called for parsing a string,
- The order in which the Parsers are invoked,
- And which parser generates the error message should they all fail. (This will always be the last parser listed in the Parsing Order, and it is quite permissible for that parser to be the same as a previously listed parser.)

A global Parse Ordering can be set and displayed and saved in your session profile. It can also be overridden for individual commands. More specifically, the Parsing Order:
- Is set by the **SET PARSEORDER** command,

- Is displayed by the **LIST PARSEORDER** command (described here),
- And is saved in your session profile by the **PROFILE SAVE** command.

It also can be displayed, set and saved by the **Profile Menuing System.**

Syntax:

LIST PARSEORDER

This command accepts no operands.

For detailed information about setting a Parse order, see HELP COMMANDS SET PARSEORDER.

Help COmmands LISt PC#

This command displays a list of the available program call numbers (PC#s) for an address space.

Syntax:

LIST PC# [space] [selection] [filter] [control]

Where:

space is:
[ASID=asid]
(Only one may be specified)

selection is:
[#=information]
[ALLCLASSES]
[[SFT] [SYSTEM] [PRIVATE]]
[address-expression ...]
(Only one may be specified)

filter is:
[SSWITCH=value]
[LOCATION=value]
[ARR=value]
[TYPE=value]
[PARMS=value]
[OWNER=asid]
[RTN=address]
(None, one, or several may be specified)

control is:
[NOEQUATES]
[SUMMARY | DETAIL]
[RESOLVE | NORESOLVE]

(None, one, or several may be specified)

Notes:

- Operands can be entered in any order.
- Operands can be abbreviated as long as they remain uniquely identifiable
- If an address expression that resembles an operand is specified, you can have it unambiguously treated as an address expression by specifying '+0' after it. For example 'SYSTEM+0' will be treated as an address expression.
- If no address space is specified on the command (the **ASID=...** operand) then:
  - The **current** address space will be used.
- If no selection operands are specified on the command then:
  - If no filtering operands have been specified, the command is processed as if the the **SYSTEM PRIVATE** selection operands were specified.
  - If any filtering operands have been specified, the command is processed as if the the **ALLCLASSES** selection operand was specified.
- If multiple filtering operands are specified, they are logically AND'd together - only PC#s that pass all filtering criteria will be displayed.
- The **SSWITCH=asid** and **OWNER=asid** filtering operands have different uses. This is explained in detail below.
- Owner information may not be available in this environment (To obtain owner information, control blocks in the PCAUTH address space must be available). If owner information is not available:
  - Owner information is not displayed in a DETAIL display
  - an attempt to filter by owner will be rejected with an error message.
- If neither **SUMMARY** or **DETAIL** are specified then:
  - If any selection operands that result in a list are used (or defaulted) (these are range numbers (b-c), all EX numbers (d**), ALLCATEGORIES, SFT, SYSTEM, and PRIVATE) the **SUMMARY** list format is used - In other cases the **DETAIL** list format is used
- Unless the NOEQUATES operand is used, or the environment is not authorized (see below) equates will be produced if any selection operands that result in a list are used (or defaulted) (these are range numbers (b-c), all EX numbers (d**), ALLCATEGORIES, SFT, SYSTEM, and PRIVATE).
- PC#s can be split up into 3 classes:
  Class 1 is those PC#s supplied by the operating system. These are available to all address spaces (jobs, etc) in the system.
  In this command these are **SFT** PC#s.
  Class 2 is those PC#s supplied by other facilities (i.e. not the operating system) and are available to all address spaces (jobs, etc.) in the system.
  In this command these are **SYSTEM** PC#s.
  Class 3 is those PC#s supplied by other facilities (i.e. not the operating system) that are only available to selected address spaces (jobs, etc.) in the system. Note that these can only be displayed by this command if the nominated (or defaulted) address space has them available.
  In this command these are **PRIVATE** PC#s.

Operands:

ASID=asid

the address space for which the program call number list is to be produced. If this operand is omitted, the 'current' address space is used. Note that **JOBNAME= ASNAME= ASPACE** are all aliases of **ASID=.** An asid can be specified in many ways.

For a complete list, see **HELP COMMANDS SYNTAX ASIDS**

#=information
Display PC# information about the specified number(s) and/or number rangers. This operand can be either one entry or several entries. If there is just one entry, the entry can optionally be enclosed in parenthesis ('()'). If more than one entry, the entries **must** be enclosed in parenthesis and separated by blanks or commas. Each entry is of one of these formats:

a	- A 1-to-8-digit hex number, Leading zeros are assumed. The maximum value of the number is 7FFFFFFF. the single PC# will be displayed.
b-c	- Two 1-to-8-digit hex numbers. Leading zeros are assumed. The maximum value of each number is 7FFFFFFF. c **must** be GE b. All PC#s from b to c (inclusive) will be displayed.
d**	- d must be a 1-6 digit hex number Leading zeros are assumed. The maximum value of the number is 7FFFFF. All PC numbers from d00 to dFF (inclusive) will be displayed (i.e. all EXs for a given LX)

ALLCLASSES
This operand forces the display of all program call numbers available to the nominated (or defaulted) address space. It has the same effect as specifying **SFT SYSTEM PRIVATE**

SFT
This operand controls the displaying of program call numbers that are supplied by the operating system. (the program call numbers that are in the System Function Table (SFT)).
Specifying **SFT** will cause these program call numbers to be displayed.
(MVS, OS, and ZOS are aliases of SFT)

SYSTEM
This operand controls the displaying of program call numbers that are system-wide. (that is, they are available to all address spaces). (this is a **selection** operand). Note that program call numbers that are in the **SFT** class will not be displayed by this operand, even though they are logically system PC numbers.
Specifying **SYSTEM** will cause these program call numbers to be displayed.

PRIVATE
This operand controls the displaying of program call numbers that are only made available to specific address spaces, only if the program call number is available to the nominated (or defaulted) address space. (this is a **selection** operand).
Specifying **PRIVATE** will cause these program call numbers to be displayed.

address expression ...
Displays PC# information for a PC# with the nominated routine address. Note that this operand can be specified multiple times and PC# information will be displayed for each one.
Note that the PC# displayed may not be the one wanted. Z/XDC builds a table of unique routine addresses and ASIDs based on previous **LIST PC#** commands issued in this debugging session, and the PC# associated with a routine address (for routines that support more than one PC#) is random.
An alternative way to display PC#s with a particular routine address is to use the **RTN=...** and **SSWITCH=...** filtering parameters.

SSWITCH=value
This operand can be used to restrict the displayed PC#s to only those that pass the indicated space-switch criteria:

```
  SSWITCH=NO       - only selects PC#s that do not space-switch.
  SSWITCH=YES      - only selects PC#s that space-switch.
  SSWITCH=asid     - only selects PC#s that space-switch to the indicated address
                     space. An asid can be specified in many ways. For a complete
                     list, see HELP COMMANDS SYNTAX ASIDS
```

LOCATION=value

This operand can be used to restrict the displayed PC#s to only those that pass the indicated routine location criteria:

```
LOCATION=COMMON   - only selects PC#s that commence execution in common storage.
LOCATION=PRIVATE  - only selects PC#s that commence execution in private storage.
```

ARR=value

This operand can be used to restrict the displayed PC#s to only those that pass the indicated ARR (associated Recovery Routine) type:

```
ARR=NO            - only selects PC#s that do not have an ARR.
ARR=YES           - only selects PC#s that have an ARR.
ARR=COMMON        - only selects PC#s that have an ARR, and the ARR is in common
                    storage.
ARR=PRIVATE       - only selects PC#s that have an ARR, and the ARR is in private
                    storage.
```

TYPE=value

This operand can be used to restrict the displayed PC#s to only those that pass the indicated PC type:

```
TYPE=BASIC        - only selects PC#s that are a basic PC (i.e. do not use the
                    linkage stack).
TYPE=STACKING     - only selects PC#s that are a stacking PC (i.e. use the linkage
                    stack).
```

PARMS=value

This operand can be used to restrict the displayed PC#s to only those that pass the indicated parameter type:

```
PARMS=NO          - only selects PC#s that have both parm 1 and parm 2 zero.
PARMS=YES         - only selects PC#s that have either (or both) parm 1 and parm 2
                    nonzero.
```

OWNER=asid

This operand can be used to restrict the displayed PC#s to only those that pass the indicated space-switch criteria:

```
 OWNER=asid       - only selects PC#s that are owned by the indicated address
                    space. An asid can be specified in many ways. For a complete
                    list, see HELP COMMANDS SYNTAX ASIDS
```

RTN=address

This operand can be used to restrict the displayed PC#s to only those that pass the indicated routine address criteria:

```
 RTN=address      - only selects PC#s that have the indicated PC routine address
```

NOEQUATES

This operand stops the production of equates by this command. As described below, under many circumstances equates will be produced by the **LIST PC#** command. Use of this operand will stop the production of equates in all cases. Note that there is no way to force the production of equates.

SUMMARY

this operand forces the a summary display. In a summary display, one line is displayed for each PC#.

DETAIL
this operand forces the a detail display. In a detail display, several lines are displayed for each PC#. The first line displayed will have the same information on it as in the summary display. Extra lines are displayed with more information. Each PC# is separated from the next by blank lines.

RESOLVE
When a detail display is produced, 2 of the detail lines contain addresses (1 has the PC routine address, and one has the ARR address). If this operand is specified (or defaulted, RESOLVE is the default), if Z/XDC is able to identify the location, the identification will be shown.

NORESOLVE
If this operand is specified, the additional work that Z/XDC does to identify the location of PC routines and ARRs will not be done.

Examples:

LIST PC#
A summary list of Program Call Numbers for the current address space will be displayed. The list will not include operating system-supplied numbers.

LIST PC# DETAIL NORESOLVE
the same PC#s as the previous example will be displayed. However each PC# will be shown in detail, and PC routine and ARR addresses will not have Z/XDC information produced.
be aware that this command will generate a **lot** of output.

LIST PC# JOBNAME=MYJOB SYSTEM
A summary list of 'system' Program Call Numbers available to the address space with jobname MYJOB will be displayed.

LIST PC# JOBNAME=MYJOB SYSTEM NOPRIVATE
LIST PC# PCSEQ#5 PCSEQ#6
The first command produces the same output as the previous example.
The second command displays detailed information about PC#s assigned equate sequence numbers 5 and 6 by the first command.

LIST PC# ASID=1 ALLC SSWITCH=YES ARR=YES DETAIL NORESOLVE
A detail list of all program call numbers available to ASID 1 (normally *MASTER*) that space-switch and have an ARR will be displayed.
Neither the PC routine nor the ARR address will have any Z/XDC-produced information.

LIST PC# #=(5 700-9FF 14) DETAIL NORES**
A detail list of the nominated Program Call Numbers will be displayed.
Neither the PC routine nor the ARR address will have any Z/XDC-produced information.

LIST PC# OWNER=RRS DETAIL
A detail list of the all Program Call Numbers owned by the RRS address space will be shown.

Note that this command (with the correct owner name) can be used during development of a server that supplies system PC#s to display detailed information about those PC#s (including their routine and ARR addresses).

LIST PC# #=(30B 311)
A detail list of the PC#s that are used by the **STORAGE OBTAIN** and **STORAGE RELEASE** Assembler macro instructions is displayed.
If the system nucleus has been mapped (the Z/XDC command **MAP IEANUC01** has been issued), Z/XDC will be able to display detailed information about the location of the PC# routines.

More Information:

Authorization
Equates can only be produced if the environment that this command executes in is authorized. This is because, in many cases, the equate produced will have a different ASID associated with it and these address spaces can only be accessed if authorized.
Point-and-shoot fields and routine addresses have the same considerations. However, the authorization needed is not exactly the same as XDC or the user being authorized. The actual authorization needed depends on the environment as follows:
- In the case of Z/XDC executing as an ESTAE (etc.) exit or TRAP routine, XDC authorization is required. (i.e. PSW key 0-7, supervisor state, or APF authorized)
- in the case of dump/XDC, examination of a dump does not require that Z/XDC is authorized, and Z/XDC runs unauthorized, but this command can still do things that require authorization (such as produce equates).

Equates
When equates are produced:
- all existing equates with the same prefixes (as specified below) will be deleted.
- for each PC# displayed, an equate starting with **PCSEQ#** and followed by a sequence number will be created. The equate will have a value of the PC#s routine address (including ASID). Note that this equate can be used as an address expression on subsequent **LIST PC#** commands to obtain a detailed display about that PC#. (One of the examples shows this).
- for each PC# displayed that has an ARR, an equate starting with **PCARR#** and followed by the same sequence number will be created. The equate will have a value of the PC#s ARR address (including ASID).

SSWITCH=asid vs. OWNER=asid filtering criteria
The two filtering criteria, SSWITCH=asid and OWNER=asid have different uses:
- SSWITCH=asid is used to select PC#s that execute their routine in a specific aspace (i.e. 'Space-Switch').
- OWNER=asid is used to select PC#s that are owned by a specific aspace.

Output
The command produces the following output:
- a line showing the ASID (and jobname etc) that the output is being produced for.
- One or more title lines.
- in the case of a summary list, a line for each PC number. The information shown is:
  - The PC number.

- the assigned equate sequence number. (If the command is not generating equates this field will be blank)
- an indication of the PC number being Basic (BS) or Stacking (ST).
- an indication of the Z/XDC-assigned class of the PC number (SF - System Function table (operating system), SY - System, PV - Private).
- the Authorization Key Mask (AKM).
- If ALRF (Address space and Linkage index Reuse) is in effect, the LSTESN value. If ALRF is not in effect, the value will be shown as 8 dashes.
- an indication (Y/N) that this PC number has or doesn't have an ARR.
- The PC#s parameters (1 and 2). If a value is not zero, the parameter field may be used as a point-and-shoot field.
- If available, a comment about the PC number. (only **SFT** PC#s may have a comment. The comment has been derived from the Diagnosis Reference)
- the **L** shortcut may be used on this line to produce a detail display of the PC#.

- If a detail list is being produced, each PC number will have the same summary line displayed (but note that the **L** shortcut command will not be available). That line will be followed by additional lines, showing:
  - The PC# owning ASID and jobname Note that if owner information is not available this line will not be displayed.
  - if the PC number is space-switching, the execution ASID and jobname (etc).
  - Information about the routine that will be executed when the PC number is called:
    - The initial execution mode (problem or supervisor).
    - The initial AMODE (24 or 31 or 64).
    - The initial ASC-MODE (primary or AR).
    - The initial value of the secondary address space (Old Primary or New Primary).
    - The Execution Key Mask (EKM) and an indication as to whether the value will be OR'd with the caller's current EKM or will replace it.
    - The routine's EAX value
    - The initial PSW Key. This could be the PSW key of the caller or a specific value.
    - The routine address. (the line containing this address may have a selection field at the beginning, where a shortcut can be entered to display the routine).
    - if the PC number has an ARR, information about the ARR will be shown:
      - The CANCEL option for the ARR.
      - The ASYNCH option for the ARR.
      - The ARRCOND option for the ARR.
      - the ARR routine address. (the line containing this address may have a selection field at the beginning, where a shortcut can be entered to display the ARR routine).

Help COmmands LISt PEt

This command displays information about a PET (Pause Element Token).

Note that a PET can be released by the **RELEASE** command.

Syntax:

LIST PET pet-address

Operands:

pet-address
The address of the PET that is to be listed.

Examples:

LIST PET 12345678
Information abut the PET located at address 12345678 in the current address space is shown.

LIST PET A0000000~ASID(MYJOB1)
Information about the PET located at address A0000000 in the address space of MYJOB1 is shown.

Help COmmands LISt PFkeys

This command is an alias of the **LIST KEYS** command. See HELP COMMANDS LIST KEYS for more information.

Help COmmands LISt PGms

The **LIST PGMS** command displays the following:

- **LIST PGMS JES3:** For JES3 address spaces, this displays the load modules that have been loaded via JES3 services.
- **LIST PGMS CICS:** For CICS address spaces, this displays the load modules that have been loaded via CICS services.
- **LIST PGMS tcbaddress:** This displays the load modules associated with any given TCB in any accessible address space. The TCB's **RB** and **LLE** queues are examined. (Note, the **TCB#n** equates created by the **LIST TASKS** command come in handy here when issuing this command.)
- **LIST PGMS ALLT:** This displays all references to all load modules associated with **all TCBs** in the target address space. The information is gathered by examining the **RB** and **LLE** queues for all tasks. Notes:
  - This is **not** the same as displaying an address space's "Job Pack Area".
  - Only references are reported. The Defining Queues report is not produced.
- **LIST PGMS ∗:** This displays all programs on all queues (both private area and

common storage queues) that are accessible to the target address space.

- **LIST PGMS JPA:** This displays the load modules in the target address space's Job Area Area.

- **LIST PGMS LPA:** This displays the load modules in both the System's Dynamic Link Pack Area (DLPA) and the classic Link Pack Area (LPA). This also includes both the Modified Link Pack Area (MLPA) and Fixed Link Pack Area (FLPA). (Generally, the Link Pack Areas contain modules that were loaded into common storage at IPL time.)

- **LIST PGMS PLPA:** This displays the load modules contained in the system's Pageable Link Pack Area (PLPA).

- **LIST PGMS PRIVATE:** This displays those load modules that that are unknown to the System but that you have identified via certain **MAP** and **DMAP/USING** commands. See **HELP MAPS PRIVATELYLOADED** for more information.

- **LIST PGMS NUCLEUS:** This displays the System Nucleus (IEANUC01).

- **LIST PGMS modulename:** This displays all **references to** and all **locations of** all accessible instances of the named load module.

- **LIST PGMS namemask:** This displays all locations of all accessible instances of all load modules that match the name mask. (References are not reported. Only the Defining Queues report is produced.)

- **LIST PGMS addressexpression:** This displays information about the load module (if any) into which the expression resolves.

This **LIST PGMS** command is reactive to whether or not Z/XDC is in **Foreign Address Space Mode (FASM).** When reporting on load modules described by address space related queues, LIST PGMS will examine the JPAs, RBs and LLEs located in the currently targeted address space.

The **LIST PGMS** command is sensitive to whether or not a program has a path name provided and displays either an 8-character name or a path name.

More Information

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

SYNTAX - Describes the full syntax of the LIST PGMS command.
AUTOMATICEQUATES - Discusses the Automatic Equates that can be created by the LIST PGMS command.
EXAMPLES - Offers examples for using the LIST PGMS command.
REPORT - Describes the report produced by the LIST PGMS command.

Help COmmands LISt PGms Syntax

This topic describes the syntax of the **LIST PGMS** command. For a full discussion of the command itself, see **HELP *UP (F1).**

Syntax:

	[1st operand]	[2nd operand]
LIST PGMS	**omitted**	**omitted**
	tcbaddress	**skipcount**
	ALLTASKS	**firstname**
	JPA	
	LPA	
	PLPA	
	JES3	
	CICS	
	PRIVATE	
	NUCLEUS	
	USS	
	modulename	
	namemask	
	addressexpression	
	,	

Aliases and Special Notes:

[keyword:]	[Alias name or comment]
ALLTASKS:	**ALLTCBS** and **ALLT**
JPA:	**JPQ**
LPA:	**LPQ** and **DLPA**
PLPA:	**LPDE**
PRIVATE:	**NOCDE** and **FAKE**
NUCLEUS:	**NUC**
USS:	**PATHNAME**

tcbaddress: **TCB#n** These are equates created by the **LIST TASKS** command.

ALLTASKS: **ALL** is explicitly disallowed because it would not display everything! It would report only modules described by RBs and LLEs. That behavior would be surprising and, therefore, confusing.

LIST PGMS *: THIS reports all modules everywhere.

The Rules:

- Operands appearing above in the same column are mutually exclusive.

- Either zero, one or two operands may be given.

- When the first operand is omitted but the second operand is given, the omission must be indicated by a comma (,).

Operands:

first operand omitted
This is valid only when Z/XDC is not in Foreign Address Space Mode. Omitting the first operand causes the modules associated (by RBs and LLEs) with the **current** TCB to be displayed. [It does not display the Job Pack Area. Use **LIST PGMS JPA** if you want to do that.]

,
When the first operand is omitted and the second operand is given, then a comma needs to be used to indicate the omission.

tcbaddress
This must be the address of some TCB in an accessible address space. The load modules associated (by RBs and LLEs) with this TCB are displayed.

Note that the **TCB#n** equates created by the **LIST TASKS** command are a very convenient way to reference TCB addresses.

ALLTASKS
ALLTCBS
ALLT
This shows all modules associated (by RBs and LLEs) with all TCBs in the target address space.

This does **not** display all the modules in the address space and common storage. It only shows all the module references made by TCBs.

- Use **LIST PGMS JPA** if you want a report of an aspace's Job Pack Area.

- Use **LIST PGMS** * if you want a report of all modules everywhere (that are in storage and accessible to your program).

LIST PGMS ALLTASKS also will not report any modules brought into storage via **directed LOADs.** So the report will not include CICS and JES3 modules loaded via their respective internal services. If you want a report of those, use either:
- **LIST PGMS JES3**
- **LIST PGMS CICS**
- **LIST PGMS** *

JPA
JPQ

This shows the modules contained in the target address space's **Job Pack Area.** Note, most address spaces have only one Job Pack Area. The Master Scheduler address space, however, has multiple Job Pack Areas. Accordingly, this command supports the display of multiple Job Pack Areas for those address spaces that have them.

LPA
LPQ
DLPA
This shows the modules contained in the system's **Link Pack Area** and **Dynamic Link Pack Area (DLPA)** both. This display will include all modules contained in the system's FLPA, MPLA, DLPA and LPA. It will **not** show any modules in the PLPA.

PLPA
LPDE
This shows the list of modules contained in the system's **Pageable Link Pack Area.**

JES3
When the target address space is the JES3 address space, this displays all load modules loaded by JES3's internal services.

If Z/XDC is running within a JES3 address space, or if the **SET ASID** command has been used to target a JES3 address space, then this **JES3** operand can be used.

When the target address space is a CICS address space, this displays all load modules loaded by CICS's internal services.

If Z/XDC is running within a CICS address space, or if the **SET ASID** command has been used to target a CICS address space, then this **CICS** operand can be used.

PRIVATE
NOCDE
FAKE

If a Privately Loaded load module (or program object) has been described by certain **MAP** or **DMAP/USING** commands, then Z/XDC creates an internal descriptor of the module so as to make the module's presence known to the rest of Z/XDC (including known to this LIST PGMS command). For details, please see **HELP MAPS PRIVATELYLOADED.**

The **MAP** and **DMAP/USING** commands can be used to map privately generated code located anywhere, including above the bar. This works regardless of whether or not z/OS itself supports RMODE64 modules.

The **LIST PGMS PRIVATE** command displays a report of all Privately Loaded and mapped load modules and program objects.

NUCLEUS

NUC
This shows a descriptor for the System Nucleus (IEANUC01).

USS
PATHNAME
This shows the same list of program names that the **ALLTASKS** operand produces, except that only programs that have a path name are shown.

modulename
module or program object, then all address space and system areas are searched for CDEs (and LPDEs) that describe either the named module, or aliases of the named module, or IDENTIFY'd entry points for the named module. Messages are generated and displayed as appropriate.

To allow modules with a path name to be processed, the modulename parameter may be a USS file name up to 64 characters long. If a program has a path name specified, the given name is matched to the **file name** part of the full path name. (for this match the provided name is not upcased)

Modulename cannot be the name of a Privately Loaded Load module **unless** that module has been suitably MAP'd or DMAP/USING'd. See **HELP MAPS PRIVATELYLOADED** for more information.

namemask
Use this operand to cause the command to scan all available areas but display only those modules that meet the mask's criteria. Masks may contain **?** and ***** wildcard characters. For detailed syntax, see **HELP COMMANDS SYNTAX PATTERNMATCH.**

To allow filtering of modules with path names in them, the **namemask** value is kept in an un-upcased form. If a module contains a path name, the provided **namemask** value is matched against the file name part of the path name.

addressexpression
If this operand is an address expression that resolves to an address that is contained within a load module (and is not the address of a TCB), then all address space and system areas are searched for CDEs and LPDEs that describe the module, or are aliases of the module, or are IDENTIFY'd entry points of the module. Messages are generated and displayed as appropriate.

Second operand: **skipcount**
The first **skipcount-1** modules of the generated display are suppressed.
skipcount must be a pure decimal number. It may be given in combination with any first operand except:
- addressexpression

- modulename
- namemask

Second operand: **firstname**
All generated messages are suppressed preceding the first message for the module having this name. **Firstname** must be a pure load module name or the file name part of a path name. It may be given in combination with any first operand except:
- addressexpression
- modulename
- namemask

Second operand: **omitted**
No messages are suppressed from the generated display.

Help COmmands LISt PGms Automaticequates

This topic describes the automatic equates that can be created by the **LIST PGMS** command. For a full discussion of the command itself, see **HELP *UP (F1).**

Automatic Equates

The **LIST PGMS** command (re)generates a series of Automatic Equates that label the locations of the underlying CDEs, LPDEs, JES3-JDEs and CICS-APEs from which the command's report was built. Notes:

- The names of the equates are **CDE#n, LPDE#n, APE#n, JDE#n** and **FAKE#n,** according to what the underlying cblock is.
- The fake cblocks that Z/XDC creates for Privately Loaded modules are LPDEs located within Z/XDC's private control block structures. They are unknown to the rest of the System, but are fully known to all Z/XDC commands and displays.

  When using the **LIST PGMS** command with a system that is not executing in the host system (such as a dump) the equates generated from the Z/XDC-generated control blocks are not journaled and restored. (typically these are privately loaded modules and the system nucleus).

  Using the **LIST PGMS** command on the new instance of the system will recreate the equates.
- Each time the **LIST PGMS** command is issued, all equates generated by a prior use of the command are deleted, and new equates are created.
- The number of equates generated can range from a handful to several thousand! The performance issues don't seem to be significant.

- The generated equates are static. They do not float. Consequently, it is quite possible that as LOADs and DELETEs occur, reality can change out from under them, but reissuing the LIST PGMS command will redefine the equates back into conformity. This is just something to be aware of.

- The **LIST EQUATES namemask** command can be used to display subsets of the generated equates. Example: **L EQU CDE#* LPDE#***

Help COmmands LISt PGms Examples

This topic gives examples of using the **LIST PGMS** command. For a full discussion of the command itself, see **HELP *UP (F1).**

Examples:

L PGMS
This shows the load modules and program objects associated with the current TCB. (It does not show any load modules loaded by CICS and JES3 internal services.)

L PGMS CICS
Assuming that the target address space is a CICS address space, this shows all modules loaded by CICS. (It does **not** show any load modules located in the address space's Job Pack Area.)

L PGMS 21C?+7C%
L PGMS TCB.TCBJSTCB%
These show the load modules associated with the jobstep TCB. Notes:
- The second command presumes that you have loaded and properly positioned a dsect map of the TCB.
- This is **not** the same as displaying the address space's Job Pack Area. This will show only those modules pointed to by the **RBs** and **LLEs** that exist under that TCB.
- For CICS and JES3 address spaces, this does not show any load modules loaded by their internal services.
- **21C** references, of course, the **PSATOLD** field.
- **+7C** references the **TCBJSTCB** field.

L TASKS
L PGMS,TCB#1
This shows the load modules associated with the target address space's first TCB. The L TASKS command generates the TCB#1 equate and assigns it to the address of the first TCB.

L PGMS JPA
This shows the load modules contained within the target address space's Job Pack Area (also known as Job Pack Queue). (For CICS and JES3 address spaces, it does not show any modules loaded by CICS's or JES3's internal services.)

L PGMS MYSUBS
This displays information about all occurrences and uses of the MYSUBS load modules (and their aliases, if any). If there are multiple copies of MYSUBS loaded into storage, then information about each copy is displayed.

Note, typing "L PGMS MYSUBS." (with a trailing dot) would produce a different result. This is because "MYSUBS." is not a pure load module name; it is an address expression.

L PGMS DFHA***
This scans all areas and displays all modules whose names start with **DFHA.**

L PGMS XDC***
This scans all areas and displays all modules whose names start with **XDC.**

L PGMS *SVC
This scans all areas and displays all modules whose names end with **SVC.**

L PGMS *AD????
This scans all areas and displays all modules whose names contain **AD** as the 4th and 3rd to last characters.

L PGMS ???AD????
This scans all areas and displays all modules whose names are exactly 7 characters long and whose 4th and 5th characters are **AD.**

L PGMS *
This scans all areas and displays all modules. Period.

L PGMS PSW!
This displays information about all uses and aliases, if any, of the currently executing program.

MAP HIDNMODU START=R15?
FORMAT HIDNMODU
L PGMS HIDNMODU
L PGMS PRIVATE

Suppose that a load module named **HIDNMODU** has been brought into storage via a **LOAD** macro using the **ADDR=** (or related) operand. Then z/OS does not build a **CDE** describing the module, thus the module remains hidden both from z/OS and from Z/XDC.

But if you know where the module is, then you can still load a map for it by using the **START=address** operand of the **MAP** command.

When you do this, Z/XDC, in addition to creating the map, also creates an internal record of the module, thereby making the module known to the rest of Z/XDC.

You then can reference the module explicitly by other Z/XDC command, including the **FORMAT** and **LIST PGMS** commands.

You can also collectively reference all known Privately Loaded load modules via the **LIST PGMS PRIVATE** command.

DMAP HIDNMODU.
USING HIDNMODU R15?
L PGMS HIDNMODU
L PGMS PRIVATE

Alternatively, a **DMAP/USING** sequence also will make a Privately Loaded load module known to the rest of Z/XDC.

Be aware that c/XDC will not be able to source-map Privately Loaded modules made known to Z/XDC via DMAP/USING commands. If this is something you want to do, then make the Privately Loaded module known via the **MAP** command and its **START=** operand, not via this DMAP/USING sequence. (See the prior example.)

Help COmmands LISt PGms Report

This topic describes the reports produced by the **LIST PGMS** command. For a full discussion of the command itself, see **HELP *UP (F1).**

Below is a typical report produced by the LIST PGMS command:

```
     ENTRY    PRIMARY REFRNCE USE                       SUB-  ATTRIBU ...
SEQ# NAME     NAME    TCB/RB# COUNT     ENTRY ADDRESS KEY POOL  (ATTR A ...
   1 XXXCMD   XXXCALL 8   1       0 00000000_00069E90  0   252  350242- ...
   2 XXXCMD   XXXCALL 9   1       0 00000000_00069E90  0   252  350242- ...
   3 XXXCMD   XXXCALL 13  1       0 00000000_00069E90  0   252  350242- ...
   4 XXXCMD   XXXCALL 14  1       0 00000000_00069E90  0   252  350242- ...

     ENTRY    PRIMARY DFINING USE                       SUB-  ATTRIBU ...
SEQ# NAME     NAME    QUEUE   COUNT     ENTRY ADDRESS KEY POOL  (ATTR A ...
   5 XXXCALL          JPQ-3       4 00000000_00069E90  0   252  312242- ...
```

```
  6 XXXCMD   XXXCALL  JPQ-3       0 00000000_00069E90  0   252  350242- ...
```

Depending upon the operands given, the LIST PGMS command generates either one report or two. The two reports differ in the third column. One report is captioned **REFRNCE TCB/RB#** while the other is captioned **DFINING QUEUE.**

The first of these reports is intended primarily to show what tasks and Request Blocks are referencing the displayed load modules, while the second report primarily shows in what area (ie. on what system queue) the existence of the load module is defined.

Z/XDC extracts most of the information displayed from LLEs, CDEs, LPDEs and APEs. Briefly, these are system control blocks that serve the following purposes:

Application Program Entries (APEs)
These describe load modules residing in a CICS address space and loaded into storage via CICS services (not by z/OS services).

Link Pack Directory Entries (LPDEs)
These describe load modules contained within the System’s Pageable Link Pack Area (PLPA). They provide the names, locations, entry addresses, and attributes of PLPA load modules. LPDEs exist for all PLPA load modules regardless of whether or not the modules have been referenced by user or system programs. LPDEs are built at system startup time, and they exist for the life of the System and even beyond.

Content Directory Entries (CDEs)
These describe essentially all other load modules in the System. Like LPDEs, CDEs provide the names, entry addresses, and attributes of load modules. The System maintains two kinds of CDE queues. One queue is known as either the "Job Pack Queue" or the "Job Pack Area". The other is known as the "Link Pack Queue" or "Link Pack Area" **(not** to be confused with the system’s Pageable Link Pack Area).

Each address space in the System may have (usually) one or (sometimes) more Job Pack Queues. Generally, the Job Pack Queue is queued from an address space’s jobstep TCB; although, it is quite possible for additional Job Pack Queues to be chained from other TCBs.

Each time a load module is brought into storage via a LINK, LOAD, ATTACH, or XCTL macro, a CDE is created for that module on the Home Address Space’s Job Pack Queue. A CDE is also created whenever the IDENTIFY macro is used to create an additional name for an existing load module.

The Link Pack Queue is a single queue that the System creates to describe load modules that are located in shared common storage but are not in the PLPA. Generally, these are load modules located in the **Modified** Link Pack Area (MLPA) and/or the **Fixed** Link Pack Area (FLPA) and/or the **Dynamic** Link Pack Area (DLPA).

Load List Elements (LLEs)
These are control blocks created in addition to CDEs to describe load modules

brought into storage specifically via **LOAD** macros. (LLEs are **not** created via LINK, XCTL, or ATTACH.) The System uses LLEs to keep track of which task has issued LOADs for a given load module. Whereas an address space typically has only one CDE queue (the Job Pack Queue), there can be as many as one LLE queue for each TCB in the address space. Should a TCB terminate without first DELETE'ing all of the modules that it has LOAD'd, the System will use the LLEs to determine which modules it has to delete on behalf of the terminating task.

Each LLE contains a pointer to the CDE or LPDE describing the LOAD'd module. It also contains two use counts to keep track of how many LOAD macros have been issued for the load module and, therefore, how many DELETE macros must be issued in order to remove the load module from storage.

Report Details

What follows is a column by column description of the information that can be displayed in the LIST PGMS reports.

SEQ#

This column is simply an incrementing sequence number. It is used in the construction of the names of Automatic Equates generated by this report.

ENTRY NAME
This column shows the name of a program being described. Specifically, it is the name found in the CDNAME or LPDENAME field of the CDE or LPDE from which Z/XDC has found most of the rest of the information shown. This name may be a **major** name (ie. the load module's primary name), a **minor** name (ie. an alias name for the load module), or an IDENTIFY'd name (ie. a secondary name created via use of the system's **IDENTIFY** macro).
If a specific CDE (never an LPDE) is flagged as containing a path name, this field is shown as a dash ('-').

PRIMARY NAME
If the ENTRY NAME, shown in the prior column, is a load module's major name, then this column is left blank. On the other hand, if the ENTRY NAME is a minor name or an IDENTIFY'd name, then this column shows the load module's major name.

REFRNCE TCB/RB#

This column shows two items of information. At the left (below "TCB") is shown the sequence number of the task for which the module was brought into storage. (Task sequence numbers can be displayed via the LIST TASKS command.) At the right (below "RB#") Z/XDC shows whether the module is associated with a Request Block (RB) or a Load List Element (LLE) as follows:

- If a sequence number is shown, then the load module is associated with the indicated Request Block. (Request Block sequence numbers can be displayed via the LIST RBS command.)

- If RB sequence number 1 is shown, then the load module was probably brought into storage via an ATTACH macro. (It could have been brought in via an XCTL macro, but that is uncommon.)

- If an RB sequence number other than 1 is shown, then the load module was probably brought into storage via a LINK macro. (Again, however, it could have been an XCTL.)

- If **LLE** is shown, then the load module was brought into storage via a LOAD macro. The System does not maintain any association between LOAD'd modules and the RB that was in control at the time that the LOAD macro was issued.

DFINING QUEUE
This column shows the area in which the load module exists as follows:

PLPA - The module exists in the system's Pageable Link Pack Area. It may reside in either 24-bit or 31-bit storage.

LPQ - The module exists in the system's Link Pack Queue. This is a queue of common area load modules that are not contained within the PLPA. Most modules on the LPQ are "in" the DLPA, MLPA or FLPA.

JPQ-# - The module is referenced on an address space's Job Pack Queue. Such modules may actually reside either in the address space's private area, or in the system's LPA or PLPA. **#** is the sequence number of the TCB from which the JPQ is queued. Note, Z/XDC always checks all TCBs for the presence of a Job Pack Queue.

CICS - The module is located within a CICS address space and was loaded into storage via CICS services (not z/OS services). The module is described by an Application Program Element (APE).

NUC - The module is the System Nucleus (IEANUC01).

MAP
DMAP - The MAP or DMAP command has been used to map a Privately Loaded module. This causes Z/XDC to create a fake LPDE to represent that module.

USE COUNT
This may be one of several possible use counts for the load module as follows:

- For modules brought into storage via LOAD macros (ie. **LLE** is displayed under **REFRNCE RB#),** two counts are shown (e.g. "3-2"). The first is from the LLECOUNT field, and the second is from the LLESYSCT field. The LLECOUNT value is the total number of LOAD macros that have been issued against the load modules from the current task only. (LOADs from other tasks are not counted here.) The LLESYSCT value is the number of those LOADs that have been issued from "system" (ie. authorized, key 0, and/or supervisor state) routines. The existence of these two counts allows the System to enforce a restriction that user routines cannot issue DELETEs corresponding to LOADs issued by system routines, **and** vice-versa!

- For modules found in the PLPA, this is the value contained in the LPDEUSE field. Usually this count is zero, since PLPA modules are permanently loaded into storage and therefore the System has no need to maintain accurate use counts for them.

- For all other modules, this is the value contained in the CDUSE field of the module's CDE. For minor names, this usually is zero, but for major names, this is the net sum of all references to the modules via LINKs, LOADs, ATTACHs, and XCTLs issued against both the major name and all minor names **(not** including IDENTIFY'd names) of the load module.

ENTRY ADDRESS

This is the 8-BYTE address of the load module's entry point as defined via the displayed ENTRY NAME.

KEY
This shows the access key of the storage containing the load module.
- If an **f** follows the number (e.g. **8f),** then the storage is fetch protected.
- If an **s** follows the number, then the storage is store protected.
- Otherwise, the storage is only key protected.

SUBPOOL
This describes the storage in which the load module is physically located. Possible annotations include:

DLPA
XDLPA - The module is on the Link Pack Queue and is located in the 24-bit (below the line) or 31-bit (above the line) Dynamic Link Pack Area.

MLPA
XMLPA - The module is on the Link Pack Queue and is located in the 24-bit or 31-bit Modifiable Link Pack Area.

FLPA
XFLPA - The module is on the Link Pack Queue and is located in the 24-bit or 31-bit Page Fixed Link Pack Area.

PLPA
XPLPA - The module is located in the 24-bit or 31-bit Pageable Link Pack Area.

64BCM - The module is located in 64-bit common storage.

64BPV - The module is located in 64-bit private storage.

s0C4 - The module is located in inaccessible or nonexistent storage.

In addition, the following are the most common storage subpools in which load modules are likely to be found:

241 - This is pageable and fetchable storage located in the Common Services Area (CSA). Usually, it is key 0, but it can be in any key. It often is used for dynamically built, vendor product modules that have been dynamically added to the Link Pack Queue (LPA) after IPL. Z/XDC uses this for some of its dynamically built System Interface routines.

245 - This is fixed, fetchable, key 0 storage located in the System Queue Area (SQA). It often is used for dynamically built, vendor product modules that have been dynamically added to the Link Pack Queue (LPA) after IPL. Z/XDC uses this for some of its dynamically built System Interface routines.

248 - This is DREF (Disabled Referenceable), fetchable, key 0 storage located in the System Queue Area (SQA). It often is used for dynamically built, vendor product modules that have been dynamically added to the Link Pack Queue (LPA) after IPL. Z/XDC uses this for some of its dynamically built System Interface routines.

252 - This is key 0 storage located in the Home Address Space's private area. This is used for load modules located on the Job Pack Queue that are reentrant **and** were loaded into storage from an authorized library!

251 - This is key 8 storage located in the Home Address Space's private area. This is used for all other private area load modules, including reentrant load modules loaded into storage from non-authorized libraries!

ATTRIBUTES
(ATTR ATTR2 ATTR3-ATTRB)
This column shows the attribute flags associated with the load module's current ENTRY NAME. First, the flags are shown in hex, then all flags are shown mnemonically. For CDEs the information is extracted from the CDATTR, CDATTR2, CDATTR3, and CDATTRB fields. for LPDEs the information comes from the LPDEATTR, LPDEATT2, LPDEATT3, and LPDEATTB fields.

The following attribute mnemonics may be displayed and have the following meanings:

AMODE(ANY)	means CDEANYM=1
APFLIB	means CDSYSLIB=1
APFPGM	means CDAUTH=1
CDE	means CDELPDE=0 (OFF)
CDX	means CDCDEX=1
CONTMN	means CDCONTMN=1
DLPA	means CDEDYLPA=1
EOM	means CDEOM=1
FAKE	means this information represents a mapping of a Privately Loaded module.
GLOBAL	means CDGLOBAL=1
IDENTIFY'D	means CDIDENTY=1
INACTIVE	means CDREL=1
JPA	means CDJPA=1
LPDE	means CDELPDE=1
LPOK	means CDELPOK=1
MINOR	means CDMIN=1
NFN	means CDNFN=1
NIC	means CDNIC=1
NIP	means CDNIP=1
NOXTLST	means CDXLE=0 (OFF)
OL	means CDNLR=0 (OFF)
OVLY	means CDOLY=1
PATHN	means CDPATHN=1
PMLOK	means CDPML=1
PROTP	means CDEPROTP=1
RACDTY	means CDRACDTY=1
RACF	means CDRACF=1
RENT	means CDREN=1
REUS	means CDSER=1
ÎRENT	means CDREN=0
ÎREUS	means CDSER=0
RLOCEP	means CDRLC=1
SHRHFS	means CDSYSHLB=1
SPLIT	means CDESPLIT=1
SP0	means CDSPZ=1
n EXTENTS	means #'Extents usually when other than 1

For the meanings of the flag settings, see the CDE control block mapping described in IBM's **z/OS MVS Data Areas, vol. 1 (ABEP-DALT)** (GA22-7581).

If a CDE has the Path name flag set, the path name is shown on a separate line. If the path name is longer than 64 characters, only the **last** 64 characters of the path

name are shown, and the displayed path name value will be preceded by an ellipses ('...') indicating that truncation took place.

Help COmmands LISt PId

The **LIST PID** command displays the USS Process ID.

There are circumstances where the USS process ID is not available:
- If debugging an SRB
- If debugging a dump and there is no CXU or the CXU is an SRB.
- If the current task is not a USS process, and has not been 'dubbed'.

In these cases a message will be displayed indicating the reason that the process ID is not available

Note that the use of this command **never** results in the current task being dubbed.

Syntax:

LIST PID

This command does not accept operands.

Help COmmands LISt PRIMarysize

The **LIST PRIMARYSIZE** command shows whether Z/XDC's Fullscreen Support will attempt to use the display terminal's primary or secondary screen geometry, when available.

The **LIST SECONDARYSIZE** command is a synonym of the "LIST PRIMARYSIZE" command.

Syntax:

LIST PRIMARYSIZE
SECONDARYSIZE

This command accepts no operands.

The terminal display geometry setting can be saved in your session profile. It can be changed by the SET PRIMARYSIZE and SET SECONDARYSIZE commands. It also can be displayed and changed by Z/XDC's "Profile Menuing System". For more information, please see:

HELP COMMANDS SET PRIMARYSIZE
HELP COMMANDS SET SECONDARYSIZE
HELP PROFILES MENU

Help COmmands LISt PRINt

The **LIST PRINT** command displays information about the XDCPRINT output file. It shows whether or not the file is allocated, what its dsname or sysout status is (as appropriate), what its DCB attributes are, how much output has been written to the file, and what its lines-per-page setting is.

The XDCPRINT output file can be used for printing excerpts of or the entirety of Z/XDC's Built-in Help. The information is automatically paginated and indexed. See HELP ABOUTHELP PRINTING for more information.

See also HELP COMMANDS SET PRINT for information about changing certain XDCPRINT related settings.

Some of the information displayed by the LIST PRINT command is profiled, so it can be both displayed and changed by the PROFILE command.

Syntax:

LIST PRINT

This command does not accept operands.

Examples: See HELP COMMANDS SET PRINT.

Help COmmands LISt PROfiles

The **LIST PROFILES** command scans the available profile and table libraries and displays information about any or all Z/XDC related profiles found. The information displayed includes:

- The profile's name.
- The version and release of the XDC that created the profile.
- The library name and member name in which the profile resides.
- The ddname by which the profile is accessed.
- Indications of whether the profile is the currently active profile and (if so) whether or not changed settings are pending.
- An indication when a profile is invalid.

Note, if you want to see the current profile's current **settings,** use the **PROFILE** command, without operands.

LIST PROFILES recognizes whether or not a given profile was created by Z/XDC (or by a predecessor) by the presence or absence of certain eye-catchers in the data. When those eye-catchers are not present, **LIST PROFILES** realizes that the profile or table library member belongs to some program other than z/XDC, and so it ignores it.

When **LIST PROFILES** encounters a member that does have the necessary eye-catchers but which then goes on to fail certain other validity checks, **LIST PROFILES** realizes that the profile data has been corrupted and so will display that profile as being **(INVALID!).**

WARNING! This command, depending upon the operands given, can wind up searching **all**

members of **all libraries** allocated to both //ISPPROF and //ISPTLIB. Consequently, certain operand choices can lead to potentially long run times. Accordingly, **LIST PROFILES** supports being interrupted via an attention signal. See the **LIST PROFILES ISPTLIB** example (below) for more information.

The following command syntax may be confusing. Basically, any number and type of operands can be given in any order, and the syntax rules are so free-form that sometimes operands can be (and **will** be) interpreted in **multiple** ways. Consequently, there are operand combinations that don't make sense. On the other hand, there also are many that do. For clarifications, please review the Examples at the bottom of this topic. Hopefully, they contain good examples of most of the more useful combinations.

Syntax:

```
LIST PROFILES ALL
              CURRENT (or omitted)
              SEARCHORDER

              profilenames ...
              clonenames ...
              membernames ...

              ddnames ...
              dsnames ...
```

Rules:

- The member selection operands (profilenames, clonenames and membernames) are mutually additive with respect to each other.

- The library selection operands (ddnames and dsnames) are mutually additive with respect to each other but restrictive with respect to the member selection operands.

- When all library selection operands are omitted, a default set of DDNAMEs is searched. This list includes **xxxPROF, ISPPROF, ISPTLIB** and several others. (For the complete list, see **ALL** below).

- Any number of instances of the member selection operands may be given and in any combination with each other.

- If keyword operands (ALL, CURRENT and SEARCHORDER) are given in combination with other keyword and/or selection operands, then they are treated as being selection operands, not keyword operands.

- Consequently, the keyword operands will not be effective unless they are given alone.

- Also, long keywords may not be abbreviated to less than 9 characters, and short keywords (8 characters and shorter) may not be abbreviated at all.

ALL
This operand causes the display of information about all XDC profiles from all profile and table libraries that are allocated to the following ddnames (if present):

- **xxxPROF** (where "xxx" matches Z/XDC's current clone name)
- **ISPPROF** (ISPF's standard profile library)
- **xxxTLIB** (a TLIB)
- **ISPTUSR** (a TLIB)
- **ISPTLIB** (ISPF's standard TLIB)

In addition, information about the currently active profile is displayed.

For more information about profile ddnames, see HELP PROFILES DDNAMES.

CURRENT (or omitted)
This operand causes information about only the currently active profile to be displayed.

SEARCHORDER

This operand results in the display of information about all XDC profiles in Z/XDC's default profile search order. The profiles are displayed in search order sequence. The first listed profile is the one that would be loaded via a **PROFILE READ XDC** command. For more information on the profiles search order, see HELP PROFILES DEFAULTPROFILE.

profilenames ...

This is an optional list of profile names (as would be given on either a **PROFILE READ** or a **PROFILE SAVE** command). These names can range from one to five characters in length. These names cause the display of all profile and table library members having member names whose 4th through 8th characters match these names.

Note the following special cases:

- Profile names that are three characters long will also be processed as clone names (see below).

- The profile name **XDC** will produce a list of all default profiles that exist in the default profile search order. (This is similar to the display produced by the SEARCHORDER operand. See HELP PROFILES DEFAULTPROFILE for more information.)

clonenames ...
This is an optional list of 3-character clone names. These names cause the display of all profile and table library members having member names whose first three characters match these names.

Note, clone names will also be processed as profile names (see above). I.E. if a profile member exists whose name is six characters long and whose last three characters match a given name, then that member will be displayed too.

membernames ...
For any given name whose length is between 4 and 8 characters, if the profile and table libraries contain a member whose name exactly matches the given name, then that member is included in the display.

ddnames ...
If a name is given that matches a currently allocated ddname, then it will be processed as a ddname, and the **LIST PROFILE** command will display the libraries allocated to this ddname instead of to the default libraries (xxxPROF, ISPPROF and ISPTLIB).

dsnames ...
When a name is given that contains a period (.), then that name is considered to be a dsname. If the named dataset is a PDS or PDSE, then it will be searched for XDC profiles.

Dsname can be **any** accessible profile and table library, including those belonging to other TSO sessions (if accessible).

Examples:

LIST PROFILES V31
All profile and table libraries are searched looking for profiles created by the clone named **V31.** Those will be the members that will be displayed.

LIST PROFILES ABC DEFGH
All profile and table libraries in Z/XDC's normal profile and table libraries search order (see HELP PROFILES DDNAMES) will be searched, and information will be displayed about:
- All profiles made by an XDC clone named ABC (i.e. all members whose names start with ABC).
- All profiles having a profile name of ABC or DEFGH (i.e. all members whose names, starting with the 4th character, end with either ABC or DEFGH).
- Any profile stored in a member whose name is exactly DEFGH.

LIST PROFILES TFSXDC31 WORK ISPPROF hlq.XDCV31.XDCTLIB
The following libraries will be searched:
- All PDSs and PDSEs (if any) allocated to ddname ISPPROF.
- The library named hlq.XDCV31.XDCTLIB (but if and only if it is a PDS or PDSE).

Then information will be displayed about the following members (if present):
- TFSXDC31
- xxxWORK (where **xxx** is any clone name)
- WORK

Note, the reason ISPPROF is treated as being a ddname while TFSXDC31 and WORK are

not is that Z/XDC searched the TIOT and found ISPPROF there but did not find the other two names. Thus, Z/XDC concluded that TFSXDC31 and WORK were to be treated as membernames, not ddnames.

LIST PROFILES ISPTLIB
This command examines **all members** of **all libraries** allocated to the ISPTLIB concatenation, and displays all those that are XDC created session profiles. Profiles created by any clone and any version and any release of XDC are displayed.

WARNING! Depending upon how many libraries are allocated to the ISPTLIB concatenation, this command can take an exceedingly long time to run. If you feel that it is taking too long, then you can interrupt it via an attention signal (the **ATTN** key or **ESC** key). Also, you might find it more feasible to search for profiles via dsnames instead of the ISPTLIB ddname. (See the following example.)

Note, if any profiles are displayed from ISPTLIB that have member names that do **not** start with **TFS,** then those profiles are not reachable by the **PROFILE READ** command. So that would be an installation error that would need correcting. (ISPTLIB is an ISPF table library. The only profiles that Z/XDC can read from table libraries are those whose names start with **TFS.** Profiles whose names start with anything else must be located in profile libraries, not table libraries.)

LIST PROFILES CSW.TLIB
This command examines all members of the library named CSW.TLIB and displays all those that are XDC created session profiles. Profiles created by any clone and any version and any release of XDC are displayed.

LIST PROFILES DBCOLE7.ISPF.ISPPROF
This command examines all members of the library named DBCOLE7.ISPF.ISPPROF and displays all those that are XDC created session profiles. Profiles created by any clone and any version and any release of XDC are displayed.

LIST PROFILES SEARCHORDER

All profile and table libraries in Z/XDC’s library search order are searched, and then all profiles are displayed that are eligible to be loaded as a default profile. The profiles are displayed in their search order. If a **PROFILE READ XDC** command were issued, then the first listed profile is the one that would be loaded.

LIST PROFILES XDC
All profile and table libraries in Z/XDC’s library search order are searched, and the following profiles are displayed:

- All profiles that are eligible to be loaded as a default profile (because **XDC** is a special name that means "the default profile". See HELP PROFILES DEFAULTPROFILE for a complete discussion of exactly what that means.)
- All profiles created by Z/XDC when Z/XDC was named **XDC,** (i.e. when not renamed to some other clone name). This would be all library members whose names are at least 4 characters long and start with **XDC** as the first three characters.

- A profile, if any, exactly named **XDC.** (This would be library members whose names were six characters long and ended with **XDC.)**

PROFILE SAVE ALL
LIST PROFILES ALL
LIST PROFILES ALL JUNK
The first of these commands saves a profile whose PDS membername is xxxALL (where "xxx" is Z/XDC's current clone name).

The second of these commands results in the display of information about **all** the profiles in Z/XDC's default list of profile and table library ddnames. (In addition, information about the currently active profile is displayed.)

In the third command, the presence of the additional operand (JUNK in this case) causes the **ALL** operand to be treated as a name instead of as a keyword. Consequently, the following profiles will displayed:

- All profiles made by an XDC clone named **ALL** (i.e. all members whose names start with **ALL).**
- All profiles having a profile name of **ALL** or **JUNK** (i.e. all members whose names, starting with the 4th character, end either with **ALL** or with **JUNK).**
- Any profile stored in a member whose name is exactly **JUNK.**

In all likelihood, the only profile that would exist that meet the above criteria would be the one created by the first of the above commands.

LIST PROFILES ALL ALL
This is similar to the prior example, except that a second **ALL** operand is used instead of JUNK. The presence of a second operand, no matter what it is, causes any keyword operand to be treated as a name instead of as a keyword. So in this example, the following profiles will be displayed (if they exist):

- All profiles made by an XDC clone named **ALL** (i.e. all members whose names start with **ALL).**
- All profiles having a profile name of **ALL** (i.e. all members whose names, starting with the 4th character, end with **ALL).**

Help COmmands LISt PSW

The **LIST PSW** command displays PSWs in their old **8-byte** format. It can be used to display any of the following PSWs:

- The current program's retry level PSW.
- The PSW belonging to any program running under any RB queued from any TCB located in any accessible address space.
- The PSW belonging to any suspended SRB routine located in any accessible address space.

Related Commands:

- Use the **LIST PSWE** command to display the above PSWs in their full **16-byte** format.
- Use the **LIST EPSW** command to display the **error level** PSW in its 8-byte format.

- Use the **LIST EPSWE** command to display the error level PSW in its 16-byte format.

Important! The PSW that is displayed is **not** necessarily the contents of either the **RBOPSW** field or the **XSBOPSW16** field (new in z/OS R1.13). Instead, it is a **resolved** PSW that is derived when RBOPSW and XSBOPSW16 **differ.** For more information, see HELP EXECUTIONLEVELS RBS RESUMEADDRESS.

Syntax:

LIST PSW	**rbaddress**	**FORMAT**
	ssrbaddress	**FMT**
	omitted	**omitted**

rbaddress
This must be the address of any Request Block, queued from any TCB, located in any accessible address space in the system. When the RB is found, its **resolved** resume PSW is displayed in an 8-byte format.

If the resolved address is **above-the-bar** (i.e. cannot fit in an 8-byte wide PSW), then the display **is adjusted** so that the true resume address is shown anyway.

Note, after a **LIST RBS** command is issued, any of the **RB#n** equates that it generates can be used as an rbaddress operand for this **LIST PSW** command.

ssrbaddress
This must be the address of any SSRB or SSRX located in any accessible address space in the system. When the SSRB or SSRX is found, the resume PSW contained within the SSRB is displayed in its 8-byte format.

Note, after a **LIST SSRBS** command is issued, any of the **SSRB#n** equates or **SSRX#n** equates that it generates can be used as an ssrbaddress operand for this **LIST PSW** command.

omitted
When no address operand is given, then the **retry level PSW** is displayed. (See HELP EXECUTIONLEVELS for information about retry level vs. error level environments.)

FORMAT
FMT
This operand causes the PSW to be displayed with **all** of its flags and fields interpreted.

If **FORMAT** is **omitted,** then only the condition code, addressing mode and instruction address are interpreted.

When the **FORMAT** keyword is specified, the following additional information is displayed:
- The system mask is displayed and interpreted as follows:
  - **PER:** Program Event Recording
  - **DAT:** Dynamic Address Translation
  - **I/O:** I/O Interrupt status
  - **EXT:** External Interrupt status
- The Protection Key and Problem State flag are displayed.
- The Address Space Control mode (ASC-MODE) is displayed and interpreted as follows:

    - **PRIMARY**
    - **ACCESS REGISTER**
    - **SECONDARY**
    - **HOME**
- The current Condition Code is displayed and interpreted.
- The program mask is displayed and interpreted as follows:
    - **FIX:** Fixed-Point Overflow
    - **DEC:** Decimal Overflow
    - **EXP:** Exponent Underflow
    - **SIG:** Significance
- The addressing mode (24-bit, 31-bit or 64-bit) and Instruction Address are displayed and interpreted.

Automatic Equates

When an **rbaddress** or **ssrbaddress** operand is given on the **LIST PSW** command, the following automatic equate is (re)generated:

#PSWNIA
This labels the location pointed by the **resolved** resume PSW.

This equate is **not** generated when the **LIST PSW** command is given without operands. For more information, see HELP EQUATES BUILTIN AUTOMATIC.

Notice: This automatic equate is **static!** It does not float. The locations that it targets can change only when another **LIST PSW[E] address** command is issued. In other words, the equate can become obsolete without warning simply due to the ongoing execution of the program whose PSW is being displayed.

Examples:

L PSW
This displays the **scrunched** (8-byte wide) version of the retry level PSW. Example:

```
PSW 078D1000 00069CFC (cc-LO) (24) - PRIVATE+068CFC
```

L PSW,.TCBRBP?
Presumedly, the dsect that maps the TCB has been assigned to a suitable location. This command then displays the resolved resume PSW for the newest RB chained from that TCB. The PSW is displayed in its 8-byte format.

In the 8-byte format, the highest resume address that can be held is 2GIG-2 (i.e. below-the-bar). But if the resume address is actually above-the-bar, then the display is adjusted so that the true resume address is displayed anyway.

L RBS
L PSW,RB#1
This displays the resolved resume PSW for the oldest RB chained from the current TCB. The **LIST RBS** command generated the **RB#1** equate to represent the location of the

task's oldest RB.

Because an **rbaddress** operand was given, the following automatic equate is (re)generated:

- The **#PSWNIA** equate labels the location pointed to by the **resolved** resume PSW.

Warning! The location of this equate will **not** change as the program running under **RB#1** executes. It can be changed only by reissuing the **LIST PSW[E] RB#1** command.

L SSRBS 1
L PSW SSRB#1 [or **SSRX#1]**
The **LIST SSRBS 1** command displays all SRBs that are currently suspended in the Master Scheduler address space. It also creates a series of **SSRB#n** and **SSRX#n** equates labeling the locations of all the SSRBs and SSRXs that were found.

The **LIST PSW SSRB#1** command then displays the resolved resume PSW (in its 8-byte format) for the first of those SRBs.

Because an **ssrbaddress** operand was given, the following automatic equate is (re)generated:

- The **#PSWNIA** equate labels the location pointed to by the resolved PSW.

Warning! The location of this equate will **not** change as the program running under **SSRB#1** executes. It can be changed only by reissuing the **LIST PSW[E] SSRB#1** command.

LIST PSW,FORMAT
This produces a display of a PSW in an exploded format. Example:

```
     +---------------- PER(OFF)
     |+--------------- DAT(ON) I/O(ON) EXT(ON)
     ||+-------------- KEY(8)
     |||+------------- PROBLEM STATE
     ||||+------------ PRIMARY, CC=1 (MINUS, MIXED, LOW)
     |||||+----------- FIX(OFF) DEC(OFF) EXP(OFF) SIG(OFF)
     |||||| +-+------- AMODE(31)
     |||||| | |
 PSW 078D1000 80069CFC - PRIVATE+068CFC
```

SET PSW AMODE=64
Z PSW,48_0000001E
LIST PSW
The **SET PSW AMODE=64** command sets the retry level PSW's **addressing mode** to 64 bits.

The **ZAP PSW,48_0000001E** command sets the resume address to an **above-the-bar** location.

The **LIST PSW** command displays the retry level PSW as follows:

```
PSW  078D1001 [00000048_0000001E] (cc-LO) (64) - DBC810W NOT WITHIN ANY
IDENTIFIABLE MODULE OR ANY OTHER OBJECT
```

Note that the full 64-bit resume address is displayed even though it does not actually fit within the 8-byte wide format of the PSW! This violation of the 8-byte limit is indicated in the display by the open and close square brackets.

Help COmmands LISt PSWE

The **LIST PSWE** command displays PSWs in their full **16-byte** format. It can be used to display any of the following PSWs:

- The current program's retry level PSW.
- The PSW belonging to any program running under any RB queued from any TCB located in any accessible address space.
- The PSW belonging to any suspended SRB routine located in any accessible address space.

Related Commands:

- Use the **LIST PSW** command to display the above PSWs in their old **8-byte** format.
- Use the **LIST EPSWE** command to display the **error level** PSW in its 16-byte format.
- Use the **LIST EPSW** command to display the error level PSW in its 8-byte format.

Important! The PSW that is displayed is **not** necessarily the contents of either the **RBOPSW** field or the **XSBOPSW16** field (new in z/OS R1.13). Instead, it is a **resolved** PSW that is derived when RBOPSW and XSBOPSW16 **differ.** For more information, see HELP EXECUTIONLEVELS RBS RESUMEADDRESS.

Syntax:

```
LIST PSWE  rbaddress    FORMAT
           ssrbaddress  FMT
           omitted      omitted
```

rbaddress
This must be the address of any Request Block, queued from any TCB, located in any accessible address space in the system. When the RB is found, its **resolved** resume PSW is displayed in a 16-byte format.

Note, after a **LIST RBS** command is issued, any of the **RB#n** equates that it generates can be used as an rbaddress operand for this **LIST PSWE** command.

ssrbaddress
This must be the address of any SSRB or SSRX located in any accessible address space in the system. When the SSRB or SSRX is found, the resume PSW contained within the SSRB is displayed in its 16-byte format.

Note, after a **LIST SSRBS** command is issued, any of the **SSRB#n** equates or **SSRX#n** equates that it generates can be used as an ssrbaddress operand for this **LIST PSWE** command.

omitted
When no address operand is given, then the **retry level PSW** is displayed. (See HELP EXECUTIONLEVELS for information about retry level vs. error level environments.)

FORMAT
FMT
This operand causes the PSW to be displayed with **all** of its flags and fields interpreted.

If **FORMAT** is **omitted,** then only the condition code, addressing mode and instruction address are interpreted.

When the **FORMAT** keyword is specified, the following additional information is displayed:
- The system mask is displayed and interpreted as follows:
  - **PER:** Program Event Recording
  - **DAT:** Dynamic Address Translation
  - **I/O:** I/O Interrupt status
  - **EXT:** External Interrupt status
- The Protection Key and Problem State flag are displayed.
- The Address Space Control mode (ASC-MODE) is displayed and interpreted as follows:
  - **PRIMARY**
  - **ACCESS REGISTER**
  - **SECONDARY**
  - **HOME**
- The current Condition Code is displayed and interpreted.
- The program mask is displayed and interpreted as follows:
  - **FIX:** Fixed-Point Overflow
  - **DEC:** Decimal Overflow
  - **EXP:** Exponent Underflow
  - **SIG:** Significance
- The addressing mode (24-bit, 31-bit or 64-bit) and Instruction Address are displayed and interpreted.

Automatic Equates

When an **rbaddress** or **ssrbaddress** operand is given on the **LIST PSWE** command, the following automatic equate is (re)generated:

#PSWNIA
This labels the location pointed by the **resolved** resume PSW.

This equate is **not** generated when the **LIST PSW** command is given without operands. For more information, see HELP EQUATES BUILTIN AUTOMATIC.

Notice: This automatic equate is **static!** It does not float. The locations that it targets can change only when another **LIST PSW[E] address** command is issued. In other words, the equate can become obsolete without warning simply due to the ongoing execution of the program whose PSW is being displayed.

Examples:

L PSWE
This displays the **16-byte wide** version of the retry level PSW. Example:

```
PSWE 47851001 80000000 00000000_000858B0 (cc-LO) (64) - .DIE9106Z
```

L PSWE,.TCBRBP?
Presumedly, the dsect that maps the TCB has been assigned to a suitable location. This command then displays the resolved resume PSW for the newest RB chained from that TCB. The PSW is displayed in its 16-byte format.

L RBS
L PSWE,RB#1
This displays the resolved resume PSW for the oldest RB chained from the current TCB. The **LIST RBS** command generated the **RB#1** equate to represent the location of the task's oldest RB.

Because an **rbaddress** operand was given, the following automatic equate is (re)generated:

- The **#PSWNIA** equate labels the location pointed to by the PSW.

Warning! The location of this equate will **not** change as the program running under **RB#1** executes. It can be changed only by reissuing the **LIST PSW[E] RB#1** command.

L SSRBS 1
L PSWE SSRB#1 [or **SSRX#1]**
The **LIST SSRBS 1** command displays all SRBs that are currently suspended in the Master Scheduler address space. It also creates a series of **SSRB#n** and **SSRX#n** equates labeling the locations of all the SSRBs and SSRXs that were found.

The **LIST PSWE SSRB#1** command then displays the resolved resume PSW (in its 16-byte format) for the first of those SRBs.

Because an **ssrbaddress** operand was given, the following automatic equate is (re)generated:

- The **#PSWNIA** equate labels the location pointed to by the resolved PSW.

Warning! The location of this equate will **not** change as the program running under **SSRB#1** executes. It can be changed only by reissuing the **LIST PSW[E] SSRB#1** command.

LIST PSWE,FORMAT
This produces a display of a PSW in an exploded format. Example:

```
+----------------------- PER(ON)
|+----------------------- DAT(ON) I/O(ON) EXT(ON)
||+---------------------- KEY(8)
```

```
      |||+-------------------- PROBLEM STATE
      ||||+------------------- ASC-MODE(PRIMARY), CC=1 (MINUS, MIXED, LOW)
      |||||+------------------ FIX(OFF) DEC(OFF) EXP(OFF) SIG(OFF)
      |||||| +-+-------------- AMODE(31)
      |||||| | |
 PSWE 47851000 80000000 00000000_000854D0 - XDCTESTS+0
```

Help COmmands LISt Qualifier

The **LIST QUALIFIER** command displays the current default load module name and csect name. These names are used to complete partially qualified labels that might be given in address expressions, on the DMAP command, and in other places.

It also displays the name of the default load module that the DMAP command will search when looking for an unqualified dsect map name. (See HELP COMMANDS DMAP for more information.)

Syntax:

LIST QUALIFIER

For additional information, please see:

HELP ADDRESSING

HELP COMMANDS SET QUALIFIER

Help COmmands LISt R#

The **LIST Rn** command displays the lo-order half (4 bytes) of an individual retry level general register. For more information, see HELP COMMANDS LIST GENERALREGISTERS.

Help COmmands LISt RBs

The **LIST RBS** command shows the queue of Request Blocks chained from any given TCB in any accessible address space. For each RB, the following is shown:

- The type (PRB, SVRB, etc.)
- The manner by which it was created (ATTACH PGM-CHK, etc.)
- The name of the routine associated with it (modulename, SVC-n, ABEND-code, etc.)
- Its resolved resume address, relative to the nearest relevant module, map, or equate (if any).
- If retry is permitted, the retry RB line will be hilighted.

The LIST RBS command can be used with dump/XDC to show request blocks queued to a specific task on the dump. Most of the following description is correct, but be aware that the task and RB structure cannot change.

When displaying an RB list under dump/XDC, the CU (set current) shortcut can be used (to make the owning task 'current', and the nominated RB will be used).
Also, if the CXU is the task that RBs are being displayed for the CXU RB will be indicated by **(CXU)** and the line will be hilighted.

For more information regarding a Request Block's **resolved** resume address, see HELP EXECUTIONLEVELS RBS RESUMEADDRESS.

Note, when the **error level** and **retry level** environments are **different,** additional warning messages are displayed to emphasize the importance of being aware of and understanding that condition. (See HELP EXECUTIONLEVELS for more information.)

The **LIST RBS** command numbers the Request Blocks from oldest to newest. These numbers are then referenced by various messages from Z/XDC.

When RBs queued from the current task are displayed, the newest one or two listed will have the label **(ARTIFACT)** appended to them, indicating that they are "artifacts" of the debugging session. These particular RBs were created as part of the process that intercepts ABENDs and breakpoints and passes control to Z/XDC so that it can produce displays and execute your commands. **They will be purged** whenever Z/XDC relinquishes execution back to the user program (i.e. whenever the user issues a TRACE, GO or END command). Therefore, these artifact RBs **will never be seen** by the user's program. Therefore, these artifact RBs **can be disregarded.**

Normally, an SVRB exists for running an SVC routine, but the System also uses them to run ABEND processing. So when an ABEND SVRB is being displayed, Z/XDC will show the code for the ABEND that the SVRB was created to handle. A System ABEND code is shown, of course, as a 3-digit hex value. A User ABEND code is shown as the letter **U** followed by a 4-digit decimal value. But when **both** codes are present, they are shown together as a 6-digit hex value.

Syntax:

```
LIST RBS tcbaddress
         omitted
```

tcbaddress
If this is omitted, then the RBs queued from the current TCB are displayed. If this is given, then it must be the address of any TCB located in any accessible address space. The RBs queued from that TCB are displayed.

When Z/XDC is in Foreign Address Space Mode, the tcbaddress **cannot** be omitted.

Automatic Equates

The **LIST RBS** command (re)generates a series of **RB#n** equates to label the displayed Request Blocks. For more information, see HELP EQUATES BUILTIN AUTOMATIC.

Notice: the **RB#n** equates are **static!** They do not float. The locations that they target can change only when another **LIST RBS** command is issued. In other words, when the **LIST RBS** command is used to display the Request Block queue for any task other than the current task, the **RB#n** equates can become obsolete without warning simply

due to the ongoing execution of the task whose Request Blocks are being displayed.

Examples:

For an example of using **autocloned** equates or maps to label an entire chain of RBs, see the examples in HELP COMMANDS EQUATE AUTOCLONING.

L RBS
The RBs queued from the current task are displayed.

L RBS,21C?+7C?
The RBs queued from the jobstep TCB are displayed.

L TASKS
L RBS,TCB#2
This displays the RBs queued from the second task in the Target Address Space. The L TASKS command generates the TCB#2 equate and assigns it to the location of the Target Address Space’s second TCB.

Help COmmands LISt READ

The **LIST READ** command displays the status of all **READ** command related settings. Specifically, it shows the following:

- The current default **Scripts Library name** (if any)
- The current value of the **SET READ ECHO|NOECHO** setting
- The current value of the **SET READ SEQF=** setting
- The current behavior of script processing if an error occurs.
- The current nesting depth created by **SET READ ERROR=CONTINUE/STOP/FORCESTOP.** (This affects the behavior of script processing when an error occurs.)

- The current state of script processing:
  - Whether or not a script file is currently **open**
  - If open, then the **dsname(membername)** of the opened script file
  - Whether the script is **running** or **suspended**

Syntax:

LIST READ

This command accepts no operands.

Notes:

- All **SET READ** command settings (except the **ERROR** setting) are saved in your session profile.

- All settings can be changed by the **SET READ** command.

- Except for the **ERROR** setting, they can also be changed by the **Profile Menuing System.**

Help COmmands LISt READEcho

This command has been functionally replaced by the **LIST READ** command. Please discontinue using this command and use that one instead.

Help COmmands LISt REFrprot

Starting with z/OS R1.9, MVS has a configuration setting, named REFRPROT, that affects how the System loads **refreshable** modules into storage. Previously, a load module's **REFR** attribute had no affect as to whether a module was loaded into user key storage (usually key 8) or key 0 storage. (It was the RENT attribute that affected this, not REFR). But starting with z/OS R1.9, if a data center decides to "turn on" the REFRPROT facility, then load modules and program objects having the REFR attribute will always be loaded into key 0 storage, **regardless** of whether or not it was loaded from an APF authorized, library, and **regardless** of whether or not the RENT attribute also is on.

Notes:

- In z/OS R1.9 (and newer systems) the REFRPROT facility has to be turned on by adding a REFRPROT statement to the PROGxx members of the System's PARMLIB libraries. See IBM's "z/OS MVS Initialization and Tuning Reference" manual (SA22-7592, -15 and newer) for details.

- z/OS R1.9 (and newer systems) supports both a system-wide REFRPROT setting and a task-level REFRPROT override. Z/XDC's **LIST REFRPROT** command can be used to display these settings and overrides. The **SET REFRPROT** command can be used to change the task-level overrides.

- If the LIST REFRPROT command is issued on any system older than z/OS R1.9, then after parsing and checking all operands, the command simply reports that REFRPROT support is not present.

- When the LIST REFRPROT command is issued on any system where REFRPROT is supported, then all forms of this command will always report whether or not REFRPROT is active system-wide. Then, the REFRPROT state will be reported for the selected TCBs (as indicated by the command operands).

Syntax:

```
LIST REFRPROT ALL         aspaceref
              CURRENT     omitted
              tcbaddress
              omitted
```

Rules:

- All operands are optional. Varying defaults will be taken for omissions.

- Operands may be given in any order.

- Operands appearing above within the same column are mutually exclusive.

aspaceref

This operand specifies the address space whose REFRPROT overrides are to be displayed. (Each task within an address space has its own override setting.) If omitted, then REFRPROT settings in the space currently being targeted via Foreign/Local Address Space Mode (FASM/LASM) will be displayed. (See HELP VIRTMEM XDCACCESS FASM for more information.)

The aspaceref operand may be any of the following:
- A 1-4 digit hexadecimal number.
- A 1-8 character job name or TSO userid.

- An address space keyword (HOME, PASID, SASID, etc. See HELP COMMANDS SYNTAX ASIDS for a complete list of legal keywords.)
- CR3, CR4, ECR3 or ECR4.

Note, unlike other commands that accept aspaceref operands, this command will **not** accept general register names, address expressions, equate names and dsect names as address space references. For complete details, see HELP COMMANDS SYNTAX ASIDS.

CURRENT

This operand causes the LIST REFRPROT command to display the REFRPROT setting for the current task. It may be used only:
- When the aspaceref operand is given and it resolves to the home address space.

- Or when the aspaceref operand is omitted and Z/XDC is not running in Foreign Address Space Mode (FASM).

This operand is mutually exclusive with the ALL and tcbaddress operands.

ALL

This operand causes the LIST REFRPROT command to display the REFRPROT settings for all tasks in the target address space. This operand is mutually exclusive with the CURRENT and tcbaddress operands.

tcbaddress

This operand must be an address expression that resolves to the starting address of any TCB located in any accessible address space anywhere in the system. This operand causes the LIST REFRPROT command to display the REFRPROT setting for the specified task. Notes:

- Since address expressions resolve to locations within address spaces, the aspaceref operand is not needed for specifying the address space within which the desired TCB resides.

- If both the tcbaddress and aspaceref operands are given, then they must resolve

to the same space.

- This operand is mutually exclusive with the ALL and CURRENT operands.

tcbaddress, CURRENT and ALL all **omitted**
- When the home address space is targeted, **CURRENT** is the default.
- When a foreign address space is targeted, **ALL** is the default.

Examples:

When REFRPROT support is installed, then the following examples will have the results indicated.

SET ASID
LIST REFRPROT

In Local address space mode (LASM), the LIST REFRPROT command's default action is to display the REFRPROT state for the current task.

SET ASID JES2
LIST REFRPROT

In Foreign address space mode (FASM), the LIST REFRPROT command's default action is to display the REFRPROT states for all tasks in the targeted address space.

LIST REFRPROT JES2
This is another way to accomplish the same thing: This command displays the REFRPROT states for all tasks in the targeted address space.

LIST REFRPROT ALL
In either Local or Foreign Address Space Mode, this command displays the REFRPROT states for all tasks in the targeted address space.

LIST REFRPROT CURRENT
In Local Address Space Mode (LASM), this command displays the REFRPROT state for only the current task. In Foreign Address Space Mode (FASM), this command fails.

LIST TASKS JES2
LIST REFRPROT TCB#3
The LIST TASKS command, in addition to displaying the subtasking structure within the JES2 address space, also creates a series of TCB#n equates that label the starting address of all TCBs in that address space. (See HELP COMMANDS LIST TASKS for more information.)

The LIST REFRPROT command then uses one of those equates to display the REFRPROT

state of one of those tasks (the HASJES20 task, in this example).

LIST REFRPROT TCB#4 PCAUTH
If the TCB#4 equate resolves into the PCAUTH address space, then this command will display the REFRPROT state of the 4th TCB in that Space.

If the TCB#4 equate resolves to any other space, then this command will fail.

LIST REFRPROT REAL
Come-on!. Are you serious?

Help COmmands LISt REGIsters

Z/XDC supports the display of five types of registers:

- General registers
- Access registers
- Control registers
- Floating point registers
- Vector registers

For two of these types (general and access), Z/XDC (and the System) maintains two sets:

- **An Error Level Set:** These are the registers that existed at the time the ABEND or the breakpoint occurred that caused Z/XDC to receive control.
- **A Retry Level Set:** These are the registers that will exist should Z/XDC be used to cause the user's program to resume execution.

For the remaining registers (control, vector and floating point), z/OS saves their status only at the thread level (task or SRB), not at the request block level. Consequently, their "error level" values and their "retry level" values are one and the same. (z/OS can get away with this because it carefully manages the control registers and it never alters user vector and floating point registers.)

For more information, follow the several crosslinks shown above.

Help COmmands LISt REGS

The **LIST REGS** command displays the lo-order halves (4 bytes) of the whole set of retry level general registers. For more information, see HELP COMMANDS LIST GENERALREGISTERS.

Help COmmands LISt REXx

The **LIST REXX** command displays information about the currently active rexx/XDC

Interface (if any). The information displayed includes:
- REXX's version and language code
- input ddname
- output ddname
- The REXX library's name, ddname, and search order
- The names of all subcommand environments
- The names of all function packages and the functions they contain
- the current settings of input and output routing

The **LIST REXX** command is **identical** to the **REXX LISTREXX** command.

Syntax:

LIST REXX

This command accepts no operands.

Help COmmands LISt RH#

The **LIST RHn** command displays the hi-order half (4 bytes) of an individual retry level general register. For more information, see HELP COMMANDS LIST GENERALREGISTERS.

Help COmmands LISt RHRegs

The **LIST RHREGS** command displays the hi-order halves (4 bytes) of the whole set of retry level general registers. For more information, see HELP COMMANDS LIST GENERALREGISTERS.

Help COmmands LISt RSa

The **LIST RSA** command generates a display of a chain of register save areas (RSAs).

The **LIST RSA** command understands all the RSA formats as documented in:
- The IHASAVER Assembler macro
- Chapter 2 of the Assembler unauthorized guide
- And 'format 2' RSAs

You may give the address of any RSA in the chain to be displayed. The **LIST RSA** command will then produce a list of all RSAs chained both before and after the given RSA.

Syntax:

LIST RSA [operands ...]

The following operands can be specified (in any order) (and can be abbreviated as long as they are unique):

```
starting-address
LIMIT=number
EQUATES=option
TCB=option
FORWARD=option
BIAS=option
CODEABOVEBAR=option
SUMMARY
DETAILS
```

starting-address
This must be the address of a RSA located in the chain that you want displayed, or the address of a task control block (TCB), in which case the save area pointed to by the TCBFSA field in that TCB will be used as the starting RSA. If omitted, then provided that foreign address space mode is **not** in effect, the address pointed to by the retry level R13 will be used as the default.

LIMIT=number
This operand sets a limit to the number of entries followed or displayed.
The range is 1 to 9999. The default is 99.

EQUATES=option
this operand can be used to control the generation of equates.
Specifying **EQUATES=YES** means that equates will be generated.
Specifying **EQUATES=NO** means that equates will not be generated.
The default is EQUATES=YES.

TCB=option
this operand can be used to control the detection of an RSA that is a TCB's first save area.
Specifying **TCB=YES** means that if an RSA has no previous RSA, has a 31-bit address, and is a TCBs first save area, it will be so indicated.
Specifying **TCB=NO** means that no check for an RSA being a TCBs first save area will be done.
The default is TCB=YES.

FORWARD=option
As explained below, Z/XDC can follow the RSA forward chain. This operand can be used to control this.
Specifying **FORWARD=YES** means that Z/XDC will attempt to follow the forward chain.
Specifying **FORWARD=NO** means that Z/XDC will not attempt to follow the forward chain.
The default is FORWARD=YES.

BIAS=option
when the last RSA in a chain is determined, Z/XDC may be unable to determine the RSA's type. (normally the next RSA in a chain determines the type of the previous RSA).
In this case, the **BIAS=** operand can be used to set the type of the RSA.
The values allowed are:

BIAS=LEGACY	- To indicate that the save area is a legacy format save area. (this is the default).

BIAS=FMT4 - To indicate that the save area is a format 4 save area.

CODEABOVEBAR=option
Z/XDC will attempt to format the return address and entry point address in an RSA. If the R14 and R15 slots in an RSA are 8 bytes, the high words of the R14 and R15 values may have non-address data. Using **CODEABOVEBAR=NO** will cause the high word of the R14 and R15 slots to be ignored.
Specifying **CODEABOVEBAR=YES** means that Z/XDC will use the entire 8-byte R14/R15 value when available.
Specifying **CODEABOVEBAR=NO** means that Z/XDC will ignore the high word of the R14/R15 value.
The default is CODEABOVEBAR=NO
This operand has an alias of CODE-ABOVE-BAR= and can be abbreviated to C=.

SUMMARY
The RSA display will always have only one line per RSA. The flags field in that line will indicate some things about the RSA.
(This is the default if both **SUMMARY** and **DETAILS** are omitted).

DETAILS
The RSA display may have additional lines after the first line for a specific RSA. These additional lines will have more detail about the RSA and will describe any errors found in detail.

Notes

Only some RSA formats have the ALET of the previous RSA. Because of this, the **LIST RSA** command only allows RSAs to be in the primary aspace. If the ALET in an RSA is not zero, an error is indicated.

If equates are generated (the **EQUATES=YES** parameter was specified or defaulted), equate names in the format **RSA#n** are generated, labeling each displayed RSA. Note that any previously existing equates with the same prefix are deleted.

If the **TCB=YES** operand is specified or defaulted and the first RSA listed:
- has no previous RSA.
- has an address that is less than 2G (i.e., a 31-bit address)
- has an address equal to TCBFSA in a TCB in the same aspace.

the RSA will be marked as a TCBs first save area and the save area length will be set to 144 because (as documented) the save area space provided by the system on an ATTACH[X] is 144 bytes long, even if the space is not all used.

The return address (R14 slot in an RSA) and the entry point address (R15 slot in an RSA) may have additional information in them (i.e. they may not have a 'clean' address). If the high word of these values is present in the RSA (i.e., the RSA format allows for all 8 bytes of the register) if will be ignored if it is X'FFFFFFFF' (this is the 'unused' value set by the operating system). Also, Z/XDC uses heuristics to determine a **good** address to use for analysis of the location.

The second word (i.e. offset X'04', for a length of 4) in an RSA often describes the **previous** RSA in the chain. Z/XDC understands this.

Following an RSAs forward chain is fraught with danger. However, Z/XDC uses heuristics to try to follow RSA forward pointers.

The **FORWARD=option** operand can be used to control this.

Z/XDC processes RSAs in the following order, up to the limit specified (or defaulted):
- The specified (or defaulted) starting RSA.
- previous RSAs, either until the limit is exhausted, or an RSA with no previous RSA is found.
- forward RSAs (unless the **FORWARD=NO** option is in effect), until the limit is exhausted, or an RSA with no forward RSA is found, (heuristics come into play here).

If an RSA loop is detected, the **LIST RSA** command will report the first RSA in the loop and will then stop reporting further RSAs.
This is because any following RSAs have already been reported.

The starting RSA will be indicated by a flag (and a comment if **DETAIL** is in effect) that indicates that this is the starting RSA. However if a loop is detected before the starting RSA is displayed, that information will not be shown.

For each RSA found, **LIST RSA** shows:
- A sequence number.
- Flags (see below).
- The RSA format (see below).
- The RSA length (in decimal).
- The return address found in the [G]R14 slot.
- The entry point address or the return code found in the [G]R15 slot.
- Any comments about this RSA (DETAIL only)
- Any errors found with this RSA (DETAIL only)

Subtopics Index

For additional information, type an **H** at the left below to select topics directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

↕	**TYPES**	- Save area types and formats understood by the **LIST RSA** command. (And the formats used in the output).
↕	**HEADINGS**	- Column headings use in the output of the **LIST RSA** command.
↕	**FLAGS**	- Flag values and their meaning in the output of the **LIST RSA** command.
↕	**F1SA**	- The meaning of the literal C'F1SA' in the second word of a legacy save area in the output of the **LIST RSA** command.
↕	**ERRORS**	- The possible errors that can be reported against a save area
↕	**DOC**	- Documentation related to save areas.
↕	**EXAMPLES**	- Examples of the **LIST RSA** command.

Help COmmands LISt RSa Types

The following values in the second word of an RSA are understood by Z/XDC:

- all 0 (previous RSA format is Legacy,
  current RSA format is Legacy).
  (all 0 means all binary 0)
- address (previous RSA format is Legacy,
  current RSA format is Legacy).
  (an address (and thus a save area) is at least halfword-aligned, and thus is never odd (Note that 'A' is EBCDIC X'C1' and is odd))
- C'F1SA' (no previous RSA - information on linkage stack,
  current RSA format is Legacy).
  (the value ends in 'A' and thus is odd)
- C'F2SA' (previous RSA format is Legacy,
  current RSA format is F2).
  (the value ends in 'A' and thus is odd)
- C'F4SA' (previous RSA format is F4,
  current RSA format is unknown).
  (the value ends in 'A' and thus is odd)
- C'F5SA' (previous RSA format is Legacy,
  current RSA format is F5).
  (the value ends in 'A' and thus is odd)
- C'F6SA' (no previous RSA - information on linkage stack,
  current RSA format is F4).
  (the value ends in 'A' and thus is odd)
- C'F7SA' (previous RSA format is F7,
  current RSA format is unknown).
  (the value ends in 'A' and thus is odd)
- C'F8SA' (previous RSA format is legacy,
  current RSA format is F8).
  (the value ends in 'A' and thus is odd)

The following RSA formats are displayed by the **LIST RSA** command:

- Legacy format.
  Displayed as **L**
  The RSA is a **traditional** save area that has the low 32 bits of each register. The previous and next save area addresses are 31 bit addresses. The access registers are not saved.

- Format 2 (F2).
  Displayed as **2**
  The RSA is more-or-less the same as a legacy RSA, except that the access registers are also saved.

- format 4 (F4).
  Displayed as **4**
  The RSA has all 64 bytes of each register. The previous and next save area addresses are 64 bit addresses. The access registers are not saved.

- format (5).
  Displayed as **5**
  The RSA has all 64 bytes of each register. The previous and next save area addresses are 64 bit addresses.
  The caller's high register halves are also saved.

- format 7 (F7).
  Displayed as **7**
  The RSA has all 64 bytes of each register. The previous and next save area addresses are 64 bit addresses.
  space is provided to allow the access registers to be saved.

- format 8 (F8).
  Displayed as **8**
  The RSA has all 64 bytes of each register. The previous and next save area addresses are 64 bit addresses.
  space is provided to allow the access registers to be saved.
  The caller's high register halves are also saved.

Please note:
- the format 2 RSA is only documented in some messages on the IBM-MAIN list in 2006.
- The format 3 RSA has never been defined.
- 'F1SA' and 'F6SA' mean that there is no previous RSA - the information has been saved on the linkage stack.

Help COmmands LISt RSa Headings

Column heading names for the **LIST RSA** command.

The column heading used in the **LIST RSA** display have the following titles:

RSA#
The assigned number of the RSA.

FLAGS
A set of flags that provide information about the RSA.

F
The RSA Format.

LEN
The RSA length (in decimal). Note that if the length is not known the length is shown as '-'.

RETURN ADDRESS (saved [G]R14)
The value and name of the 'Return To Caller' field in the RSA (Register 14). If this field in an RSA has the value **[unknowable RSA format]** then the value is not available for that RSA.

ENTRY ADDRESS (saved [G]R15)
The value and name of the 'Entry Address' field in the RSA (Register 15). Note that this field may also be a return code. If this field in an RSA has the value **[unknowable RSA format]** then the value is not available for that RSA.

Help COmmands LISt RSa FLags

The flags in the **LIST RSA** display are a 6-character field with each flag having the possible values as below.

Position 1 - Error indication.
If the value is **-** then the RSA had no detected errors.
If the value is **E** then the RSA has at least 1 error.

Position 2 - previous RSA.
If the value is **-** then there is no previous RSA.
If the value is **P** then the RSA has a previous RSA.
If the value is **L** then there is no previous RSA and the registers (etc.) have been saved on the linkage stack.

Position 3 - next RSA.
If the value is **-** then there is no next RSA.
If the value is **N** then the RSA has a next RSA.

Position 4 - TCBFSA indication.
If the value is **-** then the RSA is not a TCB's first RSA.
If the value is **T** then the RSA is a TCB's first RSA (i.e., the RSA address is in a TCB's TCBFSA field)

Position 5 - Used indication.
If the value is **-** then the RSA has not been used.
If the value is **U** then the RSA has been used.

Position 6 - starting or given RSA indication.
If the value is **-** then the RSA is not the starting RSA.
If the value is **G** then the RSA is the starting or given RSA (i.e., the RSA specified or defaulted on the **LIST RSA** command).

When the **DETAIL** keyword is used, lines that follow expand on the flag meanings.

Help COmmands LISt RSa F1sa

The meaning of the literal C'F1SA' in the second word of a legacy save area.

If a legacy save area has the literal C'F1SA' in the second word of the save area then:
- there is no previous save area (the same as word 2 being binary 0) **but** the registers (etc.) have been saved on the linkage stack.

If a legacy save area does not have the literal C'F1SA' in the second word then:
- the second word (if it is not binary zero and is not odd) points to the previous save area.

Help COmmands LISt RSa ERrors

The following errors can be reported for an RSA. Note that these errors are only displayed:
- If the FLAGS displayed for an RSA has the **E** character listed and
- If the **DETAILS** operand is used on the **LIST RSA** command.

The errors (and their meanings are):

- Not able to (fully) access RSA
  This error indicates that the entire RSA, for the length shown, could not be accessed (at least for reading). Sometimes this error indicates that a storage area was once an RSA but is now something else.

- Unknown RSA format
  This error indicates that the RSA format (The second word of an RSA) contained a value that was not reorganized. Sometimes this error indicates that a storage area was once an RSA but is now something else.

- Part of a loop. (Loop RSA# is n)
  This error indicates that the indicated RSA is part of a loop of RSAs. The RSA number that has been identified as the start of the loop is shown.

- The RSA is not correctly aligned
  An RSA must be (at least) fullword (4-byte) aligned.

- The previous RSA address is not zero
  This error indicates that the previous RSA pointer in the RSA should have been zero but was not.

- The previous RSA address must be below the bar
  This error indicates that the previous RSA pointer in the RSA should have had an address that is below the 2G bar (X'7FFFFFFF' or lower).

- The previous RSA ALET value is not zero
  This error indicates that the previous RSA pointer had an associated ALET and that ALET was not zero. Z/XDC requires that all RSAs reside in the primary aspace.

- The higher RSA's downchain points to me
  but my upchain does not point back to him
  This error indicates that the previous RSA pointer in the RSA did not point to the determined previous RSA. Sometimes this error indicates that a storage area was once an RSA but is now something else.

- The next RSA address must be below the bar
  This error indicates that the next RSA pointer in the RSA should have had an address that is below the 2G bar (X'7FFFFFFF' or lower).

- The next-RSA pointer does not point to
  the same RSA (shown next) that upchained to this RSA
  This error indicates that the next RSA pointer in the RSA did not point to the determined next RSA. Sometimes this error indicates that a storage area was once an RSA but is now something else.

Help COmmands LISt RSa Doc

More information on RSAs can be found in the following places:

- The IBM manual **MVS Programming: Assembler Services Guide** (SA23-1369) in chapter 2.

- The **IHASAVER** Assembler macro has most of the RSA formats documented in it.

- Format 2 RSAs are only documented in some posts on the IBM-MAIN list in 2006.

- Format 3 RSAs have not been defined.

- The 'F1SA' and 'F6SA' strings are documented in the IBM manual **MVS Programming: Extended Addressability Guide** (SA23-1394).

Help COmmands LISt RSa EXamples

Examples of the use of the **LIST RSA** command.

L RSA
L RSA R13?
These two commands are equivalent. They both display the RSA chain containing the RSA pointed to by retry level R13.

L RSA LIMIT=3
This displays up to three RSAs starting with the RSA pointed to by retry level R13.

USING TCB.TCBRBP 21C%
L RSA .TCBFSA?
The USING command assigns a dsect map to map the Task Control Block for the current task. The **L RSA** command then displays the RSA chain that is anchored from that TCB's TCBFSA field.

L RSA 21C%
Like the previous example, this **L RSA** command displays the chain of save areas for the currently active TCB. This example uses the fact that if a TCB address is provided to the **L RSA** command, the RSA anchored in that TCB's TCBFSA is used as the starting RSA.

L RBS
L RSA RB#2+54%
The "L RBS" command, in addition to displaying the current TCB's Request Block chain, creates a series of equates that label those RBs. The "L RSA" command then displays the RSA chain that contains the RSA pointed to by the copy of R13 that is saved in RB#2. (Remember that because of the way in which the System saves registers when RBs are created, the registers saved in RB#2 actually belong to the program that was running under the control of RB#1!)

Help COmmands LISt RW#

The **LIST RWn** command displays all 8 bytes of an individual retry level general register. For more information, see HELP COMMANDS LIST GENERALREGISTERS.

Help COmmands LISt RWRegs

The **LIST RWREGS** command displays the entire 8 bytes of the whole set of retry level general registers. For more information, see HELP COMMANDS LIST GENERALREGISTERS.

Help COmmands LISt SCBs

This command is an alias of the **LIST ESTAES** command. See HELP COMMANDS LIST ESTAES for more information.

Help COmmands LISt SCReen

This command is an alias of the **LIST WINDOW** command. See HELP COMMANDS LIST WINDOW for more information.

Help COmmands LISt SEArchorder

The **LIST SEARCHORDER** command shows the load module search order (as used by the **ATTACH[X], LINK[X], LOAD,** and **XCTL[X]** Assembler macros) for any given TCB in any accessible address space.

Syntax:

```
LIST SEARCHORDER [ tcbaddress ]
                 [ ALL        ]
                 [ JOBLIB     ]
                 [ LINKLIST   ]
                 [ PLPA       ]
                 [ STEPLIB    ]
                 [ TASKLIBS   ]
                 [ NONE       ]
                 [ NOJOBLIB   ]
                 [ NOLINKLIST ]
```

```
                    [ NOPLPA     ]
                    [ NOSTEPLIB  ]
                    [ NOTASKLIBS ]
```

Operands

Note that operands can be entered in any order. Operands are processed in the order specified.

A keyword can be abbreviated, as long as it is unique. (Except that A is treated as ALL, and N and NO are treated as NONE)

If no keywords are specified, processing is as if **NONE** had been specified.

tcbaddress The address of any accessible TCB. If omitted, the current TCB is used. If in Foreign Address Mode (FASM), a TCB address is required.

ALL This keyword can be used to force all dsnames to be displayed. (Note that this keyword can also be specified as A).

JOBLIB This keyword is an alias of the **TASKLIBS** keyword.

LINKLIST This keyword can be used to force the Linklist dsnames to be displayed.

PLPA This keyword can be used to force PLPA dsnames to be displayed.

STEPLIB This keyword is an alias of the **TASKLIBS** keyword.

TASKLIBS This keyword can be used to force task library dsnames to be displayed.

NONE This keyword can be used to prevent any dsnames from being displayed. (Note that this keyword can also be specified as N or NO).

NOJOBLIB This keyword is an alias of the **NOTASKLIBS** keyword.

NOLINKLIST This keyword can be used to prevent the Linklist dsnames from being displayed.

NOPLPA This keyword can be used to prevent PLPA dsnames from being displayed.

NOSTEPLIB This keyword is an alias of the **NOTASKLIBS** keyword.

NOTASKLIBS This keyword can be used to prevent task library dsnames from being displayed.

APF Authorization.

The **LIST SEARCHORDER** command displays the APF authorization of 2 items:

- if a TCB has a Task library open, the APF authorization of that open library is displayed.
- The APF authorization status of The individual dataset or datasets (if there is a concatenation) for that open library is also displayed (provided that datasets for a library are being displayed). Note that for foreign systems (e.g., dumps) this information is not available.

Note that normally if at least **one** dataset is not APF authorized then the library is not considered to be APF authorized.

However it is possible that although there is at least **one** non-APF-authorized dataset in the library concatenation that the library is still regarded as APF authorized.

There are 2 reasons why this could happen:

- the first reason is that the dataset(s) could have been removed from the list of APF Authorized datasets after the relevant library has been opened.
- The second reason is that an program with sufficient authority could have set the bit in the DEB (Data Extent Block) that indicates APF Authorization. Z/XDC may do this in some cases. (see **HELP XDCCALL TASKLIB** for more information).

Examples:

L SEARCHORDER ALL
The search order of the current task is displayed. Also, the list of PLPA and Linklist dsnames is displayed.

L SEARCHORDER ALL NOLINKLIST
The search order of the current task is displayed. This is similar to the last example, except that the list of LINKLIST dsnames is not displayed.

L SEARCHORDER 21C?+7C?
The search order of the current jobstep TCB is displayed.

L TASKS 1
L SEA TCB#20 ALL
Assuming that the XDC session is authorized...
The first command ('L TASKS 1') displays a list of the tasks in ASID 1 (normally *MASTER*).
The second command ('LIST SEA TCB#20 ALL') will show the search order of the task number 20 in address space 1. All dsnames are shown.

L TASKS
L SEARCHORDER TCB#2
This displays the search order of the second task in the Target Address Space. The L TASKS command generates the TCB#2 equate and assigns it to the location of the Target Address Space's second TCB.

Help COmmands LISt SECOndarysize

The **LIST PRIMARYSIZE** command shows whether Z/XDC's Fullscreen Support will attempt to use the display terminal's primary or secondary screen geometry, when available.

The **LIST SECONDARYSIZE** command is a synonym of the "LIST PRIMARYSIZE" command.

Syntax:

LIST PRIMARYSIZE
SECONDARYSIZE

This command accepts no operands.

The terminal display geometry setting can be saved in your session profile. It can be changed by the SET PRIMARYSIZE and SET SECONDARYSIZE commands. It also can be displayed and changed by Z/XDC's "Profile Menuing System". For more information, please see:

HELP COMMANDS SET PRIMARYSIZE
HELP COMMANDS SET SECONDARYSIZE
HELP PROFILES MENU

Help COmmands LISt SECUrity

The **LIST SECURITY** command displays information about Z/XDC's interface to your Data Center's security system. (For more information about Z/XDC's security interface, see HELP SECURITY.) Specifically, the following report messages are produced:

SECURITY SYSTEM:
The name of the installed security system is displayed. Z/XDC supports the three major security systems:
- IBM's RACF
- Computer Associates' ACF2
- Computer Associates' Top Secret (TSSF)

RULES SYNTAX:
Essentially, all this report line shows is whether or not all Z/XDC-related resource names start with the qualifier **XDC.** When the security system is RACF, then they do. When the security system is ACF2 or TSSF, then they do not.

SECURITY USERID:
This report line shows the ownerid in whose name Z/XDC makes its security calls. When debugging TSO-running programs, this will, of course, match the current TSO userid. When debugging batch jobs, this will be the job's security assigned ownerid (if any). (In all cases, Z/XDC obtains this information from the ACEEUSER field of

the ACEE control block.)

SECURITY TRACE:
The SET SECURITY command can be used to cause information to be displayed about all security calls made by Z/XDC. This report line shows whether or not such displays are enabled.

MAX CACHE EXPIRATION:
This report line shows how much time remains in the life of the youngest cache entry in Z/XDC's internal security cache. For more information about Z/XDC's internal security cache, see HELP SECURITY CACHE.

Note, if you want, you can use the SET SECURITY command to force the immediate expiration of all cache entries.

Syntax:

LIST SECURITY

This command accepts no operands.

Help COmmands LISt SESsions

This command is an alias of the **LIST LICENSE** command. See **HELP COMMANDS LIST LICENSE** for more information.

Help COmmands LISt SIGnonwait

The **LIST SIGNONWAIT** command displays the period of time that cdf/XDC is permitted to wait for a programmer to sign onto a pending debugging session. The factory default time limit is one hour.

Syntax:

LIST SIGNONWAIT

This command accepts no operands.

The SIGNONWAIT setting can be saved in your session profile. It can be changed by the SET SIGNONWAIT command. It also can be displayed and changed by Z/XDC's "Profile Menuing System". For more information, please see:

HELP COMMANDS SET SIGNONWAIT
HELP PROFILES MENU

Help COmmands LISt SIZeof

This command is available only for customers who have licensed the features necessary for debugging one or more of the supported **High Level Languages** (such as the **c/XDC feature** for debugging XL C/C++ and Metal C). For more information, see HELP SUPPORT FEATURESANDCAPS.

This command allows you to display the size of storage consumed by one or more variables, arrays, structures, unions, whatever, belonging to any High Level Language program that you have mapped. (See HELP COMMANDS MAP.)

This command applies only to variables used in HLL programs. Variables defined in Assembler are referenced via classical Z/XDC syntax rules as described in topics starting at HELP ADDRESSING PARSERS ASM RESOLUTION.

Please note that because the size of a structure is not recorded in DWARF data, c/XDC has to manually calculate the size of the structure. The size reported by c/XDC may differ by a few bytes from what is reported by C's **SIZEOF()** function.

Syntax:

```
LIST SIZEOF variable variable ...
            ALL
            omitted
```

Rules:

- **ALL** may not be combined with any other operands.
- If **ALL** is given along with other **variables,** then **ALL** is treated as being a variable, not a keyword.

variables
Any number of variables may be given. One (or more, if appropriate) lines of display will be produced for each variable.

- When referring to a variable, use whatever syntax is appropriate for the HLL language in which the variable is defined.
- Generally, the variables can be anything supported by the HLL language.
  Examples:
  - Simple scalar variables
  - Array names
  - Selected array elements
  - Structure names
  - Structure elements
  - Structures within arrays within structures within ... whatever
  - Unions
  - Whatever

- The variable's storage length will be displayed.

- Variables and structures whose names are reserved as references to registers or a PSW by Z/XDC can be specified through use of one of the variable pool built-in equates. For Example: **L V VS#3_SV.R0**

ALL
When **ALL** is given as the sole operand, then all variables are listed from all variable pools in all **known** stack frames.

When **ALL** is given in combination with other **variables,** then it is treated as being a variable, not a keyword.

omitted
When all operands are omitted, then all variables are listed from all variable pools in just the **current** stack frame.

Comments

If you wish to view the raw storage in which the variable is defined, a Shortcut Entry Field is provided at the left for doing that. Just enter **F** or **D** command there to display the raw data.

Alternatively, use a **FORMAT variablename** command. This will display the variable's raw storage too.

Help COmmands LISt SRbs

This command is an alias of the **LIST SSRBS** command. See HELP COMMANDS LIST SSRBS for more information.

Help COmmands LISt SSCt

Subsystem Control Tables (SSCTs) identify registered subsystems in the z/OS Operating System. The **LIST SSCT** command displays some or all Subsystem Control Tables currently existing in the system.

For each SSCT, the following information is displayed:

- The name of the subsystem.

- The status of the subsystem (active vs. inactive).

- Whether the subsystem has active and/or standby SSVTs (Subsystem Vector Tables).

- Whether the SSCT is for an XDC. If so, then:
  - What version of XDC created it.
  - Whether the SSCT has been deactivated by XDC.

- Whether the subsystem is the primary subsystem.

- Whether the subsystem is a dynamic subsystem.

- Whether the subsystem responds to SETSSI commands.

- A list of function codes handled by the active SSVT and, if available, descriptive information about what these function codes represent.

In addition, the **LIST SSCT** command numbers the SSCTs in queued order.

This command displays the selected SSCTs in either alphabetic order or queue order. When displayed in alphabetic order, the listed SSCT numbers will not be sequential.

Syntax:

```
LIST SSCT   ssnamepattern  FUNCTIONS                  SORT=NAME
           'ssnamepattern' FUNCTION=ssfunctionnumber  SORT=QUEUE
           "ssnamepattern" NOFUNCTIONS
```

ssnamepattern
This operand is optional. It identifies one or more subsystems to be displayed. If given, then:
- It must be a character string, 1 to 4 characters long.
- All typeable characters, not having a role in parsing, are permitted. (In other words, blanks, commas, semicolons and quote characters are not permitted. But see below.)
- All characters are treated literally except wildcard characters:
  - **?** matches any single character.
  - * matches any string of zero, one or more characters.

  For more details, see **HELP COMMANDS SYNTAX PATTERNMATCH.**
- The given name pattern is not case sensitive; it will always be upcased. (But see below.)

If this operand is omitted, then **all** registered subsystems are displayed.

'ssnamepattern'
"ssnamepattern"
If you wish to use either lowercase letters or syntactically significant characters in your SSCT name pattern, then you need to enclose it within quote characters. If your pattern includes the opening quote character, then you will need to double it up. (Note, lowercase letters will be retained as lowercase letters).

FUNCTIONS
FUNCTION=ssfunctionnumber

NOFUNCTIONS
These operands are both optional and mutually exclusive. They control whether or not the produced display is to contain information about the subsystem function exits that have been installed for each of the subsystems.

FUNCTIONS

The names and numbers of all registered subsystem functions for each displayed subsystem will be displayed. In addition, the Shortcut Entry Field for each displayed function will point to that function's starting address. (See HELP SHORTCUTCMDS ASSOCADDR for more information.)

FUNCTION=ssfunctionnumber
Only those subsystems are displayed that have registered the indicated subsystem function (and whose names match the given ssnamepattern). **Ssfunctionnumber** must be a 1- or 2-digit hexadecimal number identifying the subsystem function to be filtered for.

NOFUNCTIONS (or omitted)
The generated display will not include any information about a subsystem's registered functions. (This results in a shorter, less cluttered display.) With this option, each display line's Shortcut Entry Field points to the subsystem's SSCT (instead of to a subsystem function routine).

The default taken when none of these operands is given is NOFUNCTIONS.

SORT=NAME (or omitted)
SORT=QUEUED
These operands control whether the resulting display is sorted by subsystem name (SORT=NAME) or by the order in which the subsystem SSCTs are queued (SORT=QUEUED). Note, the queue order is the order by which the Operating System passes control to registered subsystem function routines.

The default is to display the subsystems sorted by name.

Automatic Equates

The **LIST SSCT** command (re)generates the following equates:

SSCT#n
These label the SubSystem Control Tables. All SSCTs are labeled regardless of which ones are displayed. The SSCTs are labeled in queue order.

SSVT#n
These label the SubSystem Vector Tables (when present) pointed to by the **SSCTSSVT** fields of the SSCTs.

SUSE#n
These label the Subsystem User Tables (when present) pointed to by the **SSCTSUSE** fields of the SSCTs.

Notice: the **SSCT#n, SSVT#n** and **SUSE#n** equates are **static!** They do not float. The locations that they target can change only when another **LIST SSCT** command is issued.

Help COmmands LISt SSRbs

The **LIST SSRBS** command displays a list of all suspended SRB routines found in a specified address space.

The LIST SSRBS command can be used with dump/XDC to show suspended SRBs in an address space on the dump. Most of the following description is correct, but be aware that the SSRB structure cannot change.

When displaying an SSRB list under dump/XDC, the CU (set current) shortcut can be used (to make the nominated SSRB 'current').
Also, the list includes a **CXU?** column, and if the CXU is an SSRB in the displayed address space, it is marked with YES in this column, and the line is hilighted.

In addition, the LIST SSRBS command assigns sequence numbers to the SSRBs that it displays, and it uses those to (re)generate the following automatic equates:

- **SSRB#n**
  These equates label the SSRB control blocks used to build the display. They can be used in subsequent address expressions and especially as operands of commands that call for ssrbaddress operands.

- **SSRX#n**
  In z/OS R1.13 and newer systems, these equates label the SSRX control blocks used to build the display. They can be used in subsequent address expressions and especially as operands of commands that call for **ssrbaddress** operands.

- **SSRBEP#n**
  These equates label the entry points of the SRB routines defined by the SSRBs.

If prior sets of **SSRB#n, SSRX#n** and **SSRBEP#n** equates exist, then they are deleted prior to the creation of the new equates.

Syntax:

```
LIST SSRBS  aspaceref
     SRBS   omitted
```

aspaceref
This identifies the address space for which suspended SRB information is to be displayed. Briefly, this operand can be any of the following:
- A jobname.
- A keyword indicating the home space or the retry level or error level, primary, secondary, or instruction execution address space. Examples: HOME PASID ESASID, IASID, etc.
- An ASID number.
- An address expression pointing to an ASID number.
- A register containing an ASID number.
- The name of an equate or dsect that has been assigned to represent storage in an address space. (The assigned address space is used.)

For detailed information, see HELP COMMANDS SYNTAX ASIDS.

omitted
When no operand is given, the suspended SRBs in the current target address space (either the home space or as established by the SET ASID command) will be displayed.

Examples:

LIST SSRBS JES2
If any suspended SRBs currently exist in JES2's address space, then they are displayed.

LIST SSRBS 1
If any suspended SRBs currently exist in the Master Scheduler's address space, then they are displayed. (Note, there usually is one.)

LIST SSRBS EIASID
If any suspended SRBs currently exist in the error level, instruction execution address space, then they are displayed.

LIST SSRBS PASID
If any suspended SRBs currently exist in the retry level primary address space, then they are displayed.

LIST SSRBS .ASCBASID
Assuming that the ASCB dsect map has been loaded and assigned to represent an Address Space Control Block of interest, suspended SRB routines (if any) in the address space that that ASCB describes will be displayed. (In this case, the command's operand is resolved as an address expression, and the contents of the two bytes to which that expression points is what is used.)

LIST SSRBS CR3
The retry level secondary address space's suspended SRB routines (if any) are displayed.

LIST SSRBS ECR3
The error level secondary address space's suspended SRB routines (if any) are displayed.

LIST SSRBS R5
The two low order bytes of register R5 are interpreted as an ASID, so the suspended SRB routines (if any) are displayed for the address space having that ASID.

Help COmmands LISt STATIstics

This command is intended to display a wide variety of processing statistics for z/XDC. Currently, however, it only displays information about certain GETMAINs and FREEMAINs performed internally by z/XDC.

GETMAIN/FREEMAIN Stats: The display shows both the total and net GETMAINs and FREEMAINs issued by Z/XDC, broken out by AMODE: those issued against 24-bit storage vs. those issued against 31-bit storage. It also shows information about **degraded** GETMAINs, GETMAINs that requested 31-bit storage but for which the System returned 24-bit storage.

The accounting produced by this command is fairly comprehensive. Pretty much every chuck of storage obtained or freed by Z/XDC is accounted for. (The exceptions are certain specialty allocations such as space in CSA for building SSCTs and installing service SVCs and such.)

Note, must of Z/XDC's storage is obtained from the following subpools:

SUBPOOL	**PURPOSE**
57	Permanent storage (for control blocks used to describe maps, equates, breakpoints, etc.)
58	Transient storage, i.e. storage that Z/XDC uses only while its in control and that is freed prior to returning control to the user program
59	Also transient storage

Syntax:

```
LIST STATISTICS
     STATS
```

This command accepts no operands.

Help COmmands LISt STATS

This command is an alias of the **LIST STATISTICS** command. See HELP COMMANDS LIST STATISTICS for more information.

Help COmmands LISt STEp

The **LIST STEP** command displays c/XDC's current settings related to the **STEP** command. Those settings are:

- Whether or not **AUTOSTEP** is in effect.
- Whether **STEP** (when given without operands) means **STEP IN** or **STEP OVER.**

Syntax:

LIST STEP

Example:

```
Setting     Description                                  Choices
OVER        Default action for STEP command          AC= IN OVER
YES         Auto-step through linkage routines?      AU= YES  NO
```

AC= and **AU=** refer to operands of the **SET STEP** command:
- AC= is the **ACTION=** value.
- AU= is the **AUTOSTEP=** value.

See HELP COMMANDS SET STEP for more information.

Help COmmands LISt STGlayout

The **LIST STGLAYOUT** command displays the storage layout of the current system.

The output of this command will indicate the type of system that the storage layout is being produced for:

- **HOST** will be displayed if the storage layout is for the system that Z/XDC is currently executing on.
- **DUMP** will be displayed if the storage layout is for a system in a dump.

Syntax:

LIST STGLAYOUT

This command does not accept operands.

Help COmmands LISt SUbpools

The **LIST SUBPOOLS** command shows the organization of storage subpools within any address space. Depending upon the operands chosen, it can show:
- All areas of storage allocated to a given subpool number.
- All subpools owned by or shared to a given task.
- The subpool (or free area) in which a given address falls.
- The organization of storage within CSA and SQA.
- The organization of storage within any permitted aspace’s private areas.

For Private Area reports, the information shown includes:
- The size and usage of each storage allocation region (24-bit and 31-bit) within the Private Area.
- **How much of each storage allocation region is available for GETMAINs.**
- How much low-end and and high-end storage has been used in each storage allocation region.
- The current size of the gap between the low and high usage areas.
- The start, end and length of every free block within each storage allocation region.

In addition to normal environments, the **LIST SUBPOOLS** command may be used when debugging:
- Both authorized and **non-authorized** programs,
- In **FASM** mode (Foreign Address Space Mode),
- And with **dump/XDC** for debugging dumps.

For Foreign Address Space Mode, use the **SET ASID** command (prior to the **LIST SUBPOOLS** command) to select the target aspace.

For dump/XDC, start up **IPCS** and then use the **DXDC** procedure to start a Z/XDC session against a dump.

Syntax:

```
    LIST SUBPOOLS COMMON           SUMMARY
                  PRIVATE          DETAILED
                  CURRENTTASK
                  tcbaddress
                  subpoolnumber
                  anyaddress
                  omitted
```

Notes:
- Operands appearing above within a column are mutually exclusive.
- **GLOBAL** is an alias of **COMMON.**
- **LOCAL** and **REGION** are aliases of **PRIVATE.**
- **CURRENTTCB, CURRENTCB** and **CURRENTASK** are aliases of **CURRENTTASK.**
- **DETAILS** is an alias of **DETAILED.**

- When the first operand is omitted, the default is **CURRENTTASK.**
- When the second operand is omitted, the default is **SUMMARY.**
- To give a 2nd operand while omitting the first, use two commas to denote the missing first operand. Example: **LIST SUBPOOLS,,DETAILED**

COMMON
GLOBAL
This shows the general organization of storage in the System's CSAs and SQAs.
Specifically, it shows the extents and usage levels of both the 24-bit and 31-bit:
- System Queue Area (SQA)
- System Key Common Services Area (CSA)
- User Key Common Services Area (RUCSA)

When **SQA** has **overflowed** into CSA, that fact is reported.

When **DETAILED** is given, the following are added to the report:
- A list of CSA and SQA subpools and how much storage they each use,
- Lists of unallocated storage blocks.

PRIVATE
REGION
LOCAL
This shows the general organization of storage, including free blocks, in the target aspace's private areas.

Also, enough information is shown that storage shortages can be diagnosed that result from the encroachment of high-end private area allocations (LSQA for example) into low-end private storage. (See **HELP COMMANDS LIST SUBPOOLS REPORTS** for more information.)

When **DETAILED** is given, the following are added to the report:
- A list of which Private Area subpools are in use and how much storage they each use,
- Lists of unallocated storage blocks.

CURRENTTASK
CURRENTASK
CURRENTTCB
CURRENTCB
omitted
This displays the subpools allocated to the current task. This operand cannot be used when in Foreign Address Space Mode (FASM).

tcbaddress
This must be the address of some TCB in any address space. The subpools either owned by or shared to that task are shown.

Note, The **LIST TASKS** command can be used to generate a series of **TCB#n** equates representing the locations of each TCB in the address space. These equates make perfectly good tcbaddress operands for the **LIST SUBPOOLS** command. Example:

```
LIST TASKS
LIST SUBPOOLS TCB#3
```

anyaddress
This causes the **LIST SUBPOOLS** command to display the area of storage and (if appropriate) the subpool or free block containing the given address. **Anyaddress** may be an address expression that resolves to any private or common area address located in any accessible address space. It should **not** be:
- The location of any TCB located in any accessible aspace anywhere.
- A pure decimal number less than 256. (The **LIST SUBPOOLS** command treats **10,** for example, as a subpool number but **10+0** as an address.)

When **anyaddress** is given, the **SUMMARY** and **DETAILED** operands, if also given, are ignored.

This form of the **LIST SUBPOOLS** command causes both the "current display pointer" **(@CDP)** and the "next display pointer" **(@CDP)** to be set to the given address. For more information, see **HELP ADDRESSING IMPLICIT CDP.**

subpoolnumber
Subpoolnumber must be a pure decimal number ranging from **0** to **255.**

The **LIST SUBPOOLS subpoolnumber** command produces a report of all instances of the specified subpool located either in common storage or in the private area of whatever address space is currently being accessed via FASM mode.

Notes regarding Private Area Subpools:

- It is possible for any task to create and own its own private instance of any private area subpool. Thus, **multiple instances** of a given subpool may exist within an address space.

- It is possible for any task to share any or all of its subpools to subtasks.

- **DETAILED** reports show a message for each and every storage block allocated to a subpool.

- Each detail line will include information about which task owns that storage block and to which tasks that block is shared.

- This information is conveyed in the **TCB#** column of the report. See **HELP COMMANDS LIST SUBPOOLS REPORTS TCB#** for details.

- Allocated storage blocks can be FREEMAIN’d only by owning and sharing tasks.

- When an owning task terminates, all storage belonging to subpools that it owns are automatically freed.

DETAILED
DETAILS
A detailed report is produced. The report will include various descriptive messages that show:
- (For subpool reports) all blocks of storage allocated to the subpool,
- (For storage allocation region reports) all unallocated blocks of storage within the storage allocation region.

Obviously, it is possible for this report to become quite lengthy.

DETAILED is not applicable to **LIST SUBPOOLS anyaddress** reports. If given, it is ignored.

SUMMARY
omitted

A summary report is produced. Various description messages and totals are displayed, but descriptions of individual storage blocks are suppressed.

SUMMARY is not applicable to **LIST SUBPOOLS anyaddress** reports. If given, it is ignored.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

REPORTS - Explains the reports produced by some of **LIST SUBPOOLS** operands.

EXAMPLES - Illustrates some of the reports produced by the **LIST SUBPOOLS** command.

Help COmmands LISt SUbpools Reports

The reports produced by the **LIST SUBPOOLS** command vary according to the type of operand used with the command. This topic describes those reports in detail.

LIST SUBPOOLS may produce any of six distinct reports:

- A **Summary Subpools Report** just shows total allocations for selected subpools.
- A **Detailed Subpools Report** adds a listing of individual allocated blocks to the report for each selected subpool.
- A **Private Area Report** displays key information about the 24-bit and 31-bit storage allocation regions in a selected address space’s Private Area. Note, this particular report can be used to diagnose limitations arising from improper use of the REGION= JCL parameter.
- A **Private Detailed Report** adds a listing of unallocated storage blocks and a subpool totals list to the report.
- A **Common Area Report** displays key information about about SQA, CSA and RUCSA.
- A **Common Area Detailed Report** adds a listing of unallocated storage blocks and a subpool totals list to the report.

Report Columns

The various **LIST SUBPOOLS** reports will contain anywhere from 5 to 11 columns of information selected from the possibilities described further in the following subtopics. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

CAPTION - This column identifies line by line the information being presented in the report.

SPID - This, together with the **CATEGORY** column, identifies the subpool by number (SPID) and type (CATEGORY).

WHERE - The **START, END, LENGTH** and **[DEC]** columns describe where a block of storage is located and how long it is.

USED - The **USED%** column reports the proportion of the storage block (or storage allocation region) that is in use.

NOTES - The **NOTES, REAL** and **KY** columns provide additional information specific to individual blocks of allocated storage.

TCB# - For Private Area subpools, this column lists the tasks that the subpool is owned by and shared to.

Help COmmands LISt SUbpools Reports Caption

The various LIST SUBPOOLS reports will contain anywhere from 5 to 11 columns of information selected from twelve possibilities. This topic discusses the **CAPTION** column. This column is used by **LIST SUBPOOL PRIVATE** and **LIST SUBPOOL COMMON** reports to describe the row by row information being displayed. Possible captions include the following.

For LIST SUBPOOLS PRIVATE Reports

SYSTEM REGION: This is a small area of storage located at the start of the 24-bit Private Area. It's for the use of:
- The Region Control Task (IEAVAR00)
- Started Task Control (IEESB605)
- The Dump Task (IEAVTSDT)

When you use the **LIST TASKS** command, you will see these tasks included in the report.

24-BIT PRIVATE
31-BIT PRIVATE: These are the Private Areas available to user programming.

REGION= LIMIT: These are the theoretical limits of storage available to user programming in the 24-bit and 31-bit Private Area storage allocation regions. Notes:
- The REGION= area starts at the low-ends of the Private Area storage allocation regions.
- The actual GETMAIN limits may be less if high-end usage has encroached down into the REGION= area.
- Except for encroachment, high-end usage does not count against the REGION= limit.
- The 24-bit and 31-bit REGION= limits are set from a combination of:
  - User JCL,
  - Installation limits and defaults (as expressed through an IEFUSI SMF exit

and an SMFLIMxx PARMLIB member),
- z/OS limits and defaults.

ACTUAL GM LMT: These are the actual GETMAIN limits for the 24-bit and 31-bit Private Area storage allocation regions. They will be the REGION= limit reduced by high-end encroachment, if any.

LO-END USAGE
HI-END USAGE: z/OS splits its subpool locations between those that obtain storage from the low-end of a storage allocation region vs. those that obtain from the high-end. Generally:
- User subpools and certain System subpools obtain from the low-end,
- While other System subpools obtain from the high-end.

z/OS maintains a current high pointer for low-end usage and a current low pointer for high-end usage. Storage gets constrained when the two pointers meet.

GAP: This is the current size of the gap between low-end and high-end usage. Programs start failing when the length of this gap reaches zero.

For LIST SUBPOOLS COMMON Reports

CSA
ECSA: In z/OS R2.3 and older systems, these captions label lines that report the starts, ends, lengths and usage levels for 24-bit and 31-bit Common Service Area (CSA).

SystemKey CSA
SystemKey ECSA: In z/OS R2.4 and newer systems, these captions label lines that report the starts, ends, lengths and usage levels for 24-bit and 31-bit System Key CSA.

These reports do not include space taken by the User Key CSA (RUCSA), if any.

UserKey CSA
UserKey ECSA: In z/OS R2.4 and newer Systems, if User Key CSA is present, then these captions label lines that report the starts, ends, lengths and usage levels for 24-bit and 31-bit RUCSA storage allocation regions, when present.

SQA
ESQA: These captions label lines that report the starts, ends, lengths and usage levels for 24-bit and 31-bit Common Service Area (SQA).

NOTICE:
If **SQA** has **overflowed** into CSA, a notice will be emitted alerting you to that fact.

For Both
LIST SUBPOOLS PRIVATE Reports
LIST SUBPOOLS COMMON Reports

Free Block: This caption labels lines that report the starts, ends, and lengths of an area’s unallocated blocks of storage.

Help COmmands LISt SUbpools Reports Spid

The various LIST SUBPOOLS reports will contain anywhere from 5 to 11 columns of information selected from twelve possibilities. This topic discusses the **SPID** and **CATEGORY** columns. These columns are used by most **LIST SUBPOOL** reports to identify the pertaining subpool both by number (SPID) and by type (CATEGORY).

The subpool numbers are, of course, simply decimal numbers ranging from 0 to 255.

The categories can be any of the following:

PVT - The subpool is located in 24-bit Private storage.
EPVT - The subpool is located in 31-bit Private storage.

LSQA - The subpool is located in 24-bit Local System Queue Area.
ELSQA - The subpool is located in 31-bit Local System Queue Area.

CSA - The subpool is located in 24-bit Common Services Area.
ECSA - The subpool is located in 31-bit Common Services Area.

SQA - The subpool is located in 24-bit System Queue Area.
ESQA - The subpool is located in 31-bit System Queue Area.

Note, when storage is located in User Key CSA **(RUCSA),** it is still described in this CATEGORY column as being just CSA or ECSA. However, it will be annotated in the **NOTES** column as being in **RUCSA.**

Help COmmands LISt SUbpools Reports Where

The various LIST SUBPOOLS reports will contain anywhere from 5 to 11 columns of information selected from twelve possibilities. This topic discusses the **START, END, LENGTH** and **[DEC]** columns. They are used by most **LIST SUBPOOL** reports to show **where** a storage block or section is located and how long it is.

- The **START, END** and **LENGTH** columns provide the exact starting address, ending address and length of a storage block or section. They are shown as hex values.

- The **[DEC]** column reshows the storage’s length as a scaled, approximate decimal value.

Help COmmands LISt SUbpools Reports Used

The various LIST SUBPOOLS reports will contain anywhere from 5 to 11 columns of information selected from twelve possibilities. This topic discusses

the **USED%** column. For a storage block or storage allocation region, this shows how much of that storage is in use.

The **USED%** column has some subtleties regarding what it displays...
100% - Indicates that the area is exactly 100% used.
100.0% - Indicates that the area is nearly 100% used, but not quite.
0% - Indicates that the area is completely unused.
0.0% - Indicates that the area is nearly (but not quite) completely unused.
nn.n% - Indicates the percentage of the area that is used. The value has been rounded to the nearest tenth.

Help COmmands LISt SUbpools Reports Notes

The various LIST SUBPOOLS reports will contain anywhere from 5 to 11 columns of information selected from twelve possibilities. This topic discusses the **NOTES, REAL** and **KY** columns. They are used by those **LIST SUBPOOL whatever DETAILED** report lines that describe individual blocks of allocated storage:

- The **KY** column reports a block's storage access key as a decimal number. Any value **0** to **15** can be shown.

- The **REAL** column shows where, in real storage, the block's pages would be backed in the event that they were page fixed. The possible values are:
  R24 - Restricted to 24-bit real storage
  R31 - Restricted to 24-bit or 31-bit real storage
  R64 - Not restricted. May be backed by any real storage.

- The **NOTES** column is used to report miscellaneous annotations of interest:

  ÊXEC - The storage cannot contain executable code.

  HARDC - The storage's location and length is hard-coded:
  - In the GDA (for SQA)
  - In the LDA (for LSQA)

  LIST SUBPOOLS replicates z/OS's behavior or reporting these areas as if they were in subpools 245 and 255, respectively.

  MPAGE - The storage is backed in real storage by 1-megabyte pages.

  RUCSA - The storage is located in User Key CSA.

  OFLOW - This tags SQA subpool blocks that have been allocated in CSA storage due to **SQA overflow.**

  UNUSE - The storage has been flagged (in the DQE) as being unusable.

Help COmmands LISt SUbpools Reports Tcb#

The various LIST SUBPOOLS reports will contain anywhere from 5 to 11 columns of information selected from twelve possibilities. This topic discusses the **TCB#** column. This column is used by **LIST SUBPOOL whatever DETAILED** reports that report on Private Area subpools. This column reports first the task that owns the subpool, then all tasks to which the subpool is shared. Notes:

- The tasks are identified by their TCB sequence numbers as they are shown in a **LIST TASKS** display.
- The first displayed number identifies the task that owns the subpool.
- Subsequent numbers identify the tasks to which the subpool has been shared.
- Consecutive TCB sequences numbers (e.g. 5 6 7 8 11) are gathered together and displayed in range format (e.g. 5-8 11).
- When the report shows that multiple tasks own the same subpool number, that indicates that multiple instances of that subpool do, indeed, exist. But they are completely unrelated to and independent of each other. They only happen to have the same subpool number. There is no significance in that.

Help COmmands LISt SUbpools Examples

What follows are several use case examples for the **LIST SUBPOOLS** command. For detailed syntax, see **HELP COMMANDS LIST SUBPOOLS.**

Examples: LIST SUBPOOLS tcbaddress DETAILED

These are examples of commands that produce subpool reports of all subpools either owner by or shared to a given task.

```
LIST SUBPOOLS,,DETAILED
LIST SUBPOOLS CURRENTTASK DETAILED
LIST SUBPOOLS 21C?        DETAILED

LIST TASKS
LIST SUBPOOLS TCB#7       DETAILED
```

The first three commands reference the current task. The 4th and 5th commands illustrate referencing a task other than the current task:

- The **LIST TASKS** command, as a side effect, creates a series of **TCB#n** equates to label all of the aspace's Task Control Blocks.
- The **LIST SUBPOOLS TCB#7** command then uses one of those equates to reference the TCB whose subpools are to be displayed.

Detailed reports are produced: One message is emitted for each block of storage found in each subpool.

For a Summary Report, change the DETAILED operand to **SUMMARY** (or omit it altogether). Then only a **Totals** line is shown for each subpool.

Examples: LIST SUBPOOLS subpoolnumber DETAILED

These illustrate commands that produce Subpool Reports of **all** instances of a given subpool. Any subpool number, from 0 to 255, can be given. If the subpool exists, then a display is produced that reports on all blocks of storage that are owned by any instance of the subpool.

LIST TASKS

LIST SUBPOOLS 0 DETAILED
LIST SUBPOOLS 2 DETAILED
LIST SUBPOOLS 245 DETAILED
LIST SUBPOOLS 57 DETAILED
LIST SUBPOOLS 58 DETAILED
LIST SUBPOOLS 59 DETAILED
LIST SUBPOOLS 78 DETAILED

For task owned subpools, it might be a good idea to issue a LIST TASKS command ahead of the LIST SUBPOOLS command. (Example: **LIST TASKS;;LIST SUBPOOLS 0)** That will make it easier to understand the ownership information displayed in the **TCB#** column.

Subpool Notes:
- Subpool 2 is used by the C Language and by LE for working storage.
- Subpool 57 is used by Z/XDC for all of its persistent storage allocations.
- Subpools 58 and 59 are used by Z/XDC for most of its transient storage allocations.
- In TSO, subpool 78 generally is shared across all TSO command subtasks. Subpool 0 is not!
- In batch, the opposite is true: Subpool 0 is generally shared across all subtasks, while subpool 78 has no special significance.

Subpool Categories:
PRIVATE - is comprised of subpools 0-127, 129-132, 229-230, 236-237, 249 and 251-252.
LSQA - is comprised of subpools 205, 215, 225 and 255.
CSA - is comprised of subpools 227, 228, 231 and 241.
SQA - is comprised of subpools 226, 239, 245, 247 and 248.

Examples: LIST SUBPOOLS addressexpression

These illustrate commands that identify and describe the subpool (or free area) into which a given address expression resolves.

LIST SUBPOOLS 21C%+1
LIST SUBPOOLS TCB#3+1
LIST SUBPOOLS R1?
LIST SUBPOOLS PSW?

Note, if an address expression (such as **21C%)** resolves to the starting address of a TCB, then that's a different command. In that case, the subpools owned/shared by that task are displayed.

However, if an address expression resolves to an address that's **within** a TCB (such as **TCB#3+1** or **21C%+1),** then information about the storage **containing** that TCB control block is displayed.

For **LIST SUBPOOLS addressexpression** commands, the following information is displayed:

- If the address resolves into a load module, an equate, a dsect map or some other object, then offset information is shown with respect to that object.

- Offset information is also displayed with respect to the major areas of storage (PSA private area, CSA, PLPA, nucleus, etc.) into which the address resolves.

- If the address resolves into an area of storage that is managed with subpools, (CSA, SQA, most of the private areas), then a Subpools Report also is produced. It may show any of the following:

  - **subpoolnumber**
    This indicates that the address resolves into allocated storage, and the report describes the subpool chunk into which the address resolves.

  - **FREE FRAG**
    This indicates that the address resolves into an unallocated fragment of an allocated block of storage.

  - **FREE BLKS**
    This indicates that the address resolves into an entirely unallocated block of storage.

Example: Private Area Report
Example: Common Storage Report

LIST SUBPOOLS PRIVATE
LIST SUBPOOLS COMMON

These commands produce storage allocation region reports that comprehensively describe the current layouts of Private or Common storage. Each report line is uniquely captioned. For a detailed discussion of the captions, see **HELP COMMANDS LIST SUBPOOLS REPORTS CAPTION.**

Miscellaneous Additional Examples:

LIST SUBPOOLS
LIST SUBPOOLS,,SUMMARY
LIST SUBPOOLS 21C%
LIST SUBPOOLS CURRENTTASK
All of these commands do exactly the same thing: They display a summary of all subpools owned by or shared to the current task.

SET ASID JES2

LIST TASKS
LIST SUBPOOLS,TCB#3 DETAILED
These commands display a detailed report of the subpools owned by or shared to the JES2 address space's "third" task. Notes:

- The **SET ASID** command can be issued only when Z/XDC is running authorized, and only when permitted by Security.

- The **LIST TASKS** command displays tasks in tree sequence order. So the "third" task will be the task that is shown third by the command.

- The **LIST TASKS** command also automatically creates a series of equates named **TCB#n** that label the starting locations (i.e. both address and address space) of all TCBs in the address space currently being displayed. These equates can then be used in address expressions for referencing those TCBs by address.

- So the **TCB#3** equate labels the location of the "third" TCB located in the JES2 address space.

- All **TCB#n** equates persist until the next time a **LIST TASKS** command is issued.

- In particular, **TCB#n** equates persist even after another **SET ASID** command changes or ends Foreign Address Space Mode. This allows subsequent **LIST SUBPOOLS TCB#n** commands still to reference the prior aspace.

For more information, see **HELP COMMANDS LIST TASKS.**

LIST TASKS
LIST SUBPOOLS,TCB#3+1
This describes the storage that **contains** TCB#3 (not the subpools owned by TCB#3). That's because the **TCB#3+1** operand is an address expression that resolves to a location that is **not** the starting location of a Task Control Block.

LIST SUBPOOLS 78 DETAILED
This produces a detailed report showing all instances of subpool 78 occurring in the address space.

LIST SUBPOOLS PRIVATE
This produces a summary report showing the usage of storage within an address space's private area. The information shown includes:

- The location and size of each private area's storage allocation region (24-bit and 31-bit).

- The amounts of low-end and high-end storage that is being used.

- The current size of the gap between low-end usage and high-end usage.

- The REGION= limit for each storage allocation region.

- The actual GETMAIN limit for each storage allocation region. This will be either:

    - The REGION= limit,
    - Or less if high-end usage has encroached into the REGION= area.

- The locations and sizes of all free areas within each storage allocation region.

- Information about something called the **System Region** located at the start of the 24-bit Private Area.

Help COmmands LISt TAsks

The **LIST TASKS** command shows the organization of all TCBs located in any accessible address space. Subtasks are shown below (but not necessarily immediately below) their parent tasks and indented to the right. Sister tasks are shown with the same indentation.

The LIST TASKS command can be used with dump/XDC to show tasks in an address space on the dump. Most of the following description is correct, but be aware that the task structure cannot change, and no tasks can be PENDING XDC.

When displaying a task list under dump/XDC, if a task is current and it corresponds to one of the listed tasks it will be indicated by the line being hilighted. Also, the S (Switch) shortcut cannot be used, and the CU (set current) shortcut can be used (to make the nominated TCB 'current', and the newest (top) RB will be used). Also, if the CXU is a task, and the CXU is in the nominated (or defaulted) aspace, the CXU will be indicated by **(CXU)** on the line and that line will be hilighted.

In addition to the subtask structure, the LIST TASKS command also shows the following information:

- The current task is labeled as such. The current task is the task under which Z/XDC is running. Therefore, this is a task that has ABENDed or reached a breakpoint and consequently, has passed control to Z/XDC for debugging.

- If multiple tasks have passed control to Z/XDC, then those that are not the current task are labeled as "PENDING XDC". You may use Z/XDC's SWITCH command to switch the debugging session from the current task to any pending task. See HELP COMMANDS SWITCH for more information.

- Sometimes a task may have terminated, but its TCB continues to exist. When Z/XDC discovers such a task, it labels it as having "TERMINATED".

- The task that is the jobstep task is labeled as such.

- If the task is part of a USS process or has been 'dubbed' (a z/OS term) it is labelled as such, and the USS process ID is shown.

- If the task has a TRAP handler this is indicated (and the TRAP handler type will also be indicated).
  The following TRAP handler types will be used:
    - **OTH** A trap handler was found, but it is not a Z/XDC trap handler.
    - **TH** A trap handler was found, installed by this Z/XDC.
    - **TH+** A trap handler was found, installed by this Z/XDC, and if the instruction is TRAP4 or not a Z/XDC breakpoint, the Z/XDC trap handler

will chain to the previously-existing trap handler.
- **XTH** A trap handler was found, installed by some other Z/XDC.
- **XTH+** A trap handler was found, installed by some other Z/XDC, and if the instruction is TRAP4 or not an other Z/XDC breakpoint, the trap handler set by the other Z/XDC will chain to the previously-existing trap handler.
- **THEx** An error occurred during trap handler analysis.

- For all tasks that have ABENDed, the ABEND code is displayed within parentheses.

- When a task has opened a JOBLIB, STEPLIB, or task library DCB, the ddname for that DCB is shown within parentheses. (Note, when a task has such a job-, step-, or task library, please remember that the library is available and used, not only by the task that owns it, but also by **all subtasks** of that task.)

- For the address space being displayed, both the name of the address space and the address space's id number (ASID) are displayed.

- Those tasks that are participating in the current Z/XDC debugging session are displayed hilighted. Those that are not hilighted are not participating in the current debugging session. ABENDs that may occur within those tasks will not be processed by Z/XDC.

In addition, the **LIST TASKS** command assigns sequence numbers to the TCBs that it displays. These numbers are then referenced by various messages from Z/XDC.

Syntax:

```
LIST TASKS aspaceref EQUATESONLY
     TCBS  omitted   EQONLY
                     omitted
```

Rules:

- EQUATESONLY and EQONLY are aliases of each other.

- The aspaceref and EQUATESONLY operands are both optional.

- When both operands are given, they may be given in any order.

aspaceref
This identifies the address space for which task information is to be displayed. Briefly, this operand can be any of the following:
- A jobname.
- A keyword indicating the home space or the retry level or error level, primary, secondary, or instruction execution address space. Examples: HOME PASID ESASID, IASID, etc.
- An ASID number.
- An address expression pointing to an ASID number.
- A register containing an ASID number.
- The name of an equate or dsect that has been assigned to represent storage in an address space. (The assigned address space is used.)

For detailed information, see HELP COMMANDS SYNTAX ASIDS.

omitted
When an aspaceref operand is omitted, the task structure for the current target address space (either the home space or as established by the SET ASID command) will be displayed.

EQUATESONLY
EQONLY
As discussed below, the **LIST TASKS** command, in addition to producing a display, also (re)generates a series of automatic equates to label in storage the Task Control Blocks from which the display was built.

Sometimes, it is useful to generate the equates without also producing the display. This is what the **EQUATESONLY** operand does. It suppresses the commands output but allows the equates to be (re)generated. The equate can then, of course, be used as address expressions in subsequent commands, such as:

    LIST LSTACK
    LIST RBS
    LIST SCBS
    etc.

Automatic Equates

The **LIST TASKS** command (re)generates the following equates:
- **TCB#n** (for each TCB listed)
- **STCB#n** (the secondary TCB for each TCB listed)
- **OTCB#n** (the open MVS (USS) TCB extension for each TCB listed if the address of the relevant OTCB is not 0)

For more information, see HELP EQUATES BUILTIN AUTOMATIC.

Notice: All the above equates are **static!** They do not float. The locations that they target can change only when another **LIST TASKS** command is issued. In other words, the equates can become obsolete without warning simply due to the ongoing execution of the tasks whose TCBs are being displayed.

Examples:

LIST TASKS JES2
JES2’s task structure will be displayed.

LIST TASKS 1
The Master Scheduler address space’s task structure will be displayed.

LIST TASKS EIASID
The error level, instruction execution, address space’s task structure will be displayed.

LIST TASKS PASID
The retry level primary address space's task structure will be displayed.

LIST TASKS .ASCBASID
Assuming that the ASCB dsect map has been loaded and assigned to represent an Address Space Control Block of interest, the task structure for the address space that that ASCB describes will be displayed. (In this case, the command's operand is resolved as an address expression, and the contents of the two bytes to which that expression points is what is used.)

LIST TASKS CR3
The retry level secondary address space's task structure will be displayed.

LIST TASKS ECR3
The error level secondary address space's task structure will be displayed.

LIST TASKS R5
The two low order bytes of register R5 are interpreted as an ASID, so the task structure will be displayed for the address space having that ASID.

Help COmmands LISt TCbs

This command is an alias of the **LIST TASKS** command. See HELP COMMANDS LIST TASKS for more information.

Help COmmands LISt TErminal

The **LIST TERMINAL** command displays the following terminal related settings:
- Color
- Extended hilighting
- Field intensities
- Use of extended hilighting on color terminals
- Bell/nobell status
- Terminal screen size selection
- Image optimization frequency

Syntax:

LIST TERMINAL

This command accepts no operands.

All of the terminal characteristics displayed by this command also can be both displayed and altered by Z/XDC's Profile Menuing System.

For more information, see:

HELP COMMANDS SET BELL
HELP COMMANDS SET COLORS
HELP COMMANDS SET HILIGHT
HELP COMMANDS SET INTENSITY
HELP COMMANDS SET PRIMARYSIZE
HELP COMMANDS SET OPTIMIZATION
HELP COMMANDS SET SECONDARYSIZE
HELP COMMANDS PROFILE
HELP PROFILES MENU

Help COmmands LISt TFs

The **LIST TFS** command displays the options that pertain to the various communications interfaces:

- Whether or not the user wishes to use ISPF dialog services when they are available.
- Fullscreen TPUT interface status and options.
- ISPF services interface status and options.
- cdf/XDC interface status and options.

Syntax:

LIST TFS

This command accepts no operands.

For more information, see:

HELP COMMANDS LIST ISPF
HELP COMMANDS LIST TPUT
HELP COMMANDS LIST CDF
HELP COMMANDS SET ISPF
HELP COMMANDS SET TPUT

Help COmmands LISt TIMeout

The **LIST TIMEOUT** command displays the current value of Z/XDC's multi-tasking timeout period.

When multiple tasks are running in an address space, it often is possible for more than one such task to have ABENDed (or reached a breakpoint) and, therefore, be waiting for its turn to talk to the user at the terminal. However, since there is only one terminal, only one task at a time can have control of the debugging conversation. All additional ABENDed tasks are made to wait.

There are many situations that arise wherein the task that currently owns the debugging conversation may relinquish that ownership. Sometimes Z/XDC is aware of this (such as when the SPLIT, SWAP, SWITCH, TSO, and END commands are used), while at other times (i.e. when the GO command is used) it is not sure.

The GO command is an ambiguous case because the user either may or may not be expecting the resumed task to immediately hit a breakpoint and, therefore, return very quickly to Z/XDC. There just is no way Z/XDC can know this ahead of time. Accordingly, Z/XDC implements the GO command with a conditional assumption that the task will return control to Z/XDC very quickly, thus it does **not** awaken any other tasks that might be pending debugging. However, just in case this assumption is wrong, Z/XDC also starts a timer, and if (by the time the timer interval expires) the resumed task has not returned control to Z/XDC, then Z/XDC releases another pending task so that the user may begin to debug it.

Syntax:

LIST TIMEOUT

This command accepts no operands.

The TIMEOUT setting can be saved in your session profile. It can be changed by the SET TIMEOUT command. It also can be displayed and changed by Z/XDC's "Profile Menuing System". For more information, please see:

HELP COMMANDS SET TIMEOUT
HELP PROFILES MENU
HELP COMMANDS SWITCH

Help COmmands LISt TIOt

The **LIST TIOT** command can be used to display the ddnames for any task in any accessible address space. **LIST TIOT** generates its display by scanning the Task I/O Table for a specified task.

LIST TIOT Command vs. TIOTMAP Script

As an alternative to **LIST TIOT,** Z/XDC provides a script named **TIOTMAP** that also displays information from a Task I/O Table. The difference is this:

- The **LIST TIOT** command shows a table of information (ddnames and such) built from the DD entries of a specified TIOT.
- The **TIOTMAP** script, on the other hand, labels and displays the DD entries themselves.

LIST TIOT is a Sample User Commands Exit

LIST TIOT is **not** actually a Z/XDC command. Instead, it is implemented via a sample **User Commands Exit** that is distributed with Z/XDC in both source and executable form.

Proper installation is required for this command to be available. If Z/XDC's normal installation process is followed, then the **LIST TIOT** command generally will be available for debugging sessions that are started via the **XDCCALL** program (or its aliases).

It will also be available for debugging sessions started by options **1** or **2** of the **Z/XDC STARTUP PANEL** in ISPF. (This is because the panel uses **XDCCALL[A]** under the covers to start the session.)

LIST TIOT Command is NOT Available When ...

The **LIST TIOT** command will **not** be available when a debugging session is started:

- Via option **3** of the **Z/XDC STARTUP PANEL**
- Via a dynamic or static **Hook**

However, you can make **LIST TIOT** available if you take the following steps:

- Pre-LOAD the **XDCXITS** load module into storage.
- Make its address available to Z/XDC via a **DBCPARM** control block.
  - See **HELP EXITS USERCMDS** for more information.
  - Also, read the commentary found within the **#DBCPARM** macro located in the **hlq.XDCV31.XDCMACS** library.

The Source Code for LIST TIOT is Available

The source code for the **LIST TIOT** command is named **SRCXUCMD** and can be found in the **hlq.XDCV31.XDCASM** library.

The load module containing the **LIST TIOT** command is named **XDCXITS** and can be found in the **hlq.XDCV31.XDCLINK** library.

See **HELP SUPPORT HLQ** for information about Z/XDC Product Library names.

Syntax:

```
LIST TIOT tcbaddress
          omitted
```

tcbaddress
This must be an address expression that resolves to the address of any Task Control Block in any accessible address space. The TIOT that is chained to that TCB is then scanned and its information is displayed.

omitted
The tcbaddress operand can be omitted only when Z/XDC is **not** running in Foreign Address Space Mode, in which case the Home Address Space's "current" TCB is used as the default. For information about **Foreign Address Space Mode** see **HELP VIRTMEM XDCACCESS FASM.**

Examples:

For a cool example of using **autocloned equates** to label all of the DD entries in any TIOT, see the examples in **HELP COMMANDS EQUATE AUTOCLONING.**

Also check out the **TIOTMAP** script in **hlq.XDCV31.XDCCMDS.**

L TIO
If Z/XDC is running in Local Address Space Mode, then this displays the ddnames for the current task.

L TASKS
L TIO TCB#1
The **LIST TASKS** command creates a series of **TCB#n** equates labeling all of the TCBs in the current Target Address Space.

The **LIST TIOT TCB#1** command then scans and displays the address space's top task's (the Region Control Task's) ddnames.

SET ASID JES2
L TASKS
L TIOT TCB#3
The **SET ASID** command places Z/XDC into Foreign Address Space Mode targeting JES2's address space.

The **LIST TASK** command then creates a series of **TCB#n** equates labeling all of the TCBs in the JES2 address space.

The **LIST TIOT TCB#3** command then scans and displays all of the ddnames for JES2's main task.

Help COmmands LISt TPut

The **LIST TPUT** command displays about how your 3270 terminal's displays are painted.

Syntax:

LIST TPUT

This command accepts no operands.

The **TPUT** setting can be saved in your session profile. It can be changed by the **SET TPUT** and **SET ISPF** commands. It also can be displayed and changed by Z/XDC's **Profile Menuing System.** For more information, please see:

HELP COMMANDS SET TPUT
HELP COMMANDS SET ISPF
HELP PROFILES MENU

Help COmmands LISt TRace

The **LIST TRACE** command shows the current state of the following trace related settings.

SCROLL vs. **ROLL**
This shows whether, during tracing Z/XDC's displays will remain stable with the execution pointer rolling down the display or Z/XDC's displays will scroll so as to maintain the execution pointer at the top of the display.

ALL vs. **LOCAL**
This shows whether or not, during tracing, Z/XDC will attempt to follow execution into "alien" areas of storage.

ACTUAL vs. **SIMULATE**
This shows whether, during tracing, Z/XDC simulates the execution of some user program instructions without leaving the ESTAE environment, or Z/XDC returns via RTM to the user program to let it actually run its own instructions one by one. (Note, "SIMULATE" support is only planned. It has not yet been implemented.)

ZERO vs. **TRAP2**
This shows whether breakpoints are implemented using X'00' opcodes (ZERO) or using TRAP2 machine instructions (TRAP2).

STOP vs. **IGNORE**
When the #DIE macro creates a conditional DEAD trap, it usually will create that trap inline and preceded by a conditional branch to be taken should the tested-for condition not be met. When stepping through code using the TRACE BY command, it is both rather tedious and quite uninformative for the trace to be stopping at these conditional DEAD traps when they are about to be bypassed. So this setting has been implemented to control whether (STOP) or not (IGNORE) BY-type traces are to stop at conditional DEAD traps that are about to be skipped around by execution. For more information, see:

HELP BREAKPOINTS TRACING
HELP BREAKPOINTS DEADTRAPS
HELP COMMANDS TRACE

Syntax:

LIST TRACE

This command does not accept operands.

Trace related settings can be set via the SET TRACE command.

The TRACE settings can be saved in your session profile, and so they can be both displayed and set via the Profile Menuing System. See HELP PROFILES MENU for more information.

Help COmmands LISt USErid

The **LIST USERID** command reports the TSO userid to be notified if and when a background program has abended and is waiting for a user to signon to the pending debugging session. See **HELP XDCSRVER CDF** for information about connecting to a debugging session pending in a batch job or started task.

Syntax:

LIST USERID
LOGONID [alias]

This command does not accept operands.

The **USERID** setting can be saved in your session profile.

It can be changed by the **SET USERID** command.

It also can be displayed and changed by Z/XDC's **Profile Menuing System.**

In order for the userid setting to be effective, it must preexist in your default session profile. For details, see the following:

- A **PROFILE SAVE** command has to be issued to save this setting into the user's personal profile, and...
- A **//ISPPROF** (or **//XDCPROF)** DD card pointing to the profile library must be included in the JCL for the job to be debugged via cdf/XDC.

Help COmmands LISt USS

The **LIST USS** command displays information about:
- The current USS environment
- The current setting of the **USS** value

depending on options specified on the command.

Syntax:

```
    LIST USS [ option ] ...
```

option can be:

```
  ENVIRONMENT    - display information about the USS environment.

  PROFILE        - display information about the current USS setting. Note that the
                   USS setting can be saved in your session profile. It can be
                   altered using the SET USS command or the Profile Menuing System

  ALL            - display all the above.

  (if no operands are specified, ALL is assumed).
```

For more information on altering the **USS** setting, see:

HELP COMMANDS SET USS
HELP PROFILES MENU

Help COmmands LISt VARIables

This command is available only for customers who have licensed the features necessary for debugging one or more of the supported **High Level Languages** (such as the **c/XDC feature** for debugging XL C/C++ and Metal C). For more information, see HELP SUPPORT FEATURESANDCAPS.

This command allows you to display the current values for one or more variables, arrays, structures, unions, whatever, belonging to any High Level Language program that you have mapped. (See HELP COMMANDS MAP.)

This command applies only to variables used in HLL programs. Variables defined in Assembler are referenced via classical Z/XDC syntax rules as described in topics starting at HELP ADDRESSING PARSERS ASM RESOLUTION.

Syntax:

```
    LIST VARIABLES variable variable ...
         VARS      ALL
                   omitted
```

Rules:

- **ALL** may not be combined with any other operands.

- If **ALL** is given along with other **variables,** then **ALL** is treated as being a variable, not a keyword.

variables
Any number of variables may be given. One (or more, if appropriate) lines of display will be produced for each variable.

- When referring to a variable, use whatever syntax is appropriate for the HLL language in which the variable is defined.

- Generally, the variables can be anything supported by the HLL language. Examples:
  - Simple scalar variables
  - Array names
  - Selected array elements
  - Structure names
  - Structure elements
  - Structures within arrays within structures within ... whatever
  - Unions
  - Whatever

- The variable's value will be displayed, formatted according to the variable's type.

- Variables and structures whose names are reserved as references to registers or a PSW by Z/XDC can be specified through use of one of the variable pool built-in equates. For Example: **L V VS#3_SV.R0**

ALL
When **ALL** is given as the sole operand, then all variables are listed from all variable pools in all **known** stack frames.

When **ALL** is given in combination with other **variables,** then it is treated as being a variable, not a keyword.

omitted
When all operands are omitted, then all variables are listed from all variable pools in just the **current** stack frame.

Shortcuts

The following shortcuts can be used with the displays produced by this command:

D - Displays the variable's raw storage in a hex-text format.
F - Displays the variable's raw storage, usually in a hex-text format.
Z - Allows you to zap the variable's value.
***** - Displays text data as EBCDIC.
| - Displays text data as ASCII.
**** - Displays text data as a null-terminated string.
/ - Displays the full array in which text data is contained.

Help COmmands LISt VARS

This command is an alias of the **LIST VARIABLES** command. See HELP COMMANDS LIST VARIABLES for more information.

Help COmmands LISt VDisplay

The LIST VDISPLAY command displays the current settings of the SET VDISPLAY command.
These controls affect the output of the **LIST VARIABLES** command.

Syntax:

LIST VDISPLAY

This command accepts no operands.

See **HELP COMMANDS SET VDISPLAY** for an explanation of each item displayed.

Help COmmands LISt VEctorregisters

In IBM's current incarnation of vector registers, they have the following characteristics:

- They were introduced with IBM's **Z13** processor.
- They are 16 bytes wide.
- There are 32 of them.

- All of the 16 floating point registers are mapped to the hi-order halves (8 bytes) of the first 16 vector registers. Thus any change to one also changes the other.

- The vector registers are operated upon by the 143 (or so) **SIMD** machine instructions, also introduced with the Z13 processor.

- Depending upon the instruction used, the data within a vector register is treated as being a collection of one or more distinct values (called "elements").

- For any given machine instruction, all elements are the same size. Their lengths can be only a power of two (i.e. 1 byte, 2-bytes 4-bytes, etc.).

When vector register support is **not** installed, Z/XDC disallows use of commands and symbols related to vector registers (LIST VREGS, LIST VRn, etc). However, a **PRETEND** operand can be used if you want see what these command outputs look like anyway. The **LIST FEATURES** command can be used to display whether or not vector register support is installed on the current system.

When vector register support is installed, the registers themselves will remain undefined for any task or SRB routine until they are first referenced by code running under said task or SRB. Accordingly, attempts to display the vector

registers will result in the data being dashed out. (Try it, you will see what I mean.)

Unless you use Z/XDC's **ZAP** command to intentionally set data into a vector register, Z/XDC will never do anything to activate vector register support in an execution thread in which vector register support has not yet been activated.

Unlike other registers, the Operating System does not provide any Request Block level or Linkage Stack level save areas for vector registers. It only provides a task level save area. Consequently, there is no error level vs. retry level distinction to be made for vector registers. Whatever values they contained at the time of error, those will be the values they will contain at retry time. (For more information about the error level and retry level environments, see HELP EXECUTIONLEVELS.)

Consequently, unlike other registers, Z/XDC does not maintain two sets of vector registers, only one. Collectively, they are named **VREGS.** Individually, they are named **VRn.**

Z/XDC does not limit you to displaying just **your** program's vector registers. Every task and every SRB has its own set of vector registers, so (security and authorization permitting) Z/XDC allows you to display the vector registers associated with **any** task (TCB) or **any** SRB located in **any** address space in the entire system!

Subtopics Index

Z/XDC allows you either to display individual vector registers or to display an entire register set. For specific information, type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

INDIVIDUAL - Displaying individual vector registers.
REGISTERSET - Displaying an entire vector register set.

Help COmmands LISt VEctorregisters Individual

Z/XDC allows you to display an individual vector register:
- Belonging to the current program.
- Belonging to any program running under any TCB located in any accessible address space.
- Belonging to any suspended SRB routine located in any accessible address space.

Please note that for control registers, floating-point registers and vector registers, z/OS does not maintain a distinction between the error level and retry level environments.

If programs (running under the task or SRB for which a vector register is being displayed) have not yet used any vector registers, then z/OS has not yet enabled the vector registers for that task or SRB, and so the display will show that register's data dashed out. For more information, see HELP COMMANDS LIST VECTORREGISTERS.

Vector registers are displayed showing all permitted interpretations of the register's contents both with respect to data type and with respect to element size. The data types include:

- Hex
- Text (ASCII or EBCDIC, according to the current SET FORMAT setting)
- Signed integer (shown as multiple 2-byte, 4-byte and 8-byte wide elements)
- Binary floating point (BFP), shown as two 8-byte wide elements.

If vector register support does exist but has not yet been activated in the current execution thread (task or SRB), then the LIST VRn display will show the register contents zeroed out (not dashed out).

Syntax:

LIST VRn **tcbaddress** **PRETEND**
nVR **rbaddress**
ssrbaddress
addressomitted

LIST VRn [address]
nVR

The register's data is displayed in all useful formats. (described above).

LIST vregname **tcbaddress**
ssrbaddress

The specified vector register is displayed that is owned either by the specified task or by the specified SRB routine.

Note, if no code running under the specified task has yet referenced any vector register, then the display will be dashed out. (See HELP COMMANDS LIST VECTORREGISTERS for more information.)

Note, the **TCB#n** equates created by the **LIST TASKS** command can, of course, be used as the **tcbaddress** operand

Also, the **SSRB#n** and **SSRX#n** equates created by the **LIST SSRB** command can be used as the **ssrbaddress** operand.

LIST vregname **rbaddress**

Request Block support for this command is provided only for convenience. Since z/OS saves vector registers only at the task level, not the RB level, Z/XDC treats a given RB address as if it were the address of the TCB to which the RB is queued.

LIST vregname **addressomitted**

When an address operand is omitted, the current program's vector register will be displayed. The value shown will be what was the register's contents at the

time of error and what will be its contents at program resumption time. (z/OS does not maintain a distinction between error level and retry level vector registers.)

LIST vregname [...] **PRETEND**

Normally, when the LIST vregname command is issued while running on a computer that does not support vector registers, the command fails (with message DBC123E). However, if you want to see what the display looks like anyway, you can use this **PRETEND** operand, and Z/XDC pretends that the registers exist after all.

You also can use the PRETEND operand on the **ZAP** command to make changes to the vector registers. But of course, whenever Z/XDC releases control back to your program (i.e. you issue a GO or TRACE command), all zapped changes will be lost.

The **Z** shortcut commands does not need a PRETEND operand.

Examples:

LIST VR0
Lists and formats the contents of the current program's vector register 0 in every possible format.

If vector register support has not yet been activated in the correct thread (task or SRB), zeros will be displayed. in the current thread (task or SRB), (Activation will not occur.)

LIST SSRBS 1
LIST VR1 SSRB#1 [or **SSRX#1]**
If Z/XDC is running authorized, and if System Security permits the access, then the **LIST SSRBS 1** command displays a list of all SSRBs suspended in the Master Scheduler's address space. It also creates a series of **SSRB#n** and **SSRX#n** equates labeling the locations of all the SSRBs and SSRXs that were found.

The **LIST 1VR SSRB#1** command (or a **LIST VR1 SSRX#1** command) then lists and formats the contents of the SRB routine's vector register 1. Note, if the SRB routine has not used or referenced any vector registers, the registers will not have been created, and so their displayed data will be zeroed out.

Help COmmands LISt VEctorregisters Registerset

The **LIST VREGS** command allows you to display the entire set of vector registers:
- Belonging to the current program.
- Belonging to any task located in any accessible address space.
- Belonging to any suspended SRB routine located in any accessible address space.

If programs running under the task or SRB for which the vector registers are being displayed have not yet referenced any vector registers, then z/OS has not yet

enabled the vector registers for that task or SRB, and so the display will show those registers data dashed out. For more information, see HELP COMMANDS LIST VECTORREGISTERS.

The **LIST VREGS** command displays the vector registers **only** in a raw hex-EBCDIC (or hex-ASCII) format. If you want to see element by element numeric interpretations, then display them individually. See HELP COMMANDS LIST VECTORREGISTERS INDIVIDUAL for more information.

Syntax:

```
LIST VREGS  WIDE    EBCDIC  tcbaddress      PRETEND
            NARROW  ASCII   rbaddress
                            ssrbaddress
                            addressomitted
```

LIST VREGS **WIDE**
NARROW

These are optional. They control whether the resulting display will show 8 words of data per line **(WIDE)** or just 4 words **(NARROW).** If omitted, then the current **SET FORMAT** command setting is used. This setting can be displayed by the **LIST FORMAT** command and saved into your session profile by the **PROFILE SAVE** command. For more information, see:

HELP COMMANDS LIST FORMAT
HELP COMMANDS SET FORMAT
HELP COMMANDS PROFILE

Wide displays are suitable if your terminal display is set to 136 columns or wider. Narrow displays are suitable when only 80 columns are displayed. See HELP FULLSCREEN TERMINALS for more information.

LIST VREGS **EBCDIC**
ASCII

These operands also are optional. They control, for the character portion of the display, whether the register contents are interpreted as EBCDIC characters or ASCII. If omitted, then the current **SET FORMAT** command setting is used.

LIST VREGS **tcbaddress**
ssrbaddress

The vector register set is displayed that is owned either by the specified task or by the specified SRB routine.

Note, if no code running under the specified task or SRB has yet referenced any vector register, then the vector registers will not yet have been enabled by z/OS, and so their displays will be dashed out. (See HELP COMMANDS LIST VECTORREGISTERS for more information.)

Note, the **TCB#n** equates created by the **LIST TASKS** command can, of course, be used as the **tcbaddress** operand

Also, the **SSRB#n** and **SSRX#n** equates created by the **LIST SSRB** command can be used as the **ssrbaddress** operand.

LIST VREGS **rbaddress**
Request Block support for this command is provided only for convenience. Since z/OS saves vector registers only at the task level, not the RB level, Z/XDC treats a given RB address as if it were the address of the TCB to which the RB is queued.

LIST VREGS **addressomitted**
When an address operand is omitted, the current program's vector registers will be displayed. The values shown will be what were the registers contents at the time of error and what will be their contents at program resumption time. (z/OS does not maintain a distinction between error level and retry level vector registers.)

LIST VREGS [...] **PRETEND**
Normally, when the LIST VREGS command is issued while running on a computer that does not support vector registers, the command fails (with message DBC123E). However, if you want to see what the display looks like anyway, you can use this **PRETEND** operand, and Z/XDC pretends that the registers exist after all.

You also can use the PRETEND operand on the **ZAP** command to make changes to the vector registers. But of course, whenever Z/XDC releases control back to your program (i.e. you issue a GO or TRACE command), all zapped changes will be lost.

The **Z** shortcut commands does not need a PRETEND operand.

Examples:

LIST VREGS
Lists the contents of the entire set of vector registers owned by the current task or SRB routine. The display is either wide or narrow according to the current setting of the **SET FORMAT** command.

LIST TASKS JES2
LIST VREGS TCB#3 WIDE
If Z/XDC is running authorized, and if System Security permits the access, then the **LIST TASKS JES2** command displays the subtask structure for JES2's address space. It also creates a series of **TCB#n** equates labeling the locations of all TCBs within the JES2 address space.

Then the **LIST VREGS TCB#3 WIDE** command displays the contents of the entire set of vector registers that belong to the program (HASJES20) running under the control of the jobstep task (TCB#3). The display is wide, ie. is suitable for a wide geometry terminal with display lines that are at least 136 columns wide. (See HELP FULLSCREEN

TERMINALS GEOMETRIES for more information.)

LIST SSRBS 1
LIST VREGS SSRB#1 [or **SSRX#1]**

If Z/XDC is running authorized, and if System Security permits the access, then the **LIST SSRBS 1** command displays a list of all SSRBs suspended in the Master Scheduler's address space. It also creates a series of **SSRB#n** and **SSRX#n** equates labeling the locations of all the SSRBs and SSRXs that were found.

Then the **LIST VREGS SSRB#1** command (or **LIST VREGS SSRX#1** command) displays all vector registers belonging to the SRB routine running under the control of that SSRB. Note, if the SRB routine has not used or referenced any vector registers, the registers will not have been created, and so their displayed data will be dashed out.

Help COmmands LISt VR#

The **LIST VRn** command displays the entire 16 bytes of an individual vector register in various formats.

The display shows all permitted interpretations of the register's contents both with respect to data type and with respect to element size. The data types include:

- Hex
- Text (ASCII or EBCDIC, according to the current SET FORMAT setting)
- Signed integer (shown as multiple 2-byte, 4-byte and 8-byte wide elements)
- Binary floating point (BFP), shown as two 8-byte wide elements.

If vector register support does not exist, normally the LIST VRn command will fail. However, if you'd like to see what the display looks like anyway, you can add the **PRETEND** operand to the command, and Z/XDC will pretend that vector registers exist after all.

If vector register support does exist but has not yet been activated in the current execution thread (task or SRB), then the LIST VRn display will show the register contents zeroed out (not dashed out).

For more information, see HELP COMMANDS LIST VECTORREGISTERS.

Help COmmands LISt VREgs

If vector register support exists on the current processor, the **LIST VREGS** command displays the entire set of 32 vector registers. All 16 bytes of each register are shown in a hex-text format. The LIST FEATURES command can be used to see whether or not vector register support is installed.

If vector register support does not exist, normally the LIST REGS command will fail. However, if you'd like to see what the display looks like anyway, you can add the **PRETEND** operand to the command, and Z/XDC will pretend that vector registers exist after all.

If vector register support does exist but has not yet been activated in the current execution thread (task or SRB), then the LIST VREGS display will show the register contents dashed out.

For more information, see HELP COMMANDS LIST VECTORREGISTERS.

Help COmmands LISt VSEttings

This command displays various settings and limitations (array dimension limits, name length limits and such) pertaining to High Level Language variables.

Variable related controls:

- Are established by the **SET VSETTINGS** command,
- Are displayed by the **LIST VSETTINGS** command (described below),

- And saved in your session profile by the **PROFILE SAVE** command.

They also can be displayed, set and saved by the **Profile Menuing** system.

Syntax:

LIST VSETTINGS

This command accepts no operands.

For detailed information about the VSETTINGS settings, see HELP COMMANDS SET VSETTINGS.

Help COmmands LISt VSTack

This command is available only for customers who have licensed the features necessary for debugging one or more of the supported **High Level Languages** (such as c/XDC for debugging XL C/C++ and Metal C). For more information, see HELP SUPPORT FEATURESANDCAPS.

An LE compliant subroutine generally has access to up to three variable pools:

- A **Global Pool** of constants
- A **Subroutine Pool** of variables declared for the current routine, subroutine or function
- A **Local Pool** of variables declared only for the current block of code

As one subroutine calls another, new Subroutine Pools and Local Pools are created, and the variables declared in the calling routine are generally unavailable to the called routine. The management of these variable pools is accomplished through structures known as **Language Environment Stack Frames.**

The **LIST VSTACK** command displays information about these Stack Frames as follows:

- The stack frames are displayed from oldest to newest.

- For each frame:
  - The resume address is displayed for the routine that owns the frame.
  - Information is displayed about each variable pool that is owned by the frame.
  - Shortcut Entry Fields are provided that can be used to display:
    - Resume address locations
    - Or variable pool storage
  - For each pool, an automatic equate is generated conveying the following information:
    - The sequence number of the stack frame
    - The type of variable pool (global, subroutine or local)
    - The location of the variable pool
    - The length of the variable pool

Syntax:

LIST VSTACK

This command accepts no operands.

Help COmmands LISt WIndow

The **LIST WINDOW** command displays the following settings for the terminal display window from which it is issued:
- That window's starting line number.
- That window's length.
- That window's default scroll amount.
- The maximum number of prior commands that can be retrieved for re-execution by the **RETRIEVE** command.

Syntax:

LIST WINDOW
SCREEN

This command accepts no operands.

The various WINDOW settings can be saved in your session profile. They can be changed by the SET WINDOW command. They also can be displayed and partially changed by Z/XDC's "Profile Menuing System". For more information, please see:

HELP COMMANDS SET WINDOW
HELP FULLSCREEN WINDOWS
HELP PROFILES MENU

Help COmmands LISt XDc

The **LIST XDC** command can be used to display information about Z/XDC's current status, about the Licensed Features available to the current debugging session, and about contacting ColeSoft's Technical Support.

The status information displayed includes Z/XDC's:

- Clone Name
- Subsystem Control Table (SSCT) name
- Release
- Assembly Date
- **Maintenance Level**
- Service SVC number
- legacy HOOK SVC number
- The state of Quick Locate Support

The licensing information includes:

- Which **Features** have and have not been licenses in the current SYSPLEX.
- Which of those **Licensed Features** have been granted for use by the current debugging session.
- What the **CAP Quotas** are for each Licensed Feature.
- What the license **expiration date** is.

A reminder also is displayed that the most **current maintenance** is always available from our website.

Syntax:

```
LIST XDC
     MAINTENANCE
```

This command does not accept operands.

Help COmmands LISt XMs

The **LIST XMS** command displays the cross memory execution environment for any program running under any Request Block in the System.

When the host program's environment is displayed, **both** the retry level environment and the error level environments are shown. (See HELP EXECUTIONLEVELS.)

For each environment, the display shows:

- The primary address space
- The secondary address space
- The home address space
- The address space control mode (ASC-MODE)
- The execution space
- The PSW key-mask (PKM)
- The authorization index (AX)
- The extended authorization index (EAX)

Syntax:

```
LIST XMS rbaddress
         omitted
```

Operands:

rbaddress
This operand must resolve to the address of a Request Block located in any accessible address space. (Note, the **LIST RBS** command and the **RB#n** equates that it generates, are good ways to find request blocks anywhere.) This causes the cross memory environment associated with the program running under the control of that request block to be displayed.

omitted
When the **rbaddress** operand is omitted, the cross memory characteristics of **both** the error level and retry level environments are displayed.

Help COmmands LISt Zap

The **LIST ZAP** command displays the current state of the zap related settings made by the following commands:

SET ZAP NORMAL
SET ZAP SPROT
This setting controls whether or not Z/XDC allows the ZAP command to store into store protected storage. **SET ZAP NORMAL** does not allow the ZAP command to store into store protected storage. **SET ZAP SPROT** does allow the ZAP command to store into store protected storage. For more information about the ZAP command, see HELP COMMANDS ZAP.

SET ILC
SET NOILC
This setting displays whether or not Z/XDC does a length match check when the user zaps a machine instruction by overtyping a displayed mnemonic. ("ILC" stands for "Instruction Length Check".) When **SET ILC** is in effect, the new machine instruction’s length must match the old instruction’s length. For more information about the SET ILC command see HELP COMMANDS SET ILC.

Syntax:

```
LIST ZAP
```

This command does not accept operands.

Example:

```
LIST ZAP
ALLOW ZAPPING STORE PROTECTED STORAGE?                  NO
REQUIRE INSTRUCTION LENGTH MATCH FOR MNEMONIC ZAPS?    YES
```

You can save ZAP settings in your session profile. For information on displaying and setting your session profile using the the Profile Menuing System, see HELP PROFILES MENU.

Help COmmands LL

This command is an alias of the **LIBRARYLISTS** command. See HELP COMMANDS LIBRARYLISTS for more information.

Help COmmands LOAd

This command cannot be used in Foreign Address Space Mode.

This command can only be used when debugging programs on the **host** system
- for example it cannot be used under dump/XDC.

This command can be used to load programs in 2 ways:
1: - **traditional** load modules or program objects in the **standard** z/OS search order or a PDS(E).
2: - program objects in a USS file.

In the descriptions below, the term **load module** is used to make the text easier to read. Use of the term **program object** means that just a **program object** is relevant.

Syntax:

```
    LOAD modulename    loadlibraryname    USS=option    DUB=option
         'modulename'  omitted            omitted       omitted
         "modulename"
```

modulename
'modulename'
"modulename"
This gives the name of the load module or path to be loaded.

The value may be quoted.

The syntax of **modulename** depends on the presence and/or value of other operands:

- If the **loadlibraryname** operand is specified or **USS=NO** is in effect then the value (regardless of quotes) must be a 1-8 character PDS(E) member name.

loadlibraryname
The name of a load libary (or program object library) the contains the desired load module.

Specification of this operand means that that **modulename** is **always** treated as a **1-8 character PDS(E) member name** and the **LOAD** macro will be used.

When given, the library dsname must be a fully qualified, UNquoted dataset name. It may contain variable symbols which Z/XDC will make substitutions for. This allows for the dsname to be reactive to the current time, date, userid, jobname, etc. These variables are most useful when the **LOAD** command is executed from a command script or a DEAD trap.

For more information, see:

HELP COMMANDS SYNTAX DSNAMES
HELP COMMANDS READ
HELP BREAKPOINTS DEADTRAPS

The library must be cataloged. It may be either a PDS or a PDSE. It is searched instead of the standard libraries.

USS=option
This operand is optional, and can be specified anywhere on the command. It is always processed before any other operands.

If this operand is omitted, the **LOAD** command behaves as if **USS=PROFILE** was specified.

More information about this operand is in the **HELP DEBUGGING USS OPERANDS USS=** help topic.

DUB=option
This operand is optional, and can be specified anywhere on the command. It is always processed before any other operands.

If this operand is omitted, the **LOAD** command behaves as if **DUB=PROFILE** was specified.

More information about this operand is in the **HELP DEBUGGING USS OPERANDS DUB=** help topic.

Notes

Loading a non-USS program
If the **LOAD** command determines that a non-USS program is to be loaded, the **LOAD** assembler macro is used.

In this case, the value for **modulename** must be a 1-8 character name that meets the following:
- containing quotes are ignored
- The value is uppercased
- the first character must be alphabetic (A-Z) or national (@, #, $) or the left bracket ({)
- the remaining characters must be alphabetic (A-Z) or national (@, #, $) or numeric (0-9)

Loading a USS program
If the **LOAD** command determines that a USS program is to be loaded, the **loadhfs** USS system call is used.

In this case, the value for **modulename** must be a 1-63 character name that meets the following:
- containing quotes are ignored

Before loading, the supplied **modulename** is examined. if it does not start with a '/' (forward slash or solidus) the Current Working Directory (CWD) is prefixed to it.

The **modulename** (and maybe the CWD prefix) is converted into an absolute path name and that value is used as the value on the **loadhfs** USS system call.

Modules that have the **Sticky Bit** set will cause a load error.

Note that RMODE64 programs are handled correctly.

Note too that if the program had previously been loaded into storage using a relative path name (one that did not start with a '/') z/OS may increment that program's use count rather than loading the program specified on the **LOAD** command. Z/XDC detects this and will write a warning message if this occurs.

Examples:

LO MYPROG,*U.MYLIB.LOAD
LO MYPROG,MYUID.MYLIB.LOAD

Both of these commands load MYPROG from MYUID.MYLIB.LOAD. (Z/XDC substitutes the current TSO userid string for "*U". See **HELP COMMANDS SYNTAX DSNAMES** for more information.)

LO IEBGENER
This loads IEBGENER from SYS1.LINKLIB.

LO IGC0001I
This returns a pointer to the Pageable Link Pack Area (PLPA) copy of IGC0001I.

LO xyzzy USS=YES
This command loads a USS program. The Current Working Directory (CWD) will be prefixed to the name.

Help COmmands LOCate

The **LOCATE** command causes a window's messages to be scrolled either up or down so as to bring either a designated message (nnn) or a designated command string (#nnn) to the top of the window.

Syntax:

LOCATE nnn

#nnn
#0
#

Note:

- Exactly one operand is required, no more, no less.

Operands

LOCATE nnn
This scrolls the current window to the message whose assigned sequence number matches **nnn.** Notes:

- This form of the **LOCATE** command can be used on any window or display that is scrollable (i.e. in which the **UP** and **DOWN** commands can be used).

- A message's assigned sequence number is generally not displayed, but:
  - It does roughly correspond to the message's position in the window's scroll area.

  - For the Primary Window, the sequence# of the top displayed message will be shown in the screen's title line at the far right.

  - It will also be displayed in the screen's title line for HELP topic displays.

- If **nnn** matches the sequence number of a message that is not currently visible then the scroll will be positioned to the next following message that is visible.

- The message reached by **nnn** can be either a normal message or the echoing of a **nn - commandstring** message. Note that the command string's displayed sequence number has no relationship with or connection to its message number.

- If the given **nnn** is less than the message sequence number of the scroll's first message, then:
  - The scroll will be positioned to that first message regardless of its visibility.
  - The display will start with the first following visible message.

- If the given **nnn** is greater than the message sequence number of the scroll's last message, then:
  - The scroll will be positioned to that last message regardless of its visibility.
  - The display may be blank if that last message is not visible.

LOCATE #nnn
LOCATE #0
LOCATE #
This scrolls the Primary Window to the command string echoing message displaying the matching command sequence number. Notes:

- This form of the **LOCATE** command can be used **only** in the **Primary Window.** It will be rejected if issued in any other window.

- **#0** is permitted. It positions the scroll to the top of Z/XDC's opening messages located prior to the first issued commands.

- **#** positions the scroll to the last command string issued so far.

This **#nnn** form operand can also be used on the **UP** and **DOWN** commands to identical effect. See **HELP COMMANDS DOWN** for more information.

Examples:

LOC 500
This command will position the scroll to the message whose sequence# is 500:

- If there are not yet 500 messages in the scroll area, then positioning will be to the last message.

- If that last message is invisible, then the window will appear empty.

- If the 500th message is invisible, then:
  - The next following visible message will be displayed.
  - For the Primary Window, that first displayed message's sequence number will be shown in the screen title line, and that number will be some value greater that 500.

- For the Primary Window, if the scroll area overflows and has to be trimmed, then it is possible that the 500th no longer exists! In this case: the scroll will be positioned to the first message that does exist, and the screen title message will (after pressing ENTER once) show how many messages have been trimmed.

LOC #10
UP #10
DOWN #10
These commands all do exactly the same thing. They all reposition the Primary Window's Scroll Area so as to bring the tenth issued command string (and its output messages) into view in the Window's display area.

Help COmmands LOG

During a debugging session, all activity (commands and their resulting displays) that are issued from the **Primary Window's** command line are saved in that Window's **scroll area.** You can then use **UP** and **DOWN** commands to review the work that you have done so far in your debugging session. (See **HELP FULLSCREEN SCROLLING** for more information.)

In addition, this scrollable history is, by default, automatically written as a SYSOUT file to JES2 spool. (You can use the **SET LOG** command to manage this, if you like.)

Only activity in the Primary Window can be logged. Activity in the Watch Window(s) cannot be logged.

The logging of the command history can be either automatic ("auto-logging") or manual. When logging is manual, the command history is not logged unless and until you issue the **LOG** command.

Normally though, logging is automatic: Pending messages are automatically written out to the log file every time the terminal keyboard is unlocked for additional input. The **LOG** command can be used to cause pending messages to be written immediately instead of waiting for keyboard unlock. This is sometimes useful in certain automatic execution tracing situations. See **HELP BREAKPOINTS AUTOCMDS** for an example.

Syntax:

LOG

This command accepts no operands.

When auto logging is turned on, the **LOG** command causes pending messages to be written to the log file immediately.

When auto logging is turned off, the **LOG** command does the following:
- Allocates the log file.
- Opens it.
- Writes to it all Primary Window activity that is currently recorded in the scrollable area and that has not previously been written to the log.
- Closes the log file.
- Deallocates it.

This management process (allocate, open, write, close, deallocate) has the following consequences:

- If the log is assigned to a spool file, then each time a **LOG** command is issued, a new spool file is created.

- If the log is assigned to a disk dataset, then each **LOG** command causes pending entries to be appended to the end of the dataset.

There are some environments within which you may need to debug in which it is not possible to write a log to JES2/JES3 spools. These include:
- When debugging JES2/JES3 itself.
- When debugging within any address space that is running under the control of the "MSTR" subsystem. These include:
  - Many address spaces that are started at IPL time prior to the startup of JES2 or JES3.
  - Any system task that is started with the SUB=MSTR keyword specified on the System Operator's START command by which the System Task is started.
- When debugging in any address space that is started under the ownership of any other subsystem that is not a normal JES (Job Entry Subsystem).

When Z/XDC receives control in such environments, and it attempts to create spool files for its logs:
- The attempts will fail.
- The System will issue several error messages, including:
  - $HASP708 with RC=13. (See "OS/390 JES2 Messages", GC28-1796.)
  - IEC141I with code 013/C0. (See "OS/390 MVS System Messages Volume 4 (IEC - IFD)", GC28-1787.)
- Z/XDC will proceed with the debugging session without use of its ability to log to spool.

In these environments, you either can proceed without session logging, or you can use the **SET LOG** command to allocate the log to a sequential file on disk. You might then use the **PROFILE SAVE** command to make these settings your personal default for future debugging sessions.

For more information, please see:
HELP FULLSCREEN LOGGING
HELP COMMANDS SET LOG
HELP COMMANDS LIST LOG
HELP PROFILES

Help COmmands Map

The **MAP** command's primary purpose is to build **Symbol Maps** and **Source Image Maps** for programs written in Assembler and various C and C++ languages. It also, when necessary, will build **Binder Maps** (also known as linkedit maps and module maps) for load modules and program objects in order to figure out where, within those modules, the target program's csects are located.

The supported C and C++ languages are:
- XL C/C++ from IBM

- Metal C from IBM
- Systems/C from Dignus

(Use the **DMAP** command to build:
- **Dsect maps** of control blocks and data areas,
- **Dsect versions** of csect maps and Binder maps that can be assigned to code whose locations are not documented by system control blocks (CDEs and such).

The **MAP** command is sensitive to which Z/XDC **Features** have been licensed for use by your company. Briefly:

- Support for **Binder maps** is a part of **base/XDC.** The MAP command is always able to build Binder maps.

- Support for **Assembler Maps** and **Dsect Maps** is a part of the **asm/XDC Feature.** This support is available only when the asm/XDC Feature has been licensed.

- Support for **C Source Maps** is a part of the **c/XDC Feature.** This support is available only when the c/XDC Feature has been licensed. For more information, see **HELP DEBUGGING C.**

- Support for mapping objects in storage dumps is part of the **dump/XDC Feature.** This support is available only when the dump/XDC Feature has been licensed. For more information, see **HELP DUMPS.**

Related Information

The following related information is available:
- For several discussion topics pertaining to various aspects of mapping, see **HELP MAPS.**
- For comprehensive information about loading maps for modules and control blocks contained within **dumps,** see **HELP DUMPS MAPPINGDATA.**
- For information about Licensed Features, see **HELP SUPPORT FEATURESANDCAPS.**
- To see what Features are licensed, issue the **LIST XDC** command.

Subtopics Index

For additional information, type an **H** at the left below to select topics directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

SYNTAX - Comprehensive Syntax for the MAP command (and notes about how the MAP command behaves if any MMEFDSNs are defined)

BINDERMAPS - About building Binder maps (aka Binder Maps or Linkedit Maps)
PRIVATELYLOADED - About mapping Privately Loaded modules
ASSEMBLERMAPS - About building maps of Assembler code from ADATA and SYM data
CMAPS - About building maps of XL C/C++ and Metal C code from DWARF data
DWARFFILES - About DWARF data files
CMDDIALOGS - About constructing and issuing MAP commands via Command Dialogs
FINDINGMAPDATA - How the MAP command decides what maps to build and how it finds the data to build them

EXAMPLES - MAP command usage examples

Help COmmands Map Syntax

The following is a combined and comprehensive description of all operands of the **MAP** command. Subsets of this description have been copied into various of the other subtopics of HELP COMMANDS MAP. Hopefully there are no conflicts.

That said, while the **START=** operand is described herein, due to the complexities introduced by that operand, a better, more focused discussion is provided in HELP COMMANDS MAP PRIVATELYLOADED.

If any MMEFDSN tokens are defined (using the **DEFINE MMEFDSN** command), then the MAP command works differently. as follows:

1: If no specific datasets have been specified on the MAP command, the MMEFDSN tokens are searched in order based on the location of the module being mapped in storage (private area, PLPA, nucleus) and the resulting definition is used.

2: If a specific dataset name has been specified on the MAP command, that dataset name is searched for in the MMEFDSN tokens (i.e. the datasets that were originally processed by module mapping extraction utility (XDCMMEF), not the MMEFDSN names themselves), and, if found, that dataset's information will be used. This can be overridden by the use of the **USELDS** operand. Z/XDC assumes that you know what you are doing when you use this operand.

Modules loaded from a USS (Unix System Services) file system may be mapped. If the module name contains characters that have a meaning to Z/XDC (especially the '.' (fullstop or period)) the module name must be enclosed in quote characters.

If all of the following are true:
1: Neither the **USELDS** or **NOUSELDS** operands (or their aliases) have been specified.
2: The **MMEFDSN=** operand has not been specified.
3: A dataset name was specified on the command that could be defined in an MMEF dataset (i.e. a positional dataset name or the **SYMDATALIB=** operand but not the **ADATALIB=** operand or the **DWARFLIB=** operand or the **SOURCELIB=** operand).
4: This is not the 'host' system (i.e. this is a dump).
5: No MMEF datasets have been defined.
then processing of the command behaves as if the **USELDS** operand was specified.

Complete MAP Command Syntax:

```
    MAP addressexpression  l-or-p-name      ...
        modulename         'pathname'
                           "pathname"
                           LIB=libname
                           PATH=pathname
                           omitted

    ... ADATA                      SYMDATA                      ...
```

```
        ADATALIB=libraryreference  SYMDATALIB=loadlibraryname
        ADATAPATH=pathname         SYMDATAPATH=pathname
        ADATAFILE=pathname         SYMDATAFILE=pathname

    ... DWARFDATA                  SOURCELIB=libraryreference
        DWARFLIB=libraryreference  SOURCEDIR=pathname
        DWARFPATH=pathname         SOURCEPATH=pathname
        DWARFFILE=pathname

    ... MMEFLIB=datasetname
        MMEFID=id
        MMEFDLPA={IPL|NO|YES}
        MMEFPOS={FRONT|BACK}

    ... [NO]ALLMESSAGES  START=addressexpression  ...

    ... [NO]USELDS

    ... LIBACCESSERRORS={IGNORE|FAIL}

    ... USS=option

    ... DUB=option
```

Shortcut: M

Command Dialog Syntax:

```
    ?
    MAP
    MAP ?
```

[Invokes a Command Dialog]

addressexpression
This is the MAP command's **first operand.** It must be an address expression that resolves to some location within a load module or program object. This identifies both the load module (or program object) that is to be mapped as well as the csect (within that module or object) that also is to be mapped.

Once the module is mapped, the given address is examined again to see if it resides within a mappable control section. If so, and if the appropriate feature **(asm/XDC** or **c/XDC)** is licensed, then the control section too is mapped.

Note, **modulename.csectname** is a perfectly valid address expression. (Example: MAP MYPROG.MYCSECT). But then so are **PSW?, R12?** and **0FFFFFF** (for the System Nucleus). See HELP ADDRESSING for complete details.

Note, when the **START=** operand also is given, then certain constraints are imposed upon this first operand. See HELP COMMANDS MAP PRIVATELYLOADED for details.

modulename

If the given address expression is a **pure** load module name (e.g. **MAP MYLOAD),** i.e. without any modifiers whatsoever, without even a trailing dot, then only a Module Map is brought into storage. No attempt is made to build a Csect Map.

If you want a Csect Map built as well, then type, for example, any of the following:
- MAP MYLOAD. (with a dot)
- MAP MYLOAD+0
- MAP MYLOAD.MYCSECT
- Or any other address expression that resolves to any location within the csect to be mapped.

In other words, use any addressexpression that is **not** a pure module name.

l-or-p-name
'pathname'
"pathname"
LIB=libname
PATH=pathname
Often Z/XDC can figure out for itself the name of the load library (dataset) or the USS file that contains the DASD-resident copy of the module to be mapped. When it cannot, then you must supply this operand to tell Z/XDC where to look.

If this operand is supplied, one of the following formats must be used:

l-or-p-name
The name of a load library (dataset) or USS path name. An unquoted name could be ambiguous, and Z/XDC will decide how it is processed based on the USS option in effect (see the description of the **USS=option** operand).

'pathname'
"pathname"
The quoted name of a USS path.

LIB=libname
Tbe name of a load library (dataset).

PATH=pathname
The name of a USS path. Note that the path name may be quoted.

If specifying a dataset then:
- the dataset name must be a fully qualified, **UNquoted** dataset name.
- the dataset may be a PDS or PDSE
- the dataset must contain load modules (PDS) or program objects (PDSE)
- the dataset cannot contain ADATA or GOFF
- a member name may be specified after the dataset name by enclosing it in parenthesis. If a member name is not specified it defaults to the member name specified by the first operand. Note that if mapping a USS path, and specifying an overriding dataset, a member name is required.
- The dataset **must** be catalogued. Z/XDC does not support uncatalogued datasets.
- The dataset name may contain variables. For their syntax please see **HELP COMMANDS SYNTAX DSNAMES**

If specifying a path name then:
- If the path name contains characters that have a special meaning to Z/XDC, the path name must be enclosed in quotes.
- the path name may contain variables. For their syntax please see **HELP DEBUGGING USS NAMES**

ADATA
ADATALIB=libraryreference
ADATAPATH=pathname
ADATAFILE=pathname

These operands coerce the **MAP** command towards building Csect Maps from **ADATA.** (SYM data and DWARF data are not considered.)

libraryreference can be any of the following:
- The name of a sequential dataset
- The name of a PDS or PDSE
- The name of a specific member within a PDS or PDSE

pathname can be any of the following:
- The unquoted name of a USS file or path containing a program object
- The quoted name of a USS file or path containing a program object

If the ADATA dataset or library or path is specified, then it may be any of the sequential files or libraries listed in HELP COMMANDS MAP ASSEMBLERMAPS.

If a dataset or library or path is not specified, then the currently active **MAPLIBS list** is searched. See HELP MAPS ADATA MAPLIBS for more information.

When a dataset or library is specified, it is automatically added to the currently active **MAPLIBS list.** If it is already in the list, then it is moved to the top of that list's search order. (at this time, USS files are not added to the MAPLIBS list).

These operands have no affect upon the loading of Module Maps.

The **USELDS** operand has no affect on the location of the ADATA dataset. It is always treated as a local dataset.

SYMDATA
SYMDATALIB=loadlibraryname
SYMDATAPATH=pathname
SYMDATAFILE=pathname

These operands coerce the **MAP** command towards loading maps from **SYM data.** (ADATA and DWARF data are not considered.) If a library is specified, then it must be a load library that contains the load module to be mapped.

pathname can be any of the following:
- The unquoted name of a USS file or path containing a program object
- The quoted name of a USS file or path containing a program object

If a load library or pathname is not specified, then Z/XDC examines the current environment's load module search order to find the load module being mapped.

When a load library or pathname is specified, and a Module Map has not yet been loaded, Z/XDC will search the given library or file for **both** the Module Map and the Csect Map. Z/XDC will skip examining the search order, and instead it will look only in this library or file to find **both** those maps.

When **both** the ADATALIB= and SYMDATALIB= operands are given, then:

- If a Module Map has not yet been loaded, then it will be loaded from the load library or file specified via the **SYMDATA?=** operand.

- If a Csect Map has not yet been loaded, then the ADATA?= dataset or library or file will be searched first.

- If the Csect Map is not found in the ADATA?= dataset or library or file, then the SYMDATA?= load library or file is searched.

DWARFDATA
DWARFLIB=libraryreference
DWARFPATH=pathname
DWARFFILE=pathname

These operands coerce the **MAP** command towards building Csect Maps from **DWARF data.** (ADATA and SYM data are not considered.)

libraryreference must be the name of a partitioned dataset library (PDS or PDSE) or USS File System folder. The referenced location must contain a member or file whose name matches the csect to be mapped. That member/file must contain the DWARF data to be used. Notes:

- **libraryreference** may **not** be a reference to either:
  - A sequential dataset
  - A specific member of a partitioned dataset
  - An individual file in a USS File System

- **libraryreference MAY** be a reference to a USS File System Folder.

- When the name of a csect to be mapped is longer than 8 characters, a truncated version of that name is used for selecting the library member from which DWARF data is read. The truncation rules are:
  - (1) When a name contains a **#,** both the leftmost # as well as all following characters are discarded.
  - (2) If more than 8 characters remain, then only the leftmost 8 characters are retained; all excess characters on the right are discarded.

- Use of the **DWARFLIB=** operand causes an entry to be added to the **DWARF** Family's Redirect List. This entry causes all future DWARF file references to be redirected to the library or folder that you specify.

The member or file selected will be the one whose name matches the csect being mapped. Specifically, the following command is issued internally to make all this happen:

```
LIBRARYLISTS REDIRECT DWARF MASK=**(*) USE=libraryreference(&2)
```

Where **libraryreference** is the libraryreference provided by the **DWARFLIB=** operand.

If you have a situation wherein you cannot use the DWARFLIB= operand (or the DWARFPATH= or DWARFFILE= operands), you will have to use the **LIBRARYLISTS** command instead. See HELP COMMANDS LIBRARYLISTS for more information.

If the **DWARFPATH=** or **DWARFFILE=** operands are used, the reference is to a USS file.

SOURCELIB=libraryreference
SOURCEDIR=pathname
SOURCEPATH=pathname

This operand notifies the **MAP** command of the location to search for any **C** source files needed for Source Image Maps.

libraryreference must be the name of a library (PDS or PDSE) or USS File System folder that contains a member or file whose name matches the csect to be mapped.

Notes:

- **libraryreference** may **not** be a reference to either:
  - A sequential dataset
  - A specific member of a partitioned dataset
  - An individual file in a USS File System
- **libraryreference MAY** be a reference to a USS File System Folder.
- **pathname MUST** be a reference to a USS File System Folder.
- When the name of a csect to be mapped is longer than 8 characters, a truncated version of that name is used for selecting the library member from which source data is read. The truncation rules are:
  - (1) When a name contains a **#,** both the leftmost # as well as all following characters are discarded.
  - (2) If more than 8 characters remain, then only the leftmost 8 characters are retained; all excess characters on the right are discarded.

Use of the **SOURCELIB=** or **SOURCEDIR=** or the **SOURCEPATH=** operands causes an entry to be added to the **CSOURCE** Family's Redirect List. This entry causes all future source file references to be redirected to the library or folder that you specify. The member or file selected will be the one whose name matches the csect being mapped. Specifically, the following command is issued internally to make all this happen:

```
LIBRARYLISTS REDIRECT CSOURCE MASK=**(*) USE=libraryreference(&2)
```

Where **libraryreference** is the libraryreference provided by the **SOURCELIB=** operand.

If you have a situation wherein you cannot use the SOURCELIB= or SOURCEDIR= or SOURCEPATH= operands, you will have to use the **LIBRARYLISTS** command instead. See HELP COMMANDS LIBRARYLISTS for more information.

START=addressexpression
When you need to build maps for a load module or program object that has been **Privately Loaded,** you will need to use this operand to let Z/XDC know where the module's **Load Point** is.

This operand is not permitted when USS is in effect. (see the description of the **USS=option** operand).

When you use this **START=** operand, the MAP command's **first** operand (the **addressexpression** operand) will need to reference the **name** of the Privately Loaded module. In other words, the command's **first** operand **must** start with a load module name. It **may not** start with a PSW reference, a register reference, an equate symbol a raw address or anything else that is not the name of the Privately Load module.

The reason is, the name you give will need to be used to find the proper load library member from which the module's Module Map will be built.

Once the Module Map has been built, the rest or Z/XDC is made aware of the existence of this module so that it can be used in commands and displays just like any other module in the system.

The MAP command's First Operand also is used to determine the offset (from its Load Point) to the Privately Loaded module's **Entry Point.** For complete details, please see HELP COMMANDS MAP PRIVATELYLOADED.

MMEFLIB=datasetname

If the **MMEFLIB** operand is used, the dataset named on the **MMEFLIB** operand and other **MMEF...** operands are used to internally issue a **DEFINE MMEFDSN** command to add the named MMEF dataset to the current MMEF datasets list. If the dataset is already defined the request is ignored. If definition errors occur, an error message is issued and the **MAP** command fails. Refer to the **DEFINE MMEFDSN** command for more details.

MMEFID=id

If the **MMEFLIB** operand is used, the **MMEFID** operand can be used to supply a specific id value for the MMEF definition. Refer to the **DEFINE MMEFDSN** command for more details. If the **MMEFLIB** operand is not used, the **MMEFID** operand is validated but ignored.

MMEFDLPA={NO|YES!IPL}

If the **MMEFLIB** operand is used, the **MMEFMSGS** operand can be used to supply a specific value for the DLPA operand of the **DEFINE MMEFDSN** command. Refer to the **DEFINE MMEFDSN** command for more details. If the **MMEFLIB** operand is not used, the **MMEFDLPA** operand is validated but ignored.

MMEFPOS={FRONT|BACK}

If the **MMEFLIB** operand is used, the **MMEFPOS** operand can be used to supply a specific values for the POS operand of the **DEFINE MMEFDSN** command. Refer to the **DEFINE MMEFDSN** command for more details. If the **MMEFLIB** operand is not used, the **MMEFPOS** operand is validated but ignored.

ALLMESSAGES

When searching for map data, the **MAP** command may search in a lot of locations before finding the data it is looking for. Normally, when a map is loaded successfully, the error messages generated by one or more unsuccessful look-see's are discarded. This **ALLMESSAGES** operand causes those transitory error messages to be displayed instead of being discarded.

NOALLMESSAGES

This is the default action: All transitory error messages are discarded unless the **MAP** command was unable to find the map data anywhere, in which case they are all displayed.

USELDS ["use local dataset"]
this operand can also be specified using one of the following aliases:
USELFILE
USELPATH
If this operand is given, all defined MMEFDSN tokens are ignored. If a library was explicitly named on the MAP command, that library must exist in the local ('host') system (as if no MMEFDSN tokens had been defined).

NOUSELDS
this operand can also be specified using one of the following aliases:
NOUSELFILE
NOUSELPATH

This is the default action: If any MMEFDSN tokens have been defined and a specific dataset name has been specified on the **MAP** command the dataset name must be found in a LIBx definition loaded by the **DEFINE MMEFDSN** command.

LIBACCESSERRORS={IGNORE|FAIL}
This operand can be used to control the processing if MMEFDSN tokens have been defined and the **MAP** command has identified that it is trying to map something in the private area of an address space. Mapping will search the libraries specified (for example) on the STEPLIB ddname before searching the linklist. However some of the libraries may not be found in any MMEFDSN token.
Note that **LIBACCESSERRORS=IGNORE** is the default.

If **LIBACCESSERRORS=IGNORE** is in effect **(this is the default)** the error is ignored unless the item being mapped is not found at all, in which case an error is reported.

If **LIBACCESSERRORS=FAIL** is in effect the error is reported when the missing library is detected.

USS=option
This operand is parsed and removed from the command before other operands are processed, regardless of its position in the command. It (or its default) affects the meaning of some of the operands on the command.

For a description of the possible values, their meaning, and the interaction with the USS profile facility please see **HELP DEBUGGING USS OPERANDS USS=**

DUB=option
This operand is parsed and removed from the command before other operands are processed, regardless of its position in the command. It (or its default) controls the processing if Z/XDC determines that dubbing is required.

For a description of the possible values, their meaning, and the interaction with

the DUB profile facility please see **HELP DEBUGGING USS OPERANDS DUB=**

Help COmmands Map Bindermaps

Binder Maps

Load module maps and program object Maps (collectively, **Binder Maps)** consist of linkedit time labels (also known as bind time labels or **ESD** symbols). Such labels include:

- Class names
- Control section names (i.e. csect names)
- Entry names
- Etc.

Binder Maps can always be loaded for any load module or program object. Also, the ability to load and use Binder Maps is a part of **base/XDC.** So this capability is **always present regardless** of what Z/XDC features are or are not licensed.

Binder Maps may or may not be flagged as being **case-sensitive.** If the map contains **mixed-case** names, then it will be flagged as being case-sensitive; otherwise, it will not.

When a map is flagged as being case-sensitive, references to names containing lowercase letters requires the use of quotes. When a map is not so flagged, the use of quotes is irrelevant. For more information about the consequences of this, see HELP ADDRESSING PARSERS ASM MIXEDCASE.

In general **any** load module or program object in any accessible address space can be mapped. This includes:

- Modules on any accessible address space’s Job Pack Queues.
- Modules on the System’s Dynamic Link Pack Queue (DLPA, LPQ, MLPA and FLPA).
- Modules in the System’s Pageable Link Pack Area (PLPA).
- The System Nucleus itself (IEANUC0x). (Z/XDC contains special code to sort out the way in which the Nucleus is built.)
- Any module whose presence is described by standard system control blocks (LPDEs and CDEs and CICS APEs).
- Modules loaded by the USS (Unix System Services) have control blocks built by the system (as per the previous point). Z/XDC has logic to handle these modules, which are always program objects.
- Even modules that have been **Privately Loaded** and, thus, are not on any system queue at all.
- Modules that have been loaded into a CICS aspace via CICS’s loader.

Help COmmands Map Privatelyloaded

Mapping Privately Loaded Load Modules

If a load module has been loaded into storage through mechanisms that do not create a system control block (a CDE usually) that describes its existence and location, then the module is invisible to Z/XDC, and so a normal MAP command cannot be used to map it.

However, you can use the MAP command's **START=** operand to tell Z/XDC where the module actually is. Then Z/XDC can go ahead and build a Module Map.

Additionally, once the Module Map is built, the rest of Z/XDC is notified of the existence of the module. That allows the module to be referenced by other commands and to show up in displays just like any other load module in the System. Examples:

- The module shows up in **LIST PGMS** displays,
- It shows up in the header lines of **FORMAT** and **WHERE** commands,
- It can displayed by the **LIST LKEDMAP** command,
- Privately Loaded XL C/C++ and Metal C modules can be **source image mapped** by a normal MAP command.
- etc.

Relevant MAP Command Syntax

This a partial syntax description focusing on those operands that are important to mapping Privately Loaded load modules. For the comprehensive syntax description, see **HELP COMMANDS MAP SYNTAX.**

```
MAP modulename[.csectname][+offset]  loadlibraryname ...
                                     omitted

... START=addressexpression ...

... [NO]ALLMESSAGES
```

MAP modulename[.csectname][+offset]
Normally, the MAP command's first operand would be a normal address expression that would resolve to a storage location within the module to be mapped. But when the **START=** operand is present, it provides the target module's location, and so this first operand is treated differently.

Although this first operand is not needed for the target module's address, it is still needed for two other purposes:

- First, it provides the name of the load library member from which to read the mapping data.
- Second, it provides the offset into the load module to its Entry Point.

Accordingly, while this first operand's syntax is similar to an address expression, it is restricted with respect to a normal address expression's syntax. Here are the rules:

- 1) The expression must start with the name of the load library member that contains the load module or program object whose mapping data is to be read. Then going forward, Z/XDC will use this name as the Privately Loaded module's

name.

- 2) If the module's entry address is not the same as its load point, then this first operand must contain additional information indicating the offset of the Entry Point from the Load Point. That additional information may consist of either or both of the following:
  - A csect name or any other external name (i.e. an ESD symbol from the load module)
  - Plus or minus a hex offset value that is relative either to the csect name (if given) or to the module's load point otherwise.

This first operand may not contain:
- A PSW reference
- A register reference
- Any pointer operation (% ? ! etc.)
- Any reference to any symbol or label other than an ESD symbol from within the module itself.

loadlibraryname
Typically, the PDS or PDSE containing the copy of your USS loaded module might not be in for environment's Load Library Search Order. So you probably have to provide that name manually.

When given, **loadlibraryname** must be a fully qualified, **UNquoted** dataset name.

Loadlibraryname may contain **variable symbols** for which Z/XDC will make substitutions. (This allows for the dsname to be reactive to the current time, date, userid, jobname, etc.) These variables are most useful when the **MAP** command is executed from within a command script or a DEAD trap. For more information, see **HELP COMMANDS SYNTAX DSNAMES.**

Loadlibraryname must be the name of a **load** library. It may be either a PDS or a PDSE. It may **not** be the name of an ADATA library or GOFF file.

Loadlibraryname must be cataloged. (Z/XDC does not contain any support for finding datasets that are not cataloged.)

START=addressexpression
This operand must resolve to your USS loaded or privately loaded module's **Load Point** (not its Entry Point). See examples below for one way to determine a USS loaded module's Load Point.

Once the Module Map has been built, the rest of Z/XDC is made aware of the existence of this module so that it can be used in commands and displays just like any other module in the system.

Note, privately Loaded load modules can also be identified by use of the **DMAP** and **USING** commands. Such modules then can be referenced just as freely throughout the reset of Z/XDC but with the **single exception** that they cannot be C source imaged mapped when identified that way.

For more information, see **HELP MAPS PRIVATELYLOADED.**

Example:

Suppose that a module named CSAMPLE has been loaded into storage by the LOAD macro using the ADDR= operand to place the module at a specific address. In this case the system does not build a CDE to describe the module.

Also, the load module was loaded from DBCOLE.MY.LOADLIB, using an opened DDName and the DCB for that opened DDName was specied on the LOAD macro (The DCB= operand).

Let’s suppose that the address that the module was loaded at (the value of the ADDR= operand on the load macro) is **1d948000.**

So to map this beast, we can do the following:

EQUATE CSAMPEPA 1D948EE2
EQUATE CSAMPLPT CSAMPEPA-EE2
I’m creating a couple of equates just to make things a bit easier.

MAP CSAMPLE.CEESTART DBCOLE.MY.LOADLIB START=CSAMPLPT
Here’s what’s happening:

CSAMPLE.CEESTART
DBCOLE.FROMUSS.LOADLIB
These operands tell Z/XDC the following:
- A copy of the load module can be found in DBCOLE.MY.LOADLIB(CSAMPLE).
- The module’s Entry point is at the CEESTART label.

START=CSAMPLPT
This tells Z/XDC that the load module is located in storage starting at 1D948000.

Upon successful completion of these commands, Z/XDC will now know where CSAMPLE is, and it will be able to treat the module no differently from any other module in the system.

In particular, you can now use the following command to load a C source map for this module:

- **MAP CSAMPLE.**
  Important: The trailing dot matters.

Help COmmands Map Assemblermaps

Assembler Maps

When the **asm/XDC Feature** is licensed, Z/XDC supports two kinds of Assembler Maps: SYM Data Maps and ADATA Maps.

- **SYM Data Maps** contain assembly-time statement labels and field names. They are built from **SYM** records. SYM records are available only for csects that have been **both** assembled and linkedited with **PARM=TEST** specified for **both** the Assembler and the Binder. For more information, see **HELP MAPS SYM.**

- **ADATA Maps** contain complete source program statement images. They are built from **ADATA** records. These records are created by:

  - IBM's High Level Assembler (www-01.ibm.com/software/awdtools/hlasm). Use **PARM=ADATA.**

  - Tachyon Software's Cross Assembler and z/Assembler (www.tachyonsoft.com/txaover.html).

  - Dignus' Systems/ASM (www.dignus.com).

  Z/XDC is capable of reading Assembler ADATA from any of the following sources:

  - Files and libraries containing **//SYSADATA** output created by any mainframe Assembler (see above) that supports producing IBM-compatible ADATA. (If using IBM's High Level Assembler, then specify **PARM='ADATA'.)**

  - GOFF files containing embedded ADATA. (If using IBM's High Level Assembler, then specify **PARM='GOFF(ADATA)'.)**

  - Files containing streamed ADATA produced by a PC based mainframe Assembler and then binary-uploaded **as a stream** to the mainframe into a **RECFM=FB** file of arbitrary LRECL. (When reading the Fixed Blocked file, Z/XDC will ignore the record boundaries and instead identify and piece together the ADATA records based solely upon record length information found within the ADATA records themselves.)

  - **ADATAnnnn** classes found within program objects. These classes will be present in a program object when the Binder finds that ADATA is present in the GOFF file produced by the Assembler.

  For most of these cases, the **SET MAPLIBS** command must be used to make the ADATA side files known to Z/XDC and, therefore, available to the **MAP** command. For more information, see **HELP MAPS ADATA.**

if this is not a 'host' system (such as a dump) the 'DEFINE MMEFDSN' command can be used to define one or more load module information sets created (on the original system) by the load module information extract utility. In this case the 'MAP' command will extract SYM or ADATA information from the relevant information set rather than the current system. If a specific dataset name is specified on the 'MAP' command, that dataset name must be found in the information set (i.e. one of the datasets on the original system where the information was extracted from) rather than the current system. However, the 'USELDS' operand can be used to force the dataset to be on the 'host' (current) system - Z/XDC assumes that you know what you are doing in this case.

Assembler Maps are **never** flagged as being case-sensitive. This is because even though the Assembler supports mixed-case names, it treats those names in a case-insensitive way: When two names are identical except for case, the Assembler considers them to be the exact same name. For more information about the consequences of case-insensitivity (and the lack thereof), see **HELP ADDRESSING PARSERS ASM MIXEDCASE.**

Relevant MAP Command Syntax

This a partial syntax description focusing on those operands that are important to mapping Assembler Programs. For the comprehensive syntax description, see **HELP COMMANDS MAP SYNTAX.**

```
MAP addressexpression  loadlibraryname  ...
                       omitted

... ADATA                       SYMDATA                       ...
    ADATALIB=libraryreference   SYMDATALIB=loadlibraryname

... [NO]ALLMESSAGES  START=addressexpression  ...

... [NO]USELDS
```

addressexpression
This is the MAP command's **first operand.** It must be an address expression that resolves to some location within a load module or program object. This identifies both the load module (or program object) that is to be mapped as well as the csect (within that module or object) that also is to be mapped.

Once the module is mapped, the given address is examined again to see if it resides within a mappable control section. If so, and if the **asm/XDC** feature is licensed, then the control section too is mapped.

Note, **modulename.csectname** is a perfectly valid address expression. (Example: MAP MYPROG.MYCSECT) But then so are **PSW?** and **R12?.** See **HELP ADDRESSING** for complete details.

modulename
If the given address expression is a **pure** load module name (e.g. **MAP MYLOAD),** i.e. without any modifiers whatsoever, without even a trailing dot, then only a Module Map is brought into storage. No attempt is made to build a Csect Map.

If you want a Csect Map built as well, then type, for example, any of the following:
- MAP MYLOAD. (with a dot)
- MAP MYLOAD+0
- MAP MYLOAD.MYCSECT
- Or any other address expression that resolves to any location within the csect to be mapped.

In other words, use any addressexpression that is **not** a pure module name.

ADATA
ADATALIB=libraryreference
These operands coerce the **MAP** command towards building Csect Maps from **ADATA.** (SYM

data and DWARF data are not considered.)

libraryreference can be any of the following:
- The name of a sequential dataset
- The name of a PDS or PDSE
- The name of a specific member within a PDS or PDSE

If the ADATA dataset or library is specified, then it may be any of the sequential files or libraries listed above under the heading: **Assembler Maps.**

If a dataset or library is not specified, then the currently active **MAPLIBS list** is searched. See **HELP MAPS ADATA MAPLIBS** for more information.

When a dataset or library is specified, it is automatically added to the currently active **MAPLIBS list.** If it is already in the list, then it is moved to the top of that list's search order.

The **ADATA** and **ADATALIB=** operands have no affect upon the loading of Module Maps.

SYMDATA
SYMDATALIB=loadlibraryname
These operands coerce the **MAP** command towards loading maps from **SYM data.** (ADATA and DWARF data are not considered.) If a library is specified, then it must be a load library that contains the load module to be mapped.

If a load library is not specified, then Z/XDC examines the current environment's load module search order to find the load module being mapped.

When a load library is specified, and a Module Map has not yet been loaded, Z/XDC will search the given library for **both** the Module Map and the Csect Map. Z/XDC will skip examining the search order, and instead it will look only in this library to find **both** those maps.

When **both** the ADATALIB= and SYMDATALIB= operands are given, then:

- If a Module Map has not yet been loaded, then it will be loaded from the load library specified via the **SYMDATALIB=** operand.
- If a Csect Map has not yet been loaded, then the ADATALIB= dataset or library will be searched first.
- If the Csect Map is not found in the ADATALIB= dataset or library, then the SYMDATALIB= load library is searched.

Help COmmands Map CMAps

XL C/C++ and Metal C Program Maps

When the **c/XDC feature** is installed, Z/XDC supports Source Image Maps for programs written in the XL C/C++ and Metal C languages.

Source Image Maps contains complete source program statement images as well as comprehensive information about all C variables used in the program. The Source Image Maps themselves are built from **DWARF data** and the original source files (usually pointed to by information hardcoded in the DWARF data).

Relevant MAP Command Syntax

This a partial syntax description focusing on those operands that are important to mapping XL C/C++ and Metal C Programs. For the comprehensive syntax description, see HELP COMMANDS MAP SYNTAX.

```
MAP addressexpression  loadlibraryname  ...
    modulename         omitted

... DWARFDATA                  SOURCELIB=libraryreference  ...
    DWARFLIB=libraryreference

... [NO]ALLMESSAGES  START=addressexpression  ...
```

addressexpression
This is the MAP command's **first operand.** It must be an address expression that resolves to some location within a load module or program object. This identifies both the load module (or program object) that is to be mapped as well as the csect (within that module or object) that also is to be mapped.

Once the module is mapped, the given address is examined again to see if it resides within a mappable control section. If so, and if the **c/XDC** feature is licensed, then the control section too is mapped.

Note, **modulename.csectname** is a perfectly valid address expression. (Example: MAP CSAMPLE.srccsamp#C_C_CODE). But then so are **PSW?, R12?** and **0FFFFFF** (for the System Nucleus). See HELP ADDRESSING for complete details.

Also, the **CEESTART** csect is specifically misinterpreted to mean, "The csect containing the first C source statement". So **MAP CSAMPLE.CEESTART** will work too.

modulename
If the given address expression is a **pure** load module name (e.g. **MAP MYLOAD),** i.e. without any modifiers whatsoever, without even a trailing dot, then only a Module Map is brought into storage. No attempt is made to build a Csect Map.

If you want a Csect Map built as well, then type, for example, any of the following:
- MAP MYLOAD. (with a dot)
- MAP MYLOAD+0
- MAP MYLOAD.MYCSECT
- Or any other address expression that resolves to any location within the csect to be mapped.

In other words, use any addressexpression that is **not** a pure module name.

DWARFDATA
DWARFLIB=libraryreference
These operands coerce the **MAP** command towards building Csect Maps from **DWARF data.** (ADATA and SYM data are not considered.)

libraryreference must be the name of a partitioned dataset library (PDS or PDSE) or USS File System folder or //ddname. The referenced location must contain a member or file whose name matches the csect to be mapped. That member/file must contain the DWARF data to be used. Notes:

- **libraryreference** may **not** be a reference to either:
  - A sequential dataset
  - A specific member of a partitioned dataset
  - An individual file in a USS File System

- **libraryreference MAY** be a reference to a USS File System Folder.

- **libraryreference MAY** be the //ddname of an existing allocation to an suitable library or File System folder.

- When the name of a csect to be mapped is longer than 8 characters, a truncated version of that name is used for selecting the library member from which DWARF data is read. The truncation rules are:
  - (1) When a name contains a **#,** both the leftmost # as well as all following characters are discarded.
  - (2) If more than 8 characters remain, then only the leftmost 8 characters are retained; all excess characters on the right are discarded.

- Use of the **DWARFLIB=** operand causes an entry to be added to the **DWARF** Family's Redirect List. This entry causes all future DWARF file references to be redirected to the library or folder that you specify.

The member or file selected will be the one whose name matches the csect being mapped.

Specifically, the following command is issued internally to make all this happen:


```
LIBRARYLISTS REDIRECT DWARF MASK=**(*) USE=libraryreference(&2)
```

Where **libraryreference** is the libraryreference provided by the **DWARFLIB=** operand.

If you have a situation wherein you cannot use the DWARFLIB= operand, you will have to use the **LIBRARYLISTS** command instead. See HELP COMMANDS LIBRARYLISTS for more information.

SOURCELIB=libraryreference
This operand notifies the **MAP** command of the location to search for any source files needed for Source Image Maps.

libraryreference must be the name of a library (PDS or PDSE) or USS File System folder or //ddname that contains a member or file whose name matches the csect to be mapped.

Notes:
- **libraryreference** may **not** be a reference to either:
  - A sequential dataset
  - A specific member of a partitioned dataset
  - An individual file in a USS File System

- **libraryreference MAY** be a reference to a USS File System Folder.

- **libraryreference MAY** be the //ddname of an existing allocation to an suitable library or File System folder.

- When the name of a csect to be mapped is longer than 8 characters, a truncated version of that name is used for selecting the library member from which source data is read. The truncation rules are:
  - (1) When a name contains a **#,** both the leftmost # as well as all following characters are discarded.
  - (2) If more than 8 characters remain, then only the leftmost 8 characters are retained; all excess characters on the right are discarded.

Use of the **SOURCELIB=** operand causes an entry to be added to the **CSOURCE** Family's Redirect List. This entry causes all future source file references to be redirected to the library or folder that you specify. The member or file selected will be the one whose name matches the csect being mapped.

Specifically, the following command is issued internally to make all this happen:

```
LIBRARYLISTS REDIRECT CSOURCE MASK=**(*) USE=libraryreference(&2)
```

Where **libraryreference** is the libraryreference provided by the **SOURCELIB=** operand.

If you have a situation wherein you cannot use the SOURCELIB= operand, you will have to use the **LIBRARYLISTS** command instead. See HELP COMMANDS LIBRARYLISTS for more information.

Help COmmands Map Dwarffiles

DWARF Data Files

DWARF data is created by the C compiler (CCNDRVR) for XL C/C++ and by the CDA Assembler (CDAHLASM) for Metal C. See HELP DEBUGGING C SETUP on how to generate DWARF data.

When c/XDC builds a DWARF data based map:

- Programs can be formatted and display by source line number or function name.

- Variables and their contents can be displayed and zapped.

In order for the mapping process to be successful, c/XDC's knowledge of the location of DWARF and source files must be correct. For XL C/C++ programs, the location of the associated DWARF file is embedded in a metadata area in each csect. If you rename the files or copy the DWARF data into other datasets or libraries, then you will have to use the **LIBRARYLISTS** command to override the dsname information saved

at compile time.

For Metal C programs, the location of the DWARF file is **not** available in the generated csects, so you will have to use the LIBRARYLISTS command regardless.

In **all** cases (XL C/C++ and Metal C), DWARF files contain the locations of the program's source and header files.

If the DWARF file or the source files of the compilation unit you wish to map have been moved or renamed, or if you are trying to map a Metal C program, you will have to use the **LIBRARYLISTS** command to set up DWARF and/or CSOURCE **redirects** (either or both). The LIBRARYLISTS command maintains separate redirect lists for DWARF files vs.C source files.

LIBRARYLISTS redirects essentially override the file name information saved within a program. They inform c/XDC where to look for the required DWARF and/or source files.

Library Lists and the whole redirect process are discussed in detail in **HELP DEBUGGING C LIBRARYLISTS.**

The **DWARFLIB=** and **SOURCELIB=** operands of the **MAP** command can also be used to override the location information of a program's DWARF and SOURCE files. See HELP COMMANDS MAP CMAPS for more information.

Help COmmands Map CMDdialogs

Command Dialog

A Command Dialog is available for assisting with constructing and issuing MAP commands. Just issue the command either with no operands or with a lone question mark as its sole operand. For information about Command Dialogs, see **HELP CMDDIALOGS.**

Relevant MAP Command Syntax

This is a partial syntax description focusing only in the operands used for invoking the Command Dialogs. For the full syntax description for everything else you can do with the MAP command, see **HELP COMMANDS MAP SYNTAX.**

?
MAP
MAP ?

[Invokes a Command Dialog]

Help COmmands Map Findingmapdata

The Map Search and Construction Process

To recap: The MAP command attempts to build either one or two maps: First a Module Map, then a Csect Map.

A Module Map is always built if it has not already been built by a prior MAP command. A Csect Map may or may not be built depending upon:

- Whether or not map information (SYM data, ADATA or DWARF data) for a csect is available.
- Whether or not the MAP command's first operand is a pure module name or a more complex address expression. (In the pure module name case, the attempt to build a Csect Map is bypassed.)
- Whether or not the **asm/XDC** feature and/or **c/XDC** feature have been licensed for use.
- Whether or not the **START=** operand is present. (START= inhibits the loading of csect maps.)

If a Module Map is to be built, Z/XDC must find the library in which the load module or program object resides, and then read that library to obtain the ESD information necessary for building the MAP. (Module Maps **cannot** be built from ADATA, SYM data or DWARF data.)

If the MAP command was given with a library name provided either as a second positional operand or as the value of a SYMDATALIB= operand, then Z/XDC presumes that it is the load library in which the load module or program object resides, and proceeds to open it and read it accordingly.

On the other hand, if no library name was given (either via the MAP command's second positional operand or via the value of a SYMDATALIB= operand), then Z/XDC will search system structures in an effort to find the library in which the load module resides. The search stops at the first library that contains a member whose name matches the load module or program object.

If no such member is found, then the MAP command fails.

The following libraries are searched.
- The current task's task library, if any.
- Task libraries owned by any ancestor task, ranging from newest back to oldest ancestor. (In an ISPF environment for example, //ISPLLIBs are task libraries.)
- The STEPLIB or JOBLIB library. (One or the other of these will be the task library for the home address space's jobstep task.)
- DLPA libraries.
- FLPA libraries.
- MLPA libraries.
- PLPA libraries.
- The linklist libraries.

You can use the **LIST TASKS** command to see what task libraries are assigned to which tasks.

You can use the **LIST SEARCHORDER** command to see your current task's load library search order.

If the module being mapped is located in common storage, then only the DLPA, FLPA, MLPA, and PLPA libraries and the linklist libraries are searched. Task libraries, STEPLIB libraries, and JOBLIB libraries are **not** searched.

Once a Module Map has been built, the MAP command then determines whether or not it also should try to build a Csect Map. (See above.) If so, then it searches for ADATA, SYM data or DWARF data according to the following process:

If a **loadlibraryname** operand (second positional operand) was **not** given on the MAP command, then the Csect Map search proceeds as follows:

- If the **c/XDC Feature** has been licensed, then Z/XDC will check to see if the target load module contains XL C/C++ and Metal C csects. If so, then an XL C/C++ and Metal C Source Image Map will be built. Also, XL C/C++ and Metal C variable pools will be mapped.

- **Otherwise,** if the **asm/XDC Feature** has been licensed, then Z/XDC will start searching for an Assembler program map as follows:

  - Z/XDC will search for ADATA in the currently active MAPLIBS list, from first dataset to last. (SYM data is not searched for here.) The datasets may contain either pure ADATA as written to a //SYSADATA file by the Assembler, or GOFF files containing ADATA.

    Each dataset in the MAPLIBS list may be any of the following:
    1) A sequential file.
    2) A specific member of a partitioned dataset.
    3) A partitioned dataset without any particular member specified.

    When a MAPLIB dataset is a sequential file or a specific member of a partitioned dataset, then that dataset will be searched unconditionally. No match is attempted between any part of the dataset name or member name and the name of the csect being mapped. (This is because typically an ADATA file may contain mapping data for multiple csects, not just one csect whose name matches the member name.)

    When a MAPLIB dataset is a partitioned dataset without any particular member specified, then at most only two members of that library will be searched as follows:

    - If the library contains a member whose name matches the name of the csect being mapped, then that member will be searched first.

    - If that search fails, then the library is checked for a member whose name matches the name of the load module or program object containing the csect being mapped. Note, member name case-sensitivity does not apply to load module or program object names.

  - If the MAPLIBS search fails to find appropriate ADATA (or if the MAPLIBS search was bypassed), then the load module or program object itself is searched twice: First for ADATA, then for SYM data.

  - If this last search fails, then the MAP command fails.

If a **loadlibraryname** operand **was** given on the MAP command, then the Csect Map search proceeds as follows:

- If the **c/XDC Feature** has been licensed, then Z/XDC will check to see if the target load module contains XL C/C++ and Metal C csects. If so, then an XL C/C++ and Metal C Source Image Map will be built. Also, XL C/C++ and Metal C variable pools will be mapped.

- If the **asm/XDC Feature** has been licensed, then Z/XDC will search for an Assembler Program Map as follows:

  - The load module or program object is searched twice: First for ADATA, then for SYM data.

  - If this last search fails, then the MAP command fails.

  Note: when a **loadlibraryname** operand is given, a search for ADATA in the MAPLIB libraries is **not** done.

For additional information, see HELP MAPS.

Help COmmands Map Examples

Examples:

For a detailed example illustrating how to map programs loaded into storage by Unix System Services (USS), see **HELP COMMANDS MAP PRIVATELYLOADED.**

For other examples, read on.

M FFFFFF
Address 00FFFFFF is contained within the System Nucleus, so a Module Map for the Nucleus is built.

Usually, the Nucleus is IEANUC01, but even if an alternate Nucleus has been loaded, Z/XDC knows this. The right map is always built.

SET MAPLIBS MYUID.ADATA PROJECT.ADATA
MAP PSW?
Maps for the currently executing load module and csect (if available) are built. The process is as follows:

- The **SET MAPLIBS** command establishes two datasets (MYUID.ADATA and PROJECT.ADATA) as map libraries, i.e. places that Z/XDC can look when searching for ADATA. (In this case, these are in addition to any MAPLIBs that may have been established by prior SET MAPLIBS commands.) See **HELP MAPS ADATA MAPLIBS** for more information.

- Then the **MAP** command builds the load module's Module Map from ESD information found in the load library from which the load module was loaded. (Z/XDC automatically searches task libraries, STEPLIBs, JOBLIBs, LPALIBs, and linklist libraries in order to find the load module.)

- The **MAP** command then searches the MAPLIB datasets (MYUID.ADATA and PROJECT.ADATA) for ADATA that describes the csect into which the PSW points.

- If ADATA is not found in the MAPLIBs, then the **MAP** command checks the load module itself. If the load module actually is a program object, and if the program object contains **ADATAnnnn** classes, then Z/XDC builds a Source Image type Csect Map from the ADATA.

- If the load module is not a program object, or if it is a program object but does not contain ADATAnnnn classes, then Z/XDC sees if it contains SYM data. If so, then it builds a Symbol type Csect Map.

M PSW?,*U.MYPROG.LOAD
M PSW?,MYUID.MYPROG.LOAD
Both of these commands are similar to the preceding examples, but now I have explicitly identified the library dataset that contains the DASD-resident copy of the desired load module. Notes:

- Z/XDC substitutes the current TSO userid string for "*U". See **HELP COMMANDS SYNTAX DSNAMES** for more information.
- Because a load library name has been explicitly given, the MAPLIBs search for ADATA is bypassed.
- Nonetheless, the load module or program object itself is still searched for ADATAnnnn classes and SYM data.

M MYPROG.MYCSECT
The Module Map for MYPROG is built. If a Csect Map for MYCSECT is available (DWARF data, ADATA or SYM data), then it too is built.

M MYPROG
Only the Module Map for MYPROG is built. No attempt is made to build any Csect Map from any source. This is because the address expression operand ("MYPROG") actually is a pure load module name.

M MYPROG+0
M MYPROG.
Both of these commands do the same thing. First, the Module Map for MYPROG is built. Then an attempt is made to build a Csect Map for whatever csect is located at MYPROG's entry point.

M MYPROG.X#1
The Module Map for MYPROG is built. Then an attempt is made to build a Csect Map for

whatever csect is located at MYPROG's load point. (Note, it is often the case that a module's load point address and entry point address are not the same address!)

M CSAMPLE.
M CSAMPLE.main
M CSAMPLE.CEESTART
All of these commands do the same thing:

- If a Module Map for CSAMPLE has not yet been built, then one is built now.
- After that, if the **c/XDC Feature** is licensed, then an attempt is made to read the Source Image Map for the XL C/C++ and Metal C module's main program. (Source Image Map data is built from DWARF data associated with the CSAMPLE program object.)
- If there is no DWARF data, and if the **asm/XDC Feature** is installed, then Z/XDC will attempt to load a Csect Map from either ADATA or SYM data.
- If ADATA or SYM data is either not allowed or not found, then no Csect Map is loaded.

Note, while it is true that for C programs, CEESTART is not the name of the csect containing the **main()** C program, c/XDC has special logic to pretend that it is that csect. But this special misinterpretation occurs only for references to CEESTART's exact address. CEESTART+1, for example, is not adjusted and would not work for loading C maps.

Help COmmands Note

During a debugging session, you may wish to make notes and comments about various addresses of interest. This can be done with the **NOTE** command. What you do is type **NOTE** followed by an address expression followed by arbitrary commentary. This information is then saved by Z/XDC into a list. Later, you can display the entire list of notes with the LIST NOTES command.

Syntax:

NOTE addressexpression comment
'comment'

Shortcut: N

addressexpression
This is any address expression defining the location to be noted.

comment
'comment'
This is any comment message to be saved, along with the address, in the note list.

This operand is optional.

- **comment**
  If the comment is **not** framed by quote characters, then it may not contain semicolons (;) or colons (:), It may contain embedded quote characters. If it does, then they need not be doubled.

- **'comment'**
  If the comment **is** framed by quote characters, it may contain semicolons and colons. The framing quote characters will be removed, and embedded double quote characters will be singlized.

For more information, see **HELP COMMANDS SYNTAX CHARACTERSTRINGS.**

The note list can be displayed and updated at any time with the **LIST NOTES** command. See **HELP COMMANDS LIST NOTES** for more information.

Help COmmands Off

The **OFF** command deletes various categories of breakpoints and hooks.

When a breakpoint or hook is "deleted":
- The original opcode is restored to the breakpoint or hook location (unless other breakpoints remain at that same location).
- The breakpoint's or hook's definition is purged.

The **OFF** command accepts any combination of breakpoint selection operands repeated any number of times. Some combinations (such as **OFF ALL ACTIVE)** are redundant; however, they still are permitted.

The **OFF** command is a shorthand for the **SET BREAKPOINTS OFF** command, but notice that **SET BREAKPOINTS** can do more than just delete breakpoints. It also can disable and reenable them without deleting them. See **HELP COMMANDS SET BREAKPOINTS** for the details.

Syntax:

The syntax for the **OFF** command is exactly the same as for **SET BREAKPOINTS OFF.** Briefly, it is:

```
OFF                  selections  [NO]MSG     [USS=option]
SET BREAKPOINTS OFF  selections  [NO]MSG
```

Where **selections** is one or more operands identifying the breakpoints to be deleted by either name or category or type or location.

The operands are explained and exampled in detail at **HELP COMMANDS SET BREAKPOINTS.**

Help COmmands PAnelid

When Z/XDC is using ISPF Display Services to display its displays, the **PANELID** command functions in the same way it does in ISPF: It enables or disables ISPF's display of the current panel's name in the upper left-hand corner of the display screen.

Syntax:

```
PANELID
PANELID ON
PANELID OFF
```

This command either turns on, turns off or toggles the display of panel names when it's using ISPF Display Services.

This setting will survive the end of the debugging session. It will also affect ISPF when invoked recursively via Z/XDC's **ISPF** command.

Note, if you turn this on, you will discover very quickly that all panels generated within Z/XDC will always be named **XDCV31A.**

With respect to ISPF's **PANELID** command, see IBM's **ISPF User's Guide, Vol. I** (SC34-4822) for details.

Help COmmands POst

The **POST** command posts a given Event Control Block (ECB) with a given code. The ECB may be located in any accessible address space, as well as in common storage.

Syntax:

```
POST addressexpression hi-order-code low-order-code
                       omitted       omitted
                       ,
```

addressexpression
This should be the address of a valid ECB located in any accessible foreign or local address space.

If the ECB is located in common storage, then you must first use the **SET ASID** command to switch to the address space that is WAIT'ing on that ECB; otherwise, the POST will fail.

hi-order-code
This must be two hexadecimal digits giving the code to store into the left-most byte of the ECB. The given code must fall into the range of 40-7F (hex). In other words

the given code must have the post-bit (X'40') on and the wait-bit (X'80') off. If a given code does not conform to these requirements, then it is adjusted as necessary to make it conform. If omitted, then code X'40' is stored.

If the hi-order-code is omitted and the low-order-code is given, then the omission must be indicated by an extra comma.

low-order-code
This must be an address expression. It is resolved to a 3-byte value (regardless of the program's current addressing mode) and stored into the low-order three bytes of the ECB. If omitted, then the value X'000000' is stored.

Examples:

POST .MYECB,FF
The value X'7F000000' is POST'd into the ECB located at the label "MYECB".

POST R10?,,R5%
The ECB pointed to by R10 is to be POST'd. The value X'40' is POST'd into the first byte of the ECB. The 24-bit address pointer contained in R5 is stored into the last three bytes of the ECB.

POST R5!+4,21,20D
The ECB is located at R5!+4. The value X'6100020D' is POST'd into that ECB. (The post bit, X'40', is OR'd with the X'21' to yield a hi-byte of X'61'.)

POST 1DC004~ASID(JES2)
The ECB located at X'001DC004' in JES2's address space is POST'd. The ECB's contents are set to X'40000000'.

SET ASID MYJOB
POST <an ecb located somewhere in common storage>
- Suppose that you have a program running in an aspace named MYJOB.
- Suppose that program has issued a WAIT against "an ECB located somewhere in common storage".
- Suppose now that you are running Z/XDC in a different aspace, and you want to use its **POST** command to wake up your program.

Then you must first use the **SET ASID** command to switch to the MYJOB aspace before issuing the **POST** command.

Help COmmands PRInt

The **PRINT** command can be used to print certain information (see below). The information selected to be printed is written to an output file whose ddname is XDCPRINT.

Z/XDC contains support for automatically recording a log of the debugging session into an output file written to spool. That logging facility is completely unrelated to the PRINT described here. In particular, please note the following:

- The PRINT command writes its information to a dataset that is different from the LOG dataset.

- The PRINT command is part of Z/XDC's base component and, therefore, is always available. The session logging support, on the other hand, is a part of Z/XDC's Fullscreen Support; therefore, it is available only when that support is turned on.

- Most information that you might want to print can be displayed by issuing various Z/XDC commands, and all such displays are automatically written to the LOG dataset; therefore, a separate PRINT command is not necessary for such information. The PRINT command is reserved for certain information that would not be appropriate to LOG (see below).

You may preallocate, allocate, deallocate, and reallocate the XDCPRINT file at any time you wish during a debugging session. This can be done either with Z/XDC's "SET PRINT" command or TSO's "ALLOCATE" and "FREE" commands issued either prior to the debugging session or during the debugging session via Z/XDC's "TSO" command (assuming, of course, that Z/XDC is running within TSO). Which method you would use to allocated XDCPRINT depends upon the specifics of what you would want to accomplish. For more information, please see the following:

```
HELP COMMANDS SET PRINT
HELP COMMANDS TSO
```

Whenever you issue a PRINT command, Z/XDC checks to see if an XDCPRINT file is already allocated. If not, then Z/XDC automatically allocates XDCPRINT to a sysout file on JES2 or JES3 spool.

XDCPRINT can be preallocated to any output device or dataset supported by QSAM; however, if it is not preallocated, then Z/XDC will dynamically allocate it only to a sysout class on spool. XDCPRINT can be preallocated either via an //XDCPRINT DD card or TSO's ALLOCATE command (assuming that Z/XDC is running within TSO). ALLOCATE can be issued either prior to the start of the debugging session, or it can be issued during the debugging session via Z/XDC's "TSO" command.

The maximum number of characters that Z/XDC writes to the XDCPRINT file per output record is 80 (including an ASA carriage control character). XDCPRINT supports any reasonable DCB attributes for the XDCPRINT file:

- The record format (RECFM) may be fixed, variable, or undefined. The records may be blocked or unblocked. Z/XDC will always use ASA carriage control.

- Any legal LRECL and BLKSIZE value may be set, but if a preset LRECL is not large enough to permit 80 characters per record, then truncation will occur.

When Z/XDC opens XDCPRINT, it may discover that one, some, or all of the file's DCB attributes are not set. In this case, Z/XDC will set default values that (a) are

consistent with those values that are set, and (b) come as close as is reasonable to Z/XDC's preferred defaults: RECFM=VBA, LRECL=84, and BLKSIZE=6233.

The XDCPRINT file is not automatically closed when a PRINT command completes. Thus, subsequent PRINT commands will cause information to be appended to what's already in the file. The XDCPRINT file is not closed until one of the following commands are issued:

- SET PRINT CLOSE
- SET PRINT DELETE
- END
- GO
- TRACE

If the PRINT commands being issued generate index information (e.g. PRINT HELP), then multiple PRINT commands will result in a composite index being generated. This is because the index is not sorted and printed until the XDCPRINT file is closed.

Once XDCPRINT is closed, subsequent PRINT commands may overwrite old data:

- If XDCPRINT was preallocated to a tape or disk data set (i.e. a dataset NOT on spool), then PRINT commands issued subsequent to XDCPRINT being closed will reopen the dataset and overwrite the old data.

- On the other hand, if the XDCPRINT file was allocated to spool, then PRINT commands issued subsequent to XDCPRINT being closed will open a new spool file, thus old data will not be disturbed.

The following related commands can be used to control or display certain characteristics of the XDCPRINT file:

SET PRINT ...
This command can be used to set the SYSOUT class that Z/XDC uses if and when it allocates the XDCPRINT file to spool. It also can be used to set the lines per printed page that Z/XDC sends to XDCPRINT, and it can be used to close or delete the XDCPRINT file. For more information, see HELP COMMANDS SET PRINT.

LIST PRINT
This command displays a report describing the characteristics and status of the XDCPRINT file.

PROFILE
The various Z/XDC settings that affect XDCPRINT are saved into the session profile so that different users can have different default settings. These settings can be displayed and changed both by the LIST and SET commands described above and by Z/XDC's Profile Menuing System. See HELP PROFILES MENU for more information.

General Syntax:

PRINT type,operands

type
This selects the type of information to be printed. "Type" may be any of the following:

HELP - Prints either excerpts of or the entirety of Z/XDC's Built-in Help data base. The printout is both paginated and indexed.

Specific information can be selected directly. Type an **H** at the left above to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

Help COmmands PRInt Help

The **PRINT HELP** command can be used to print either excerpts of or the entirety of Z/XDC's Built-in Help data base.

The command can also be used to print your own personal copies of the several Z/XDC user manuals (although it would be a better idea to simply download the PDFs from our website at **colesoft.com/pdfs.)**

The printout generated by the **PRINT HELP** command is both paginated and indexed, which is one of the primary advantages that the printed manuals have over the displays produced by the **HELP** command.

Also, you can doodle and take notes in the printed manuals. (Doodling on glass usually doesn't work very well.) For more information, see HELP ABOUTHELP PRINTING.

The Overall Structure of the Built-in Help

The Built-in Help database has a pyramidal or hierarchical structure. (See HELP ABOUTHELP for a more detailed description.) With the **PRINT HELP** command, you can select any topic within the database for printing. You also can control whether or not any descendant topics of the selected topic are printed, and you can control to what depth the descendant topics are printed. (The default is to print **all** descendant topics.)

The **PRINT HELP** command accepts any number of operands, but they do **not** each designate a topic to print. Instead:

- Only the **last** operand (or 2nd to last operand) designates the name of a topic at which printing is to **start.**
- All the operands leading up to that last operand designate the **path** to that starting topic.
- An additional operand can be given that is a decimal number providing the **depth** to which topics will be selected for printing. For example, a depth of **2** would cause the selected topic to be printed plus as all 1st level **descendant** topics.

The following is a simplified diagram of a pyramidal structure that is similar to the Built-in Help database. The **PRINT HELP** commands following the diagram reference this structure.

```
        HELP        <-- Master topic
        /|\
       / | \
      B  E  F       <-- Topics
```

```
      /|    / \
     / |   /   \
    C  D  G     K    <-- Subtopics
        /|\
       / | \
      H  I  J       <-- Etc.
```

PRINT HELP F G J
This command would print only topic **J.** This is because the operands **HELP** and **F** and **G** only designate the **path** to **J;** they do **not** specify topics to be printed.

PRINT HELP F G
This command would print topics G H I and J, in that order. The operands **HELP** and **F** provide the **path** to **G.** That topic and all its descendant topics (H, I and J) are printed.

PRINT HELP F
This command would print topics F G H I J and K.

PRINT HELP F 2
This command would print only topics F G and K because a "depth limitation operand" of 2 was given.

PRINT HELP F G 1
This command would print only topic G. topics H I and J would not be printed.

Syntax:

PRINT HELP name1 name2 ... depth
-or- -or-
rref1 rref2 ...

PRINT HELP USERGUIDE depth
UGUIDE

Notes:

- The syntax of the **PRINT HELP** command is similar to that of the **HELP** and **LIST HELP** commands:
  - Multiple topic referencing operands may be given.
  - These topic references may be either relative or absolute.
  - Together, they select a single topic by naming the path to that topic from

the top of the Built-in Help database's hierarchical structure. See HELP COMMANDS HELP for more comprehensive details.

- Like the LIST HELP command, the **PRINT HELP** command also accepts a final **depth limitation** operand to control how much information is printed. See below for more details.

Operands:

names
These are names of HELP topics. They, together with relative references, help define a path into the Built-in Help database. Each name given must be the name of a topic that resides directly below (in the Built-in Help hierarchy) the topic identified by the command's preceding operand.

rrefs
These are any of the following relative references.

***Up** - This refers to the topic that is next higher than the topic indicated by the preceding operand.

***Down** - This refers to one of the next lower topics.

***Forward** - This refers to the next following topic (if any) that is at the same level.

***Back** - This refers to the previous topic that is at the same level.

***Next** - This refers to the sequentially next topic in the depth-first sequential ordering through the HELP's tree structure. It may be at the same level or a different level.

***Previous** - This refers to the sequentially previous topic on the depth-first sequential path. It may be at the same or a different level.

*
***Repeat** - These refer to the same topic as the current topic. * and ***Repeat** are synonyms of each other.

The topic references (both **names** and **rrefs)** can be intermixed. Altogether, they define a **starting point** for the PRINT HELP command. The topics located at and below that starting point are written to the **//xxxPRINT** file for printing.

- Please see HELP DDNAMES PRINT for more information about about **//xxxPRINT** allocations.

- Please see HELP COMMANDS HELP for more complete information about Built-in Help topic references.

USERGUIDE
UGUIDE
These keywords are aliases of each other. They are not the names of any actual topic in the Built-in Help database. Instead, they are recognized by the PRINT HELP command as being special, and they cause the command to use a special exclusion table so as to create a User's Guide that contains only the topics that it should contain.

depth

When given, this must be a decimal number equal to or greater than 1. Once PRINT HELP has selected a starting point, this operand controls to what further depth the structure of Built-in Help topics is printed. If omitted, a default depth of 999 is used. This default guarantees that all topics located below the starting topic will be printed since the database's maximum depth currently is only 7 or so.

The **PRINT HELP** command writes the selected topics to the output file identified by the ddname **//xxxPRINT.**

- If //xxxPRINT was not already allocated, then PRINT HELP allocates it to a sysout file on JES2 or JES3 spool.
- If //xxxPRINT was not already opened, then PRINT HELP also opens it.

The //xxxPRINT file is **not** automatically **closed** when a PRINT HELP command completes. Thus, subsequent PRINT commands will cause information to be **appended** to what's already in the file.

The **index** information that is generated by the PRINT HELP command is not written out to //xxxPRINT until the file is closed. Therefore, when multiple PRINT HELP commands are issued, the index information generated by those commands **is accumulated.** When the //xxxPRINT file finally is closed, the index information is then sorted and written to //xxxPRINT prior to its being closed.

The //xxxPRINT file is closed only when one of the following commands is issued:

- SET PRINT CLOSE
- SET PRINT DELETE

- END

- GO

- TRACE

For more information about //xxxPRINT, its allocation, opening, closing, and deallocation, and about its DCB attributes, see HELP COMMANDS PRINT.

Examples:

The following specific command sequences can be used to print your own private copies of the various Z/XDC manuals:

PRINT HELP USERGUIDE

SET PRINT CLOSE

These commands cause a copy of Z/XDC's **User Guide** to be indexed and printed. The **SET PRINT CLOSE** command causes an index to be generated and the file to be closed. If //xxxPRINT is allocated to spool, then subsequent PRINT commands will generate a new output file; otherwise (i.e. if //xxxPRINT is allocated to DASD), subsequent commands will overwrite the User Guide information.

Note, **USERGUIDE** is a special keyword that is recognized by the **PRINT HELP** command and is given special handling so that the right topics are selected and printed.

PRINT HELP COMMANDS
PRINT HELP SHORTCUTCMDS

PRINT HELP SCRIPTS
SET PRINT CLOSE
These commands cause a copy of Z/XDC's **Commands** manual to be printed. The **SET PRINT CLOSE** command causes a unified index to be generated for the output.

PRINT HELP MESSAGES
SET PRINT CLOSE
These commands cause a copy of Z/XDC's **Messages** manual to be printed.

PRINT HELP MAINTENANCE
SET PRINT CLOSE
These commands cause a copy of Z/XDC's **Maintenance and New Functionality Guide** to be printed.

PRINT HELP ADDRESSING
PRINT HELP BREAKPOINTS
SET PRINT CLOSE
These commands print an indexed document that describes address expressions and breakpoints only.

PRINT HELP DEBUGGING 2
This command prints the topic named **DEBUGGING** and the topics that are first level direct descendants of the DEBUGGING topic (GETTINGSTARTED, AMODE64, etc.). Topics that are 2nd level descendants are skipped. Note, the **2** in this command means that only two levels of topics are to be printed: the **DEBUGGING** topic and its first level descendants.

PRINT HELP DEBUGGING 1
PRINT HELP SECURITY RULES 1
SET PRINT CLOSE
These three commands print just the DEBUGGING topic and the SECURITY RULES topic all by themselves. (In the case of the SECURITY RULES topic, the "1" is unnecessary because that topic has no descendants.)

Help COmmands PROfile

The **PROFILE** command is used to managed Z/XDC's session profiles. Profiles can be loaded, saved, displayed, changed, and reset to factory defaults through use of the PROFILE command. For general information, see HELP PROFILES.

Syntax:

```
PROFILE READ ...
PROFILE SAVE ...
PROFILE RESET ...
PROFILE omitted
```

For more information, select the following topics. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

READ - Reads a named or default session profile from a profile or table library.
SAVE - Writes a named or default session profile to a profile library.
RESET - Resets the currently active session profile to factory defaults.
omitted - Enters the Profile Menuing System for displaying and changing all profiled settings.

Help COmmands PROfile REAd

The **PROFILE READ** command loads a default or named session profile into Z/XDC and activates it.

Z/XDC profiles can be loaded from **any** FB-80 library located anywhere. The default, however, is to load them from locally allocated ISPF profile and table libraries.

Syntax:

```
PROFILE READ profilename  clonename  LIBRARY=dsname
             XDC          TFS                dsname(membername)
                                             ddname
```

Rules:

- The **clonename** operand may not be given unless the **profilename** operand also is given.
- The **membername** operand is mutually exclusive with both the **profilename** and the **clonename** operands.

Operands:

profilename
This operand is optional, if given, then it is the name of the profile to be read. It must be from 1 to 5 characters long. It must consist of alpha, numeric, and/or national ($ # @) characters. The first character cannot be numeric. (However, this last restriction can be circumvented via use of the LIBRARY=dsname(membername) operand. See below.)

If profilename is omitted, then the alias name **XDC** is used and a default session profile is loaded. See HELP PROFILES DEFAULTPROFILE for detailed information.

XDC
This is an alias name that refers to the default session profile. The default profile's true name changes from one release to the next, so using **XDC** is a stable way of referring to the default profile regardless of its true name. (In the current release of Z/XDC, the default profile's true name is **DPV31.)**

clonename
If you need to load a profile that was created by a Z/XDC that had been renamed to something that is different than the name of the Z/XDC that you are currently running, then specify the 3-character name of that other Z/XDC here. See HELP PROFILES PROFNAMES for related information.

If you want to give this **clonename** operand, then you **must** also provide the **profilename** operand.

TFS
If you need to explicitly load a profile from a table library, then specify **TFS** as the clonename. See HELP PROFILES TLIBPROFILES for related information.

LIBRARY=dsname
If you need to load a profile from a library other than the standard ISPF allocations, you can use this operand to name the library that you wish to use. **Dsname** can be **any** FB-80 partitioned dataset.

See the **PROFILE READ LIBRARY=** examples (below) for a discussion of accessing session profiles when debugging programs **running in the batch.**

LIBRARY=dsname(membername)
Normally, Z/XDC profiles can be loaded only from library members whose names follow the syntax of **xxxpname,** where:

- **xxx** is a Z/XDC's clone's name,
- And **pname** is a 1 to 5 character name that must not start with a digit.

However, by explicitly providing the member name, you can load a Z/XDC profile from any member **regardless** of its name (assuming of course that the contents of the member are, in fact, a Z/XDC created session profile). In other words, pretty much **all** of the rules listed above for profile names and clone names can be broken.

When a **membername** is given, neither **profilename** nor **clonename** may also be given.

LIBRARY=ddname
If you happen to have a profile library preallocated to some arbitrary ddname, you can use this operand to cause the **PROFILE READ** command to load the profile from this allocation instead of from the default allocations (//xxxPROF, //ISPPROF and //ISPTLIB).

Notes:

- The **LIST PROFILES CURRENT** command can be used to display the true and complete name of the currently active session profile.

- It is quite possible that the Systems Programmer who installed Z/XDC at your Data Center declined to create a default profile for Z/XDC. When that is the case, a **PROFILE READ XDC** command:
  - Will issue message **DBC716W** to warn you that your personal profile does not exist.
  - Will internally issue a **PROFILE RESET** command to cause Factory Default settings to be loaded instead. See HELP COMMANDS PROFILE RESET for more information.

- Eventually, you may decide that some settings are not to your liking, and so you may want to create your own personal default profile that gets loaded automatically instead of your Data Center's default and instead of our Factory Defaults. Here's how to do that:
  - First, get to a known starting point by issuing either of the following:
    - **PROFILE READ XDC** to load either your Data Center's default profile or a prior personal default profile that you've created previously.
    - Or **PROFILE RESET [name]** to load one of our Factory Default profiles.
    - Or **PROFILE READ [various operands]** to load any other profile you'd like to start from.
  - Then use either:
    - The **Profile Menuing System**
    - Or individual **LIST** and **SET** commands

    to tweak those settings you'd like to change.
  - Finally, use a **PROFILE SAVE XDC** command to save your own personal default profile.

  If you like, you can then use a **LIST PROFILE** command to see how your profile got named and where it was saved.

Examples:

PROF READ
PROF READ XDC
These two commands do exactly the same thing:

- The name **XDC** is an alias name that represents a chain of "default" profile names.

- The default profile's true name and internal format change with each new release of Z/XDC.

- The **XDC** profile name exists to provide the user with a stable way of referencing a default profile without regard to what its true name currently is.

- So when **XDC** is given (or implied), Z/XDC will search all profile library allocations for the newest default profile that it can find. It does this by:
  - Running down a historical list of prior default profile true names. (See HELP PROFILES DEFAULTPROFILE.)
  - For each name, it searches all profile library allocations. (See HELP PROFILES DDNAMES.)
  - If the true name cannot be found in any profile library allocation, it will move on to the next older true name and try again.

- When Z/XDC successfully finds a default profile to load, it will convert it to current format so that it can be used. (It does **not** write the converted profile back to disk unless explicitly requested to by a **PROFILE SAVE** command.)

If you issue a **PROFILE SAVE XDC** command, your profile will be written to the default profile's current true name **(xxxDPV31** for **Z/XDC v3.1).**

PROF READ XDC TFS
If your SysProg has created a System Wide default profile, then this command explicitly loads it.

Normally, if you do not have a personal default profile, then a simple **PROF READ XDC** will do the trick. But if you **do** already have a personal default profile, then that would get in front of the System Wide default profile, so you would have to use this command if you wanted to coerce Z/XDC into loading the System default instead of your personal default.

In order to create a System Wide default profile, the SysProg has done more or less the following:

- Starting from a Factory Default profile (probably), he has tweaked it to meet your Data Center's local standards.

- Then he has saved it in a **Table Library** (//ISPTLIB for example), replacing the clone name with **TFS.**

- The command the SysProg would use would look more or less like this: **PROF SAVE XDC TFS LIBRARY=whatever.ISPTLIB**

PROF READ WORK1;LIST PROFILES CURRENT
Z/XDC loads a profile named **WORK1.** It then displays some descriptive information (including its full name) about the profile.

PROF READ WORK1 V31
Z/XDC loads a profile named **WORK1** that was created by a clone of Z/XDC whose name had been changed from **XDC** to **V31.**

PROF READ LIBRARY=DBCOLE.ISPF.ISPPROF
Normally, session profiles are loaded from libraries allocated to ddnames //xxxPROF, //ISPPROF and //ISPTLIB, but one not so obvious thing about this occurs when debugging programs running in the batch:

- The xxxPROF, ISPPROF and ISPTLIB allocations will be used only when they are allocated to the batch job!

- Even though you might be communicating to the batch job from a TSO session, the allocations in TSO will not be used!

Unfortunately, customers rarely think to add the necessary DD cards ahead of time, and in fact there are plenty of situations where it's simply not feasible to do so at all!

The **LIBRARY=dsname** operand provides a workaround for this problem. It allows you to load profiles from any FB-80 library that contains Z/XDC created session profiles.

PROF SAVE LIBRARY=MY.PROFILES.LIBRARY(@0000000)
[...]
PROF READ LIBRARY=MY.PROFILES.LIBRARY(@0000000)
PROF READ 00000 @00 LIBRARY=MY.PROFILES.LIBRARY
The **PROFILE SAVE** command causes a profile to be saved with a name that violates the normal rules for profile names and clone names.

The first **PROFILE READ** command causes that session profile to be reloaded from the **@0000000** member of the **MY.PROFILES.LIBRARY** Library.

The second **PROFILE READ** command would have the same result **except** that Z/XDC does not allow profile names (00000 in this case) to start with digits. So an error message occurs instead.

Help COmmands PROfile Save

The **PROFILE SAVE** command saves the currently active session profile (and its description) into your personal profile library.

Syntax:

```
PROFILE SAVE profilename  clonename  LIBRARY=dsname
             XDC          TFS               dsname(membername)
             <----- omitted ------>         ddname
```

Rules:

- The **clonename** operand may not be given unless the **profilename** operand also is given.

- The **LIBRARY=dsname(membername)** operand is mutually exclusive with both the **profilename** and the **clonename** operands.

Operands:

profilename
This operand is optional, if given, then the name of the currently active profile is changed, and the profile is saved under the new name. The name must be from 1 to 5 characters long. It must consist of alpha, numeric, and/or national ($ # @) characters. The first character cannot be numeric.

XDC
This is an alias name that refers to the default session profile. The default profile's true name changes from one release to the next, so using **XDC** is a stable way of referring to the default profile regardless of its true name. (In the current release of Z/XDC, the default profile's true name is **DPV31.)**

omitted
If **profilename** is omitted, then the active profile's current name is used.

If **profilename** is omitted, then **clonename** also must be omitted.

clonename

If this operand is omitted, then the profile library member name into which the profile is saved includes Z/XDC's current clone name (usually **XDC)** as the member name's first three characters. If you want the member name to start with a different three characters, then specify them here.

TFS
If you eventually want to move the saved profile from your profile library to a table library (for use as a System wide default), then specify **TFS** as the clonename.

See the examples below for an easy way to create System wide profiles using both **TFS** and the **LIBRARY=** operand.

For related information, see HELP PROFILES PROFNAMES.

LIBRARY=dsname
If you want to save a profile into a library other than the standard ISPPROF allocation, you can use this operand to name the library that you wish to use. **Dsname** can be any FB-80 partitioned dataset.

See the **PROFILE SAVE LIBRARY=** examples (below) for a discussion of saving session profiles when debugging programs running in the batch.

Also see below for an example of saving a session profile into a TLIB (an ISPF Table Library) for use as a System wide default.

LIBRARY=dsname(membername)
Normally, Z/XDC profiles can be saved only into library members whose names follow the syntax of **xxxpname,** where:

- **xxx** is a Z/XDC's clone's name,
- And **pname** is a 1 to 5 character name that must not start with a digit.

However, by explicitly providing the member name, you can save a Z/XDC profile into any member regardless of its name.

When a **membername** is given, neither **profilename** nor **clonename** may also be given.

LIBRARY=ddname
If you happen to have a profile library preallocated to some arbitrary ddname, you can use this operand to cause the **PROFILE SAVE** command to save the profile into this allocation instead of into the xxxPROF/ISPPROF allocations.

Examples:

PROF SAVE
Z/XDC saves the currently active profile without changing its name.

PROF SAVE WORK1

Z/XDC changes the name of the currently active session profile to **WORK1** and saves it into your profile library under that name coupled with Z/XDC's current clone name (i.e. in member name **xxxWORK1** where **xxx** usually would be **XDC).**

PROF SAVE XDC

Z/XDC changes the name of the currently active session profile to the alias name **XDC** and saves it into your profile library as your personal default profile. The name used will be the current release's true name for default profiles coupled with Z/XDC's current clone name (usually **XDC).** So in this case, the member name created will be **xxxDPV31** where **xxx** will probably be **XDC.** See HELP PROFILES DEFAULTPROFILE for related information.

PROF SAVE TEMP1 LIBRARY=DBCOLE.ISPF.ISPPROF

Normally, session profiles are saved into the library allocated to ddname xxxPROF or ISPPROF, but one not so obvious thing about this occurs when debugging programs running in the batch: The xxxPROF or ISPPROF allocation will be used only when it is allocated to the batch job! Even though you might be communicating to the batch job from a TSO session, the allocations in TSO will not be used!

Unfortunately, customers rarely think to add the necessary DD cards ahead of time, and in fact there are plenty of situations where it's simply not feasible to do so at all!

The **LIBRARY=dsname** operand provides a workaround for this problem. It allows you to save profiles into any FB-80 library you choose.

PROF SAVE XDC TFS LIBRARY=CSW.TLIB
PROF SAVE LIBRARY=CSW.TLIB(TFSDPV31)
Assuming your Data Center's standard ISPF Table Library is named **CSW.TLIB,** either of these commands can be used to create a System wide default profile that will be loaded for any user that has not created his own personal profile. (See HELP PROFILES TLIBPROFILES for related information.)

PROF SAVE BIGPR TFS LIBRARY=CSW.TLIB
PROF SAVE LIBRARY=CSW.TLIB(TFSBIGPR)
If you have a development project you're calling **Big Project,** you might want to create a default profile that's specific to the project and that is easily available to both yourself and your colleagues. Both of these commands create such a profile. It is named **BIGPR,** and it is saved into a Table Library named **CSW.TLIB** where it will be available to all developers for whom **CSW.TLIB** is included in their **ISPTLIB** allocations.

Help COmmands PROfile RESet

The **PROFILE RESET** command loads one of Z/XDC's **Factory Default** profiles and names it **XDC.**

Z/XDC's Factory Default profiles are hard-coded within Z/XDC itself. Currently, there are six of them. They each are suitable for particular combinations of program language and terminal display width according to the following chart: (Note that while 'Dumps' is not a program language it is used in this table as if it was - it applies when running under IPCS).

PROFILE NAME	PROGRAM LANGUAGE	DISPLAY WIDTH (COLUMNS)
A80	Assembler	80-99
AWIDE	Assembler	100+
C80	C	80-99
CWIDE	C	100+
D80	Dumps	80-99
DWIDE	Dumps	100+

In addition, **PROFILE RESET** supports certain nicknames that let the command dynamically select the profile to load based upon what Z/XDC sees in the current environment: (Note that while 'Dumps' is not a program language it is used in this table as if it was - it applies when running under IPCS).

NICKNAME	PROGRAM LANGUAGE	DISPLAY WIDTH (COLUMNS)
ASM	Assembler	A80 or AWIDE
C	C	C80 or CWIDE

CEE	C	C80 or CWIDE
D	Dumps	D80 or DWIDE
DUMP	Dumps	D80 or DWIDE
XDC	A80 or C80 or AWIDE or CWIDE or D80 or DWIDE	
[omitted]	A80 or C80 or AWIDE or CWIDE or D80 or DWIDE	

If, after loading a Factory Default profile, you then issue a **PROFILE SAVE** command (without operands):
- The profile will be saved under the name **XDC.**
- That profile will then be your personal default profile.

For more information, see HELP COMMANDS PROFILE SAVE.

Syntax:

```
PROFILE RESET name
             omitted
```

name
This must be either:
- The name of a predefined Factory Default profile, as listed above,
- Or a nickname (also as listed above) that allows the command to choose a profile based on current environmental conditions. The conditions checked are:
  - For language, Z/XDC sees if it is running under IPCS, and, if so, will use the dumps-related 'language'. If not running under IPCS, Z/XDC will check to see whether or not c/XDC debugging permission has been granted. (Use the **LIST XDC** command to see the current state of license grants.)
    - If under IPCS, then D80 or DWIDE is loaded.
    - If use of c/XDC has been granted, then C80 or CWIDE is loaded.
    - If use of c/XDC has not yet been granted, then A80 or AWIDE is loaded.
  - For display width, Z/XDC checks your terminal's current display width:
    - If it's less than 100, then A80, C80 or D80 is loaded.
    - If it's 100 or more, then AWIDE, CWIDE or DWIDE is loaded.

omitted
If no operand is given, then the command is processed as if **XDC** had been given: Z/XDC will load whichever profile best fits the current environment (as described above).

For more information, see HELP PROFILES FACTORYDEFAULTS.

Examples:

PROF RESET
PROF RESET XDC
Both of these commands allow Z/XDC complete freedom to decide which of the six Factory Default profiles to load. Whichever one is picked, once it is loaded, it will be named **XDC.**

PROF RESET C
Z/XDC loads either the **C80** or **CWIDE** Factory Default profile according to whether your terminal's width is narrow or wide. Both of these profiles are suitable for C program debugging. Whichever one is picked, once it is loaded, it will be named **XDC.**

PROF RESET D
Z/XDC loads either the **D80** or **DWIDE** Factory Default profile according to whether your terminal's width is narrow or wide. Both of these profiles are suitable for examining dumps. Whichever one is picked, once it is loaded, it will be named **XDC.**

PROF RESET AWIDE
Z/XDC loads the **AWIDE** Factory Default profile. This profile is suitable for Assembler debugging on wide terminals. Note, once it is loaded, it will be named **XDC.**

Help COmmands PROfile Omitted

The **PROFILE** command, given without operands, enters you into Z/XDC's "Profile Menuing System" where you can display and change all profiled settings. For more information, see HELP PROFILES MENU.

Syntax:

PROFILE

Help COmmands Quick$init

The **QUICK$INIT** command is an internally issued command that normally is invisible to users. Its purpose is to force certain internal initializations to occur earlier in Z/XDC's startup processing than they otherwise would.

When Z/XDC first receives control, normally much of Z/XDC's initialization does not occur until just prior to sending its first display screen to the user's terminal. However, when a startup script is processed, that initial display can be delayed for a very long time!

For information about startup scripts, see:

- **HELP DDNAMES CMDS**
- **HELP DEBUGGING ISPF PANEL**

This delay can cause certain failures when the delay results in the first display occurring when Z/XDC is running in a constrained environment (such as SRB debugging) where such things as opening an ACB are disallowed.

To address this problem, when the **READ** command processes a startup script, it issues a **QUICK$INIT** command before processing any commands from the script. This causes all environmentally sensitive initializations to occur in a friendly execution environment (i.e. before commands from the script are processed that might cause a delay in first display construction).

Normally, we would not bother documenting hidden, internal commands, but in this case, if **SET READ ECHO** is in effect, the internally issued **QUICK$INIT** command will be displayed. So you might want to know what it is. Well, this is what it is.

SYNTAX:

Undocumented

When issued by a user, the command has no visible or deleterious effect.

Help COmmands REAd

The **READ** command causes Z/XDC to read subsequent commands from a sequential file.
- The file may be either a sequential dataset or a specified member of a partitioned dataset.
- The file may have any reasonable record format, record length, and block size.
- It may or may not be line numbered. If it is line numbered, then
  - For fixed length records (RECFM=F), the line numbers must appear in the last 8 bytes of those records that are numbered;
  - Otherwise (for RECFM=V or RECFM=U), they must appear in the first 8 data bytes.

The dataset is read until one of the following is encountered.
- An error occurs while processing one of the commands read from the script (unless **ERROR=CONTINUE** is in effect).
- A **READ SUSPEND** command is encountered in the script.
- A **READ CLOSE** or **READ dsname** command is encountered in the script.
- The end of the script file is reached.
- A **GO/GOT/GOX** command is encountered in the script.
- A **TRACE** command (but not TRACE W command) is encountered in the script.

If reading stops for either of the first two reasons (command processing error or READ SUSPEND command), then it is **only suspended.** The file remains opened and positioned.

A subsequent **READ** to a suspended file will resume with the next record following the one that caused the suspension. (Note, even if the failing record contained unprocessed commands, the balance of the record is skipped, and the unprocessed

commands remain unprocessed.)

For useful information pertaining scripts, their uses, their limitations and their workarounds, see **HELP SCRIPTS.**

ColeSoft provides a number of expertly written command scripts, which can be found in the library named **hlq.XDCV31.XDCCMDS.** (You will have to ask your Systems Programmer for its actual name at your Data Center.) Many good examples of interesting techniques can be found there. For more information, see **HELP SCRIPTS OURSCRIPTS.**

I recommend that you browse these scripts and lift from them whatever you think might be useful.

Syntax:

```
READ [DSNAME=]dsname [MUST contain a period] SEQUENCEFIELD=PRESENT
     [DSNAME=]dsname(member)                  SEQFIELD=     ABSENT
     [DSNAME=](member)                        SEQF=         DETECT
     [DSNAME=]member

     SUSPEND
     RESUME
     CLOSE
     omitted
```

Rules:

- The various variations of the **[dsname][(member)]** operand can be given either as the operand of the **DSNAME=** keyword or as a **positional parameter.** I.e. they may be given either with or without the **DSNAME=** keyword.

- **SUSPEND, RESUME, CLOSE** and **omitted** are mutually exclusive both with each other and with all other operands. If given, they must be given alone.

- The **SEQF=** operand (and its variations), if given, must be given only in combination with the various variations of the **[dsname][(member)]** operand. It may not be given stand-alone, and it may not be given in combination with any other operand.

Operands:

dsname [for sequential datasets]
DSNAME=dsname [for sequential datasets]
dsname(member) [for partitioned dataset members]
DSNAME=dsname(member) [for partitioned dataset members]
This gives the filename of the script to be read. It must meet the following requirements:

- The dataset name must consist of **two or more qualifiers.** Simple single qualifier names are not permitted. (They will be mistaken for member names. See below.)

- The **DSNAME=** part of this operand is optional. Its presence or absence does not matter.

- The given dataset name must be **fully qualified.** (Z/XDC does **not** follow the TSO convention of prefixing unquoted dsnames with your userid. You have to do that yourself.)

- The dataset name must be **unquoted.** (Again, Z/XDC does not follow TSO conventions for dsnames.)

- The dataset name must conform to **classic JCL Syntax rules.** (Z/XDC does not yet have HFS or ZFS file system support.)

- The dataset name may contain **variable symbols** for which Z/XDC will make substitutions. This allows the dsname to be reactive to the current time, date, userid, jobname, etc. (These variables are most useful when the **READ** command is executed from another command script or from a DEAD trap.) For more information, see:

    HELP COMMANDS SYNTAX DSNAMES
    HELP BREAKPOINTS DEADTRAPS

- The dataset must exist and be **cataloged.** (Z/XDC generally does not have support for accessing uncataloged datasets.)

- The dataset may be either sequential (DSORG=PS) or a partitioned dataset (PDS or PDSE) member.

- If the dataset is partitioned, then a **member name** must be given.

- The dataset may have any reasonable **DCB attributes:** RECFM, LRECL and BLKSIZE. (But keep in mind, while command strings can be of any length up to 32767 bytes, individual commands may not be longer than 255 characters.)

- Records within a script file must be received by Z/XDC as **EBCDIC** text. (ASCII text would have to be translated by the file access method ["driver" to all you PC heads out there] prior to being received by Z/XDC.)

- If script records are **sequence numbered,** then the sequence numbers must be positioned as follows:

    RECFM=F - The **last 8** bytes of each record.
    RECFM=V - The **first 8** bytes of each record.
    RECFM=U - The **first 8** bytes of each record.

  Also, see **SEQF=** (below) for related information.

If a previously read file was **suspended** and remains open, then that prior file is closed before this file is opened. (To resume processing a suspended script, issue a **READ RESUME** command instead. See below for more info.)

If the **dsname(member)** form of this operand is given, then the dataset is a **library,** so the libraryname is saved as the default Scripts Library going forward. Subsequent scripts can be read from the same library without having to respecify the full library name. (The **SET READ DSN=** command can also be used to set a default

Scripts Library name.)

member
(member)
DSNAME=member
DSNAME=(member)
If a default Scripts Library has previously been established, then you can run any script in the library just by providing a member name (either with or without enclosing parentheses, it doesn't matter). Please note the following:

- The **DSNAME=** part of this operand is optional. Its presence or absence does not matter.

- There are four ways to establish or change the default Scripts Library:
  - By issuing a **READ dsname(member)** command (see above).
  - By issuing a **SET READ DSN=dsname** command.
  - By using the **Profile Menuing System.**
  - By issuing a **PROFILE READ name** command that loads a new Debugging Session Profile in which a different default Scripts Library name was saved.

- The **READ** command does not support reading scripts from datasets having **simple** names. Only datasets having **compound** names can be used. In other words, only names containing **one or more periods** can be used. (This is because simple dataset names are interpreted to be library member names, not dataset names.)

SEQUENCEFIELD= PRESENT
SEQFIELD= ABSENT
SEQF= DETECT
When sequence fields are present in a script, they usually contain 8 decimal digits, but nowadays with the advent of **git** and the decline of the importance of sequence numbers in modern Software Change Management methodologies, they can easily contain non-numeric and even arbitrary characters that **the coder might still want ignored.**

The **SEQFIELD=** operand (and its variations) allows you to control whether or not Z/XDC will attempt to automatically detect the presence or absence of sequence fields in the script to be read. (This choice overrides the default setting established by the **SET READ** command.) Here are the details:

SEQUENCEFIELD=PRESENT
SEQFIELD=PRESENT
SEQF=PRESENT
- The sequence field is considered to be present in all records regardless of content. The field will be ignored in all records.

SEQUENCEFIELD=ABSENT
SEQFIELD=ABSENT
SEQF=ABSENT
- The sequence field is considered to be absent from all records. Whatever content is found in the sequence field position will be considered to be part of the data found in the rest of the record.

If what is found is, in fact, a sequence number, then a command parsing error

will likely be the result (unless you get unlucky).

SEQUENCEFIELD=DETECT
SEQFIELD=DETECT
SEQF=DETECT

- Z/XDC itself will determine the presence or absence of sequence fields based upon its examination of the file's **first record.** The determination will be made according to the rules discussed in **HELP SCRIPTS RYOSCRIPTS SEQUENCEFIELDS.** (Note, since sequence fields can contain non-numeric data, the detection rules are probably a bit more complicated than you might expect.)

SEQFIELD=DETECT is the factory default.

SUSPEND
If a script file is currently being processed, then it is suspended. (This command is useful only when it appears within the script being processed.)

RESUME or **all operands omitted**
If a script file is currently open and suspended, then it is resumed. Processing continues with the command string occurring on the next following logical record.

Note, if the script was processing a string of multiple commands when it was suspended, and if the script was suspended before completing the processing of all of the commands in that string, then this **READ RESUME** command does **not** cause the unfinished command string to be resumed. Instead, all of the unprocessed commands are skipped, and script execution resumes with the command string occurring on the next following logical record.

CLOSE
If a script file is currently open, then it is closed. Note, the **LIST READ** command shows whether or not a script is open.

Examples:

The following examples are focused on using the **READ** command. For discussions and examples pertaining to the content of scripts, start at **HELP SCRIPTS.**

R ∗P.CMDLIB(CMDS)
R GROUPID.CMDLIB(CMDS)
Both of these commands do the same thing (assuming, of course, that the TSO profile prefix string is currently set to "GROUPID"). The member named CMDS of the partitioned dataset named GROUPID.CMDLIB is opened and Z/XDC commands are read from it and processed. (Note, Z/XDC substitutes the current TSO session's profile prefix string for **∗P.** See **HELP COMMANDS SYNTAX DSNAMES** for more information.)

Suppose that one of the commands from the above file has an error and causes READ processing to be suspended.

R RESUME
R
Both of these commands do the same thing: **READ** processing is resumed with the record following the one that contained the command that had the error.

Note, a single record can contain multiple commands separated by semicolons. **RESUME** causes any unprocessed commands in the same record as the failing command to be skipped.

R CLOSE
(Building upon the preceding examples) this command closes the suspended file without processing any further commands from it.

R CMDS
R (CMDS)
R GROUPID.CMDLIB(CMDS)
All of these commands do the exact same thing. They all (in the context of these examples) close and reopen the **CMDS** script. The script is reprocessed from its beginning.

R MORECMDS
R (MORECMDS)
These commands do the same thing. They both read a member named **MORECMDS** from the default Scripts Library (GROUPID.CMDLIB, in the context of these examples).

SET READ DSN=hlq.XDCV31.XDCCMDS
PROFILE SAVE XDC
READ SDWAMAPS
These commands do the following:

SET READ DSN=hlq.XDCV31.XDCCMDS
This command establishes **hlq.XDCV31.XDCCMDS** as the default Scripts Library. There are two consequences of this:
- Subsequent **READ** commands will, by default, read their scripts from **hlq.XDCV31.XDCCMDS.**
- **hlq.XDCV31.XDCCMDS** is saved in the currently active debugging session's profile data as the default scripts library going forward in the current debugging session. (You will have to use the **PROFILE SAVE** command if you wish to harden this setting.)

(Ask your Systems Programmer what **hlq** is at your shop.)

PROFILE SAVE XDC
This command hardens profile changes by saving the currently active profile data to disk as your personal **default profile.** (It will be automatically reloaded each time you start a new debugging session.)

Absent this command, profile changes would be effective only for the remainder of the current debugging session and would be lost when the current session is ended and a new session is started.)

READ SDWAMAPS
This command reads an SDWA mapping script from hlq.XDCV31.XDCCMDS.

READ hlq.XDCV31.XDCCMDS(SDWAMAPS)
This command does the same thing in a single command that the following two commands do, taken together:

SET READ DSN=hlq.XDCV31.XDCCMDS
READ SDWAMAPS

R MYUID.CMDFILE
This reads a script from a sequential file named MYUID.CMDFILE. Note, because this is a sequential file, the default Scripts Library name, if any, is unaffected.

R FRANK.SCRIPTS(SENARIO1)
This reads a script named SENARIO1 from a library named FRANK,SCRIPTS.

It also changes the default scripts, but only for the current debugging session.

R SENARIO2
This reads the SENARIO2 script from the same library: **FRANK.SCRIPTS**

PROFILE SAVE XDC
This command hardens **FRANK.SCRIPTS** as the default library for **READ** commands.

WARNING! This command also hardens all other profiled settings you may have made. Sometimes that's a good thing, sometimes it's a bad thing. If in your case it's a bad thing, then there are two ways to deal with it:

1) Before you do the **PROFILE SAVE:**

- First, do a **PROFILE READ XDC** to reset all settings to a known state.

- Then use either:
  - The **Profile Menuing System,** and/or
  - Suitable **SET** commands, and/or
  - Suitable **SET WINDOW** commands (if you're changing the Windows layouts), and/or
  - Suitable **KEYS** or **SET KEYS** commands (if you're changing your PF key definitions).

- Finally, issue **PROFILE SAVE XDC** to save only the changed profile settings

that you intend to save.

2) Alternatively, use **PROFILE SAVE name** to save all your changed settings into a non-default, named profile that you can reload at will with a simple **PROFILE READ name** command.

Help COmmands RECall

This command is an alias of the **RETRIEVE** command. See HELP COMMANDS RETRIEVE for more information.

Help COmmands REFresh

The **REFRESH** command causes Z/XDC's fullscreen management routines to fully repaint the display screen. The REFRESH command can be used in command scripts that may cause display disrupting messages to be issued outside of Z/XDC's awareness and control. The REFRESH command repairs a corrupted display by causing Z/XDC to send a complete display image to the workstation. (Normally, Z/XDC tries to keep track of what is being displayed and to send only modifications to the display.) For more information, see HELP FULLSCREEN DISPLAYERRS.

Syntax:

REFRESH

This command accepts no operands.

Generally this command is most useful when it is issued from an automatic source such as from an automatic command associated with a breakpoint or from a commands file being executed by the READ command. This is because when a workstation display is corrupted, it usually is not possible to issue commands from it. (Instead, press **PA2** to accomplish the refresh function manually.)

Please understand... this command only reshows the display. It does not rebuild the display. So if your intent is to show content that was not previously shown, then **REFRESH** is not the command that you want to use.

Instead, it is likely that you will want to issue a **FORMAT +0** command (or perhaps a **WHERE** command. For example, when:

- You have issued a **WHERE** command or **FORMAT** command to display storage,
- And now you issue a breakpointing command **(TRAP, AT** and friends) to set a breakpoint into the code that you are displaying,
- But you did not do so by using a **T** or **A** shortcut command,
- Instead, you issued the **TRAP** or **AT** command from the command line...

When you do this, the breakpoint does not automatically appear in the display.

Instead, you have to issue another command to rebuild the display. Usually, that would be a **WHERE** or **FORMAT +0** command.

Help COmmands RELease

The **RELEASE** command can be used to release a Pause Element Token (PET).

An optional release code can be supplied

Note that if XDC is executing unauthorized, only PETs allocated in the same address space and with an authority level of **IEA_UNAUTHORIZED** can be released.

Syntax:

RELEASE petaddress [releasecode]

Operands:

petaddress The address of the PET. Note that this address can include an ASID() function if the PET to be released is in another address space (see the example below).

releasecode The release code to use. If specified, the value must be a hexadecimal number from 0 to FFFFFF. (A decimal value can be specified by suffixing the decimal value with N. For example, 10 and 16N are the same). If omitted, a release code of 0 is used.

Examples:

RELEASE 12345768
Releases a PET where the PET can be found at address 12345678. A release code of 0 is used.

RELEASE A0000000 100
Releases a PET where the PET can be found at address A0000000. A release code of X’100’ is used.

RELEASE A0000000~ASID(MYJOB1) 100
Releases a PET where the PET can be found at address A0000000 in the address space with name ’MYJOB1’. A release code of X’100’ is used.

Help COmmands RETRIeve

The **RETRIEVE** command causes a prior command string to be retrieved from a retrieval

stack and redisplayed on a window's command line. Then you can modify it or correct it and then re-execute it (by pressing the ENTER key).

Each window has its own stack of previously issued, retrievable command strings.

Usually RETRIEVE is issued via a PF key. **F12** is the factory default.

The **RETRIEVE** command can retrieve command strings in both the **forward** and **backwards** directions! If you press the RETRIEVE PF key **too many times** and move past the command that you want, you can recover by issuing **RETRIEVE +1** to return to the missed command. **shift-F12** is the factory default for this.

There are four different ways to use the **RETRIEVE** command, each with its own, distinct characteristics:

- When used by itself, **RETRIEVE** will retrieve a previously issued command string from the retrieval stack.

- When appended at the end of a string of other commands, the other commands will be processed, and then the command string will be retained on the command line. (Retrieval will not occur from the stack.)

- When issued from within a HELP Topics display, **RETRIEVE** will retrieve fully resolved **HELP** commands that will redisplay prior topics.

- When used with its **LIST** operand, **RETRIEVE LIST** will produce a specialty display of all previously issued command strings. You can then select the command string you want to retrieve and re-execute. You can also edit command strings and purge command strings from the stack.

For more information, see **HELP COMMANDS RETRIEVE USAGE.**

Syntax:

RETRIEVE
RETRIEVE cmd#

RETRIEVE LIST
RETRIEVE LIST window#

Aliases: **RECALL**
RETRY

Notes:

When the **LIST** operand is used:
- The command name (RETRIEVE, RETRY or RECALL) can be abbreviated down to just three characters: **RET** or **REC**
- The LIST operand itself also can be abbreviated down to just one character: **L**

However, when a numeric operand is used (or implied), the command name **cannot** be

abbreviated. Example: **RETRIEVE +1**

Operands:

RETRIEVE [blank]
RECALL [blank]
RETRY [blank]
When **no operands** are given, a prior command string is retrieved to the command line as follows:

- If **other commands precede** the RETRIEVE command in the command string (example: **LIST R0;RETRIEVE),** then after the given commands have been executed, the same command string will be retrieved to the command line so that it may be modified and issued again.

- If the RETRIEVE command is the **only** command in the command string, then the next previously issued command string will be retrieved to the command line so that it may be modified and issued again.

Note, this no-operands form of the RETRIEVE command may **not** be given abbreviated.

RETRIEVE cmd#
(The aliases, RECALL and RETRY, are also permitted.)

cmd# is any of the following:
- A signed positive number
- A signed negative number
- An unsigned number

If signed negative **or unsigned,** cmd# causes the nth **prior** command string to be retrieved to the command line.

If signed positive, cmd# causes the nth **following** command string to be retrieved to the command line.

The command string retrieved is located relative to the most recent command string previously issued or retrieved.

If **cmd#** is signed positive, and if the current retrieval position is at the newest command string in the retrieval stack, then the retrieval process will wrap to the oldest stacked command string and count forward from there.

Note, this form of the RETRIEVE command may **not** be given abbreviated.

For more usage details, see **HELP COMMANDS RETRIEVE USAGE.**

RETRIEVE LIST
RETRIEVE LIST window#
(The aliases, RECALL and RETRY, are also permitted.)

A specialty display is opened up, and an editable list of all recently issued

command strings is displayed from the retrieval stack.

If you provide a **window#** operand, then the commands that are displayed are those that have been issued on the indicated window's command line: The **window#** may range in value from **1** up to the number of window command lines that currently exist. Specifically:

- **RETRIEVE LIST 1** refers to the display's **Primary Window** (headed by the display's top command line).
- **RETRIEVE LIST 2** refers to the display's first **Watch Window** (headed by the display's 2nd command line).
- **RETRIEVE LIST 3** etc.

If you **omit** the **window#,** then the window that currently contains the display **cursor** will be the one whose retrieval stack is displayed.

Note, this **LIST** form of the RETRIEVE command **may** be given abbreviated. Example: **RET L**

For more usage details, see **HELP COMMANDS RETRIEVE LIST.**

Further Information

The following subtopics contain additional information. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

USAGE - RETRIEVE command usage rules and characteristics.
HELPMODE - RETRIEVE processing differences when used within HELP topics.
LIST - Displaying the retrieval stack.
EXAMPLES - RETRIEVE command usage examples.

Also, the following side topics might be of interest:

- **HELP FULLSCREEN RETRIEVE**
- **HELP COMMANDS SET WINDOW RETRIEVE**
- **HELP FULLSCREEN PFKEYS**

Help COmmands RETRIeve Usage

The following rules govern the use and characteristics of the **RETRIEVE** command (and its aliases):

- RETRIEVE commands can be given either from a **PF key** or from a **command line.**
- If a command string **starts** with a RETRIEVE command, then:

  - It may contain **multiple** RETRIEVE commands,
  - But it may **not** contain either non-RETRIEVE commands or RETRIEVE LIST commands,
  - And it will retrieve previously issued command strings.

- If a command string starts with a **non-** RETRIEVE command, then:
  - It may contain at most only **one** RETRIEVE command,
  - And that RETRIEVE command must be the **last** command in the string.
  - And it will allow that command string to be executed, but then retained on the command line for modification and re-execution.
  - It will **not** retrieve previously issued command strings.

- If **multiple** RETRIEVE commands are issued in a row, then successively preceding command strings will be retrieved to the command line.

  Successive RETRIEVE commands can be issued in any of the following ways:

  - Repeatedly typing RETRIEVE and pressing ENTER.

  - Typing several RETRIEVE commands separated by semicolons (;), and then pressing ENTER.

  - If you have a RETRIEVE command defined in a **PF key,** then you can press that PF key several times.

    Z/XDC's factory default PF keys include the following settings:
    - **F12** is set to **RETRIEVE.** It retrieves the next **older** command string.
    - **shift-F12** is set to **RETRIEVE +1.** It retrieves the next **newer** command string (in case you press F12 too many times).

The above rules are not affected by which alias you choose to use (RETRIEVE, RETRY or RECALL). They apply to all of them.

How Deep is the Retrieve Stack?

The maximum number of command strings that can be saved for retrieval is user settable via the **SET WINDOW RETRIEVE** command. This number can be different for each window on your terminal's screen.

However, the default number is **255.** That's also the maximum number, so it is unlikely you will ever need to change it.

Displaying the Retrieve Stack

The **RETRIEVE LIST** command can be used to display the current stack of retrievable command strings. You can then:
- Select **(S)** any string for re-execution.
- Purge **(P)** unwanted command strings from the stack.
- Edit any strings as you see fit.

For more information, see **HELP COMMANDS RETRIEVE LIST.**

RETRIEVE LIST commands may **not** be given in combination with other types of RETRIEVE

commands.

If a **RETRIEVE LIST** command is given in a command string, then it must be the **last** command in that string.

Abbreviations of the RETRIEVE command's name:
- **Are** accepted for the **RETRIEVE LIST** command,
- But are **not** accepted for any **other** forms of the RETRIEVE command.

For usage examples, see **HELP COMMANDS RETRIEVE EXAMPLES.**

Help COmmands RETRIeve Helpmode

Normally, the RETRIEVE command will retrieve to the command line a previously issued command string. But when it is issued during the display of a **HELP topic,** its behavior changes subtly in several ways...

Instead of retrieving a previously issued command string, RETRIEVE retrieves the **fully resolved** HELP command needed for redisplaying that topic. Then pressing ENTER causes that topic to be redisplayed.

When a retrieved topic is redisplayed in this way, it is **automatically scrolled** downwards to the same location in the topic at which it was last viewed.

In order to achieve this automatic scrolling, positioning information has to be maintained in the retrieval stack. Then when the **RETRIEVE LIST** command is used to display the stack, the display will show that auto-scrolling information. See **HELP COMMANDS RETRIEVE LIST** for more information.

Using RETRIEVE with the SCANLOG Command

There is no PF key whose factory default is SCANLOG, so effecting a repeat SCANLOG cannot be done with just a simple key press. Nevertheless, there are a couple of simple ways to do repeat SCANLOGs:

- Issue **SCANLOG whatever;RETRIEVE** - The SCANLOG command will be retained on the command line for as long as you keep pressing ENTER.

- Issue **SCANLOG whatever** then press **F12** once or twice - The SCANLOG command will be returned to the command line for reissuance.

Help COmmands RETRIeve List

Normally, the RETRIEVE command will retrieve to the command line for re-execution a previously issued command string. However, when you use the command's **LIST** operand, the **RETRIEVE LIST** command does something entirely different: It opens up a specialty display which allows you to peruse (and edit) the entire retrieval stack.

The display allows you to do the following:

- You can **edit** any displayed command string simply by overtyping it.

- You can **purge** any unwanted command strings by using the **P** shortcut at the left.

- You can **select** one command string for re-execution by using the **S** shortcut at the left.

Where RETRIEVE LIST Can be Used

You can issue the RETRIEVE LIST command from any of the following windows:

- **The Primary Window:** That window's retrieval stack will be displayed. The stack will contain most previously issued command strings, both those that processed successfully, and those that failed.

- **Any Watch Window:** Ditto.

- **A HELP Topic Display:** Retrieval stacks for the HELP display are a bit different. They do not contain prior command strings. Instead, they contain the **fully resolved** HELP commands needed to redisplay the prior topics.

  Also, when a HELP topic is retrieved for redisplay, it is **automatically scrolled** down to the point at which it was last viewed. (This does not happen for retrieves to normal window command lines.)

  The RETRIEVE LIST display shows the **DOWN** command that will be used for this repositioning when a topic is redisplayed.

Notes:

- The **RETRIEVE LIST** command takes an optional **window#** operand. You can use this to display the retrieval stack for a window different from the command line on which you issued the command. See **HELP COMMANDS RETRIEVE** for more information.

- The retrieval stack can hold up to 255 entries. When full, the oldest entries are stolen and reused.

  You can change the number of permitted entries, but you cannot make it more. You can only make it less, so why bother?

- This **LIST** form of the RETRIEVE command may be given abbreviated. Example: **RET L** (Other forms of the RETRIEVE command may not be abbreviated.)

Closing the Retrieval Stack Display

The retrieval stack display starts when you issue a **RETRIEVE LIST** command. It

remains in control until you use one of the following three commands:

- **END:** This command ends the display without making a selection. But all purges and editing changes that you may have made are hardened.

- **CANCEL:** This command also ends the display without making a selection. But any purges and editing changes that you may have made are discarded.

- **S:** This shortcut both ends the display and selects a command string for re-execution. (Like END, all purges and editing changes are hardened.)

For more information about these and other permitted commands, see **HELP COMMANDS RETRIEVE LIST COMMANDS.**

Further Information

The following subtopics contain additional information. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

EDITING - Editing command strings in and purging command strings from the retrieval stack.
COMMANDS - Commands that are permitted during retrieval stack display.
PFKEYS - pfkey settings during retrieval stack display.

Help COmmands RETRIeve List Editing

When you issue the **RETRIEVE LIST** command, a specialty window is opened up, and an editable list of all recently issued command strings is displayed. Editing is very easy:

- To **change** a saved command string, simply overtype it. You may change as many strings as you like.

- To **purge** a command string from the stack, use the **P** shortcut at the left. You can purge as many strings as you see fit.

Hardening or Discarding Your Changes

All edits and purges remain pending until you close the display, at which time the changes are either hardened or discarded:

- If you use either and **END** command or an **S** shortcut, your changes will be hardened.

- If you use a **CANCEL** command, your changes will be discarded.

See **HELP COMMANDS RETRIEVE LIST COMMANDS** for details about these (and other)

commands.

What's Editable and What's Not

Basically, all saved command strings are editable, but only when a command string is fully displayed on your screen. Editing is disabled for affected strings when they cannot be fully displayed. This can occur under the following circumstances:

- If your display is scrolled rightwards by more than four characters, all displayed command strings will be made non-editable. That's because rightwards scrolls of 5 characters or more cut off the starts of all command strings. Try scrolling left a bit to avoid cutting off the left ends of the command strings.

- If a command string is too long to fit into the display, it is made non-editable. You **may** be able to make it editable if you do one of the following:

  - If the command string is only four or fewer characters too long, then a **RIGHT 4** command will do the trick.

  - Otherwise, if your display terminal is set to show only an 80-character width, you should take steps to switch your terminal to a large geometry display. See **HELP FULLSCREEN TERMINALS GEOMETRIES** for a full discussion.

  - Or if you willing to execute the edited command string:
    - Use the **S** shortcut to retrieve it to the command line.
    - Make your changes.
    - Press ENTER to process the edited string.

    The newly changed string will be pushed back onto the retrieval stack. You can then (if you like) return to the RETRIEVE LIST display and use the **P** shortcut to purge the original command string. (Or not, depending upon how compulsive you are... or are not).

Help COmmands RETRIeve List Commands

When the retrieval stack display is open, most Z/XDC commands and PF keys are disallowed. Attempts to issue them will be blocked by a **DBC560E** message.

The only commands that are permitted are the following:

- **LEFT, RIGHT, UP** and **DOWN:** The scrolling commands do what you'd expect.

- **END:** This command ends the display without making a selection. But all purges and editing changes that you may have made are hardened.

- **CANCEL:** This command also ends the display without making a selection. But any purges and editing changes that you may have made are discarded.

  (Note, CANCEL is not a normal Z/XDC command. It is a specialty command that

exists only during the display of certain specialty panels.)

- **S:** This shortcut can be used to select a command string for re-execution. It does so as follows:
  - The retrieval stack display is closed.
  - (Like END) All purges and editing changes are hardened.
  - The selected command string is retrieved to the command line.
  - And you are given an opportunity to modify it before pressing ENTER.

- **P:** This shortcut can be used to purge one or more command strings from the retrieval stack. The purge remains pending, but the stack is redisplayed with the purged strings omitted:
  - The purges are hardened when an **END** command or **S** shortcut is issued.
  - The purges are discarded when a **CANCEL** command is issued.

For information about changes to the PF keys, see **HELP COMMANDS RETRIEVE LIST PFKEYS.**

Help COmmands RETRIeve List Pfkeys

During RETRIEVE LIST displays, your normal PF key definitions are unavailable. Instead, certain commands are forced into some keys, and all other keys are disabled.

Generally, the PF keys, when within a retrieval stack display, are set to scrolling commands and display closing commands.

Scrolling Commands

The following scrolling commands are forced into the following PF keys:

- **F7** is set to **UP -**
- **F8** is set to **DOWN -**
- **F10** is set to **LEFT -**
- **F11** is set to **RIGHT -**

Note the trailing **-** on each of these commands. This indicates that whatever you type onto the command line is appended to the command before it is executed. Example:

- Type a **4** onto the command line.
- press **F11**

A **RIGHT 4** command will be executed, and the display will be scrolled rightwards by four columns.

Closing Commands

The following closing commands are forced into the following PF keys:

- An **END** command is forced into **F3.** It closes the display and **hardens** all edits and purges that you may have made. It does not, however, retrieve any command

string for re-execution. (If you want to close the display **and** retrieve a command string for re-execution, use the **S** shortcut.)

- Ditto for the **QUIT** command. It is forced into **F5.** It also closes the display and hardens all edits that you may have made. It does not, however, retrieve any command string for re-execution.

  Notes:
  - For RETRIEVE LIST displays, QUIT functions identically to END. It is only in **Profile Menuing System** displays that QUIT has its own, unique meaning.
  - The current meaning of F5 should not be relied upon. It may change in the future. (Use **F3** instead.)

- A **CANCEL** command is forced into **F4.** It closes the display and **discards** all edits that you may have made.

All other PF keys are rejected with a **DBC560E** error message.

Help COmmands RETRIeve Examples

What follows are several examples of valid and failing uses of a variety of RETRIEVE commands. I hope you find them helpful.

Valid Usage Examples:

RETRIEVE -4
RETRIEVE;RETRIEVE -2;RETRIEVE
RETRIEVE 6;RETRIEVE +2
RETRIEVE -6;RETRIEVE +2
These four command strings are equivalent. They all cause the 4th prior command string to be retrieved to the command line for possible modification and re-execution.

LIST R5;RETRIEVE
The **LIST R5** command is executed, then the entire command string is retrieved back to the command line for possible modification and re-execution.

Perhaps:
- You want to display several registers.
- Then upon each retrieval, you can change the register number and press ENTER.
- Starting with the second time you do that, you will find the display cursor to be conveniently placed at the register number.

LIST R5

LIST PSW;RETRIEVE -2
Ok, here's what's going on:

- When you type the **LIST R5** command and press the ENTER key, that command is executed and the command line is blanked out for you to type your next command.

- If you then type the **LIST PSW;RETRIEVE -2** command, the **LIST PSW** command will be executed and then the **LIST R5** command will be retrieved to the command line for possible modification and re-execution.

LIST RBS;RETRIEVE LIST
The **LIST RBS** command is executed. Then the **RETRIEVE LIST** command also is executed. This causes a display of all previously executed command strings (including the current one) to be displayed. You may then select and modify one of those strings for re-execution.

LIST RBS TCB@4;RETRIEVE
Z/XDC attempts to execute the **LIST RBS TCB@4** command but that fails. To recover, just press **F12,** make your correction (by changing TCB@4 to TCB#4) and press ENTER again. This time, the command string will process successfully and will be retained so that you can change TCB#4 to TCB#5 (for example) and press ENTER again (and again and again ...)

RETRIEVE LIST 2 [or]
RET L 2
[Either form is correct.]

A specialty window is opened up, and all recent commands issued on the parent screen's **2nd** command line are displayed therein:
- You may then select one of those commands for re-execution.
- And/or you may change any of the displayed commands.
- And/or you may purge any of the displayed commands.

Incorrect Usage Examples:

RETR
RET 3
These forms of the RETRIEVE command may not be abbreviated. So in this case error message **DBC783E** is displayed.

LIST PSW;RETRIEVE;LIST REGS
The **RETRIEVE** command must be **last.** When non-RETRIEVE commands are present, they must all precede the RETRIEVE command(s). So in this case error message **DBC763E** is displayed.

LIST PSW;RETRIEVE;RETRIEVE LIST
RETRIEVE LIST commands (like non-RETRIEVE commands) may not follow other forms of the RETRIEVE command. So this command string also fails with **DBC763E.**

Help COmmands RETRY

This command is an alias of the **RETRIEVE** command. See HELP COMMANDS RETRIEVE for more information.

Help COmmands REXx

REXX is a command that allows the user to run user-written subcommands written in REXX.

Syntax:

REXX STARTREXX
ENDREXX
LISTREXX
execname argumentstring

STARTREXX
This command initializes a rexx/XDC Interface. It causes Z/XDC to call IRXINIT to do the following:
- Locate Z/XDC's REXX interface function package (load module xxxEFMVS) and load it into storage.
- Locate the user's REXX exec library (//xxxREXX ddname) and open it.
- Build a REXX Environment Block as well as all other structures necessary for running REXX execs.

Notes:

- This command is optional. If a REXX exec is invoked prior to this command having been issued, then Z/XDC will perform the STARTREXX processing anyway.

- This command will fail if the user's REXX exec library(ies) have not been preallocated to ddname //xxxREXX (where "xxx" is Z/XDC's current clone name).

ENDREXX
This is an optional command to explicitly shut down a rexx/XDC Interface. It does the following:
- It deletes Z/XDC's REXX interface function package (load module xxxEFMVS) from storage.
- It closes the REXX exec library (//xxxREXX ddname).
- It purges the REXX Environment Block and all related structures.

Notes:

- This command is optional. If an **END** command is issued (except for END KEEP) and the rexx/XDC Interface still exists, then Z/XDC will perform the ENDREXX processing automatically.

- If the debugging session should end by any means other than an **END** command, then the rexx/XDC Interface (if any) is **not** automatically taken down.

LISTREXX
This command is identical to the LIST REXX command. It displays information about the rexx/XDC Interface. See HELP COMMANDS LIST REXX for more information.

execname
This is the name of the REXX exec to be run.
the following formats are recognized:
  name
  name(member)
if **name** has at least 1 fullstop within it, the value is always regarded as a dataset name.
if **name** has no fullstops within it, the value is regarded as **simple** and the value could be:
  1: A procedure name.
  2: A DDname.
The rules used to decide which applies are:
  - If immediately followed by an open parenthesis (’(’) then the value is treated as a ddname.
  - If **SET REXX NAME=PROCEDURE** is in effect then the value is treated as a procedure name.
  - Otherwise, the value is checked to see if it is an allocation:
    - If a current allocation is **not found** the value is treated as a procedure name.
    - If a current allocation is **found** the value is treated as a ddname.
Based on the above, an execname could be:
  1: A procedure name - naming a member in the xxxREXX library
  2: A ddname - defining a sequential file allocated to that ddname.
  3: A dsname - defining a sequential file by dataset name.
  4: A member of a library defined by a specific ddname.
  5: A member of a named PDS(E) library.
Note that, in all cases, the REXX execs library must be allocated (ddname: xxxREXX, see below).

argumentstring
This is the argument string (if any) that may be required by the REXX exec. The user’s exec can receive this string via an **ARG** or **PARSE ARG** instruction.

Note, due to Z/XDC’s own syntax rules, argumentstring may not contain semicolons or colons unless the string is enclosed within quotes (’). See COMMANDS SYNTAX CHARACTERSTRINGS for more information.

Requirements

The following are the requirements for enabling the ability to run REXX execs as Z/XDC commands:

- The REXX execs may reside in any library or sequential dataset. However the **xxxREXX** ddname must be allocated in all cases (to a library or concatenation of libraries). If all the procedures are explicitly executed by dataset name (and member name), the **xxxREXX** library can be empty. (the prefix **xxx** must match Z/XDC's current clone name, usually **XDC,** see HELP XDCCLONES for details).

Implicit REXX execution

The **%** command can be used instead of the **REXX** command to request execution of a REXX procedure.

Z/XDC Services Available to User-Written Execs

The rexx/XDC Interface provides several built-in functions by which user-written execs can call a selection of Z/XDC's internal services. Generally, these services do such things as:

- Control the source of REXX input and output
- Parse address expressions
- Extract or zap data in storage
- Extract or zap data in registers
- Extract or alter the retry level PSW
- Send messages to Z/XDC's Display Management for display at the user's terminal.
- Request data from a user

For complete and detailed information, see HELP REXX XDCSERVICES.

Help COmmands RIght

The **RIGHT** command causes a display window to be scrolled rightwards. Columns to the right are brought into view.

The window that is scrolled is always the one that contains the cursor at the time that the command is issued.

shift-F11's factory default value is **RIGHT -.** The factory default setting for F11 is **NOT** "RIGHT". So be careful!

In the definition of shift-F11, the dash is important: It permits you to optionally place an operand on a window's command line and have that operand merged with the PF key's definition. If the dash were not included in the definition, then any potential operand placed on the command line would be ignored. For more information about this, see HELP FULLSCREEN PFKEYS.

Syntax:

```
RIGHT CURSOR
      FULL   (Alias: PAGE)
      DATA
      HALF
```

MAX
nnn
omitted

CURSOR
If the cursor is within a window's data area, that window is scrolled rightwards so as to move the column containing the cursor to the left-hand edge of the window.

If the cursor is located on the window's command line, then RIGHT CURSOR functions like RIGHT FULL: The window is scrolled rightwards by its full width so as to bring data to the right of the window into view.

FULL
PAGE
The window containing the cursor is scrolled rightwards by its full width so as to bring data to the right of the window into view.

DATA
The window containing the cursor is scrolled rightwards by nearly its full width so as to bring the column that was at the window's right-hand edge over to its left-hand edge.

HALF
The window containing the cursor is scrolled rightwards by half its width so as to bring the column that was at the window's right-hand edge over to the middle of the display.

MAX
The window containing the cursor is scrolled rightwards so as to bring the rightmost column of data into view.

nnn
The window containing the cursor is scrolled rightwards by the number of columns specified.

omitted
The window containing the cursor is scrolled rightwards by its default amount. This default is established by the SET WINDOW HORIZONTAL command.

The default scroll amount can be displayed by the LIST WINDOW command.

Help COmmands SCAnlog

The **SCANLOG** command searches a window's scroll area looking for a message containing a specified character string. It works in much the same way as ISPF's FIND command works.

The search will be case sensitive or insensitive according only to the syntax used to give the search string operand.

Syntax:

```
SCANLOG   string  PREVIOUS col-a col-z
         'string'
         "string"
        c'string'
        c"string"
          *
          omitted
```

The String Operand

string
'string'

"string"
c'string'
c"string"
This defines the character string to be searched for. It may be given as follows:

- **string**
  When **UNquoted,** the search string must be a single word, and it may not be the same as any other operand that the **SCANLOG** command accepts. Notes:
  - It may contain special characters, including quote characters.
  - If an unquoted string does contain quote characters, then they may not be the string's first character, and they must **not** be paired. Example: **SCAN don't**
  - The search will be case **insensitive,**

- **'string'**
- **"string"**
  When **quoted,** the string may contain any characters, including blanks, commas and the quote character. Notes:
  - If a quoted string does contain the quote character, then the quote character **must** be doubled. Example: **SCAN 'don''t'**
  - The search will be case **insensitive,**

- **c'string'** - **c"string"**
  When a quoted search string is lead by the letter **C,** this signals that the search is to be case **sensitive.** Note:
  - Only the leading-c syntax is supported. A trailing-c syntax is not supported.

If the string is given within quote characters, then:
- The enclosing quote characters will be removed,
- Embedded pairs of the quote characters will be singlized,
- The search will be either case-sensitive or insensitive according to whether or not the leading-c syntax was used.

On the other hand, if the string is given **unquoted,** then:
- Embedded quote characters are permitted,
- But they should not be doubled,
- And the search will always be case-insensitive.

For more information, see **HELP COMMANDS SYNTAX CHARACTERSTRINGS.**

The Other Operands

`*` (or **omitted)**
The same string that was used in the previous search is used again.

PREVIOUS
The window's scroll area is searched backwards. Notes:

- If **PREVIOUS** is omitted but **string** is given, then the scroll area is searched forwards.
- If **PREVIOUS** is omitted and **string** also is omitted or is given as just `*`, then the scroll area is searched in the same direction as was used by the previous search.

col-a
If just **col-a** (without **col-z)** is given, then a matching string will be found only if it starts in the specified column. (Column #1 is the first character of each logged record.)

col-a col-z
If both **col-a** and **col-z** are given, then a matching string will be found only if its first character falls within the specified columns, inclusively.

If **col-a** and **col-z** are both omitted and **string** also is omitted or just given as `*`, then repeated scans will use previously set column limits, if any.

SCANLOG vs. PF keys

In ISPF and other IBM products "FIND" means to search the current file for a string. But in Z/XDC **FIND** searches storage, not text files. In Z/XDC, **SCANLOG** serves that purpose.

So while Z/XDC's factory default for **F5** is **FIND,** it cannot be used to search a scroll area, only storage.

However, a **SCANLOG -** command is defined in factory default **shift-F5,** so shift-F5 can be used instead for scanning scroll areas. (Note, see **HELP FULLSCREEN PFKEYS** for a discussion of the significance of the dash in PF key definitions.)

However#2, shift-F5 does not exist when displaying HELP topics, so doing repeat SCANLOGs via PF key is not an option there.

However#3, keep reading.

Doing Repeat SCANLOGs with the Help of RETRIEVE Commands

There are a couple of simple ways to do repeat SCANLOGs without PF keys when you do it in combination with **RETRIEVE** commands:

SCANLOG whatever;RETRIEVE
The SCANLOG command will be retained on the command line for as long as you keep pressing ENTER.

SCANLOG whatever [then press F12 once or twice]
The SCANLOG command will be returned to the command line for reissuance.

Help COmmands SCRname

When ISPF is active **and** Z/XDC is communicating via its ISPF interface, the **SCRNAME** command functions in the same way it does in ISPF: It assigns a name to the ISPF window in which Z/XDC is running. That name can then be used by a SWAP command, issued anywhere within ISPF, to cause focus to switch back to Z/XDC's window.

Syntax:

SCRNAME
SCRNAME ON
SCRNAME OFF
SCRNAME name
SCRNAME name PERM

Z/XDC does not check the operands. They are simply passed through for processing by ISPF. See IBM's **ISPF User's Guide, Vol. I** (SC34-4822) more information.

Help COmmands SEt

The **SET** command establishes or changes a large variety of settings, lists, etc. for Z/XDC.

Syntax:

SET keyword operands
S

keyword
This identifies the item to be set. It may be abbreviated to varying degrees depending upon the specific keyword used. The keyword may be any of the following. (Use an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed

sequentially. Use **HELP *FORWARD (F5)** to skip.

ACCESSTO - Establish access to dataspaces.
ASID - Sets the address space from which Z/XDC is to display storage and control blocks. (authorized users only)
ASN - Alias of ASID.
AUTOMAP - Controls various aspects of AUTOMAPing.
AUTH - Can be used by an authorized debugging session to make a non-authorized debugging session authorized.
BANG - In z/OS systems, this controls how the exclamation point (! - a.k.a. "bang"). is to be interpreted in address expressions.
BELL - Controls whether or not the terminal's alarm is sounded for error messages.
BKPT - Alias of BREAKPOINTS.
BPT - Alias of BREAKPOINTS.
BREAKPOINTS - Used for enabling, disabling, and clearing breakpoints.
CANDET - Controls whether or not Z/XDC will intercept CANCEL-type and DETACH-type ABENDs.
COLORS - Tells Z/XDC what colors to use for the various fullscreen display fields.
COLOURS - Alias of COLORS.
CONSOLE - Sets Z/XDC's user interface to either start or stop using Operator Consoles for conducting the debugging session.
CURRENT - Used in dump/XDC to define a given task or SRB to be considered **current** for the purposes of those commands that make use of the notion of a "current" task, RB or SRB.
CXDC - Sets the availability of the c/XDC Licensed Feature either on or off. (Does not license the Feature. Only works if it is already licensed.)
DUB - alter the value of the **DUB** profile setting.
DBC640Q - When a debugging session signon is pending, this controls whether or not Z/XDC issues a **DBC640Q** WTOR to offer the user alternatives to signing in.
DUMP - Enables or disables the taking of a system dump for recovered ABENDs occurring in certain external services called by Z/XDC.
EXITS - Enables or disables the availability of those exits that comprise Z/XDC's Miscellaneous Exits Interface when it is not otherwise installed by XDCCALL or Dynamic Hook processing.
FLC - Changes the current Function Leader Character.
FORMAT - Sets several storage display related settings including source map vs. object code displays, addresses vs. offsets, EBCDIC vs. ASCII, WIDE vs. NARROW, etc.
HICOLOR - Tells Z/XDC to use extended color and hilighting on terminals that support both.
HICOLOUR - Alias of HICOLOR.
HILIGHT - Tells Z/XDC what hilighting to use for the various fullscreen display fields.
HKEYS - Controls Z/XDC's usage of special PF key definitions during HELP topic displays.
ILC - Tells Z/XDC that character mode opcode mnemonic zaps should be checked for instruction length changes.
ILIT - sets the value of an IPCS literal equates.
INTENSITY - Tells Z/XDC what field intensities to use for the various text

fields displayed at the terminal.

ISPF - Tells Z/XDC to paint its 3270 displays using ISPF Display Services, when possible. (Fullscreen TPUTs are used when ISPF is unavailable.)

ISR@PRIM - Sets the Primary Options Menu ISPF is to use when it's invoked via Z/XDC's **ISPF** command.

KEYS - Defines PF key commands and assigns PF key sets to terminal function key ranges.

LINES - Sets the default number of lines of data to be displayed by the DISPLAY and FORMAT commands.

LOCKS - Controls the system locks that Z/XDC will attempt to (re)acquire upon user program resumption.

LOG - Sets the characteristics and status of Z/XDC's external log file. Also sets the Primary Window's scroll limit.

LOGONID - Alias of USERID.

MAPLIBS - Builds, manages, and saves lists of MAPLIB datasets (for providing ADATA to the MAP and DMAP commands).

NOBELL - Prevents error messages from sounding the terminal's alarm.

NOHICOLOR - Tells Z/XDC not to use extended hilighting on terminals that support BOTH hilighting and color.

NOHICOLOUR - Alias of NOHICOLOR.

NOILC - Tells Z/XDC that character mode opcode mnemonic zaps should not be checked for instruction length changes.

NOREADECHO - Deprecated. Please use **SET READ** instead.

NOWTOR - Prevents Z/XDC's Cross Domain Facility from issuing its "awaiting programmer signon" WTOR (DBC640Q).

OPTIMIZATION - Defines the circumstances under which the Z/XDC and cdf/XDC interfaces will optimize the painting of their screen displays.

PARSEORDER - Sets both which and the order by which language-specific parsers are called when parsing High Level Language variables.

PFKEYS - Alias of KEYS.

PRIMARYSIZE - Tells Z/XDC to use the primary screen dimensions for the current terminal.

PRINT - Sets the status and characteristics of the XDCPRINT output file.

PROFILE - Sets certain characteristics of the currently active session profile.

PROXYTASKS - ATTACHs one or more **Formal Proxy Tasks** to any suitable TCB located in any accessible address space.

PSW - Sets various fields in the retry level PSW.

PSWE - Alias of PSW.

QUALIFIER - Sets, changes, or clears the current default label qualifiers.

READ - Sets all options pertaining to the READ command.

READECHO - Deprecated. Please use **SET READ** instead.

REFRPROT - Allows you to change the task-level REFRPROT settings for any TCB in any accessible address space.

REXX - Allows you to change some REXX interface settings.

SCREEN - Alias of WINDOW.

SECONDARYSIZE - Tells Z/XDC to use the secondary screen dimensions for the current terminal.

SECURITY - Enables or disables a security trace, resets Z/XDC's security cache, and/or resets several security related settings.

SIGNONWAIT - Defines the length of time that Z/XDC's Cross Domain Facility will wait for a user to signon to the debugging session.

STEP - Makes various setting regarding STEP'ing through the execution of

HLL programs.

TFS - Turns Z/XDC's Fullscreen Support on or off.

TIMEOUT - During multi-tasking debugging, this tells Z/XDC how long to wait, after losing control of the debugging conversation under one task, before permitting another task to proceed with debugging services.

TPUT - Tells Z/XDC to paint its 3270 displays using TSO based fullscreen TPUT/TGETs instead of ISPF based display services.

TRACE - Sets various tracing related characteristics.

USERID - Sets the TSO userid to be notified by cdf/XDC when a background program is awaiting connection to a programmer for a debugging session.

USS - alter the value of the **USS** profile setting.

VARIABLES - Alias of VSETTINGS

VARS - Alias of VSETTINGS

VDISPLAY - Sets various controls pertaining to the display of High Level Language (HLL) variables.

VSETTINGS - Sets various controls and limitations pertaining to the management of HLL variables.

WINDOW - Sets the characteristics and status of the currently active display window.

WTOR - Causes - Z/XDC's Cross Domain Facility to issue a WTOR (DBC640Q) to permit system operators to abort a debugging session that is awaiting programmer signon.

ZAP - Controls whether or not Z/XDC will allow you to zap into store protected storage.

Help COmmands SEt ACCessto

This command establishes access to various things. After issuing this command various Z/XDC command will be able to access whatever it is whose access has been established.

Syntax:

SET ACCESSTO object ...

Operands:

object
This identifies the kind of object whose access is being established. Supported values are listed below. For specific information, type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

DATASPACE - Access to a dataspace is to be established. Fore more information, see HELP COMMANDS SET ACCESSTO DATASPACE.

Help COmmands SEt ACCessto Dataspace

The **SET ACCESSTO DATASPACE** command can (z/OS permitting) be used to establish Z/XDC access to a dataspace. Given either the name of or STOKEN for a dataspace, this command issues an **ALESERV** macro that attempts to add the dataspace to z/XDC's home task's Access List (DU-AL).

If the ALESERV succeeds, then using the newly created ALET, you can issue z/XDC commands to access that dataspace. Also, an automatic equate named **#dspacename** is created that provides an easy way to reference the dataspace in Z/XDC commands.

If the ALESERV fails, then a message is generate giving a detailed reason for the failure.

Syntax:

```
SET ACCESSTO DATASPACE NAME=dspacename  OWNERASPACE=aspaceref
                       STOKEN=stoken
```

Rules:

- The **NAME=** and **STOKEN=** operands are mutually exclusive.
- The **STOKEN=** and **OWNERASPACE=** operands are mutually exclusive.

Operands:

NAME=dspacename
Dataspace names are 8 characters long. They can be any mix of alpha, numeric and national ($ # @) characters. Unlike other names in IBM's world, the first character may be numeric.

If you know the name of the dataspace that you want to access, and the address space that owns that dataspace (see below), then Z/XDC will search the owning address space to determine the dataspace's **STOKEN.** Then Z/XDC will attempt to gain access to that dataspace by issuing an **ALESERV** macro.

OWNERASPACE=aspaceref
OWNERASPACE=omitted
omitted
This is an optional operand. Its presence or absence affects the **NAME=** operand.

OWNERASPACE= indicates which address space owns the dataspace to be accessed.
The **aspaceref** value can be any of the following:
- The address of an **ASCB**
- A hexadecimal **ASID** (up to 4 digits long, max value: X'FFFE')

- A **decimal** ASID followed by the letter **N** (max value: 65534 [=2**16-2])
- Any address space's **jobname**
- Certain keywords such as **HOME, PASID** etc.
- An **address expression** that resolves to a two-byte storage location containing an **ASID**
- **Omitted,** in which case, the **home** address space is taken as being the owning address space.

For more information, see HELP COMMANDS SYNTAX ASIDS.

STOKEN=stoken
If you know a dataspace's STOKEN (see below), then you can use that to identify the dataspace to be accessed. STOKENs are exactly 16 hexadecimal digits long. Leading zeros, if any, **must** be given. Z/XDC permits you to insert underscores (_) to make the string a bit more human friendly. All underscores are discarded by parsing.

There are a couple of Z/XDC commands that you can use to find dataspace names and STOKENs:

- **LIST DSPACES** can be used to find the names of all dataspaces owned by a given address space.
- **LIST ACCESSLISTS** can be used to reveal the details of one or more Access Lists within any address space.

Help COmmands SEt ASID

This command can be issued **only** when Z/XDC is running in a privileged mode (APF authorized, supervisor state, or key 0-7). In addition, the user must be permitted by Z/XDC's security interface routine.

The **SET ASID** command establishes or clears either Real Addressing Mode or "Foreign Address Space Mode" (FASM). It sets the Target Address Space for Z/XDC's various display commands and the ZAP command. In Real Addressing Mode, Z/XDC permits you to display and zap the System's real (not virtual) storage. In Foreign Address Space Mode, Z/XDC permits you to display and zap the virtual storage in the private area of other address spaces.

Any attempt to use the SET ASID command to target either a foreign address space or real storage for subsequent displays (and zaps) is first check with system security. Z/XDC makes separate checks for subsequent display requests vs. zap requests. Thus it is possible for a computer center to permit some users to display and zap some address spaces (or real storage), and to permit other users only to display (but not zap) address spaces, and to deny still other users from any foreign address space access at all. For more information, please see HELP SECURITY.

Note: When debugging a space-switching PC routine, Z/XDC will automatically display the address space containing the PC routine. Use of the SET ASID command is not necessary in this situation.

For more information about Foreign Address Mode, and for lists of which commands are and are not valid in Foreign Address Space Mode, see HELP VIRTMEM XDCACCESS FASM.

Syntax:

```
    SET ASID aspaceref
        ASN  omitted
```

aspaceref
This identifies the address space (or real storage) that is to become Z/XDC's Target Address Space. Briefly, this can be any of the following:

- A jobname that uniquely identifies a specific address space. If there exists multiple spaces having the same jobname ("INIT", for example), then the SET ASID command fails.
- A keyword indicating the home space or the retry level or error level, primary, secondary, or instruction execution address space. Examples: HOME, PASID, ESASID, IASID, etc.
- The keyword **REAL** (indicating real storage).
- An ASID number.
- An address expression pointing to an ASID number.
- A register containing an ASID number.
- The name of an equate or dsect that has been assigned to represent storage in an address space. (The assigned address space is used.)

For detailed information, see HELP COMMANDS SYNTAX ASIDS.

omitted
If the operand is omitted, then the ASID is reset to that of the Home Address Space.

Examples:

SET ASID EPASID

The Target Address Space is set to the error level instance of the user program's Primary Address Space. Note, this is meaningful only when Z/XDC is being used to debug a program that is running in cross memory mode. See HELP DEBUGGING PCROUTINES for more information.

S AS JES2
The Target Address Space is set to JES2's address space.

S AS 19

S AS 25N
Both of these commands set the target ASID to 0019 hex (or 25 decimal). If address space number 0019 exists, then Z/XDC targets it for storage and control block displays.

S AS .ASCBASID
The Target Address Space is set to that space whose ASID matches the value that is contained in the ASCBASID field of whichever ASCB is mapped by the ASCB dsect.

EQUATE ASID @ASCB+24 HEX 2
S AS ASID
Z/XDC attempts to set the Target Address Space to the address space named ASID. (This is probably **not** what the user is trying to do.)

EQUATE ASID @ASCB+24 HEX 2
S AS ASID+0
The Target Address Space is set to that space whose ASID matches the value that is contained in the 2-byte location labeled by the "ASID" equate.

S AS
S AS HASID
S AS HOME
All of these commands set the Target Address Space to the Home Space.

S AS REAL
The Target Address Space is set to real storage. Subsequent displays and zaps are directed to real storage instead of virtual storage. So now maybe you can see what **really** is happening behind all that smoke and mirrors called virtual storage (maybe not).

Help COmmands SEt ASN

This command is an alias of the **SET ASID** command. See HELP COMMANDS SET ASID for more information.

Help COmmands SEt AUTOmap

Auto mapping is the automatic loading of load module, csect and source image maps at the point where they are first needed.

When AUTOMAPing is set **ON,** then whenever Z/XDC receives control, it checks both the error level PSW and the retry level PSW to see what load modules and csects they point into. If the mapping of those modules/csects has not yet been attempted, then Z/XDC will automatically map them now (or at least try to).

Be aware that with dump/XDC, when AUTOMAPing is set **ON,** then the startup time of dump/XDC may be slow, as Z/XDC needs to scan the CDE chain to attempt to locate the current PSW.

For more information about maps, see HELP MAPS.
For more information about the MAP command, see HELP COMMANDS MAP.

AUTOMAPing controls:
- Are established by the **SET AUTOMAP** command (described here),
- Are displayed by the **LIST AUTOMAP** command,
- And are saved in your session profile by the **PROFILE SAVE** command.

They also can be displayed, set and saved by the **Profile Menuing System.**

Syntax:

```
SET AUTOMAP  ON
             YES
             OFF
             NO
```

All operands are mutually exclusive.

ON
YES
These enable AUTOMAPing.

OFF
NO
These disable AUTOMAPing.

Comments

AUTOMAPing will cause the loading of both module maps (as necessary) and csect or source image maps (when available) whenever Z/XDC receives control with the retry level PSW and/or the error level PSW pointing into a load module and csect whose mapping has not yet been attempted. Once Z/XDC has determined that a map is not available (either for a load module or for a csect), it remembers that fact for the duration of the debugging session so as to avoid repeated AUTOMAPing attempts. (Explicit MAP commands will cause this experience record to be updated as necessary.)

Help COmmands SEt AUTH

The **SET AUTH** command is the first command in a two command process by which an authorized debugging session can make a non-authorized debugging session authorized. This command must be issued from an **already authorized** debugging session.

The second command is **GO NOWHERE** which must be issued from the targeted, non-authorized debugging session to complete the authorization process.

The non-authorized debugging session that you wish to change will, most commonly, be

located in a different address space. So you will need to have Foreign Address Space Mode **(FASM)** access to that other space.

The **SET AUTH** command:
- Must be issued from an authorized debugging session,
- And must target a non-authorized debugging session that's already running in another address space.

Typically, to start up an authorized debugging session, you would navigate to ISPF's Command Shell (=6) and issue **XDCCALLA IEFBR14.** Then you would:
- Use the **SET ASID** command to target the debugging session you want to change.
- Use the **LIST TASKS** command to display the task structure and to create **TCB#n** equates.
- Use the **LIST ESTAES TCB#n** command to display the recovery routines queued to the task of interest, and to create **SCB#n** equates.
- Then issue your **SET AUTH SCB#n** command to start the process of making the target debugging session authorized.

How it Works

SET AUTH works by setting the **SCBKEY0** and **SCBSUPER** flags on in the **SCB** (STAE Control Block) for the debugging session that you wish to change. Then the next time that session gets called by the Recovery/Termination Manager, **RTM** will cause the session to run authorized.

This means that **SET AUTH** will **not** work for those cases in which the target debugging session is not running as an ABEND recovery routine. In particular, it will not work for:
- Z/XDCs that are debugging **SRB** routines,
- Z/XDCs that are debugging routines protected by **FRRs,**
- Z/XDCs that are running as Trap Handlers due to **TRAP2-type** breakpoints.

(This process would always, of course, be unnecessary in the **SRB** and **FRR** cases, since those forms of debugging are already running authorized.)

SET AUTH will work only for debugging sessions that are running as Problem State, **ESTAE-type** Recovery Routines.

SET AUTH is only part one of the process. For **part two** you will need to switch over to that target debugging session and issue a **GO NOWHERE** command. This will cause that session to resume your program in such a way that it will immediately program check and, thereby, return control to Z/XDC without changing anything other than Z/XDC's execution state. It will now be in **Supervisor State.**

For more information, see **HELP COMMANDS GO NOWHERE.**

IMPORTANT!

It is important to understand that **your** program will **not** be made authorized! **Only** Z/XDC will be made authorized. Your program will continue in the same non-authorized problem state it had before the debugging session was made authorized.

Syntax:

SET AUTH scbaddress

Operands:

scbaddress
This must be an address expression that resolves to a STAE Control Block **(SCB)** queued from any TCB located in any address space (security permitting) in the System.

The ABEND recovery routine pointed to by that SCB must be one that runs Z/XDC. It must be:

- Either Z/XDC itself,
- Or any clone of XDC,
- Or XDCCALL's ESTAI routine that interfaces to Z/XDC,
- Or any user written ABEND recovery routine that follows the rules for BASR'ing to Z/XDC.

How to Find the Correct SCB

Here I think is the easiest way to find the correct SCB...

- First, use **LIST TASKS [aspacename]** to produce a tasks report and create **TCB#n** equates for all tasks in the target address space.

- Then examine the tasks report to decide under which task your target debugging session is running. You might find the **LIST RBS TCB#n** command to be helpful here.

- Then use **LIST ESTAES TCB#n** to produce an SCB report and create **SCB#n** equates for each ABEND recovery routine's STAE Control Block currently queued to that task.

- Now examine the SCB report to decide which SCB describes the Z/XDC debugging session that you want to change. It usually will be the one for the exit named **XDC** or **XDCCALL** and captioned as being **ACTIVE.**

- Then issue the appropriate **SET AUTH SCB#n** command to begin the authorization process.

Requirements

In order to use the **SET AUTH** command, several requirements must be met:

- The instance of Z/XDC from which you are issuing the command must be running authorized. One very easy way to create such an instance is to navigate to **ISPF's** Command Shell panel **(=6)** and then issue **XDCCALLA IEFBR14.**

- System Security must permit you to access the target address space.

- The target Z/XDC must be already established and running as an ABEND recovery routine in that address space.

- The target Z/XDC must **not** be running as a Trap handler.

- After using the **SET AUTH** command, you must switch to the target debugging session and then issue a **GO NOWHERE** command to complete the authorization process.

Help COmmands SEt BAng

Warning: This command is deprecated! It may be dropped without notice from a future release of Z/XDC. Please refrain from using it.

Going forward, if you need to use an AMODE sensitive indirect operator, you should use **~INDIRECT(AMODE)** (instead of "!").

In address expressions, there are three characters that Z/XDC interprets as being "indirect operators". They are the percent sign (%), the question mark (?), and the exclamation point (! - a.k.a. "bang"). The percent sign loads a 24-bit wide pointer, and the question mark loads a 31-bit wide pointer, but the bang character can have either of two meanings:

- **AMODE:** The bang is amode sensitive. This means that it loads a pointer whose width matches the retry level PSW's current addressing mode: 24-bit, 31-bit, or (for z/OS systems) 64-bit.
- **64BIT:** The bang always loads a 64-bit wide pointer (z/OS only).

For more information about indirect operators, see HELP ADDRESSING SYNTAX BETWEENTERMS INDIRECT.

The **SET BANG** command sets the "!" character's current meaning.

Syntax:

```
SET BANG AMODE
         64BIT
```

AMODE
The ! indirect operator will be sensitive to the retry level PSW's current addressing mode.

64BIT
The ! indirect operator will be resolved as a 64-bit wide pointer (z/OS only).

The BANG setting can be saved in your session profile. It can be displayed by the LIST BANG command. It also can be displayed and changed by Z/XDC's "Profile Menuing System". For more information, please see:

HELP COMMANDS LIST BANG
HELP PROFILES MENU

The initial factory default setting is **64BIT.** If you wish to change this default, then you can use the following commands to do so:

- **PROFILE READ** (to ensure that no unwanted changes creep in)
- **SET BANG AMODE** (to cause "!" to mean AMODE-sensitive indirect)
- **PROFILE SAVE** (to save your updated profile)

You might notice that Z/XDC has something of a problem regarding indirect operators: There are at least four distinct useful indirect operations (more, actually), but only three characters (% ? !) are available to represent them. So to pick up the slack, there is a built-in function that can be used to perform a variety of indirect operations. Its name is "~INDIRECT(...)". See HELP FUNCTIONS INDIRECT for more information.

Help COmmands SEt BEll

The **SET BELL** command is used to control whether or not Z/XDC rings your terminal's alarm either (a) when an error message is displayed or (b) every time any panel is displayed or (c) never.

Syntax:

```
SET BELL ALL (or ON)
         ERROR (or omitted)
         OFF
```

Aliases:
- SET BELL ON is an alias for SET BELL ALL.
- SET BELL is an alias for SET BELL ERROR.
- SET NOBELL is an alias for SET BELL OFF.

SET BELL ALL
SET BELL ON
The terminal alarm is sounded every time Z/XDC sends a display to the terminal.

SET BELL ERROR
SET BELL
The terminal alarm is sounded only when Z/XDC sends an error message to the terminal.

SET BELL OFF
SET NOBELL
The terminal alarm is never sounded.

The bell setting can be saved in your profile dataset for automatic restoration every time you start a debugging session. For more information, please see:

- HELP PROFILES
- HELP COMMANDS PROFILE

The factory default bell setting is SET BELL ERROR.

Help COmmands SEt BKpt

This command is an alias of the **SET BREAKPOINTS** command. See HELP COMMANDS SET BREAKPOINTS for more information.

Help COmmands SEt BPt

This command is an alias of the **SET BREAKPOINTS** command. See HELP COMMANDS SET BREAKPOINTS for more information.

Help COmmands SEt BReakpoints

The **SET BREAKPOINTS** command manages both active and deferred breakpoints. It permits you to enable, disable, or delete existing and deferred breakpoints. It does **not** permit you to create breakpoints. (That is done via the various breakpointing commands: **ADEFERRED AT ATX HDEFERRED HOOK TDEFERRED TRACE** and **TRAP.)**

SET BREAKPOINTS also allows you to enable, disabled and deleted **deferred hooks.** But for **active hooks,** SET BREAKPOINTS will only allow you to delete them. It will not allow you to enable or disable them.

When a **active** breakpoint is **disabled,** its trap is removed from user code, but the definition remains known to Z/XDC and remains displayable by the **LIST BREAKPOINTS** command. This is so that you may, if you wish, reenable the breakpoint later.

When a breakpoint is **deleted,** both its trap and its definition are removed.

When a **deferred** breakpoint or hook is disabled, it will suspend the creating of traps or hooks into load modules an LOAD time.

When **multiple** breakpoints exist at a single location, the trap is removed from that location only when **all** breakpoints at that location are either disabled or deleted. So long as any breakpoint remains active at that location, the trap remains in place.

Syntax:

SET BREAKPOINTS action selectionoperands [NO]MSG [DUB=option] [USS=option]
SET BKPT
SET BPT
SET B
OFF ----------

Notes:

B, BPT and **BKPT** are all aliases of **BREAKPOINTS.**

OFF is a shorthand for **SET BREAKPOINTS OFF.**

There are no shorthands for the **ENABLE, DISABLE** and **TOGGLE** actions.

The **SET BREAKPOINTS** command accepts three kinds of operands:

- **Action** operands specify the action to take: **ENABLE, DISABLE, TOGGLE** and **OFF** (i.e. delete)
- **Selection** operands select the breakpoints to be acted upon. These operands can select breakpoints either specifically, generically. categorically, by type, by family,or by location. There can be as many of these operands as you like.
- **Control** operands set various overall controls for command processing.

The action operand must be placed first, after **SET BREAKPOINTS.**

The selection and control operands can be freely intermixed following the action operand.

The **DUB=option** operand and the **USS=option** operand can be placed almost anywhere on the command. In the case of the **SET BREAKPOINTS** command (or its aliases) they can be placed anywhere after the first 2 words. In the case of the **OFF** command they can be placed anywhere after the first word.

Action operands

Selections of breakpoints can be enabled, disabled or OFF’d (deleted) according to the following action operands.

SET BREAKPOINTS DISABLE
Z/XDC marks the selected breakpoints as being disabled. For active breakpoints:
- Their traps are removed from user code,
- And their conditional expressions (if any) do not participate further in breakpoint processing decisions.
- (Active HOOKs cannot be disabled.)

For deferred breakpoints and hooks:
- They become ineligible for setting active breakpoints and hooks into load modules at LOAD time.

Their definitions, however, remain and can be displayed by the **LIST BREAKPOINT** command and can be reenabled by a future **SET BREAKPOINTS ENABLE** or **TOGGLE** command.

SET BREAKPOINTS ENABLE
Z/XDC marks the selected breakpoints as being enabled. For active breakpoints:
- Their traps are restored into user code,
- And their conditional expressions (if any) resume participation in breakpoint processing decisions.

For deferred breakpoints and hooks:
- They resume eligibility for setting active breakpoints and hooks into load modules at LOAD time.

SET BREAKPOINTS TOGGLE
Z/XDC toggles the selected breakpoints. Enabled breakpoints are disabled, and disabled breakpoints are enabled.

The **X** shortcut also can be used to toggle breakpoints. See **HELP SHORTCUTCMDS X** for more information.

SET BREAKPOINTS OFF
OFF
Z/XDC deletes the selected breakpoints **and HOOKs.** For active breakpoints and HOOKs:
- Their traps are removed from user code,
- And their definitions are discarded.

For deferred breakpoints and hooks:
- Their definitions are discarded.

The **O** shortcut also can be used to delete breakpoints and HOOKs. See **HELP SHORTCUTCMDS O** for more information.

OFF is shorthand for **SET BREAKPOINTS OFF.** The two commands can be used interchangeably.

Selection Operands - Categorical

Breakpoints and HOOKs can be categorized in several ways. Those categories can be selected as follows.

(Note, when **active HOOKs** are indicated, they are included only when the action is **OFF.** They are not included for the **ENABLE, DISABLE** and **TOGGLE** actions.)

ACTIVE: Selects all active breakpoints and HOOKs except ATXs, STEPs and WATCHs.

ALL: Selects all active and deferred breakpoints and HOOKs **except** ATXs, STEPs and WATCHs.

DEFERRED: Selects all deferred breakpoints and HOOKs.

DISABLED: Selects all **disabled,** active and deferred breakpoints except ATXs, STEPs and WATCHs. It also selects all disabled deferred HOOKs.

ENABLED: Selects all **enabled,** active and deferred breakpoints except ATXs, STEPs and WATCHs. It also selects all enabled deferred HOOKs.

Selection Operands - Types

There are several types of hooks and breakpoints, and they can be selected by those types as follows.

(Notice that while several types of active breakpoints exist, there are only three types of deferred breakpoints: **ADs, TDs** and **HDs.** Keep this in mind when reading the following type operand explanations.

- **AD:** Selects all deferred persistent traps set by ADEFERRED commands.
- **ATS:** Selects all active and deferred persistent traps set by AT and ADEFERRED commands.
- **ATX:** Selects all active super persistent traps set by ATX commands.

H
- **HOOKs:** Selects all active and deferred HOOKs set by HOOK and HDEFERRED commands.
- **HD:** Selects all deferred HOOKs set by HDEFERRED commands.
- **STEPS:** Selects all orphaned trace points created by STEP commands.

T
TR
- **TRA:** Selects all active and deferred transient traps and orphaned trace points set by TRAP, TDEFERRED and TRACE commands.
- **TD:** Selects all deferred transient traps set by TDEFERRED commands.
- **TRACES:** Selects all orphaned trace points created by TRACE commands.
- **TRAP:** Selects all active transient traps set by TRAP commands.
- **WATCHS:** Selects all enabled and disabled active WATCHs set by TRACE W commands.

Selection Operands - Families

All active and deferred breakpoints, and all deferred hooks belong to numbered **families.** (Active HOOKs do not.) Families can contain multiple breakpoints of multiple types.

Breakpoint selections can be made by family number and (optionally) by type within a family. For example, suppose the following active breakpoints exist:

```
.+6E 90EC D00C  E0283ZID STM      R14,R12,X'00C'(R13)
.+72 001D       TR00008K LR       R1,R13
.+74 00D0 F088  AT00008L LA       R13,X'088'(,R15)
```

```
    .+78 00D0 1008  TR00008M ST       R13,X'008'(,R1)
    .+7C 0010 D004  AT00008N ST       R1,X'004'(,R13)
    .+80 5810 1018           L        R1,X'018'(,R1)
    .+84 00F4 0026  AT00009O J        *+X'004C'
```

Family **#8** contains two types of breakpoints: **ATs** and **TRAPs:**
- **OFF T8** would delete both TRAPs in family 00008.
- **S B DISABLE #8** would disable all four traps in family 00008.
- **S B OFF T#9** would fail to delete any breakpoints because family 0009 does not contain any T type traps.

Notice:
- Leading zeros do not need to be given when referencing family numbers.
- A quoted value can never be a family name.
- The trailing letter in the trap names is not part of the family number.

In the following operand descriptions, **nnnnn** represents the 1-6 digit family number.

Annnnnn
ATnnnnn: Selects all active and deferred **ATs** within the given family. It does **not** select ATXs.

ATXnnnn: Selects all active **ATXs** within the given family. It does not select ATs.

Hnnnnnn
HDnnnnn: Selects all deferred HOOKs within the given family.

Snnnnnn
STnnnnn
STEnnnn
STEPnnn: Selects all orphaned STEP points within the given family.

Tnnnnnn
TRnnnnn
TRAnnnn
TRACnnn
TRACEnn
TRAPnnn: Selects all active and deferred traps as well as all orphaned TRACE points within the given family. (No distinction is made here between orphaned trace points and transient traps.)

Wnnnnnn
WAnnnnn
WATnnnn
WATCnnn
WATCHnn: Selects all WATCHs within the given family.

#nnnnnn: Selects all members of the given family regardless of active/deferred status, but of only most types. The **excluded** types are **ATXs, STEPs** and **WATCHs.** For example, suppose the following active breakpoints exist:

```
    .+6E 90EC D00C  E0283ZID STM      R14,R12,X'00C'(R13)
    .+72 001D       TR00088K LR       R1,R13
    .+74 00D0 F088  AT00088L LA       R13,X'088'(,R15)
    .+78 00D0 1008  TR00088M ST       R13,X'008'(,R1)
```

```
.+7C 0010 D004  AT00088N ST       R1,X'004'(,R13)
.+80 5810 1018           L        R1,X'018'(,R1)
.+84 00F4 0026  ATX0088P J        *+X'004C'
```

Then **OFF #88** will delete the four ATs and TRs, but not the ATX.

Selection Operands - Names

Hooks and breakpoints can be selected both by their own names and by the name of the load module in which they occur.

breakpoint names have three parts:
- The breakpoint's type is at the start of the name.
- The breakpoint's family number is in the middle.
- A uniqueness letter is at the end.

When naming a breakpoint to be deleted, the name you give must match all three parts, except:

- You do not need to provide a family number's leading zeros. For example, regarding the code snippet given above, **OFF AT88N T00088M** works just fine.

- If you use **#** instead of a breakpoint type, that will match any breakpoint type except ATX, WATCH and STEP. For example, **OFF #88N** works, but **OFF #88P** doesn't. (Use **OFF ATX88P** instead.)

- A quoted value can never be a breakpoint name.

hooknames must match exactly. For example, a HOOK named **HOOK0034** cannot be deleted with **OFF HOOK34.** You must use the full name: **OFF HOOK0034.**

loadmodulenames (including **'loadmodulenames' "loadmodulenames"** including program object names and USS program names) will delete all breakpoints located within the named load module (etc.). (Hooks will not be deleted.)

A load module name may be quoted to ensure that it is never treated as a family name or breakpoint name.

Selection Operands - Addresses

Z/XDC allows multiple breakpoints to exist at a single location. Also, no restriction is placed on the types of breakpoints that are set. Selecting by address is a way to process those breakpoints without regard to type.

addressexpression: If a mix of breakpoints exist at the resolved location, then all breakpoints (except ATX breakpoints) located there will be processed. ATX breakpoints will not be processed.

If only ATX breakpoints exist there, then they **will** be processed after all.

Control Operands

MSG or **NOMSG**
These operands control whether (MSG) or not (NOMSG) the SET BREAKPOINT command reports how many breakpoints it was able to process. Unless an error occurs, the default is NOMSG. If an error occurs, then MSG is forced.

USS=option
This operand controls the processing of other operands on the **SET BREAKPOINT** command. This operand is processed first and can be specified anywhere on the command.
More information about the **USS=** operand can be found in **HELP DEBUGGING USS OPERANDS USS=**

DUB=option
This operand determines what is to be done if a non-dubbed program needs to be dubbed. This operand is processed first and can be specified anywhere on the command.
More information about the **DUB=** operand can be found in **HELP DEBUGGING USS OPERANDS DUB=**

Examples:

SET BREAKPOINTS OFF
OFF
All breakpoints (except ATXs, WATCHs and STEPs) are removed.

SET BKPT ENABLE DISABLED
All disabled breakpoints (except ATXs, WATCHs and STEPs) are enabled.

SET BPT DISABLE #5 PSW?
All breakpoints associated with family **#5** are disabled.

Also, all breakpoints located at the address pointed to by the retry level PSW are disabled.

The affected breakpoint definitions are marked disabled and remain defined, but are ignored during Z/XDC breakpoint analysis.

ATXs, WATCHs and STEPs, if any, are not processed.

If after disablement no enabled breakpoints remain at a trap location, then the trap is removed from user code. (On the other hand, if enabled breakpoints remain, then the trap is not removed.)

S B ENABLE TRAPS AT23
All breakpoints created by the TRAP and TDEFERRED commands are enabled.

In addition, all AT and ADEFERRED breakpoints contained in family #23 are also

enabled.

All other breakpoints created by the AT and ADEFERRED commands, as well as all ATXs, WATCHs and STEPs, if any, are not processed even if they are in family #23.

SET BREAKPOINTS OFF DEFERRED TR3C
OFF DEFERRED TR3C
All breakpoints and hooks created by the ADEFERRED TDEFERRED and HDEFERRED commands, as well as the breakpoint named **TR00003C,** are removed from program code, and the definitions are deleted permanently.

Tip: The **SET BREAKPOINTS** command can be used as an automatic command for a breakpoint. You can cause entire families to be disabled, enabled, or cleared when an active enabled breakpoint is accepted. For example, you can define several families of breakpoints and mark them all disabled. Then define a family of enabled breakpoints that, when triggered, will issue an automatic command to enable the next family breakpoints, disable the previous family, and automatically continue processing. The possibilities are endless.

Help COmmands SEt CAndet

The **SET CANDET** command allows you to control whether or not Z/XDC will allow you to debug **termination** ABENDs, Examples:
- Operator cancel (s122 and s222)
- Forced DETACH (s13E)
- Various others (s00C-04, etc.)

SET CANDET defines Z/XDC’s response to those ABENDs for which recovery either is disallowed or is impossible:

- Recovery is disallowed when the System sets **SDWACLUP=1** into the SDWA. This occurs for all of the above listed ABEND codes and probably others as well.

- Recovery is impossible when a retry level environment does not exist. This occurs, for example, when ATTACH fails with an 806 ABEND and Z/XDC receives control as an ESTAI in the task that was being ATTACH’d.

Note, in order for **SET CANDET** to have any effect at all, it is necessary that Z/XDC actually receive control for an **SDWACLUP=1** ABEND in the first place. For the **s122** and **s222** ABENDs (the Operator Cancel ABENDs), this is controlled by whether or not **TERM=YES** is specified on the **ESTAE[X]** macro (or ATTACH with **ESTAI=)** by which Z/XDC receives control. When **TERM=NO** is in effect, all recovery routine processing **(INCLUDING Z/XDC itself)** will be bypassed, and therefore **SET CANDET** will have no effect at all.

Syntax:

SET CANDET DEBUG
IGNORE

DEBUG (aliases: **ON** and **YES)**
If Z/XDC receives control at all, then it will allow the debugging of **all** ABENDs, including all **SDWACLUP=1** ABENDs.

IGNORE (aliases: **OFF** and **NO)**
Z/XDC will **not** allow the debugging of any ABEND for which SDWACLUP=1. Instead, those ABENDs will be percolated as if Z/XDC did not exist.

Generally, there is no point in "debugging" SDWACLUP=1 ABENDs because...
- Their causes often are obvious.
- There causes are totally external to the program being debugged.
- Recovery from such ABENDs is not possible.

So Z/XDC's default action (ie, when SET CANDET IGNORE is in effect) is to bypass such ABENDs and simply let them percolate.

However, If you do want a debugging session for SDWACLUP=1 ABENDs, then do the following:
- Issue **SET CANDET DEBUG.**
- You may want to issue **PROFILE SAVE XDC** to make this setting your default.
- You may want to add **TERM=YES** to the ESTAE[X] macro by which Z/XDC is invoked (either directly or indirectly).
- Or you may want to add **TERM=YES** to the ATTACH macro if Z/XDC is invoked as an ESTAI.

Then when Z/XDC is given control for an SDWACLUP=1 ABEND, it will open a debugging session instead of percolating. You can then issue any Z/XDC commands you like **except** TRACE, GO, GOT or GOX.

As indicated above, this setting can be saved in your session profile. It can be displayed by the **LIST CANDET** command. It also can be displayed and changed by Z/XDC's "Profile Menuing System". For more information, please see:
HELP COMMANDS LIST CANDET
HELP PROFILES MENU

Help COmmands SEt COLORs

Screen images generated by Z/XDC contain up to four different kinds of fields:

(A) - **Low** intensity **input** fields for displaying data in secondary input areas and in overwritable fields (such as storage displays).

(B) - **High** intensity **input** fields for primary input areas such as command lines.

(C) - **Low** intensity **protected** fields for displaying nonoverwritable information.

(D) - **High** intensity **protected** fields for displaying title and header lines, warning messages, etc.

For those terminals that have the necessary hardware support, display colors and extended hilighting can be set by the following commands:

- SET COLORS
- SET HILIGHT
- SET INTENSITY
- PROFILE

The **SET COLORS** command defines the desired field colors.

Syntax:

SET COLORS abcd
SET COLOURS

abcd
This must be a 4-character string. Each character position corresponds to a field type as listed above in the sequence listed above. Each character must be one of the following codes:

B - Blue.
C - Cyan (same as Turquoise).
G - Green.
M - Magenta (same as Pink).
P - Pink (same as Magenta).
R - Red.
T - Turquoise (same as Cyan).
W - White.
Y - Yellow.
D - Restores the corresponding field to its hardware default color.

The current color settings can be displayed by any of the following commands:

- LIST COLORS
- LIST TERMINAL
- PROFILE

The colors can also be both displayed and set by Z/XDC's Profile Menuing System.

Your color choices can be saved in your profile dataset for automatic restoration every time you start a debugging session. For more information, please see HELP PROFILES.

The factory default color settings **RWGY** (red, white, green and yellow).

Example:

SET COLORS RYCW;L TERMINAL
The SET command sets input fields to be displayed in red or yellow (depending upon importance), and it sets protected fields to be displayed in cyan or white. The LIST command then immediately reports the results.

Note: Even if you have a color terminal, your color choices can be ignored under the

following circumstances:
- Your VTAM Systems Programmer has not properly defined your terminal as supporting "Extended Attribute Bytes" (done via a suitable PSERVIC= value on a MODEENT macro in a VTAM mode table definition).
- You are using a PC workstation program to emulate a terminal, but you have not properly configured your emulation to support "color orders" from VTAM.

See HELP FULLSCREEN TERMINALS for more information and for suggestions about remedying these problems.

Help COmmands SEt COLOUrs

This command is an alias of the **SET COLORS** command. See HELP COMMANDS SET COLORS for more information.

Help COmmands SEt CONsole

The **SET CONSOLE** command can be used to direct Z/XDC's user interface to either start or stop using Operator Consoles for conducting the debugging session.

When the user interface is directed towards operator consoles, Z/XDC uses WTORs to display its prompt at one or more System Operator Consoles. The user can then issue commands by replying to the prompt. For more information, see **HELP USERINTERFACES WTOWTOR.**

The first time that Z/XDC attempts to use an Operator Console as a user interface, it issues a message **(DBC829Q)** via WTOR to ask for permission from the System Operators for the user to do this. The **SET CONSOLE** command will succeed only if operations permits it to.

Syntax:

```
SET CONSOLE ON
            name
            number
            OFF
            TOGGLE
            omitted
```

Operands:

ON
Z/XDC's user interface is switch to use Operator Consoles. Subsequent displays and command prompts are sent via WTOs and WTORs.

The initial WTO[R] is routed via ROUTCDE=11 ("write to programmer"); however,

whatever console is used to reply to Z/XDC's WTOR, the resulting messages and subsequent reprompt will be routed back to that replying console.

name
number
Z/XDC's user interface is switch to use Operator Consoles. Subsequent displays and command prompts are sent via WTOs and WTORs.

The initial WTO(R) is routed to that console whose name or 4-byte ID# matches the given operand; however, whatever console is used to reply to a Z/XDC WTOR, the resulting messages and subsequent reprompt will be routed back to that replying console.

The given name or number is parsed as follows:

- If the string consists entirely of hexadecimal digits, then it is considered to be a number, and it is used to select a console by 4-byte ID#.

- If the string consists entirely of hexadecimal digits, and if the first character is any digit A through F, then it is considered to be **both** a number and a name, and it is used to select the first console found whose 4-byte ID# or console name matches the given value.

- If the string consists entirely of decimal digits but with the last character being the letter N, then it is considered to be a decimal number, and it is converted from decimal to hex and used to select a console by 4-byte ID#.

- Otherwise, the string is used to select a console by name.

OFF
Z/XDC makes an attempt to switch the user interface from using operator consoles to using the user's terminal. If the attempt fails, then the command is ignored.

TOGGLE
omitted
The use of WTO[R]s as the user interface is toggled:
- If it's on, it is turned off.
- If it's off, it is turned on.

Help COmmands SEt CUrrent

The **SET CURRENT** command can be used (under dump/XDC only) to designate any address space, task, request block, or SRB routine as being the "current" execution space or thread. This has the effect of causing Z/XDC to treat the designated control block structure, PSW, registers and other state data as being "current" as far as Z/XDC's commands are concerned. In other words, the designated "current" structures will

used as the default target structures of such commands as:

- **LIST TASKS**
- **LIST RBS**
- **LIST ESTAES**
- **LIST XMS**
- **LIST PSW**
- **LIST RWREGS**
- **LIST FREGS**
- [etc.]

About CAS and CXU

You can designate an aspace without also designating a specific task within that space. In this case, dump/XDC sets something it calls a **CAS** (Current Address Space), but does not set a **CXU** (Current eXecution Unit). This means that Z/XDC commands that require a current aspace will work (example: LIST TASKS), but those that require a current task or SRB won't work unless you add the operands needed for designating a target task (LIST RBS), RB (LIST REGS) or SSRB.

If you designate a specific task, request block or SSRB, then dump/XDC sets the **CXU** as indicated, and the **CAS** is mostly ignored (since current execution thread information includes current address space information). See **HELP DUMPS EXECUTIONTHREAD** for more information about dump/XDC's current execution settings.

You can also revert the "current" execution aspace and thread to that chosen (if any) by dump/XDC at the beginning of the dump/XDC session.

Syntax:

```
SET CURRENT rbaddress
            ssrbaddress
            tcbaddress
            ascbaddress
            asidreference
            CPU=hh
            REVERT
```

Operands:

The operands designate the target address space or execution thread as follows:

rbaddress
The CXU is set to the designated RB, and the CAS is set to the home address space.

ssrbaddress
The CXU is set to the chosen SSRB, and the CAS is set to the home address space.

tcbaddress
The CXU is set to the newest RB for this TCB (see below), and the CAS is set to the home address space.

ascbaddress
The CAS is set to the nominated aspace. The CXU is not set.

asidreference
The CAS is set to the nominated aspace. The CXU is not set.

Asidreferences are formally defined in **HELP COMMANDS SYNTAX ASIDS,** but in summary, they include:
- Jobnames, as long as they are unique,
- Keywords such as HOME, PRIMARY and such,
- ASID numbers
- etc.

CPU=hh

If the dump being examined is a **stand-alone dump,** then you can use this operand to designate whatever thread was running on the specified CPU to be the CXU. **hh** is a 2-digit hex id number of a CPU engine.

REVERT

This reverts to the CAS/CXU settings (if any) that dump/XDC chose when you started the dump/XDC session.

Resolving Ambiguous Values

As you can see, the **given value** can be any of a large number of things. So there will be cases where a given value can reasonably be interpreted in two different ways. Here are how such cases are resolved.

- **Simple Address Expressions:** An address expression can be as simple as a single name. These are syntactically not distinguishable from jobnames, so that is how they will be interpreted (as jobnames). But you can fix this ambiguity simply by appending **+0** to the name. Example:

  - **LIST TASKS:** This creates a series of **TCB#n** equates.

  - **SET CURRENT TCB#5:** This attempts to set the CXU to be under the 5th TCB, but it likely fails because:
    - **TCB#5** is a simple name.
    - So the command interprets it as a jobname.
    - There very likely is no job named TCB#5.

    - **SET CURRENT TCB#5+0:** This succeeds at setting the CXU because:
        - **TCB#5+0** is not a simple name,
        - So the command interprets it as an address expression,
        - Which resolves to the address of the aspace's 5th TCB,
        - Which makes the **SET CURRENT** command happy.

- **Some ASIDs:** When an ASID starts with one of the **A-F** hex digits, it becomes syntactically indistinguishable from a simple name. You can fix this ambiguity simply by prepending a **0** to the ASID. Example:

    - **SET CURRENT A234:** This attempts to set the CAS to be whatever address space has A234 as its ASID, but it likely fails because:
        - **A234** is a simple name.
        - So the command interprets it as a jobname.
        - There very likely is no job named A234.

    - **SET CURRENT 0A234:** This succeeds at setting the CAS because:
        - **0A234** is not a simple name,
        - It is a simple hex number that resolves into the proper range (0000-FFFF).
        - So the command interprets it as an ASID,
        - Which makes the **SET CURRENT** command happy.

When the CAS and CXU Are and Are Not Set

A **CAS** can be nominated by a unique jobname, an ASID (hex or decimal) or ASCB address. In all cases the nominated address space must be available in the dump. In these cases a **CXU is not set.**

A **CXU** can be nominated by a TCB address, an RB address or an SSRB address. In these cases, the **CAS** will be set to that thread's **home** aspace.

When Only a TCB Is Given, Which RB Is Deemed Current?

If a **CXU** is nominated by a TCB address, then **SET CURRENT** has to choose which RB, under that task, is to be considered. There is only one possibility:

- The task's **newest** RB is designated as being "current".

When a CXU is set, what is the CAS Set To?

Regardless of the cross-memory mode of the chosen CXU, the current address space is always set to the **home** aspace.

Help COmmands SEt CXdc

The **SET CXDC** command enables or disables the c/XDC Licensed Feature. It does not

license the Feature. It only enables or disables it if it is already Licensed.

This setting is saved in your Session Profile. Its Factory Default value is **ON.** See HELP PROFILES for more information.

The setting's current value can be displayed by the **LIST CXDC** command.

Syntax:

```
SET CXDC  ON
          YES
          OFF
          NO
```

All operands are mutually exclusive.

ON
YES
These enable the c/XDC Licensed Feature. If the Feature is already licensed, then it will now become usable.

On the other hand, if the Feature is not currently licensed, then this command has no effect.

OFF
NO
These disable the c/XDC Licensed Feature. Even if the Feature is licensed, it will no longer be usable. Examples:

- You will not be able to load C program source maps.
- You will not be able to use the **STEP** command.
- You will not be able to display C program variables.

Help COmmands SEt DUB

This command may be used to set the **DUB** profile setting.

Syntax:

```
SET DUB  NO
         YES
         ASK
```

NO
The current **DUB** setting in the profile is set to NO.

YES
The current **DUB** setting in the profile is set to YES.

ASK
The current **DUB** setting in the profile is set to ASK.

The DUB setting can be saved in your session profile. It can be displayed by the LIST DUB command. It also can be displayed and changed by Z/XDC's "Profile Menuing System". For more information, please see:

HELP COMMANDS LIST DUB
HELP PROFILES MENU

The initial factory default setting is **ASK.** If you wish to change this default, then you can use the following commands to do so:

PROFILE READ (to ensure that no unwanted changes creep in)
SET DUB <whatever>
PROFILE SAVE

Help COmmands SEt DBc640q

This command relates to cdf/XDC.

When a background batch job or system task ABENDs, and when that job is properly set up for debugging via Z/XDC, cdf/XDC gains control and waits for an appropriate user to signon to a debugging session with the ABENDed program. (See **HELP XDCSRVER CDF** for more information.)

Normally in this situation, cdf/XDC will wait only a limited time (as controlled by the **SET SIGNONWAIT** command) for a user to signon to the debugging session.

If **SET DBC640Q** has been issued (possible either via the current profile or via an automatically issued command), then during this waiting period cdf/XDC will send a **DBC640Q** WTOR to the System Log, giving the user some choices about what he might want to do instead of signing on to the session.

The **DBC640Q** WTOR also gives the user the opportunity to cause the abended program to resume execution as if Z/XDC (and the **breakpoint** or **DEAD trap** that caused it to receive control) had not been present at all.

Of course, whether this resumption would succeed or fail depends entirely upon the logic of the interrupted program and the reason execution reached the breakpoint or DEAD trap.

The **SET DBC640Q OFF** command prevents cdf/XDC from issuing the DBC640Q WTOR. In this case the operators will not be explicitly given an opportunity to abort the waiting period. Instead, cdf/XDC will wait until:

- A programmer signs on to the debugging session,
- Or the waiting period times out,
- Or the ABENDed program is canceled by the operators.

In order for the **DBC640Q** setting to be effective, it must preexist in your default session profile. For details, see the following:

- A **PROFILE SAVE** command has to be issued to save this setting into the user's personal profile.
- And a **//ISPPROF** or **//XDCPROF** DD card, pointing to the profile library, must be included in the JCL for the job to be debugged via cdf/XDC.

Syntax:

```
SET DBC640Q ON
            OFF
```

This command accepts no further operands.

This **DBC640Q** setting can be saved in your profile dataset for automatic restoration every time you start a debugging session. For more information, please see:

HELP PROFILES
HELP COMMANDS PROFILE

The factory default for this setting is **DBC640Q ON.**

Help COmmands SEt DUMp

The **SET DUMP** command can be used to enable or disable the possibility of producing a system dump in the event that an external service, called by Z/XDC, fails with an ABEND.

This command affects only the taking of dumps for certain recovered or suppressed ABENDs occurring within Z/XDC processing. It does not affect:

- Z/XDC's handling of such ABENDs. (They are always suppressed.)
- ABENDs occurring during user program execution. (Such dumps can be requested via the **END** command.)
- The dumping or handling of unexpected ABENDs occurring within XDC and that cause Z/XDC to fail completely. (Such dumps are controlled by the presence or absence of a //xxxSDUMP DD DUMMY allocation.)

Also, the dumps produced or suppressed by this command are not affected by the presence or absence of a //xxxSDUMP DD DUMMY allocation. As noted above, that allocation affects the production of dumps only for ABENDs that Z/XDC cannot recover from or suppress. (For more information about that, see HELP DDNAMES SDUMP.)

The **SET DUMP** command can enable dump production for ...
All ABENDs,
All system ABEND codes,
All user ABEND codes,
Or specific ABEND codes (system or user) occurring in eligible services.

Currently (Z/XDC release v3.1), ABENDs in the following services are setup to be

responsive to the **SET DUMP** command:
- Calls made by Z/XDC to internal subroutines written in the C Language.
- ABENDs occurring when Z/XDC invokes anything that runs in an LE environment (Language Environment).

Such services are invoked by the c/XDC Licensed Feature.

Recovered or suppressed Z/XDC ABENDs occurring under all other circumstances are never affected by the SET DUMP command. (An example of such an ABEND would be the 0C4 that occurs when Z/XDC is directed by a user's command to display non-existent storage.)

Note, if this command leads to a system dump being produced, then one will, in fact, be produced regardless of whether or not Z/XDC is running authorized. (Other factors, such as z/OS's DAE service, may still cause the dump to be suppressed.) Message DBC834W will be issued reporting on the success or failure of the dump attempt.

If a system dump is produced, then it will contain **all** allocated storage in the primary address space, including the PLPA, the CSA, the SQA and the System Nucleus. So the dump will be large.

Generally, this command should be used only at the direction of ColeSoft programmers.

The current status of the SET DUMP setting can be displayed by the **LIST DUMP** command.

Syntax:

```
SET DUMP DISABLE
         ABEND=SYSTEM  INSTANCE=nn  LIMIT=nn
              =USER
              =ANY
              =Shhh
              =Udddd
```

Rules:

- At least one operand is required.
- Operands may be given in any order.
- When dumps are already disabled, the ABEND= operand is required to enable them.
- The DISABLE operand is mutually exclusive with all other operands.
- Whenever the ABEND= operand is given, the count of eligible ABENDs that have occurred so far is zeroed. (This affects the operation of the INSTANCE= and LIMIT= operands.)

DISABLE
If dumps are enabled, then this operand disables them. (If dumps are already disabled, then this operand has no affect.) This operand is mutually exclusive with all other operands.

ABEND=ANY
This operand enables dumps for any eligible ABEND regardless of the ABEND code.

ABEND=SYSTEM

This operand enables dumps for any eligible ABEND having a system ABEND code (Shhh).

ABEND=USER
This operand enables dumps for any eligible ABEND having a user ABEND code (Unnnn).

ABEND=Shhh
This operand enables dumps for any eligible system-code ABEND whose ABEND code matches the given **Shhh** value, where:
- **S** is the letter "S" and indicates that a system ABEND code is being given.
- **hhh** is a 3-digit hexadecimal number.

ABEND=Unnnn
This operand enables dumps for any eligible user-code ABEND whose ABEND code matches the given **Unnnn** value, where:
- **U** is the letter "U" and indicates that a user ABEND code is being given.
- **nnnn** is a 4-digit decimal number.

INSTANCE=nn
After an eligible ABEND passes filtering by the ABEND= information, it is counted. When the count reaches this INSTANCE= value, a dump is produced.

nn must be a decimal number ranging between 1 and 100.

When this operand is omitted and the ABEND= operand is given, then the default INSTANCE= value is **=1.**

LIMIT=nn
This operand limits how many dumps can be taken. **nn** must be a decimal number ranging between 1 and 10.

When this operand is omitted and the ABEND= operand is given, then the default LIMIT= value is **=1.**

After the SET DUMP command is processed, a **LIST DUMP** command is internally issued to display for you the current status of ABEND dump filtering.

Examples:

SET DUMP A=S0C4 I=3 L=2

This command enables dumps for ABENDs that may occur within the LE environment or C subroutines that are invoked by Z/XDC. Only s0C4 ABENDs are filtered for and counted. (All other ABENDs that may occur in this interface are messaged (DBC834W) but then are ignored and suppressed.)

Dumps are taken only for the 3rd and 4th s0C4 that may occur after this command is issued. Dumps for the 1st, 2nd, 5th and all subsequent ABENDs are suppressed.

Also suppressed are dumps for all ABENDs that are not s0C4’s.

SET DUMP I=7 l=1
This changes the INSTANCE= and LIMIT= filters without resetting the ABENDs

occurrence counter.

Help COmmands SEt EXits

The **SET EXITS** command can be used to make Z/XDC's default **Miscellaneous Exits** module (XDCEXITS) available for use in situations where it is not automatically loaded and used by Z/XDC. For more information about the Miscellaneous Exits module, see HELP EXITS MISCXITS.

Generally, XDCEXITS is loaded only when a debugging session is started via the XDCCALL[A] utility.

If XDCEXITS has been preloaded by XDCCALL[A], then **SET EXITS** has no effect.

Changes to the availability of the **Miscellaneous Exits** module take effect immediately after the completion of the next Z/XDC **TRACE, STEP** or **GO** command.

The current status of the SET EXITS setting can be displayed by the **LIST EXITS** command.

Syntax:

```
SET EXITS ACTION=INSTALL
                =REMOVE
```

Rules:

- The ACTION operand is required.

ACTION=INSTALL
If a **DBCPARM** block does not exist, or if one does but it does not provide the entry point to a Miscellaneous Exits module, then **ACTION=INSTALL** requests that Z/XDC load the default Miscellaneous Exits module **(XDCEXITS)** into storage thereby making it available for use by Z/XDC going forward.

Note, the XDCEXITS module implements the **LIST TIOT** command, so if it hadn't been working before, it will start working after the next issued GO, STEP or TRACE command.

ACTION=REMOVE
This operand reverses the effect of a successful **ACTION=INSTALL.**

Examples:

SET EXITS A=I
This command enables the facility if it is not already available.

SET EXITS A=R
This command disables the **Miscellaneous Exits Interface** if previously enabled via **SET EXITS A=I.**

Help COmmands SEt FLc

The **SET FLC** command can be used to change the current Function Leader Character. This is a character that sometimes must precede a function name appearing within an address expression. For example, if tilde (~) is the current FLC character, then in the following address expression, the tilde notifies Z/XDC that the word following the tilde ("EXTRACT") is the name of a function:

```
DISPLAY .TCBTIO%+~EXTRACT(.DCBTIOT,2)
```

(Note, assuming that the proper dsect maps have been loaded and assigned appropriate base addresses, this address expression displays the TIOT DD entry associated with a given OPEN'd DCB.)

The FLC character often is optional in address expressions because if it is possible for Z/XDC to distinguish a function name by context, then it will do so.

Syntax:

```
SET FLC character
```

character
This is a required operand. It must be a single character to be used as the Function Leader Character in all subsequent address expressions. It may only be one of the following:

- **¢** - Cent sign
- **`** - Back accent
- **~** - Tilde

The factory default FLC character is the tilde (~). The current FLC character can be displayed via the LIST FLC command.

The FLC character can be saved in your session profile, and so it can be both displayed and set via the Profile Menuing System. See HELP PROFILES MENU for more information.

Example:

```
SET FLC `
```

This sets the FLC to a back accent. This new value remains in effect for the rest of your debugging session. If you then issue a PROFILE SAVE command, then this value becomes your personal default for all future debugging sessions.

Help COmmands SEt FOrmat

The **SET FORMAT** command affects the displays produced by the **DISPLAY, FIND, FORMAT, SHOW** and **WHERE** commands. It is used to set the following storage formatting related controls:

- Whether formatted displays of source mapped areas are to be displayed as source image statements **(SOURCE)** or as disassembled machine instructions **(OBJECT)** or as both **(BOTH).**

- In source image displays, whether code from macro expansions is to be included in the display **(SHOWMCODE** or **CURRENTMCODE)** or suppressed from the display **(HIDEMCODE).**

- Whether locations (in the display's address column) are to be shown as **ADDRESSES** or as **OFFSETS** into an appropriate object.

- Whether the displacement fields of machine instructions are displayed as **DECIMAL** or **HEXADECIMAL** values.

- Whether text is to be derived from an **EBCDIC** interpretation or an **ASCII** interpretation of the underlying hex values. This setting affects the following:
  - How the text portion of hex/text style storage displays are shown by the DISPLAY, FORMAT [E]WHERE and SHOW commands
  - How text string operands are translated by the ZAP, VERIFY and FIND commands
  - How the **UC** operand of the **FIND** command will perform upcasing.

- Whether data displayed by the various storage displaying commands **(DISPLAY, FORMAT, etc.)** will display:
  - In a **NARROW** format (4 words per line) for 80 character 3270 displays.
  - Or a **WIDE** format (8 words per line) when wider geometries are set up.

- Whether the initial formatting bias for FORMAT, WHERE, and SHOW commands will be towards **INSTRUCTION** or **DATA** or neither **(NOBIAS).**

Syntax:

```
SET FORMAT SOURCE ADDRESSES DECIMAL     EBCDIC WIDE   INSTRUCTION ...
           OBJECT OFFSETS   HEXADECIMAL ASCII  NARROW DATA
           BOTH                                       NOBIAS

       ... SHOWMCODE     POINTERS
           HIDEMCODE     NOPOINTERS
           CURRENTMCODE
```

Notes:

- All operands are optional, but at least one operand must be given.
- Operands may be given in any order.
- Operands appearing above within the same column are mutually exclusive.
- All operands can be abbreviated down to the shortest, non-ambiguous length (one

two or three characters). However:
- **O** is coerced to mean **OFFSETS** (not OBJECT).
- **N** is coerced to mean **NARROW** (not either NOBIAS or NOPOINTERS).

Operands:

SOURCE
OBJECT
BOTH
These operands set how source image maps are used when storage is displayed from within a mapped program (source maps or csect maps) or a mapped control block (dsect maps).

- **SOURCE**

If a source level map **(ADATA** or **DWARF** data) is available for a program or control block, then if a storage formatting command **(FORMAT, WHERE** or **FIND)** is used to display storage occupied by that program or control block, then that storage will be formatted according to information contained in the source level map; i.e. source level language statements will be displayed.

If a storage formatting command is used to display storage **not** described by a source level map, then the formatter will fall back to using its disassembly process just as if **SET FORMAT OBJECT** had been specified.

- **OBJECT**
This causes Z/XDC's storage formatter to ignore available source image statements. Instead, Z/XDC disassembles storage. Notes:

  - If the storage is covered by any map, be it **SYM** data, **ADATA,** or **DWARF** data, the disassembled display will be annotated by whatever statement names and field names the map might contain.

  - If any **equate** names have been assigned to the storage being disassembled, then they will be displayed as well.

- **BOTH**
If a source level map has been loaded describing the storage being displayed, then Z/XDC's storage formatter will display storage using information **both** from the source level map and from its disassembly process. The source level map statements will appear interspersed with disassembled machine instructions.

For the SOURCE/OBJECT/BOTH group of operands, **SOURCE** is the factory default.

SHOWMCODE
HIDEMCODE
CURRENTMCODE
These operands are effective only when **SOURCE** is in effect.

These operands control whether or not source image displays formatted via **ADATA** maps (for Assembler programs) and **DWARF** data maps (for C programs) will include the display of the machine code existing behind the Assembler macro expansions or C language statements.

- **SHOWMCODE:** The machine code underlying all Assembler macros and C statements is shown.

- **HIDEMCODE:** The underlying machine code is not shown for any Assembler macro or C statement.

  (Data generating Assembler macros, such as control block mapping macros, are unaffected.)

- **CURRENTMCODE:** The underlying machine code is shown only for the currently executing Assembler macro or C statement. The machine code is suppressed for all other macros and statements.

  (Again, data generating Assembler macros, such as control block mapping macros, are unaffected. They are never suppressed.)

Note the following:

- The intent of the **CURRENTMCODE** operand is to improve the display of Assembler code written using structured programming macros. It also works well for C|C++ language statements.

- The showing or hiding of macro expansion code affects only formatted displays from the FORMAT, WHERE, EWHERE and FIND commands. It does **not** affect the **actions** of the various TRACE commands.

  For example, **TRACE, TRACE BY** and **TRACE BNC** remain machine instruction driven. They do not become macro driven or C|C++ statement driven.

- (For stepping through C|C++ programs, you are better off using the **STEP** command.)

- Also, the hiding of macro expansions is effective only for storage formatted via **ADATA** maps (for Assembler) and **DWARF** data maps (for C). Storage formatted via **SYM** data maps (Assembler) is unaffected.

- The hiding of macro expansions does not occur for dsect maps that map data areas and control blocks.

- **CURRENTMCODE** is the factory default for the **A80** and **AWIDE** reset profiles.

- **HIDEMCODE** is the factory default for the **C80** and **CWIDE** reset profiles.

For detailed information about what happens when Assembler macro expansions are suppressed, see **HELP MAPS ADATA MACROS.**

ADDRESSES
Storage display locations (the leftmost column of a storage display) are shown as virtual addresses.

OFFSETS
O
Storage display locations are shown as offsets into an appropriate object such as a load module, a csect, a dsect, or an Area Equate.

The rules are pretty complex that Z/XDC follows when deciding what object to use for an offsets base, but the **overriding rule** is this: If one or more of the objects covering the display location is an **Area Equate** that is at least **32 bytes** long, then the displayed offset will be based upon whichever of those equates has the nearest preceding starting address.

This means that if Z/XDC's choice for an offsets base is not what you would prefer, then you can change that choice simply by using the **EQUATE** command to create an Area Equate having whatever name, base address and area length you like.

For the ADDRESSES/OFFSETS pair of operands **OFFSETS** is the factory default.

DECIMAL
In formatted displays, the displacement fields of all machine instructions are shown as decimal numbers.

HEXADECIMAL
In formatted displays, the displacement fields of all machine instructions are shown as hexadecimal numbers.

For the DECIMAL/HEXADECIMAL pair of operands **HEXADECIMAL** is the factory default.

ASCII
EBCDIC
In general, these operands control how Z/XDC interprets text strings. This affects the following:

- Text strings used on the ZAP, VERIFY, and FIND commands.
- The displays of character data produced primarily by the DISPLAY command and also produced by the FORMAT, SHOW, WHERE, and FIND commands.
- The translations used by the **Z** shortcut when overtyping character data for the purposes of zapping that data.
- The **UC** operand of the **FIND** command.

Note, when Z/XDC displays ASCII data, it frames the display with vertical bars (|). When it displays EBCDIC data, it frames the display with asterisks (∗). Examples:

- **EBCDIC** text is displayed like this: **∗...TEXTSTUF..3.h∗**
- **ASCII** text is displayed like this: **|...TEXTSTUF..3.h|**

EBCDIC is the factory default.

More information on EBCDIC and ASCII can be found at **HELP CHARACTERSETS.**

WIDE
NARROW
N
When registers or storage are displayed in a hex-text format, this setting controls whether the displayed lines will show up to **4 words** (16 bytes) of information per line (NARROW) or up to **8 words** (32 bytes) of information (WIDE). NARROW is suitable for terminals that display only **80 characters** per line. WIDE is best used when the terminal displays **136 characters** or more per line. See **HELP FULLSCREEN TERMINALS** for

more information.

The factory default depends upon which reset-level session profile is selected by the **PROFILE RESET** command:

- **NARROW** is the default when **A80** or **C80** is selected.
- **WIDE** is the default when **AWIDE** or **CWIDE** is selected.

Your personal default will depend, of course, on the setting made by your personal default profile.

INSTRUCTION
DATA
NOBIAS

The **FORMAT, WHERE,** and **SHOW** commands generally attempt to format storage contents as instructions, but this bias can be influenced by many factors, such as whether or not a PSW points to the storage being formatted, as well as the attributes of maps, fields, and equates that label the storage being displayed. As a part of the management of this process, these commands maintain a concept called the **formatting bias** which, for a given displayed line, retains information about how the preceding line was formatted and, therefore, contributes information about how the current displayed line should be formatted. In other words, if the prior displayed line was formatted as data, then the formatting command will be biased towards formatting the current displayed line also as data (unless new information, such as label attributes, suggests otherwise).

The **INSTRUCTION, DATA,** and **NOBIAS** operands set the default initial formatting bias for the **FORMAT, WHERE,** and **SHOW** commands as follows:

INSTRUCTION
The command's initial formatting bias will be to interpret storage contents as being machine instructions. This will be done even if other factors might suggest that the storage contents should be interpreted as data. This will **not** be done, however, if the initial opcode is invalid.

DATA
The command's initial formatting bias will be to interpret storage contents as being data. This will be done even if other factors might suggest that the storage contents should be interpreted as machine instructions.

NOBIAS
The command's initial formatting bias will not be forced. Absent other factors, the storage contents will initially be interpreted as machine instructions, but if other factors suggest otherwise, then those factors won't be ignored.

For the **FORMAT** and **WHERE,** commands, the **INSTRUCTION, DATA,** and **NOBIAS** operands set only the **initial** formatting bias. Once the display gets going, the formatting bias for the second and subsequent display lines will be influenced only by the normal factors discussed above.

In the case of the **SHOW** command, the initial formatting bias will be effective for every display line generated (not just the first). This is because the SHOW command's output is really just a collection of 1-line displays.

For the **INSTRUCTION/DATA/NOBIAS** group of operands **NOBIAS** is the factory default.

POINTERS
NOPOINTERS
When a formatted display **(FORMAT, [E]WHERE** and **SHOW)** is produced, if the display contains 3-byte, 4-byte and/or 8-byte wide data fields containing addresses that point to other locations ("pointer fields"), then the **[NO]POINTERS** operand controls whether or not those other locations are resolved and displayed as comments.
- **POINTERS:** The pointed-to locations are resolved and identified.
- **NOPOINTERS:** The pointed-to locations not are resolved and identified.

The factory default is **POINTERS.**

Z/XDC will attempt to resolve pointer fields only when it is formatting storage using its built-in disassembler. When storage is formatted using source code images, pointers are not resolved. See **HELP PROFILES MENU FORMATTRACEFIND POINTERRESOLUTIONS** for more details.

When very complex structures of **floating maps and equates** are present, the **POINTERS** setting may result in slower displays. In that case, you may wish to use **NOPOINTERS** instead. (For more information about floating maps and equates, see **HELP MAPS FLOATING.)**

Final Comments

Most of the storage display commands **(DISPLAY, FORMAT, SHOW, WHERE,** and **FIND)** accept operands that allow these settings to be temporarily overridden when the command is used.

The current settings can be saved in your session profile.

The current storage formatting settings can be displayed via the **LIST FORMAT** command.

The current settings can also be displayed by the **Profile Menuing System.** See **HELP PROFILES MENU FORMATTRACEFIND** for more information.

Help COmmands SEt HICOLOR

This command has been removed and not replaced. Its functionality was made obsolete by the demise of hardware 3270s having limited functionality. These days, all 3270s are emulated by software (PCOMM, Vista, Attachmate, etc.) as having full functionality.

Help COmmands SEt HICOLOUR

This command has been removed and not replaced. Its functionality was made obsolete by the demise of hardware 3270s having limited functionality. These days, all 3270s are emulated by software (PCOMM, Vista, Attachmate, etc.) as having full functionality.

Help COmmands SEt HILight

Screen images generated by Z/XDC contain up to four different kinds of fields:

(A) - **Low** intensity **input** fields for displaying data in secondary input areas and in overwritable fields (such as storage displays).

(B) - **High** intensity **input** fields for primary input areas such as command lines.

(C) - **Low** intensity **protected** fields for displaying nonoverwritable information.

(D) - **High** intensity **protected** fields for displaying title and header lines, warning messages, etc.

For those terminals that have the necessary hardware support, display colors and extended hilighting can be set by the following commands:

- **SET COLORS**
- **SET HILIGHT**
- **SET INTENSITY**

The **SET HILIGHT** command defines the desired extended hilighting.

Syntax:

SET HILIGHT abcd

abcd
This must be a 4-character string. Each character position corresponds to a field type as listed above. Each character must be one of the following codes:

B - Causes the corresponding field type to blink.
R - Sets the corresponding field type to reverse video.
U - Underlines the corresponding field type.
D - Restores the corresponding field to its hardware default: no hilighting.

The current extended hilight settings can be displayed by any of the following commands:

- **LIST HILIGHT**
- **LIST TERMINAL**
- **PROFILE**

The extended hilights can also be both displayed and set by Z/XDC's Profile Menuing System.

Your hilighting choices can be saved in your profile dataset for automatic restoration every time you start a debugging session. For more information, please see **HELP PROFILES.**

The factory default extended hilight settings are **DDDD** (no hilighting).

Example:

SET HILIGHT BBRR;L TERMINAL
The **SET HILIGHT** command causes input fields to blink (not recommended unless you like headaches), and it causes protected fields to appear in reverse video.

The **LIST TERMINAL** command then immediately reports the results.

Note: Even if you have a terminal that supports extended hilighting, your choices can be ignored under the following circumstances:

- Your VTAM Systems Programmer has not properly defined your terminal as supporting "Extended Attribute Bytes" (done via a suitable **PSERVIC=** value on a **MODEENT** macro in a VTAM **mode table** definition).

- You are using a PC workstation program to emulate a terminal, but you have not properly configured your emulation to support "hilight orders" from VTAM.

See **HELP FULLSCREEN TERMINALS** for more information and for suggestions about remedying these problems.

Help COmmands SEt HKeys

Whenever a Built-in Help topic (such as this one) is displayed, Z/XDC temporarily suspends your normal PF key definitions and provides a special set of keys that are appropriate for navigating through Built-in Help displays. In addition, the special keys are displayed in a 5-line area at the bottom of the terminal’s screen.

The **SET HKEYS** command can be used to control both whether or not these special PF key definitions are used during Built-in Help displays, and whether or not the definitions are displayed at the bottom of the screen.

SET HKEYS also can be used to change the meanings of individual PF key definitions used during Built-in Help displays.

Syntax:

```
SET HKEYS ON   SHOW
          OFF  NOSHOW

SET HKEYS nn text
             'text'
```

SET HKEYS ON
This enables the use of special PF key definitions during Built-in Help displays.

SET HKEYS OFF
This disables the use of special PF key definitions during Built-in Help displays. Your normal PF keys will be available for use instead.

SET HKEYS SHOW
If HKEYS are **on,** then this causes the HELP PF key nicknames to be displayed in the bottom five lines of your terminal's screen. If HKEYS are **off,** then "SET HKEYS SHOW" is ignored and the PF key nicknames remain undisplayed.

SET HKEYS NOSHOW
This suppresses the display of the HELP PF key nicknames at the bottom of the screen. This creates a larger display area for the Built-in Help topics.

SET HKEYS nn text
SET HKEYS nn 'text'
You can use this form of the SET HKEYS command to assign a command string to a specific PF key.

nn
This is the decimal number of the HELP PF key to which the given command text is to be assigned. The PF key number must fall within the range of 1 through 12.

text
'text'
"text"
This is the command string to be assigned to the specified PF key. It must be separated from **nn** by one or more blanks. It may be either quoted or unquoted:

- **text**
  If the text is **not** framed by quote characters, then it may not contain semicolons (;) or colons (:), and it will **not** be upcased and embedded blank strings will be compressed down to single blanks. Also, it may contain embedded quote characters If it does, then they need not be doubled.

- **'text'**
- **"text"**
  If the text **is** framed by quote characters, then it will not be upcased and it may contain semicolons and colons. The framing quote characters will be removed, and embedded doubled quote characters will be singlized.

For more information, see HELP COMMANDS SYNTAX CHARACTERSTRINGS.

The **SET HKEYS** command should be given with operands. If operands are **not** given, then this command becomes an alias of the **HKEYS** command, and is processed accordingly. See HELP COMMANDS HKEYS for more information.

Your HKEYS settings can be saved in your profile dataset for automatic restoration every time you start a debugging session. For more information, please see:

HELP PROFILES
HELP COMMANDS PROFILE

Z/XDC's factory default is **SET HKEYS ON SHOW.** The factory defaults for all profiled settings can be loaded with the **PROFILE RESET** command.

Help COmmands SEt ILC

When virtual storage is displayed in fullscreen mode, Z/XDC permits the user to zap instructions and data by simply typing a **Z** at the left of a display line and then overtyping various parts of the displayed data. In particular, if a display includes machine instructions, the instruction opcodes can be changed just by typing the new opcode's mnemonic directly on top of the old opcode's mnemonic. Example: a displayed BCTR instruction can be changed to an LR instruction simply by typing "LR" on top of the "BCTR" in the display.

When machine instruction opcodes are zapped in this manner, the **SET ILC** and **SET NOILC** commands control whether Z/XDC will (ILC) or will not (NOILC) enforce a restriction that the new machine instruction must be of the same length as the old one.

Although **SET NOILC** will permit you to, for example, change a "ST" instruction into an "MVC", it does **not** do so by shifting the following instructions in order to make room. Instead, what had been the first halfword of the following instruction simply becomes (in this example) the last halfword of the MVC. Note that in general this is not a desirable result and additional zapping would probably have to be done to produce a correct sequence of machine instructions.

The **SET ILC** and **SET NOILC** commands have no effect when machine instructions are changed via overtyping hex displays.

Syntax:

```
SET ILC
SET NOILC
```

These commands accept no operands.

The status of instruction length checking can be saved in your profile dataset for automatic restoration every time you start a debugging session. For more information, please see:

```
HELP PROFILES
HELP COMMANDS PROFILE
```

The distributed version of Z/XDC has ILC set in the profile.

Help COmmands SEt ILIt

This command can be used (only under dump/XDC) to set the value of an IPCS literal equate.

Syntax:

SET ILIT name value

Operands:

name

The name of the IPCS literal to be set. An IPCS literal name can be up to 31 characters long, and must start with an alphabetic or national character. The rest of the name must consist of alphanumeric or national characters.

Note that names that start with DXDCS are not recommended, as dump/XDC uses literals that start with this value, for internal processing.

value

The value that the IPCS literal is to be set to. The following value formats can be used:

string-no-blanks
the literal value is set to the specified string (there must be no blank characters in the string). The value is upcased.

'quoted string'

"quoted string"
the literal value is set to the specified quoted string (without the quotes). The string can contain blanks and the value is **not** upcased. You can represent the containing quotes in the value by using doubled-up quotes in the string.

Examples

SET ILIT MYLITERAL my-value
The IPCS literal equate called MYLITERAL will be set to the value MY-VALUE.

SET ILIT MYLIT2 'my-value2'
The IPCS literal equate called MYLIT2 will be set to the value my-value2.'

Help COmmands SEt INtensity

Screen images generated by Z/XDC contain up to four different kinds of fields:

(A) - **Low** intensity **input** fields for displaying data in secondary input areas and in overwritable fields (such as storage displays).

(B) - **High** intensity **input** fields for primary input areas such as command lines.

(C) - **Low** intensity **protected** fields for displaying nonoverwritable information.

(D) - **High** intensity **protected** fields for displaying title and header lines, warning messages, etc.

Display colors and extended hilighting can be set by the following commands:

- **SET COLORS**
- **SET HILIGHT**
- **SET INTENSITY**

The **SET INTENSITY** command can be used to override the normal field display intensities (bright vs dim); **however,** intensity settings will be honored only for those fields whose **colors** are set to **"DEFAULT"** (hardware default, not factory default).

The **LIST INTENSITY** command displays the current settings for field intensities.

Syntax:

SET INTENSITY abcd

abcd
This must be a 4-character string. Each character position corresponds to a field type as listed above. Each character must be one of the following codes:

H - Displays the corresponding field with high intensity (bright).
L - Displays the corresponding field with low intensity (dim).
D - Allows the corresponding field to be displayed with a system default intensity.

The current intensity settings can be displayed by any of the following commands:

- **LIST INTENSITY**
- **LIST TERMINAL**
- **PROFILE**

The intensity settings can also be both displayed and set by Z/XDC's **Profile Menuing System.** They will be shown on the **03 Color and Hilighting** submenu.

Your intensity choices can be saved in your profile dataset for automatic restoration every time you start a debugging session. For more information, please see **HELP PROFILES.**

The factory default intensity settings are **DDDD** (use system defaults for all fields).

Example:

SET COLORS DDDD
SET INTENSITY HHLL
LIST TERMINAL

The **SET COLORS** command makes way for the **SET INTENSITY** command. Intensities will be honored only for those fields whose "color" is **D.**

The **SET INTENSITY** command sets **all** input field to display bright and all protected fields to display dim.

The **LIST TERMINAL** command reports all terminal related settings, including the

intensity settings.

To report just the intensity settings, use the **LIST INTENSITY** command.

Help COmmands SEt ISPf

The **SET ISPF** command tells Z/XDC to use ISPF's Display Services for painting its displays to your 3270 terminal (but only when ISPF is available in the Home Address Space).

If ISPF is not currently running (or if Z/XDC is running authorized), then Z/XDC uses fullscreen TPUTs to paint its displays until ISPF becomes available (via Z/XDC's **ISPF** or **TSO ISPF** commands or by other means).

For more information, see:
- **HELP USERINTERFACES**
- **HELP COMMANDS LIST ISPF**
- **HELP COMMANDS LIST TPUT**
- **HELP COMMANDS SET TPUT**
- **HELP COMMANDS TSO**
- **HELP COMMANDS ISPF**

Syntax:

SET ISPF

This command accepts no operands.

Z/XDC's ISPF interface can be turned off by either of the following commands:
- The **SET TPUT** command causes Z/XDC to switch to using fullscreen TPUTs for displaying its panels.
- The **SET TFS OFF** command causes Z/XDC to discontinue using its Fullscreen Support and instead to start using linemode TPUTs for displaying its messages.

When you issue the **SET ISPF** command, if ISPF display services are, for some reason, not currently usable, then Z/XDC will report this fact, but it still will set a flag such that if and when ISPF services do become usable, then Z/XDC will automatically start using them.

Note, ISPF displays are not useable (of course), when ISPF is not present in the TSO session. They also are not useable when Z/XDC is running authorized (an ISPF restriction, not Z/XDC's).

Use the **LIST ISPF** command to see whether or not ISPF services are usable, and if not, then why not.

The choice of using or not using the ISPF fullscreen interface can be saved in your session profile for automatic restoration every time you start a debugging session.

For more information, please see:

HELP PROFILES
HELP COMMANDS PROFILE

Help COmmands SEt ISR@prim

The **SET ISR@PRIM** command establishes the initial panel to be used by ISPF when it is invoked only when **both** of the following are true:

- ISPF is reinvoked recursively via Z/XDC's own **ISPF** command (and **not** via Z/XDC's **TSO ISPF** command),
- And Z/XDC itself is already running under ISPF.

ISR@PRIM, in addition to being this command's name, is also the name of ISPF's home panel.

Syntax:

```
SET ISR@PRIM name
            ISR@PRIM
```

Notes:

The factory default for this setting is **ISR@PRIM.**

Operands:

name
This is a 1-8 character name of an initial panel to be used by ISPF when it is invoked under the recursive conditions described above.

This panel name can be saved in your session profile for automatic restoration every time you start a debugging session. For more information, please see:

HELP PROFILES
HELP COMMANDS PROFILE

ISR@PRIM
This is the name of ISPF's default home panel. To restore this default, issue **SET ISR@PRIM ISR@PRIM.**

Help COmmands SEt Keys

The **SET KEYS** command can be used to define an individual debugging session PF key command or to assign sets of PF key definitions to ranges of terminal function keys. Z/XDC supports terminals having up to 36 function keys. Z/XDC organizes PF key definitions into up to 3 sets of twelve each. For general information about Z/XDC's PF key support, see **HELP FULLSCREEN PFKEYS.**

For information about PF key sets, see **HELP FULLSCREEN PFKEY PFKEYSETS.**

Syntax:

```
SET KEYS   operands
    PFKEYS
```

The **SET KEYS** command is very versatile. It accepts a variety of operands to let you define PF key commands and make PF key set assignments in many different ways. For specific information, type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

ONEKEY - Assigning a command string to a specific PF key.
KEYSPANEL - Displaying a Keys Panel for making fullscreen updates to a specific PF key set.
PFKTOFKEY - Assigning a PF key set to a function key range.
FKEYTOPFK - Assigning a a function key range to a PF key set.
SPECIAL - Using the UPPER, LOWER, SWAP, RESET and ASIS operands.

If the **SET KEYS** command is given without operands, then this command becomes an alias of the the **KEYS** command, and is processed accordingly. (It displays a panel where you can update the various PF key definitions.) See **HELP COMMANDS KEYS** for more information.

Help COmmands SEt Keys Onekey

You can use this form of the **SET KEYS** command to assign a command string to a specific PF key.

Syntax:

```
SET KEYS   nn text
    PFKEYS    'text'
```

nn
This is the decimal number of the PF key to which the given command text is to be assigned. The PF key number must fall within a terminal function key range to which a PF key set has been assigned. (See **HELP FULLSCREEN PFKEYS PFKEYSETS** for information about PF key set assignments.)

text
'text'
This is the command string to be assigned to the specified PF key. It must be separated from **nn** by one or more blanks. It may be either quoted or unquoted:

- **text**
  If the text is **not** framed by quote characters then it will not be upcased, and it may not contain semicolons (;) or colons (:), and embedded blank strings will be compressed down to single blanks. Also, it may contain embedded quote characters. If it does, then they need not be doubled.

- **'text'**
- **"text"**
  If the text **is** framed by quote characters, then it will not be upcased and it may contain semicolons and colons. The framing quote characters will be removed, and embedded doubled framing quote characters will be singlized.

For general information about the syntax of text strings (and an unusually flexible syntax it is), see **HELP COMMANDS SYNTAX CHARACTERSTRINGS.**

Note, Z/XDC's processing of PF key definitions also provides for the automatic or manual inclusion of variable information before executing the PF key's commands. For detailed information about this, see a series of topics starting with **HELP FULLSCREEN PFKEYS.**

Examples:

SET KEYS 2 D R1?
This command assigns "D R1?" to PF key number two. It is assigned to the 2nd PF key definition in whichever PF key set is assigned to the 1-12 (or 1-10, as the case may be) function key range.

SET KEYS 15 D R1?;D R3?
This command assigns "D R1?" to PF key number fifteen (the 3rd PF key definition in whatever PF key set is assigned to the 13-24 function key range), and then it executes the "D R3?" command on the spot. The ";D R3?" part of the command string is **not** assigned to the PF key.

SET KEYS 15 'D R1?;D R3?'
This command assigns "D R1?;D R3?" to PF key number fifteen.

SET KEYS 14 'SWAP -'
When a PF key definition includes a hyphen (-), if the primary command line contains any text at the moment that the key is pressed, then that text is inserted into the string (replacing the hyphen) prior to the string being processed as a command.

SET KEYS 3 'TRAP _;GOT'
When a PF key definition includes an underscore (_), the command is displayed on the current command line, with the underscore removed, and the cursor positioned to the underscore's location. The user can then insert any desired additional text and then press ENTER to cause the modified command string to be executed.

Help COmmands SEt Keys Keyspanel

You can use this form of the **SET KEYS** command to bring up the Keys Panel for a specific PF key set. The Keys Panel is a fullscreen display that shows you all twelve command definitions within a PF key set and allows you to change those definitions just by overtyping them.

Unlike the **KEYS** command, this form of the **SET KEYS** command allows you to display a PF key set regardless of whether or not the set has been assigned to a PF key range. (The **KEYS** command, on the other hand allows you to rotate through all assigned sets.)

Syntax:

```
SET KEYS   SET-A
    PFKEYS SET-B
           SET-C
           omitted

Aliases: SET KEYS SET-A
                  SETB
                  SETC
                  A
                  B
                  C
```

SET-A or **SET-B** or **SET-C**
This identifies the PF key set to be displayed by the Keys Panel.

omitted
If no operand is given, then the **SET KEYS** command becomes an alias of the **KEYS** command, and is processed accordingly: Key Panels of all assigned PF key sets are displayed in rotation. Unassigned PF key sets are not displayed.

If the **SET KEYS omitted** command is issued from a Built-in Help display, then the **Help** PF keys are displayed instead.

Examples:

SET KEYS C
KEYS C
These two commands are synonyms. They both display PF key **SET-C** via the Keys Panel.

SET KEYS
KEYS
Both of these commands allow you to rotate through displays of all assigned PF key sets.

Help COmmands SEt Keys Pfktofkey

You can use this form of the **SET KEYS** command to assign a specific PF key set to a particular range of terminal function keys. You can also use this form to clear a set assignment from a particular function key range.

The PF-keys-to-function-keys and function-keys-to-PF-keys forms of the **SET KEYS** command produce the same result when used to assigned a PF key set. They produce differing results, however, when used to clear a range or drop a set assignment.

If the specified assignment creates a logical conflict, then other assignments are automatically adjusted until the conflict is resolved. For example, if PF key sets previously were assigned to the various 12-key ranges, and this command is issued to assign a PF key set to a 10-key range, then all assignments to 12-key ranges are automatically moved where possible, to corresponding 10-key ranges.

It is permissible to assign the same PF key set to more than one function key range.

When this command completes, it produces a display showing the new PF key set assignments.

Syntax:

```
SET KEYS   range set-name
    PFKEYS       CLEAR
```

range
This names the range of terminal function keys to be cleared or to which the assignment is to be made. Some ranges are 12 keys wide and others are only 10 keys wide. You should use the key ranges that correspond to the actual function keys that are present at your terminal. See **HELP FULLSCREEN PFKEYS PFKEYSETS** for more information. "Range" can be any of the following:

12-KEY RANGES
01-12 (Alias: **1-12)**
13-24
25-36

10-KEY RANGES
01-10 (Alias: **1-10)**
11-20

13-22

set-name
This names the PF key definitions set to be assigned to the specified function key range. It can be any of the following:

SET-A (Aliases: **SETA** or **A)**
SET-B (Aliases: **SETB** or **B)**
SET-C (Aliases: **SETC** or **C)**

CLEAR (Alias: **DROP)**
This sets the specified function key range to "unassigned". If any PF key set had been assigned to the key range, then that assignment is discontinued. (The set's PF key definitions themselves are **not** affected.) Subsequently, if you try to use any of the function keys in the cleared range, that attempt will fail with an error message.

Help COmmands SEt Keys Fkeytopfk

You can use this form of the **SET KEYS** command to assign a specific range of terminal function keys to a particular PF key set. You can also use this form to disconnect a particular PF key set from all function key ranges to which it has been assigned.

The function-keys-to-PF-keys and PF-keys-to-function-keys forms of the **SET KEYS** command produce the same results at the **KEYS** command when used to assigned a PF key set. They produce differing results, however, when used to clear a range or drop a set assignment.

If the specified assignment creates a logical conflict, then other assignments are automatically adjusted until the conflict is resolved. For example, if PF key sets previously were assigned to the various 12-key ranges, and this command is issued to assign a PF key set to a 10-key range, then all assignments to 12-key ranges are automatically moved where possible, to corresponding 10-key ranges.

It is permissible to assign the same PF key set to more than one function key range.

When this command completes, it produces a display showing the new PF key set assignments.

Syntax:

SET KEYS set-name range
PFKEYS DROP

set-name
This names the PF key definitions set to be assigned to the specified function key range. It can be any of the following:

SET-A (Aliases: **SETA** or **A)**
SET-B (Aliases: **SETB** or **B)**

SET-C (Aliases: **SETC** or **C)**

range
This names the range of terminal function keys to be cleared or to which the assignment is to be made. Some ranges are 12 keys wide and others are only 10 keys wide. You should use the key ranges that correspond to the actual function keys that are present at your terminal. See **HELP PROFILES MENU PFKTOFKEYMAP** for more information. "Range" can be any of the following:

12-KEY RANGES
01-12 (Alias: **1-12)**
13-24
25-36

10-KEY RANGES
01-10 (Alias: **1-10)**
11-20
13-22

DROP (Alias: **CLEAR)**
This drops the specified PF key set from all function key ranges to which it has been assigned. (The set's PF key definitions themselves are **not** affected.) Subsequently, if you try to use any of the function keys in the affected ranges, that attempt will fail with an error message.

Help COmmands SEt Keys Special

The following forms of the **SET KEYS** command are holdovers from more limited PF key support implemented in prior releases of Z/XDC. In Z/XDC's current release these **SET KEYS** commands create certain specific PF key set to range assignments.

Syntax:

```
SET KEYS   ASIS
    PFKEYS SWAP (alias: RESET)
           UPPER
           LOWER
```

SET KEYS ASIS
This command makes the following assignments:

12-KEY TERMINALS	10-KEY TERMINALS
01-12: SET-B	01-10: SET-B
13-24: SET-A	11-20: SET-A or cleared
25-36: cleared	13-22: SET-A or cleared

SET KEYS SWAP (alias: **RESET)**
This command makes the following assignments:

12-KEY TERMINALS	10-KEY TERMINALS
01-12: SET-A	01-10: SET-A
13-24: SET-B	11-20: SET-B or cleared

25-36: cleared 13-22: SET-B or cleared

This is the key set assignment map that is established by the **PROFILE RESET** command. For more information, see **HELP COMMANDS PROFILE RESET.**

SET KEYS UPPER
This command makes the following assignments:

12-KEY TERMINALS	10-KEY TERMINALS
01-12: SET-A	01-10: SET-A
13-24: SET-A	11-20: SET-A or cleared
25-36: cleared	13-22: SET-A or cleared

SET KEYS LOWER
This command makes the following assignments:

12-KEY TERMINALS	10-KEY TERMINALS
01-12: SET-B	01-10: SET-B
13-24: SET-B	11-20: SET-B or cleared
25-36: cleared	13-22: SET-B or cleared

Z/XDC guesses whether a terminal has 10 function keys or 12 by checking the PF key assignments prior to processing the command. If no prior assignments exist, then Z/XDC just assumes that the terminal has 12 function keys.

Help COmmands SEt LInes

The **SET LINES** command sets the default number of display lines to be generated by the DISPLAY, FORMAT, and FIND commands.

This command can be issued at any time, but its setting will be honored only when Z/XDC's fullscreen support is turned off (SET TFS OFF command). When Z/XDC's Fullscreen Support is turned on, this value is overridden by display window sizes.

Syntax:

SET LINES nn

nn
This must be a decimal number between 0 and 99. When 0 is given, only display header lines are generated.

Example:

S L 19
The default display size is set to 19 lines.

Help COmmands SEt LOCks

When:

- Z/XDC is running as an FRR,
- Or when Z/XDC is running as a TRAP handler for a task-mode routine that has at least one defined EUT FRR,
- Or when Z/XDC is running as a TRAP handler for an SRB routine,

The **SET LOCKS** command can be used to control the set of System Locks that Z/XDC will acquire or reacquire if execution of the user's program is resumed (via TRACE or GO/GOT/GOX commands).

The **SET LOCKS** command can be used only when Z/XDC is being used in a **restricted** execution environment. Examples of such environments include:

- When Z/XDC is running as an **FRR** recovery routine,
- When Z/XDC is running as a **TRAP** handler in an environment where ABEND protection by an FRR routine has been established,
- Or when Z/XDC is running in any other environment where you must use **ADDFRR** to establish ABEND protection.

You cannot use the **SET LOCKS** command when Z/XDC is running as an ESTAE-type recovery routine (ESTAE, ESTAEX, ESTAI, FESTAE, ARR).

NOTICE! Whenever Z/XDC receives control, if the program being debugged held locks, those locks are always released. If Z/XDC receives control as an ESTAE-type recovery routine, then the locks are released by the System. If Z/XDC receives control as an FRR or a TRAP handler, then they are released by Z/XDC itself.

For the reasons why Z/XDC releases all locks, and for the resulting consequences and considerations, see HELP DEBUGGING LOSTLOCKS.

Use the **LIST LOCKS** command to display what locks were held at ABEND time and what locks will be reacquired at program resumption time. Use the SET LOCKS command to change the set of locks that will be reacquired.

Syntax:

```
SET LOCKS RESET QUIET ON  locknames ... ON  ...
                      OFF              OFF
```

Notes:

- Although all operands are optional, at least one operand must be given.
- Command processing is from left to right. If conflicting operations are requested, then the rightmost request will prevail.

RESET
This restores the set of locks to be restored to that which existed at the time Z/XDC received control. This operand undoes all prior SET LOCKS commands issued since the last prior TRACE command, GO/GOT/GOX command, or user program ABEND.

When given, this operand must occur prior to all ON and OFF operands.

QUIET
Normally, when it completes, the SET LOCKS command stacks a LIST LOCKS command to display the changes that have occurred. This QUIET operand can be used to suppress the stacking of the LIST LOCKS command.

This operand can occur anywhere within the command string.

ON and **OFF**
These operands control what lock settings are to be turned on (acquired) or left off (not acquired) at user program resumption time. Notes:

- Both the ON and OFF operands may be given any number of times in the command string.
- Multiple locknames may be given following each occurrence of an ON or OFF operand.
- Each occurrence of ON and OFF must be followed by at least one lockname.
- Command processing is from left to right. If conflicting operations are requested, then the rightmost request will prevail.
- If a lock to be turned on has a prereq, then that prereq will be set to be turned on as well. Example:
  - If a CMSxxxxx lock is selected to be turned on, and if neither the LOCAL lock nor a CML lock is set, then the LOCAL lock will be set to be turned on.
- If a lock to be turned on is mutually exclusive with other locks, then those other locks will be set to to be left off. Example:
  - If the CMSSMF lock is selected to be turned on, then all other CMSxxxxx locks will be set to be left off.
- If a lock that is set to be left off has dependent locks, then the dependent locks will be set to be left off as well. Example:
  - If the LOCAL lock is selected to be left off, then any CMSxxxxx lock that was set to be turned on will now be set to be left off.

locknames
These are the names of the locks to be turned on or off. The currently supported locknames are:

LOCAL
This manipulates the local lock. Special notes:
- For ON processing:
  - If a CML lock was set to be turned on, it will now be set to be left off.
- For OFF processing:
  - If any CMS lock was set to be turned on, then it will now be set to be left off.

CML
CML=asid
This manipulates a cross memory local lock. Special notes:
- For ON processing:
  - The **CML=asid** form of this operand must be used.
  - **asid** must identify the address space whose local lock is to be acquired. It may be an asid number, an address space's name, or any other form that would be recognized by the SET ASID command. See HELP COMMANDS SET ASID for more information.
  - If **asid** resolves to the home address space, then the LOCAL lock is set to be turned on instead.
  - Otherwise, if the LOCAL lock is set to be turned on, then it is reset instead to be left off.
- For OFF processing:
  - The **CML** form of this operand must be used. (The CML=asid form may not be used.)
  - If a CMSxxxxx lock is set to be turned on, then it is reset to be turned off.

CMS
CMSEQDQ
CMSLATCH
CMSSMF
CMSALL
These are the various cross memory services locks currently recognized by IBM's SETLOCK macro. Special notes:
- For ON processing:
  - These locks (including CMSALL) are mutually exclusive. If any one is selected to be turned on, then all of the others are set to be left off.
  - If neither the LOCAL lock nor a CML lock is set to be turned on, then the LOCAL lock is reset to be turned on.
- For OFF processing, no special notes are needed. There are no dependent locks to be turned off.

CPU
This is the CPU lock. It does not have any prereq, dependent or mutually exclusive locks, so no special notes are needed regarding its processing.

Help COmmands SEt LOG

The **SET LOG** command is used to set the characteristics and status of Z/XDC's external log file. This command controls:
- Whether the log is written to a DASD dataset or to a SYSOUT file,
- The attributes of the file,
- And whether or not logging is to occur automatically or only upon explicit request.

This command also sets certain characteristics of the Primary Window's Scroll Area. Specifically:
- The capacity (in lines) of the scroll area,
- And whether or not the messages generated by Latent Commands will appear in the Scroll Area.

Syntax:

DASD Related:

```
     SET LOG DSNAME=dsn
             DSNAME
     SET LOG UNIT=unitname
             UNIT
     SET LOG VOLSER=diskname
     SET LOG VOLSER
     SET LOG MODE=APPEND
             MODE=OVERWRITE
```

SPOOL Related:

```
     SET LOG SYSOUT=class
     SET LOG SYSOUT
     SET LOG DEST=remoteid
             DEST
     SET LOG HOLD=YES
             HOLD=NO
     SET LOG OUTLIM=nnn
             OUTLIM=0
```

Logging Management:

```
     SET LOG MODE=AUTO
             MODE=MANUAL
     SET LOG RESET
```

Scroll Area Management:

```
     SET LOG SCROLL=nnn
     SET LOG LATENT='commandsstring'
             LATENT=SHOW
             LATENT=NOSHOW
```

Aliases: The following aliases are deprecated. They are retained solely to maintain backwards compatibility. They will not be further documented herein:

Preferred Operand	Deprecated Aliases

SET LOG DEST=remoteid SET LOG DESTINATION remoteid

SET LOG DEST SET LOG DESTINATION

SET LOG DSNAME=dsn SET LOG DSNAME dsn
SET LOG DATASET dsn

SET LOG HOLD SET LOG HELD
SET LOG HOLD=YES

SET LOG HOLD=NO SET LOG NOHOLD
SET LOG NOTHELD

SET LOG MODE=APPEND SET LOG DISPOSITION APPEND
SET LOG DISPOSITION MOD

SET LOG MODE=AUTO SET LOG AUTO

SET LOG MODE=MANUAL SET LOG MANUAL
SET LOG LOG
SET LOG CMD
SET LOG COMMAND

SET LOG MODE=OVERWRITE SET LOG DISPOSITION OVERWRITE
SET LOG DISPOSITION NEW
SET LOG DISPOSITION OLD
SET LOG DISPOSITION SHR

SET LOG OUTLIM=nnn SET LOG OUTLIM nnn

SET LOG SCROLL=nnnnn SET LOG nnnnn

SET LOG SYSOUT=class SET LOG SYSOUT class
SET LOG CLASS class

SET LOG UNIT=unitname SET LOG UNITNAME unitname

SET LOG UNIT SET LOG UNITNAME

SET LOG VOLSER=diskname SET LOG VOLSER diskname
SET LOG VOLUME diskname

SET LOG VOLSER SET LOG VOLUME

Notes:

- The above syntax description shows the several **SET LOG** operands separated into related groups. However, **all** operands are **mutually exclusive!** This means that the **SET LOG** command accepts at most only one keyword and value at a time. If you need to use multiple keywords, then instead you will have to issue the **SET LOG** command multiple times. So, for example, the following is illegal:
  SET LOG CLASS=A DEST=LOCAL HOLD=NO

  Instead, to accomplish this intent, it is necessary to issue three commands:
  SET LOG CLASS=A

```
SET LOG DEST=LOCAL
SET LOG HOLD=NO
```

- The chosen log status and definitions can be saved in your profile dataset for automatic restoration every time you start a debugging session. For more information, please see:

```
HELP PROFILES
HELP COMMANDS PROFILE
```

- Z/XDC's **factory default** values for the **SET LOG** command are:

```
SET LOG MODE=AUTO
SET LOG SCROLL=65535  [64K-1]

SET LOG LATENT='L PSWE;L BEA;L RWREGS'
SET LOG LATENT=NOSHOW

[The following are effective when the log is written to SYSOUT spool]
    SET LOG SYSOUT=X
    SET LOG HOLD=YES
    SET LOG DEST        [i.e. none]
    SET LOG OUTLIM=0

[The following are effective when the log is written to a DASD dataset]
    SET LOG DSNAME *U.*XLOG.*D.*T
    SET LOG MODE=APPEND
    SET LOG VOLSER      [i.e. none]
    SET LOG UNIT        [i.e. none]
```

Subtopics Index

For detailed descriptions (by grouping) of the **SET LOG** command operands, select the following. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

DATASET - Defining a log dataset on a DASD volume.
SYSOUT - Defining a log file on SYSOUT spool.
SCROLLAREA - Managing the Primary Window's Scroll Area
MANAGEMENT - Opening, closing and resetting a Session Log.

Help COmmands SEt LOG Dataset

The following commands are used to define the characteristics of a log when written to a DASD dataset.

When **automatic logging** (SET LOG MODE=AUTO) is in effect, any changes created by the

following SET LOG commands do not take effect immediately. Instead, the changes remain pending until:
- A **SET LOG MODE=AUTO** command is issued,
- Or a **SET LOG MODE=MANUAL** command is issued,
- Or a **SET LOG RESET** command is issued.

When **manual logging** (SET LOG MODE=MANUAL) is already in effect, all changes take effect immediately.

Syntax: (for DASD dataset related SET LOG commands)

```
SET LOG DSNAME=dsn
        DSNAME
SET LOG UNIT=unitname
        UNIT
SET LOG VOLSER=diskname
SET LOG VOLSER
SET LOG MODE=APPEND
        MODE=OVERWRITE
```

Aliases: See HELP COMMANDS SET LOG for a list of all deprecated aliases.

Operands:

SET LOG DSNAME=dsn
This command defines the name of a DASD dataset into which log data is to be written. If the dataset does not exist, it is created.

dsn must be a fully qualified, UNquoted dataset name. It may not include a member name.

dsn may contain variable symbols for which Z/XDC will make substitutions. This allows the dsname to be reactive to the current time, date, userid, jobname, etc.

These variables are most useful when the SET LOG DSNAME=dsn command is executed from a command script, a DEAD trap, or a session profile. For more information, see:
- HELP COMMANDS SYNTAX DSNAMES
- HELP SCRIPTS
- HELP BREAKPOINTS DEADTRAPS
- HELP PROFILES

In addition to setting the name of the log dataset, **SET LOG DSNAME=dsn** also switches logging to this dataset from wherever it had been assigned previously. However, if automatic logging is in effect, the switch remains pending until:
- A **SET LOG MODE=AUTO** command is issued,
- Or a **SET LOG MODE=MANUAL** command is issued,
- Or a **SET LOG RESET** command is issued.

SET LOG DSNAME

The SET LOG DSNAME command can be given without an **=dsname** value. In this case log switching (probably from a SYSOUT spool file) is performed without changing the name of the log dataset.

SET LOG MODE=APPEND
SET LOG MODE=OVERWRITE
These commands control whether a preexisting log dataset is **overwritten** or **appended** to.

The **OVERWRITE** setting is effective only when Z/XDC first establishes the dataset for use in a debugging session. Once usage has been established, Z/XDC appends subsequent entries to the dataset even during manual logging (where the log dataset is repeatedly opened, written to, and then closed).

Z/XDC permits only one log file to be established at a time. If commands are used to switch logging from the current dataset to another or to a spool file, then the current dataset is no longer the established log dataset. Then if later on logging is returned to this dataset, then this dataset becomes reestablished as the log dataset, and so the **OVERWRITE** setting (if in effect) again becomes effective.

Note, Z/XDC always allocates the log dataset for **exclusive** use. Z/XDC never shares it.

SET LOG UNIT=unitname
SET LOG UNIT
This provides a z/OS unit address or type or group name to be be used when allocating (or creating) a log dataset. The unit designation may be:
- A hardware address (e.g. /0A1C),
- An IBM device type (e.g. 3330-1),
- Or a groupname (e.g. SYSALLDA).

Unitname's syntax is as follows:
- It must be from 1 to 8 characters long.
- It must consist of alphabetic and numeric characters.
- Hyphens (-) and slashes (/) may also be used.
- Nationals (@ # and $) may **not** be used.

When **=unitname** is omitted, z/OS will use a default unitname for its allocation.

SET LOG VOLSER=diskname
SET LOG VOLSER
This provides the name of a DASD volume on which a log dataset is to be allocated or created.

Diskname's syntax is as follows:
- It must be from 1 to 6 characters long.
- It may consist of **any** characters, **but...**
- if it contains any characters that are not in the following list, it must be enclosed in single-quotes (').
- The following characters are permitted without requiring quotes:
    - Uppercase alphabetics

    - Numerics
    - The national characters (@ # and $)
    - And finally, a dash (-)
  - If any other characters are needed, diskname must be enclosed within single-quotes (’).
  - If the disk’s name includes a quote, then it must be given doubled up (’’).

When **=diskname** is omitted, z/OS will use a default volume for its allocation.

Examples:

V=abcdef - This resolves to **ABEDEF**
V=’abcdef’ - This resolves to **abcdef**
V=’3 ’’dEf’ - This resolves to **3 ’dEf**

Help COmmands SEt LOG SYsout

The following commands are used to define the characteristics of a log when written to a SYSOUT spool file.

When **automatic logging** (SET LOG MODE=AUTO) is in effect, any changes created by the following SET LOG commands do not take effect immediately. Instead, the changes remain pending until:
  - A **SET LOG MODE=AUTO** command is issued,
  - Or a **SET LOG MODE=MANUAL** command is issued,
  - Or a **SET LOG RESET** command is issued.

When **manual logging** (SET LOG MODE=MANUAL) is already in effect, all changes take effect immediately.

Syntax: (for spool related SET LOG commands)

```
SET LOG SYSOUT=class
        SYSOUT
SET LOG DEST=remoteid
        DEST
SET LOG HOLD=YES
        HOLD=NO
SET LOG OUTLIM=nnn
        OUTLIM=0
```

Aliases: See HELP COMMANDS SET LOG for a list of all deprecated aliases.

Operands:

SET LOG SYSOUT=class

This command causes the log to be written to spool, and it defines the SYSOUT class into which it is written.

class is is any valid SYSOUT class: A thru Z, 0 thru 9, or "*".

This command switches logging to SYSOUT spool from wherever it had been assigned previously. However, if automatic logging is in effect, the switch remains pending until:

- A **SET LOG MODE=AUTO** command is issued,
- Or a **SET LOG MODE=MANUAL** command is issued,
- Or a **SET LOG RESET** command is issued.

SET LOG SYSOUT
The SET LOG SYSOUT command can be given without an **=class** value. In this case log switching (probably from a DASD dataset) is performed without disturbing the previously defined SYSOUT class.

SET LOG DEST=remoteid
This causes the SYSOUT log file to be routed either to local printers (LOCAL, U1, etc.) or to any RJE work station defined at your computer center.

remoteid is checked for proper syntax (alphanumerics, 1-8 characters long), but beyond that it is not checked for validity. If you provide a remoteid that is unknown to JESn, then JESn will handle the error, not Z/XDC. Check with your Systems Programming staff for remote destination names that are valid for your data center.

SET LOG DEST
The SET LOG DEST command can be given without an **=remoteid** value. In this case Z/XDC’s override is cleared, and the log file is routed to the destination that is the default for your job or TSO session.

SET LOG HOLD=YES
This causes the log file to be **held** on spool. Nominally, this means that the output will not be printed until it is released either by you or an operator.

SET LOG HOLD=NO
The log file on spool will be either **held** or **not held** according to your data center’s definition for the chosen SYSOUT class. This means that your log output may still be held in spite of this operand.

For a list of held and not-held SYSOUT classes, see the Systems Programmer who is in charge of JES2 or JES3 at your site.

SET LOG OUTLIM=nnn
OUTLIM=0

This operand sets the maximum number of lines of output that JESn will accept for the log file (when the log file is being written to spool). If this limit is exceeded, then JESn will cause the debugging session to be ABENDed.

nnn may be **0,** or it may range from **1,000** up to **16,777,215.** [That's 2**24-1.] **nnn** must be given without commas. (Commas are shown here for readability.)

When **OUTLIM=0** is given, an output lines limit is not set for the log file.

However, if an output lines limit has been defined (in JCL or system defaults) for the job or jobstep in which the debugging session is running, then the lines written to the spooled log still count towards that limit, and a session ABEND can still occur should that limit be exceeded.

Generally, **SET LOG OUTLIM=nnn** should be used if Z/XDC is running within an environment for which an excessively small default output limit has been imposed upon spooled files. **SET LOG OUTLIM=nnn** can be used to override a small limit with a more reasonable value.

If **SET LOG OUTLIM=nnn** has to be used, then I guess I would recommend that a limit of **50000** lines be specified.

Help COmmands SEt LOG SCrollarea

The following commands mostly relate to the Watch Window's Scroll Area. They can be used to:

- Set the approximate size (in lines) of the Scroll Area,

- Set the Latent Commands String to be issued whenever Z/XDC receives control from the user's program,
- Control whether or not the Latent Commands messages are to be revealed in the Scroll Area.

Syntax: [for Scroll Area related SET LOG commands]

```
SET LOG SIZE=nnnnn
SET LOG LATENT='commandsstring'
        LATENT=SHOW
        LATENT=NOSHOW
```

Aliases: See **HELP COMMANDS SET LOG** for a list of all deprecated aliases.

Operands:

SET LOG LATENT='commandsstring'

This provides a string of commands to be issued under the covers every time Z/XDC receives control from the user program (typically due to an ABEND or a trap of some sort):

- The resulting messages will be recorded in the Session Log.
- They also will be recorded in the Scroll Area, **but** they usually will be hidden from view. (But see below.)

The commands string that you provide will be recorded in your **Session Profile** which you can then harden with a **PROFILE SAVE [name]** command.

The **Factory Default** command string for all default profiles is **L PSWE;L BEA;L RWREGS**

The purpose of Latent Commands is to cause information to be recorded in the Session Log without disrupting either the display or the Scroll Area. For more information, see **HELP FULLSCREEN LOGGING LATENTCOMMANDS.**

The Latent Commands String must consist entirely of commands that show information. Most action commands are disallowed. But there is one exception. The **GO** command is permitted. This is so that you can create a recording of information without requiring interactive communication with a user.

SET LOG LATENT=SHOW
This enables the display in the Scroll Area of Latent Command strings and the messages they produce. This affects both previously issued and future issued command strings.

SET LOG LATENT=NOSHOW
This disables the display of Latent Commands. Any that had been visible will suddenly disappear from the Scroll Area. (They will, of course, continue to be written to the Session Log.)

LATENT=NOSHOW is the Factory Default setting.

SET LOG SIZE=nnnnn
This command sets an approximate limit on the number of messages that can be saved in the Primary Window's Scroll Area. When this limit is exceeded, the oldest messages are discarded.

nnnnn may range from **100** [OUCH!] to **65535** (64K-1). The Factory Default is **65535.**

The limit is approximate because the discarding occurs only at command boundaries. In other words, when the discarding is done, the oldest retained message will be one echoing a command string, **never** a message generated by a command string. Consequently, the output from a given command string is guaranteed to be fully recorded in the Scroll Area, no matter how much output is generated, and no matter how small a size limit has been set.

The amount of storage needed for the Scroll Area depends upon the number of messages currently recorded within it. As of November 2018, each message requires about 800 bytes or so.

Example:

Try issuing the following commands:

MAP FFFFFF - This loads a binder map of the System Nucleus.

LIST LK FFFFFF FULL - On my system, this generates nearly seven thousand messages.

LIST LK FFFFFF FULL - Do it 10 more times. This will generate another 70,000 messages or so, which will cause the default Scroll Area size limit to be exceeded.

UP C - This will show the top of the output produced by the 2nd **LIST LKEDMAP** command.

UP C - This will show the top of the output produced by the 1st **LIST LKEDMAP** command.

UP C - This will fail because all older messages have been discarded (due to the rather large number of messages produced by the prior commands overflowing the scroll area's capacity).

LOC 0 - This will show the first visible message remaining in the scroll area after trimming. The screen title line will show you how many messages have been trimmed.

Note that even though the size limit is 65,535 messages, in this example we've managed to put around 70,000 messages into the Scroll Area. This is because when discarding messages, Z/XDC will do so only at command string boundaries. But as you issue additional commands, eventually the entirety of the early **LIST LKED** command outputs will be pushed beyond the 65,535 messages limit, at which time they will be discarded.

Help COmmands SEt LOG Management

The following commands are used to open, close write to and reset the session log.

Syntax: (for log management SET LOG commands)

```
SET LOG MODE=AUTO
        MODE=MANUAL
SET LOG RESET
```

Aliases: See HELP COMMANDS SET LOG for a list of all deprecated aliases.

Operands:

SET LOG MODE=AUTO
This command does the following:
- If a log file is currently open, then it is **closed.**
- If any changes to logging are **pending,** then they are applied.
- Then the log is **(re)opened,**

- And thereafter, all activity from the Primary Window is automatically recorded therein as it happens.

SET LOG MODE=MANUAL
This command does the following:
- If a log file is currently open, then it is **closed.**
- If any changes to logging are **pending,** then they are applied.
- The log is then left closed and unallocated until a **LOG** command is issued, upon which:
  - a) Allocates and opens the log file,
  - b) Writes to it all Primary Window activity that is currently recorded in the scroll area and that has not previously been written to the log,
  - c) Closes the log file and deallocates it.

  See HELP COMMANDS LOG for more information.

If the log is assigned to a **spool** file, then each time a LOG command is issued, a new spool file is created.

If the log is assigned to a **DASD** dataset, then each time a LOG command is issued:
- The first time the dataset is opened it is opened with a DISP=NEW, =OLD or =MOD, depending:
  - Upon whether the dataset was just now created or preexisted,
  - And upon the **MODE=APPEND|REWRITE** setting currently in effect.
- For each subsequent time, a DISP=MOD is used. Thus, once the use of a disk dataset has been established, each LOG command appends new entries to the dataset.

SET LOG RESET
This command does the following:
- If a log file is currently open, then it is closed.
- If any changes to logging are pending, then they are applied.
- If automatic logging is in effect, then the log is then reopened;
- Otherwise, it is left close.

This command is effective only when auto logging is in effect. When auto logging is turned off, this command has real no effect.

More Information

For more information, please see:
HELP COMMANDS LOG
HELP FULLSCREEN LOGGING
HELP FULLSCREEN SCROLLING

Help COmmands SEt LOGOnid

This command is an alias of the **SET USERID** command. See HELP COMMANDS SET USERID for more information.

Help COmmands SEt Maplibs

"MAPLIBs" is Z/XDC's name for those sequential and partitioned datasets in which the MAP and DMAP commands can find ADATA for building source image maps of programs and control blocks. The MAPLIB datasets may be any of the following:

- SYSADATA output created by any mainframe Assembler that supports producing IBM-compatible ADATA. As of this writing, there are three such assemblers that I know of:
  - IBM's High Level Assembler (www-01.ibm.com/software/awdtools/hlasm). Use PARM=ADATA.
  - Tachyon Software's Cross Assembler and z/Assembler (www.tachyonsoft.com/txaover.html).
  - Dignus' Systems/ASM (www.dignus.com).

- A GOFF file containing embedded ADATA. (If using IBM's High Level Assembler, then specify "PARM='GOFF(ADATA)'".)

- A file containing streamed ADATA produced by a PC based mainframe Assembler and then binary-uploaded to the mainframe into a RECFM=FB file of arbitrary LRECL.

A "MAPLIB list" is a list of MAPLIB files. In Z/XDC multiple MAPLIB lists can be defined, but at most only one MAPLIB list can be active. For more information, see HELP MAPS ADATA MAPLIBS.

MAPLIB lists can be created and managed by the SET MAPLIBS command. They can be displayed by the LIST MAPLIBS command. They can be purged by the DELETE MAPLIBS command.

The **SET MAPLIBS** command can be used to:
- Add dsnames to the currently active MAPLIBs list.
- Merge a saved MAPLIBs list into the active MAPLIBs list.
- Replace the active MAPLIBs list with a saved list.
- Permanently save a named copy of the active MAPLIBs list for use in future debugging sessions.
- Purge the active MAPLIBs list.

Note, saved MAPLIBs cannot be purged by the SET MAPLIBS command. Use the DELETE MAPLIBS command to do that. The DELETE MAPLIBS command also can be used to purge the active MAPLIBs list as well as individual datasets from the active list.

Syntax:

```
SET MAPLIBS dsname           ...
            dsname(member)
            #nnn

            READ name
            SAVE name
            RESET
```

SHOW
QUIET

FAILOK
FAILNOK

Any number of operands can be given in any order. Operands are processed from left to right.

Several operands add datasets to the currently active MAPLIBs list. Those operands are:
    dsname
    dsname(member)
    READ name
The first dataset added to the list will be added to the front of the list. The next dataset added will follow the first. And so on.

dsname
dsname(member)
These provide the names of one or more datasets to be added to the currently active MAPLIBs list.

- **dsname:** This must be the fully qualified UNquoted name of a sequential or partitioned dataset. If partitioned, then it may be a PDS or PDSE.

    - If **sequential,** then the dataset will be searched whenever ADATA is needed, regardless of the name of the object for which the ADATA is needed.

    - If **partitioned,** then the name of the object for which ADATA is needed determines which member is searched. See HELP MAPS ADATA MAPLIBS for more information.

- **dsname(member):** This must be the fully qualified UNquoted name of a specific member of a partitioned dataset. Like a sequential dataset, this member will be searched whenever ADATA is needed, regardless of the name of the object for which the ADATA is needed.

If the "dsname" or "dsname(member)" operand names a dataset that already is in the active MAPLIBs list, then that dataset is moved to the start of the list.

#nnn
This operand moves an existing member of the active MAPLIBs list to the front of the list (following any other datasets that may have already been added by prior operands of the same command).

"#nnn" must be given as the "#" character followed by a decimal number. It refers to a MAPLIB list entry’s sequence number as displayed by a preceding LIST MAPLIBS command.

If the sequence of entries in the active MAPLIB list has changed since the last time the list was displayed, then these sequence number operands cannot be used.

READ name
This operand inserts a saved MAPLIBs list into the front of the currently active MAPLIBs list (or immediately following any other datasets that may already be at the front of the list due to prior operands of the same command). It does **not** replace the active list.

"name" must be from one to 8 characters long. It must consist entirely of alphanumerics and/or the following: _ @ # $. The first character may not be numeric.

If "name" is omitted, then the default saved list is loaded. The name of the default saved list is "DEFAULT". (Clever, huh?)

If you wish to entirely replace the active MAPLIBs list with the saved list, then precede the "READ" operand with a "RESET" operand. Example: SET MAPLIBS RESET READ OLDLIST

SAVE name
This operand causes the currently active MAPLIBs list to be named and saved. If a prior instance of the named list already exists, then that prior instance is purged and replaced.

Warning! The saved list is not permanently saved unless and until a PROFILE SAVE command is issued. Once this is done, the saved list will remain available for use in future debugging sessions.

"name" must be from one to 8 characters long. It must consist entirely of alphanumerics and/or the following: _ @ # $. The first character may not be numeric.

If "name" is omitted, then the list is saved under the default name. The default name is "DEFAULT".

RESET
CLEAR
The currently active MAPLIBs list is emptied. Saved MAPLIB lists are unaffected. (Note, this is the same as the DELETE MAPLIBS ALLACTIVE command.)

SHOW
QUIET (Alias is NOSHOW)
This operand either causes (SHOW) or prevents (QUIET) Z/XDC from displaying the the results of the SET MAPLIBS command. "SHOW" is the default action.

FAILOK
FAILNOK
This operand sets the action that Z/XDC will take in the event that a **READ name** operand (following on the same command) should fail:
- If FAILNOK is in effect, then a subsequent READ operand failure will cause the SET MAPLIBS command to issue error messages and terminate.
- If FAILOK is in effect, then a subsequent READ operand failure will be ignored. Command processing will continue as if the READ operand had not been present.

The default setting is FAILNOK.

The effect of a FAILOK/FAILNOK operand does not last past the processing of the SET MAPLIBS command in which it occurs.

Examples:

SET MAPLIBS MY.ADATA1 MY.GOFF2 HIS.ADATALIB
This command adds three ADATA datasets or libraries to the front of the currently active MAPLIBs list. They are added in the order listed. The resulting active list

is automatically displayed.

LIST MAPLIBS
SET MAPLIBS #2 #3
The SET MAPLIBS command rearranges the active MAPLIBS list into the following order:
    MY.GOFF2
    HIS.ADATALIB
    MY.ADATA1

SET MAPLIBS #1 READ OLDLIST
This command inserts all of the MAPLIBs from the saved list named OLDLIST into the currently active MAPLIBS list following MY.GOFF2.

SET MAPLIBS RESET READ OLDLIST
This command purges the active MAPLIBs list and replaces it with the saved list named OLDLIST.

SET MAPLIBS RESET FAILOK READ OLDLIST READ NEWLIST FAILNOK READ MASTERL
This example illustrates the use of the FAILOK and FAILNOK operands. This command does the following:
- (RESET) It purges the currently active MAPLIBS list.
- (READ OLDLIST) Then it reads in a saved list named OLDLIST.
- (READ NEWLIST) And then it appends to it another saved list named NEWLIST.
- (READ MASTERL) And then it appends still another saved list named MASTERL.
- (FAILOK) If either OLDLIST or NEWLIST, for some reason, does not exist, then the command just ignores that list and continues processing with the next operand.
- (FAILNOK) However, if MASTERL does not exist, then the command terminates with error messages.

Help COmmands SEt NOBell

This command is an alias of the **SET BELL OFF** command. See HELP COMMANDS SET BELL for more information.

Help COmmands SEt NOHICOLOR

This command has been removed and not replaced. Its functionality was made obsolete by the demise of hardware 3270s having limited functionality. These days, all 3270s are emulated by software (PCOMM, Vista, Attachmate, etc.) as having full functionality.

Help COmmands SEt NOHICOLOUR

This command has been removed and not replaced. Its functionality was made obsolete by the demise of hardware 3270s having limited functionality. These days, all 3270s are emulated by software (PCOMM, Vista, Attachmate, etc.) as having full functionality.

Help COmmands SEt NOIlc

When virtual storage is displayed in fullscreen mode, Z/XDC permits the user to zap instructions and data by simply typing a **Z** at the left of a display line and then overtyping various parts of the displayed data. In particular, if a display includes machine instructions, the instruction opcodes can be changed just by typing the new opcode's mnemonic directly on top of the old opcode's mnemonic. Example: a displayed BCTR instruction can be changed to an LR instruction simply by typing "LR" on top of the "BCTR" in the display.

When machine instruction opcodes are zapped in this manner, the **SET ILC** and **SET NOILC** commands control whether Z/XDC will (ILC) or will not (NOILC) enforce a restriction that the new machine instruction must be of the same length as the old one.

Although **SET NOILC** will permit you to, for example, change a "ST" instruction into an "MVC", it does **not** do so by shifting the following instructions in order to make room. Instead, what had been the first halfword of the following instruction simply becomes (in this example) the last halfword of the MVC. Note that in general this is not a desirable result and additional zapping would probably have to be done to produce a correct sequence of machine instructions.

The **SET ILC** and **SET NOILC** commands have no effect when machine instructions are changed via overtyping hex displays.

Syntax:

SET ILC
SET NOILC

These commands accept no operands.

The status of instruction length checking can be saved in your profile dataset for automatic restoration every time you start a debugging session. For more information, please see:

HELP PROFILES
HELP COMMANDS PROFILE

The distributed version of Z/XDC has ILC set in the profile.

Help COmmands SEt NOReadecho

This command has been functionally replaced by the **SET READ** command. Please discontinue using this command and use that one instead.

Help COmmands SEt NOWTOR

This command has been replaced. Use **SET DBC640Q** instead.

Help COmmands SEt Optimization

For writing displays to the terminal screen, Z/XDC may use either of two mechanisms:

- **ISPF** Display Services
- Or fullscreen **TPUTs**

The following discussion concerning optimization of screen displays applies **only** when Z/XDC is communicating via fullscreen **TPUTs.** It does not apply when **ISPF** Display Services are being used.

For more information, see **HELP COMMANDS SET TPUT.**

Optimization

Generally, when Z/XDC writes display screens to the terminal, it tries to reduce transmission time and network traffic by writing only those fields that are actually different from the previously written screen image. This process is called **optimized writes.**

Optimization Problems

Unfortunately, there are a variety of circumstances where Z/XDC will not be in full control of all output to the screen. When Z/XDC first receives control during a debugging session, one of the first things it does is read and save a copy of the terminal’s current screen image so that it can be restored to the screen prior to control being returned to the program being debugged; however, while Z/XDC is in control, it assumes that it has full control of the screen and that no one else will simultaneously be writing to the screen. Accordingly, whenever Z/XDC writes a new screen, it assumes it knows what was already there, and so when optimizing, it writes only the changed fields.

Unfortunately, in multi-tasking environments, Z/XDC can have no knowledge of or control over what other nonABENDed tasks might do, and if such a task also writes to the screen, Z/XDC's screen image will be corrupted and Z/XDC will be unaware of this. And this is only one of several possible ways in which screen data can be corrupted.

Turning Optimization Off

One way to deal with this problem is to turn Z/XDC's optimizing routine off. When this is done, then each time Z/XDC writes to the screen, it will rewrite the ENTIRE screen. This may clear up screen corruption problems, but there are tradeoffs. On fast terminals there will be an annoying "flash" of the screen as it is erased just before the rewrite. On slow terminals screen displays will take substantially longer to appear.

Syntax:

```
SET OPTIMIZATION ON [or omitted]
                 OFF
                 TRACE
```

Operands

ON [or omitted]
This causes Z/XDC to attempt to optimize its screen image. (Optimization may not be possible in all TSO debugging situations.)

OFF
This prevents Z/XDC from optimizing screen images. The screen will be erased and repainted every time a new display is generated.

TRACE
This causes Z/XDC to optimize most of its writes to the terminal; however, writes occurring after a **TRACE** or **GO** command will not be optimized: They will repaint the entire screen.

TRACE and **GO** commands cause Z/XDC to release control back to the user's program, thus the next write that Z/XDC will do will be after the user's program has either reached a breakpoint or reABENDed thereby causing control to return to Z/XDC

The issues are:
- If the user program has, while it was in control, written to the screen, Z/XDC won't know that, so its optimized writes will result in a corrupted display.
- On the other hand, if the display is fully repainted every time you step the program with the **TRACE** command, the constant screen flashes will be annoying, and the full display writes will slow you down.

Your choice for optimization can be saved in your session profile for automatic restoration every time you start a debugging session. For more information, please

see:
    HELP PROFILES
    HELP COMMANDS PROFILE

The factory default setting for optimization is **SET OPTIMIZATION ON.**

Help COmmands SEt PARseorder

This command is available only for customers who have Licensed Features for debugging one or more of the supported **High Level Languages** (such as XL C/C++ and Metal C). For more information, see HELP SUPPORT FEATURESANDCAPS.

The syntax rules for variables differ from one language to the next. Consequently, one parser cannot be used to parse a variable for all languages.

Accordingly, Z/XDC provides a unique parser for each supported language type, and it allows you to define which parsers will be used to resolve a given variable name. This definition is called the **Parse Order,** and this SET PARSEORDER command allows you to control this order.

More specifically, this **SET PARSEORDER** command allows you to:
- Set which Parsers are called for parsing a string,
- The order in which the Parsers are called,
- And which parser generates the error message should they all fail. This will always be the last parser set in the Parsing Order. (And it is quite permissible for that parser to be the same as the first parser - or any other previously listed parser.)

A global Parsing Order can be set and displayed and saved in your session profile. It can also be overridden for individual commands. More specifically, the Parsing Order:
- Is set by the **SET PARSEORDER** command (described here),
- Is displayed by the **LIST PARSEORDER** command,
- And is saved in your session profile by the **PROFILE SAVE** command.

It also can be displayed, set and saved by the **Profile Menuing System.**

Syntax:

    SET PARSEORDER parsername parsername ...

parsernames
These operands identify which (and in what order) parsers are to be given a crack at parsing a given variable name. Currently, the following parsers are available:
    ASM - Names are parsed according to Assembler syntax rules.
    CEE - Names are parsed according to XL C/C++ and Metal C syntax rules.

The Parse Order rules:

- Up to eight Parser names are supported (even though only two names are currently defined).
- Parser names may be given in any order. (This defines the order in which parsers are called.)
- A given Parser name may be given more than once. (See next for why you might want to do that.)
- In the event that all Parsers fail to resolve a name successfully, the error messaging that is displayed will be that which is provided by the last Parser called.

  This last rule means that Parse Orders such as the following actually make sense:

  SET PARSEORDER CEE ASM CEE

  - A given variable will first be parsed as an XL C/C++ and Metal C variable.
  - If that fails, then it will next be parsed as an Assembler variable.
  - If that also fails, then it will be parsed as a XL C/C++ and Metal C variable again. Of course, this will fail again, **but** the error messaging resulting from this failure will be what is displayed.

Help COmmands SEt PFkeys

This command is an alias of the **SET KEYS** command. See HELP COMMANDS SET KEYS for more information.

Help COmmands SEt PRIMarysize

Some fullscreen terminals have a display screen that may be formatted in two different ways. They have a primary set of dimensions (lines and columns) and a secondary (or alternate) set. Normally, a terminal's "primary" dimensions are 24x80 (24 lines by 80 columns), and its "secondary" dimensions are larger.

The **SET PRIMARYSIZE** command causes Z/XDC to format the screen using that terminal's primary set of dimensions.

Syntax:

SET PRIMARYSIZE

This command accepts no operands.

The terminal's secondary dimensions can be requested via the "SET SECONDARYSIZE" command.

The screen size choice can be saved in your profile dataset for automatic restoration every time you start a debugging session. For more information, please see:

    HELP PROFILES
    HELP COMMANDS PROFILE

The distributed version of Z/XDC has the default set to SECONDARYSIZE.

Help COmmands SEt PRINt

The **SET PRINT** command can be used both to set certain characteristics of and to change the status of the **XDCPRINT** output file.

The **XDCPRINT** output file can be used for printing excerpts of or the entirety of Z/XDC's Built-in Help. The information is automatically paginated and indexed. See **HELP ABOUTHELP PRINTING** for more information.

See also **HELP COMMANDS LIST PRINT** for information about displaying the current settings for and status of the **XDCPRINT** file.

The settings that can be made by the **SET PRINT** command can be saved in your Z/XDC session profile. Accordingly, these settings can be both displayed and changed by Z/XDC's Profile Menuing System. See **HELP PROFILES MENU** for more information.

Syntax:

```
    SET PRINT CLASS c

              HOLD  YES
                    NO
                    DEFAULT

              LPP   n

              CLOSE
              DELETE
```

Notes:

- None of the operands are mutually exclusive with each other. The operands can all be given in any combination with each order.

- The operands can be given in any order. They are processed by Z/XDC from left to right, and they are **not** presyntax checked. So it is possible for a **SET PRINT** command with multiple operands to partially succeed and to partially fail.

- Three of the supported operands **(CLASS, LPP** and **HOLD)** require values. Those

values must be given without the use of equal signs or parentheses. Also, those values can be saved in Z/XDC's session profile, so they can be both displayed and changed by Z/XDC's **PROFILE** command.

Operands:

CLASS c
This specifies the sysout class to which the output will be printed. **c** may be any alphabetic (A-Z) or numeric (0-9) character, or it may be an asterisk (*), in which case the output will be routed to the same output class as is used for your job or TSO session's JCL. The exact characteristics of the chosen class are determined by system parameters. Speak with your Systems Programmer for a recommendation of which output class is best to choose here.

The **CLASS** operand is effective only when **XDCPRINT** is allocated to a sysout file on JES2 or JES3 spool. If **XDCPRINT** is preallocated to a disk or tape dataset, then the **CLASS** operand has no effect until **XDCPRINT** is reallocated.

The effect of the **SET PRINT CLASS** command is implemented by Z/XDC both at file allocation time and at file deallocation time. Accordingly, this command can be effectively issued at any time up until the **XDCPRINT** file is closed.

The **LIST PRINT** command or the **PROFILE** command can be used to display the current **CLASS** value assigned to the **XDCPRINT** file.

The class established by the **SET PRINT CLASS** command can be saved in your Z/XDC session profile. The factory default value for **CLASS** is **X.** See **HELP PROFILES** for more information.

HOLD YES
HOLD NO
HOLD DEFAULT
This specifies whether the output written to the **XDCPRINT** file is to be held on spool for access from a terminal session or whether it is to be immediately released for printing at a JESx controlled printer. Specifying **HOLD YES** or **HOLD NO** forces the output to be held or not held, respectively. Specifying **HOLD DEFAULT** causes the output to be held or not held according to the characteristics of the sysout class to which the output is routed. Such characteristics are established by system parameters. If you have any questions, please ask your Systems Programmer.

The **HOLD** operand is effective only when the **XDCPRINT** output file is allocated to a SYSOUT dataset on JES2 or JES3 spools. If **XDCPRINT** is allocated to a disk or tape dataset, then the **CLASS** operand has no effect until **XDCPRINT** is reallocated.

The effect of the **SET PRINT HOLD** command is implemented by Z/XDC at file deallocation time. Accordingly, this command can be effectively issued at any time up until the **XDCPRINT** file is closed.

The **LIST PRINT** command or the **PROFILE** command can be used to display the current **HOLD** setting assigned to the **XDCPRINT** file.

The setting established by the **SET PRINT HOLD** command can be saved in your Z/XDC session profile. The factory default setting for **HOLD** is **DEFAULT.** See **HELP**

PROFILES for more information.

LPP n
This controls the number of lines Z/XDC sends per printed page. When Z/XDC writes to the **XDCPRINT** file, it inserts a page title line at every nth output line. This title line has an incrementing page number and is written with a "page eject" carriage control character ("1").

n may range from **10** to **255,** or it may be **0,** in which case, Z/XDC will insert very few title lines, and the index that Z/XDC generates will be essentially useless.

The factory default value for **LPP** is **108.**

The **LIST PRINT** command or the **PROFILE** command can be used to display the current **LPP** value assigned to the **XDCPRINT** file.

The value established by the **SET PRINT LPP** command can be saved in your Z/XDC session profile. See **HELP PROFILES** for more information.

CLOSE
This causes the **XDCPRINT** file to be closed and perhaps deallocated.
- If Z/XDC itself dynamically allocated **XDCPRINT,** then it will be deallocated.
- On the other hand, if **XDCPRINT** was preallocated prior to the first use of the **PRINT** command, then the file will only be closed, it will not be deallocated.

In addition, all tabled index information is sorted, formatted, and written to **XDCPRINT** prior to its being closed. A Table of Contents also is written following the index.

DELETE
This causes the **XDCPRINT** file to be closed, deallocated, and perhaps deleted. The file is deallocated regardless of whether or not the file was preallocated.

If **XDCPRINT** was allocated to spool, then the deletion attempt will succeed. On the other hand, if it was preallocated to a disk dataset, then the deletion attempt can fail for any of a variety of reasons, including that it was allocated shared instead of exclusive.

When the **XDCPRINT** file is deallocated by a **SET PRINT CLOSE** command or a **SET PRINT DELETE** command, a subsequent **PRINT** command will cause the **XDCPRINT** file to be reallocated to a sysout output file.

The **XDCPRINT** command is always automatically CLOSE’d whenever a **GO, TRACE** or **END** command is issued.

Examples:

PROFILE READ

SET PRINT CLASS H HOLD YES LPP 80
LIST PRINT
PROFILE SAVE
This sequence of commands changes several **XDCPRINT** file characteristics and saves them to your session profile as follows:

PROFILE READ
Issuing this command first ensures that the myriad of settings profiled by Z/XDC are all reset to a "correct" state as defined either by your own session profile or by your installation's default profile.

SET PRINT CLASS H HOLD YES LPP 80
This command establishes the following settings for the **XDCPRINT** file:
- Sysout class **H**
- Forced **HOLD=YES**
- **80** lines per page

LIST PRINT
This displays the results of the **SET PRINT** command.

PROFILE SAVE
This command saves the new settings as your own personal defaults.

PRINT HELP MAINTENANCE
SET PRINT CLOSE
LIST PRINT
The **PRINT HELP** command prints those Built-in Help topics that constitute Z/XDC's **Maintenance and New Functionality Guide.**

The **SET PRINT CLOSE** command then appends an index to the output and then closes the file.

Finally, the **LIST PRINT** commands shows the closed status of the file and it shows whether or not the file was deallocated.

Help COmmands SEt PROFile

The **SET PROFILE** command can be used to set characteristics of a session profile (as opposed to the contents of a profile which are set mainly by a wide variety of other SET commands).

Syntax:

```
SET PROFILE DESCRIPTION=text
                       ='text'
                       ="text"

DESCRIPTION=text
           ='text'
```

This associates a string of arbitrary descriptive text to the currently active session profile. This text can be used to give a brief description of the profile, its purpose, whatever.

The syntax of the text data is as follows:

- **text**
  If the text is **not** framed by quote characters, then it may not contain semicolons (;) or colons (:), and it will **not** be upcased. Also, it may contain embedded quote characters. If it does, then they need not be doubled.

- **'text'**
- **"text"**
  If the text **is** framed by quote characters, then it will not be upcased, and it may contain semicolons and colons. The framing quote characters will be removed, and embedded doubled framing quote characters will be singlized.

For more information, see HELP COMMANDS SYNTAX CHARACTERSTRINGS.

Examples:

SET PROFILE DESCRIPTION=This is an unquoted (') description;L PROF CURRENT
- The description that will be saved is: **This is an unquoted (') DESCRIPTION**
- The description ends at the semicolon (;)
- The embedded quote (') did not have to be doubled.

SET PROFILE DESC='This is a quoted ('') desc; OK?';L PROF CURRENT
- The embedded quote (') had to be doubled. Parsing then singlized it.
- The embedded semicolon (;) was accepted as being a part of the text.

Help COmmands SEt PROXytasks

The **SET PROXYTASKS** command is used to create **Formal Proxy Tasks.**

Proxy Tasks are do-nothing tasks used to run the **Remote Phase** of Z/XDC processing when Z/XDC is running as an **FRR** (in order to debug **SRB** routines and **FRR** protected Task Mode routines).

Proxy Tasks are also used when Z/XDC is running as a **TRAP Handler** in...

- Either an FRR-protected task mode environment,

- Or an SRB environment.

Proxy Tasks don't do anything; **they simply exist.** As soon as they start, they immediately go into a WAIT state. But by just existing, they serve as safe places where IRBs can be scheduled for running Z/XDC's Remote Phase. For more information, see **HELP DEBUGGING FRR.**

Formal Proxy Tasks are those that have been created...

- By **XDCCALLA** pursuant to the presence of a **//xxxFPTnn DD DUMMY** allocation.

- Or through use of this **SET PROXYTASKS** command.

Formal Proxy Tasks have a particular structure that the **DELETE PROXYTASKS** command can look for when told to delete Proxy Tasks.

You can tell this **SET PROXYTASKS** command to create any reasonable number of Proxy Tasks. It does so by ATTACH'ing them below **any chosen TCB** located in **any address space** (security permitting, of course).

This **SET PROXYTASKS** command will decline to ATTACH Proxy Tasks to TCBs that are either **ending, ABENDing, ended** or **frozen.** But if you disagree with the command's decision, you can use a **FORCE** operand to make it try to do it anyway.

Syntax:

SET PROXYTASKS [tcbaddress] [nn] [FORCE]

Notes:
- All operands are optional.
- The operands may be given in any order.

tcbaddress

When given, this must be an address expression that resolves to the location of the Task Control Block (TCB) to which one or more Proxy Tasks are to be ATTACH'd.

The TCB can be located in any accessible address space.

The command checks to ensure the TCB is not:
- Ending
- ABENDing
- Ended
- Or frozen (permanently non-dispatchable)

Absent use of the **FORCE** operand, if the command detects any of these conditions, it will decline to ATTACH the Proxy Tasks.

The command also checks for certain other delaying conditions that potentially can be long term. Examples include:
- Tasks that are non-dispatchable due to an SMC ENQ (Set Must Complete) in effect elsewhere,
- Tasks to which the queuing of IRBs is not currently permitted,

In these (and other similar) cases, the command will briefly wait and recheck a handful of times before giving up and moving on.

tcbaddress is optional. If omitted, then Z/XDC will scan the current home address space to find the task that owns the current debugging session, That will be the TCB selected to anchor the Proxy Tasks.

Note, the **LIST TASKS** command uses **hilighting** to show which tasks are participating in the debugging session. The first of the hilighted tasks is the one that owns the

session.

nn
This is a decimal number that specifies how many Proxy Tasks to ATTACH. This value can range from **1** to **10.** If omitted, the default value is **2.**

FORCE
Before ATTACH'ing Proxy Tasks to a target task, the command first checks to see if the task is in the process of **ending** or **ABENDing,** or if it has **ended** or is **frozen.** If so, then the command will decline to do the ATTACHs.

But if you disagree, and you think the target task would be a good anchor anyway, then you can use this **FORCE** operand to override and cause the ATTACHs to be attempted regardless.

If there is a longterm blocking condition, then the **FORCE** operand may not work. The command will complete normally, but the Proxy Tasks will not show up. Specifically:

- A **LIST TASKS** command will not show that the Proxy Tasks have been started.

- A **LIST RBS TCB#n** command, directed towards the intended parent task, will probably include a line looking something like this:

  ```
  _  3    IRB    CIRB   XDC31+19E1C
  ```

  This shows that the **SET PROXYTASKS** command has scheduled the IRB needed to ATTACH the Proxy Tasks, but the IRB is being blocked (by the System) from running.

The **FORCE** operand does not cure the blocking condition, it only ignores it.

The TCB Address Expression

Any address expression is valid so long as it resolves to the address of a Task Control Block located anywhere in any accessible address space.

As a side effect, the **LIST TASKS** command creates a series of **TCB#n** equates labeling the locations of all listed TCBs. Every time the **LIST TASKS** command is reissued, the **TCB#n** equates are recreated. These **TCB#n** equates are very convenient for use as **tcbaddress** operands.

Keep in mind that this **SET PROXYTASKS** command results in changing the address space's Subtask Tree. That means that the next time you issue a **LIST TASKS** command, the values of at least some **TCB#n** equates will change! This can be confusing if you're not paying attention.

Examples

In the following examples, I am using **XDCCALLA** to debug a program named MYPROG. I have started the session by issuing **XDCCALLA MYPROG** from ISPF's Command Shell (=6).

At the start of the debugging session, a **LIST TASKS** command shows this:

```
XDC ===> list tasks
   TCB#  PROGRAM NAME (ASID 00A5, DBCOLE3)
_  1     IEAVAR00
_  2     . IEESB605
_  3     . . IKJEFT01, JOBSTEP
_  4     . . . IKJEFT02
_  5     . . . . XDCCALLA
_  6     . . . . . "MYPROG", CURRENT (XDCT0004) (ABEND: S0C1)
_  7     . . . . . . MYSUBTSK
_  8     . . . . IKJEFT02
_  9     . . . . . IKJEFT09
_  10    . . . . . . ISPF
_  11    . . . . . . . ISPTASK (ISPLLIB)
_  12    . . IEAVTSDT
```

From this display, we know the following:

- The hilighting shows that the scope of the debugging session encompasses two tasks: **TCB#6** and **TCB#7.**

- The task that owns the debugging session is the first of these, **TCB#6.**

- Even though the debugging session was started within ISPF, **XDCCALLA** is not running within ISPF's subtask structure. Instead, the System has created a **sister TMP (TCB#4)** and is running **XDCCALLA** there. This is because **XDCCALLA** is running authorized, and so for integrity reasons, the System runs authorized programs outside of the original TMP's subtask structure **(TCB#8** thru **TCB#11).**

- The System also freezes the original TMP's tasks until the authorized program completes.

  (TMP stands for **Terminal Monitor Program** and is an ancient and otherwise unpublished name for TSO.)

I'll now illustrate several **SET PROXYTASKS** commands and explain what happens.

```
XDC ===> SET PROXYTASKS
   2 proxy tasks scheduled to be created under TCB#6
   Issue LIST TASKS to verify
```

When no operands are given, Z/XDC attempts to create two Proxy Tasks as subtasks to the debugging session's owning task **(TCB#6** in this scenario). Now a **LIST TASKS** command shows the following:

```
XDC ===> list tasks
   TCB#  PROGRAM NAME (ASID 00A5, DBCOLE3)
_  1     IEAVAR00
_  2     . IEESB605
_  3     . . IKJEFT01, JOBSTEP
_  4     . . . IKJEFT02
```

```
_  5     . . . . XDCCALLA
_  6     . . . . . "MYPROG", CURRENT (XDCT0004) (ABEND: S0C1)
_  7     . . . . . . "PROXYXDC"
_  8     . . . . . . "PROXYXDC"
_  9     . . . . . . MYSUBTSK
_  10    . . . . IKJEFT02
_  11    . . . . . IKJEFT09
_  12    . . . . . . ISPF
_  13    . . . . . . . ISPTASK (ISPLLIB)
_  14    . . IEAVTSDT
```

```
XDC ===> SET PROXYTASKS TCB#10
.  DBC541E TCB#10 Not Suitable - Is either ending, ABENDing, ended or frozen (code
           08)
```

As noted above, when a sister TMP is running authorized programs, the System freezes the original TMP and all its subtasks. This is among the conditions that the **SET PROXYTASKS** command detects, and so the command fails.

```
XDC ===> SET PROXYTASKS TCB#10 FORCE
.  DBC541W TCB#10 Not Suitable - Is either ending, ABENDing, ended or frozen (code
           08)
.  DBC541W However, FORCE was given. Will attempt to use anyway

   2 proxy tasks scheduled to be created under TCB#10
   Issue LIST TASKS to verify
```

OK, let’s be stubborn and force the command to try to create the Proxy Tasks anyway. Well, sometimes that will work, but in this case it’ll sorta work, but not in any useful way.

To see what happened, let’s start with a **LIST TASKS...**

```
XDC ===> list tasks
   TCB#  PROGRAM NAME (ASID 00A5, DBCOLE3)
_  1     IEAVAR00
_  2     . IEESB605
_  3     . . IKJEFT01, JOBSTEP
_  4     . . . IKJEFT02
_  5     . . . . XDCCALLA
_  6     . . . . . "MYPROG", CURRENT (XDCT0004) (ABEND: S0C1)
_  7     . . . . . . "PROXYXDC"
_  8     . . . . . . "PROXYXDC"
_  9     . . . . . . MYSUBTSK
_  10    . . . . IKJEFT02
_  11    . . . . . IKJEFT09
_  12    . . . . . . ISPF
_  13    . . . . . . . ISPTASK (ISPLLIB)
_  14    . . IEAVTSDT
```

The two Proxy Tasks we created before are still there (of course), but the ones we wanted to create under **TCB#10** are not!

To look a little further, let's use a **LIST RBS** command to see what's happening under **TCB#10...**

```
XDC ===> LIST RBS TCB#10
   RB#  TYPE   CREATED BY    NAME          CURRENT EXECUTION LOCATION
_  1    PRB    ATTACH        IKJEFT02      IKJEFT02+A7A
_  2    IRB    CIRB                        XDC31+19E1C
```

This shows indeed that an IRB (for performing the ATTACHs needed to create the Proxy Tasks) has been scheduled to run under **TCB#10,** but it's not running because the System has frozen the task.

The task will remain frozen until the authorized program **(XDCCALLA** in this case) running under the sister TMP completes. Unfortunately, when that happens, it will be too late to be of use to the current debugging session.

However, once the current debugging session ends, **TCB#10** will be unfrozen, thus allowing the proxy tasks to be created after all! Then if you start a new debugging session, the tasks will be there and ready for use, and a **LIST TASKS** command will show this.

```
XDC ===> list tasks
   TCB#  PROGRAM NAME (ASID 00A5, DBCOLE3)
_  1     IEAVAR00
_  2     . IEESB605
_  3     . . IKJEFT01, JOBSTEP
_  4     . . . IKJEFT02
_  5     . . . . XDCCALLA
_  6     . . . . . "MYPROG", CURRENT (XDCT0004) (ABEND: S0C1)
_  7     . . . . . . MYSUBTSK
_  8     . . . . IKJEFT02
_  9     . . . . . IKJEFT09
_  10    . . . . . . ISPF
_  11    . . . . . . . "PROXYXDC"
_  12    . . . . . . . "PROXYXDC"
_  13    . . . . . . . ISPTASK (ISPLLIB)
_  14    . . IEAVTSDT
```

Notice! The proxy tasks will not be usable when frozen (such as in the above example when a sister TMP is running). But if your method of running an SRB does not lead to the freezing of the proxy tasks, then you're good to go.

Help COmmands SEt PSW

The **SET PSW** command can be used to set many fields of the retry level PSW. Specifically, the following fields can be set:

- The Addressing mode (AMODE)
- The Address Space Control Mode (ASC-MODE)

- The Condition Code (CC)
- The Execution Key (KEY)
- The Execution State (STATE)

The Instruction Address can be change too but only by the **ZAP PSW** command. For more information, see HELP COMMANDS ZAP.

Other fields in the PSW cannot be directly changed by Z/XDC. Examples include:
DAT ON/OFF
PER ON/OFF
Various masks

PSW vs. PSWE:

All changes seemingly made to 64-bit "PSWs" are actually made to 128-PSWEs. Then the 64-bit "scrunched" **view** of the PSWE is rebuilt.

In other words, when **SET'ing** or **ZAP'ing** the PSW, there is no difference between setting the PSWE and setting the PSW: All changes are first made to the 128-bit **PSWE.** Then the 64-bit PSW view is rebuilt from the updated PSWE information.

Accordingly, the **SET PSWE** command is nothing more than an alias of **SET PSW.**

Note, if the 128-bit PSWE contains an above-the-bar address, then the **scrunched** 64-bit view of the PSW will contain a **truncated** instruction address with the **lo-order bit turned on!**

General Syntax:

```
SET PSW   [AMODE=]value
    PSWE  [ASCMODE=]value
          CC[=]value
          EXT[=]value
          IO[=]value
          KEY{=| }value
          PROG=value
          [STATE=]value
```

Rules:

- Any combination of operands can be given as long as they are:
  - Non-redundant
  - Non-conflicting
  - Permitted by Security
  - Permitted by Authorization

Operands: The operand values are described more specifically in the following subtopics. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

AMODE - The Addressing Mode: 24, 31, or 64.
ASC-MODE - The Address Space Control Mode: Primary, Secondary, AR, or Home.
CC - The Condition Code: Equal, Low, High, or Overflow.
EXT - The External Interrupt Mask: Enable or Disable.
IO - The I/O Interrupt Mask: Enable or Disable.
KEY - The Execution Key: 0-15 or 0-F.
PROG - The Program Interrupt Masks: 0-F.
STATE - The Execution State: Supervisor or Problem.

Help COmmands SEt PSW AMode

The **SET PSW** command can be used to set the current default addressing mode (31-bit vs. 24-bit vs. 64-bit) both in the retry level PSW and for Z/XDC's use when addressing user program storage. More specifically, this setting is used by Z/XDC:

- When user program execution is to be retried (via GO or TRACE).
- When resolving the "!" indirect operator in address expressions (when SET BANG AMODE is in effect),
- When determining the width of an address value to be stored via the "address-data" form of the ZAP command.
- When determining whether Z/XDC's GETMAIN command is to obtain storage from 24-bit storage or 31-bit storage. (Obtaining 64-bit storage is not yet supported.)

The current addressing mode (AMODE) can be displayed by:

LIST AMODE
LIST PSW
LIST PSW FORMAT

Syntax:

```
SET PSW   ... [AMODE=]24 ...
    PSWE              31
                      64
```

24
AMODE=24
This sets 24-bit addressing.

31
AMODE=31
This sets 31-bit addressing.

64
AMODE=64
This sets 64-bit addressing (z/OS only).

Examples:

S PSW 24
S PSW AMODE=24
The addressing mode is set to 24-bit. If user program execution is resumed (GO or TRACE), then it proceeds in 24-bit mode.

ZAP R1,1F943CD8
- In 24-bit mode, the value of the given address expression is truncated to 24-bits, A(X'943CD8'), and stored into the low-order 3 bytes of R1 without disturbing the high byte.
- In 31-bit mode, this address expression's value is stored without truncation into the low-order 31 bits of R1 without disturbing the high bit.
- In 64-bit mode (z/OS only), this address expression's value is padded to 64 bits and then truncated back down to 32 bits and then stored into all 32 bits of R1 (i.e. the low half of RW1). The high half of RW1 (i.e. RH1) remains unchanged.

SET BANG AMODE
FORMAT .DCBBUFCB!
The SET BANG command causes the ! to be AMODE sensitive. Then:
- In 24-bit mode, the storage location displayed is pointed to by the low-order 24 bits (3 bytes) of the DCBBUFCB field.
- In 31-bit mode, the displayed location is pointed to by the low-order 31 bits which, in this case, would display a location that is **not** the dataset's buffer control block because the hi-order byte of the DCBBUFCB field is very likely to be nonzero.
- In 64-bit mode, the doubleword starting at the location of the DCBBUFCB field would be considered to be a 64-bit wide address. So again, a location would be displayed that is not the dataset's buffer control block.

FORMAT .DCBBUFCB%
This treats the .DCBBUFCB field as containing a 24-bit pointer **regardless** of the current AMODE setting.

Help COmmands SEt PSW ASc-mode

The **SET PSW** command can be used to set the PSW's Address Space Control Mode (not to be confused with Z/XDC's "Addressing Mode").

The current ASC-MODE can be displayed by: **LIST PSW FORMAT**

Syntax:

```
SET PSW   ... [ASCMODE=]PRIMARY  ...
    PSWE                SECONDARY
                        HOME
                        AR
```

PRIMARY
ASCMODE=PRIMARY
This sets the System to run in "primary" Address Space Control Mode (its normal mode). In this mode, both instructions and data are fetched by the System from the Primary Address Space, and access registers are not available for addressing data.

SECONDARY
ASCMODE=SECONDARY
This sets the System to run in "secondary" Address Space Control Mode. Machine instructions are fetched from the Primary Address Space while data is fetched from a Secondary Address Space.

HOME
ASCMODE=HOME
This sets the System to run in "home" Address Space Control Mode (ASC-MODE). In this mode, both instructions and data are fetched by the System from the Home Address Space. Z/XDC must be running authorized before this mode can be selected.

AR
ASCMODE=AR
This sets the System to run in "access register" Address Space Control Mode. In this mode instructions are fetched from the Primary Address Space while data is fetched either from the Primary Address Space or from permitted data spaces and address spaces that are linked to the user program via access registers.

SUMMARY CHART	PRIMARY MODE	SECONDARY MODE	HOME ASCMODE	AR ASCMODE
May be set from problem state	Y	Y	Y	N
May be set from supervisor state	Y	Y	Y	Y

Help COmmands SEt PSW Cc

The **SET PSW** command can be used to change the user program’s current condition code (in the retry level PSW).

The current condition code can be displayed by:
    LIST PSW
    LIST PSW FORMAT

Syntax:

```
    SET PSW   ... CC0     ...
        PSWE      CCEQUAL
                  CCZERO
                  CC=0
                  CC=EQUAL
                  CC=ZERO

    SET PSW   ... CC1     ...
        PSWE      CCLOW
                  CCMINUS
                  CCMIXED
                  CC=1
                  CC=LOW
                  CC=MINUS
                  CC=MIXED

    SET PSW   ... CC2     ...
        PSWE      CCHIGH
                  CCPLUS
                  CC=2
                  CC=HIGH
                  CC=PLUS

    SET PSW   ... CC3     ...
        PSWE      CCONES
                  CCOVER
                  CC=3
                  CC=ONES
                  CC=OVER
```

CC0
CCEQUAL
CCZERO
CC=0
CC=EQUAL
CC=ZERO
The retry level PSW’s condition code is set to 0 (B’00’).

CC1
CCLOW
CCMINUS
CCMIXED
CC=1
CC=LOW
CC=MINUS
CC=MIXED
The retry level PSW’s condition code is set to 1 (B’01’).

CC2
CCHIGH
CCPLUS
CC=2
CC=HIGH
CC=PLUS

The retry level PSW's condition code is set to 2 (B'10').

CC3
CCONES
CCOVER
CC=3
CC=ONES
CC=OVER
The retry level PSW's condition code is set to 3 (B'11').

Help COmmands SEt PSW Ext

The **SET PSW** command can be used to enable or disable the External Interrupt Mask in the retry level PSW. Needless to say...

- This can be done only for programs that are running authorized.
- This should be done only by the most expert programmers who truly know what they are doing.

The current state of the retry level External Interrupt Mask can be displayed by: **LIST PSW FORMAT**

Syntax:

```
SET PSW   ... EXTON       ...
    PSWE      EXTENABLE
              EXTOFF
              EXTDISABLE
              EXT=ON
              EXT=ENABLE
              EXT=OFF
              EXT=DISABLE
```

EXTON
EXTENABLE
EXT=ON
EXT=ENABLE
When this command is issued, Z/XDC turns on the retry level PSW's External Interrupt Mask, thereby permitting the target program to be interrupted by External Signals (timer, other CPUs, etc.). Eventually when the user program resumes execution, the CPU will be (as it is right now) enabled for External Interrupts. This is the CPU's normal running state.

EXTOFF
EXTDISABLE
EXT=OFF
EXT=DISABLE
When this command is issued, Z/XDC turns off the retry level PSW's External Interrupt Mask, thereby preventing the target program from being interrupted by External Signals (timer, other CPUs, etc.). Eventually when the user program resumes execution, the CPU will be disabled for External Interrupts. This command can be issued **only** when Z/XDC is running authorized. This command cannot be issued when Z/XDC is non-authorized.

Help COmmands SEt PSW Io

The **SET PSW** command can be used to enable or disable the I/O Interrupt Mask in the retry level PSW. Needless to say...

- This can be done only for programs that are running authorized.
- This should be done only by the most expert programmers who truly know what they are doing.

The current state of the retry level I/O Interrupt Mask can be displayed by: **LIST PSW FORMAT**

Syntax:

```
SET PSW   ... IOON       ...
    PSWE      IOENABLE
              IOOFF
              IODISABLE
              IO=ON
              IO=ENABLE
              IO=OFF
              IO=DISABLE
```

IOON
IOENABLE
IO=ON
IO=ENABLE
When this command is issued, Z/XDC turns on the retry level PSW's I/O Interrupt Mask, thereby permitting the target program to be interrupted by I/O Events. Eventually when the user program resumes execution, the CPU will be (as it is right now) enabled for I/O Interrupts. This is the CPU's normal running state.

IOOFF
IODISABLE
IO=OFF
IO=DISABLE
When this command is issued, Z/XDC turns off the retry level PSW's I/O Interrupt Mask, thereby preventing the target program from being interrupted by I/O Events. Eventually when the user program resumes execution, the CPU will be disabled for I/O Interrupts. This command can be issued **only** when Z/XDC is running authorized. This command cannot be issued when Z/XDC is non-authorized.

Help COmmands SEt PSW Key

The **SET PSW KEY** command sets the storage access key in the retry level PSW. If Z/XDC is running non-authorized, then this command can be used to set only those key values that are permitted by the retry level's PSW key-mask.

Normally, the key-mask permits only keys 8 and 9. But if Z/XDC is running authorized, then any key value may be set ranging from 0 to 15.

The current storage access key can be displayed by: **LIST PSW FORMAT**

Syntax:

```
    SET PSW   ... KEY key# ...
        PSWE      KEY=key#
```

KEY key#
KEY=key#
Key# must be either a decimal number ranging from 0 to 15 or a hexadecimal number ranging from 0 to F. Either way, it specifies the value to which Z/XDC is to set the execution key in the retry level's PSW[E].

Key# may be either one or two digits long.

Example:

```
SET PSW  K=10
SET PSWE KEY=A
SET PSW  KEY A
SET PSWE KEY 0A
```

All these forms set the retry level PSW's key value to ten.

Internally, there is no distinction between the fictional 8-byte (or "scrunched") PSW and the actual 16-byte PSWE. ("extended") PSWE. Externally, the only difference is in the way that they are displayed. See HELP COMMANDS LIST PSW for more information.

Help COmmands SEt PSW Prog

The **SET PSW** command can be used to enable or disable the four Program Mask bits in the retry level PSW. Those four mask bits govern whether the System will accept or ignore certain mathematical program checks. Those checks are:

- Fixed-point overflow
- Decimal overflow
- Hex-Floating-point underflow
- Hex-Floating-point significance

The current state of the retry level Program Interrupt Mask can be displayed by: **LIST PSW FORMAT**

Syntax:

```
    SET PSW   ... PROG=h ...
        PSWE  ... PGM=h  ...
```

PROG=h

PGM=h [where **h** is a single hex digit]
When this command is issued, Z/XDC sets the four Program Mask bits in the retry level PSW to the 4-bit value represented by the given hex digit.

To get a clear idea of which bit represents which mask, use the **LIST PSW FORMAT** command to get a bit-by-bit breakout of the settings.

Help COmmands SEt PSW State

The **SET PSW** command can be used to set the execution state of the retry level PSW either to **supervisor** state or **problem** state.

The current execution state can be displayed by: **LIST PSW FORMAT**

Syntax:

```
    SET PSW   ... [STATE=]SUPERVISOR ...
        PSWE              PROBLEM
```

SUPERVISOR
STATE=SUPERVISOR
When this command is issued, Z/XDC turns off the "problem state" flag in the retry level PSW. Eventually when the user program resumes execution, it will do so in supervisor state. This command can be issued **only** when Z/XDC is running authorized. This command cannot be issued when Z/XDC is non-authorized.

PROBLEM
STATE=PROBLEM
When this command is issued, Z/XDC turns on the "problem state" flag in the retry level PSW. Eventually when the user program resumes execution, it will do so in problem state. Note that this command can be issued when Z/XDC is running either authorized or non-authorized; however, using this command when Z/XDC is non-authorized does not accomplish anything since almost by definition the retry level PSW already would be set to problem state.

Help COmmands SEt PSWE

All changes seemingly made to 64-bit "PSWs" are actually made to 128-PSWEs. Then the 64-bit "scrunched" **view** of the PSWE is rebuilt.

In other words, when **SET'ing** or **ZAP'ing** the PSW, there is no difference between setting the PSWE and setting the PSW: All changes are first made to the 128-bit **PSWE.** Then the 64-bit PSW view is rebuilt from the updated PSWE information.

Accordingly, the **SET PSWE** command is nothing more than an alias of **SET PSW.**

Help COmmands SEt Qualifier

The **SET QUALIFIER** command sets, resets, or clears the default label qualifiers. These consist of a default load module name and a default csect name. They are used in situations where such names are needed but have been omitted. This occurs frequently in address expressions. (See **HELP ADDRESSING PARSERS ASM RESOLUTION** for information about how default qualifiers are used.)

Syntax:

```
SET QUALIFIER addressexpression  SILENT=YES
              omitted            omitted
```

Shortcut: Q

Operands:

addressexpression
This must be an address expression that resolves to some location within a **mapped** load module. The names of the load module and csect that contain the address become the default qualifiers. In the case of the mapped module being loaded from a USS file containing a program object, the **file name** part of the path name is used as the qualifier. For a detailed description of address expressions, see **HELP ADDRESSING.**

omitted
The previously defined default qualifiers, if any, are cleared.

SILENT=YES
Normally, the **S Q** command issues report messages showing what the default qualifiers have been set or cleared to. However, sometimes it would be better not to disrupt the existing display. When that is the case, just use this **S=Y** operand.

The default varies:
- When **S Q** is issued from a command line, the default is for the command to display its report messages.
- When the **Q** shortcut command is issued, the default is to not disrupt the existing display.

Examples:

Suppose the following:
- A module named XDCSYMED has been mapped.
- The XDCSYMED module contains a csect named DBCSYMED.
- The DBCSYMED csect contains an entry point named SYMEDZ.
- The retry level PSW points to XDCSYMED.DBCSYMED+D0.

Then a **WHERE** command might show the following:

```
TCB#10 RB#1 (DBC496W!) --------------- z/XDC ISPF INTERFACE -------------
XDC ===> w
_ (BRANCHED)(T I) TRACE - J
_ 000AF00C 8f (A.S.DBCOLE7) --- XDCSYMED.DBCSYMED+84, @R15+84, XDCSYMED...
_   +84                    @PRIOR   EQU  *
_   +84 A7F4 0026          @BEA     J    *+X'004C' (XDCSYMED.DBCSYMED.E028
_   +88 @R13
_   +88 SYMEDRSA 00000000 000AB468 00000000 00000000  *...................
_   +98          00000000 00000000 00000000 00000000  *...................
_   +A8          00000000 00000000 00000000 00000000  *...................
_   +B8          00000000 00000000 00000000 00000000  *...................
_   +C8          00000000 00000000                    *............*
_   +D0                   >@CDP     EQU  *
_   +D0 A7F4 0004         >E0283END J    *+X'0008' (XDCSYMED.DBCSYMED.OVER
_   +D4          00000000                             *....*
_   +D8 47F0 D078          over0283 B    X'078'(,R13)
_   +DC @EXLST   000AF07C                              *..0@*
_   +E0                    OLDIO    EQU  *
_   +E0 E3E0 D074 0056     XAMODE31 OY   R14,X'00074'(,R13)
_   +E6 0B0E                        BSM  0,R14
```

S Q PSW?
This does the following:
- The default qualifiers are set to XDCSYMED.DBCSYMED.
- The **primary window's** display is replaced with the following messages.
  _ **DEFAULT MODULE NAME IS DBCSYMED**
  _ **DEFAULT CSECT NAME IS DBCSYMED**
  DEFAULT DSECT MAPS MODULE IS DBCSYMED
- **Watch Windows** that happen to be displaying within the newly defaulted csect will by updated to show **.+offsets**

F PSW?;S Q +0
This has the same result as the preceding example. The **FORMAT** command sets the "Current Display Pointer". The "+0" references it.

S Q XDCSYMED.DBCSYMED
This has the same result as the preceding examples.

S Q XDCSYMED.SYMEDZ
This has the same result as the preceding examples. "XDCSYMED.SYMEDZ" is an address expression that resolves to a location within DBCSYMED. Note, "SYMEDZ" cannot be part of the default qualifiers because it is not a csect name.

S Q PSW? S=Y [or **Q** shortcut]
This does the following:
- The default qualifiers are set to XDCSYMED.DBCSYMED.

- If the **primary window** happens to be showing a display of the newly qualified code, it will be updated to show **.+offsets**
- Otherwise, the **primary window** will not be disturbed.
- **Watch Windows** that happen to be displaying within the newly defaulted csect will by updated to show **.+offsets**

S Q
This clears the default qualifiers.

In the current example, when all is said and done, a **WHERE** command's display will look like this:

```
TCB#10 RB#1 (DBC496W!) --------------- z/XDC ISPF INTERFACE -------------
XDC ===> w
_ (BRANCHED)(T I) TRACE - J
_ 000AF00C 8f (A.S.DBCOLE7) --- XDCSYMED.DBCSYMED+84, @R15+84, XDCSYMED...
_  .+84                    @PRIOR   EQU  *
_  .+84 A7F4 0026          @BEA     J    *+X'004C' (.E0283END)
_  .+88 @R13
_  .+88 SYMEDRSA 00000000 000AB468 00000000 00000000  *...................
_  .+98          00000000 00000000 00000000 00000000  *...................
_  .+A8          00000000 00000000 00000000 00000000  *...................
_  .+B8          00000000 00000000 00000000 00000000  *...................
_  .+C8          00000000 00000000                    *............*
_  .+D0                   >@CDP     EQU  *
_  .+D0 A7F4 0004         >E0283END J    *+X'0008' (.OVER0283)
_  .+D4          00000000                             *....*
_  .+D8 47F0 D078         over0283 B    X'078'(,R13)
_  .+DC @EXLST   000AF07C                              *..0@*
_  .+E0                   OLDIO    EQU  *
_  .+E0 E3E0 D074 0056    XAMODE31 OY   R14,X'00074'(,R13)
_  .+E6 0B0E                       BSM  0,R14
```

Notice:
- The offsets are shown as **.+offset**
- The jump targets have been shortened by removing the leading qualifiers: **XDCSYMED.DBCSYMED**

Help COmmands SEt READ

The **READ** command is used to read and execute Z/XDC command scripts.
The **SET READ** command makes various settings pertaining to the **READ** command.
The **LIST READ** command displays these settings.

The settings managed by **SET READ** are:

- The default Scripts Library to use when only a **membername** is given on the **READ** command.

- Whether the commands issued from the script (and the displays produced by those commands) are displayed as they are processed **(ECHO=ON)** or are delayed until the script completes **(ECHO=OFF).**

- Whether z/XDC should automatically detect the presence or absence of sequence fields in script records.

- Whether z/XDC should halt processing of the script if an error occurs in a command.

Syntax:

```
SET READ SCRIPTLIBRARY=dsname   ECHO=YES  SEQUENCEFIELD=PRESENT ...
         LIBNAME=      NONE          NO   SEQFIELD=     ABSENT
         LIBRARY=      omitted       ON   SEQF=         DETECT
         DSNAME=                     OFF
                                ECHO
                                NOECHO
         ... ERROR=CONTINUE
                   STOP
                   FORCESTOP
```

Rules:

- At least one operand must be given.

- Operands may be given in any order.

- Operands shown above within the same column are mutually exclusive with each other.

- No setting changes are made unless the entire command parses cleanly.

Operands:

DSNAME=
LIBNAME= **omitted**
LIBRARY= **NONE**
SCRIPTLIBRARY= **dsname**
This specifies the name of the Scripts Library. Rules:

- DSNAME=, LIBNAME=, LIBRARY= and SCRIPTLIBRARY= are all aliases of each other.

- When the **dsname** value is given, it must follow these rules:

  - **dsname** must be a dataset name that follows the classic JCL rules for dsname syntax...

  - ... Except that the dsname must consist of **two or more** elements. (I.E. the dsname must contain at least one period.)

- In other words, the **READ** command does not support "simple" dataset names (i.e. single element names) for script files and libraries. They must have **compound names.**

- TSO syntax rules are **not** supported. Therefore:
  - The given name must be **fully qualified.** In other words, a TSO userid will never be prefixed to the name that you give. You will have to do that yourself.
  - The given name must be **unquoted.** I.E. the syntax DSNAME='dsname' is not accepted.

- The dataset referenced by dsname **should:**
  - Exist.
  - Be cataloged.
  - Be a PDS or PDSE.
  - Contain members that contain Z/XDC scripts in EBCDIC text format.
  - Have any reasonable DCB attributes (RECFM, LRECL, BLKSIZE).

  However, **none** of the above "shoulds" are verified at **SET READ** time. Violations of the above will be detected only at **READ** command time. (This is so that this SET READ DSN= command can be issued prior to the actual existence of the library.)

- **NONE** is a keyword indicating that you wish to proceed without a default Scripts Library. Specifying **DSN=** (i.e. omitting the value) is equivalent to specifying **DSN=NONE.**

- **NONE** may not be abbreviated.

SEQUENCEFIELD= PRESENT
SEQFIELD= ABSENT
SEQF= DETECT

When sequence fields are present in a script, they usually contain 8 decimal digits, but nowadays with the advent of **git** and the decline of the importance of sequence numbers in modern Software Change Management methodologies, they can easily contain non-numeric and even arbitrary characters that the coder **might still** want to have ignored.

The **SEQFIELD=** operand (and its variations) allows you to control whether or not Z/XDC will attempt to automatically detect the presence or absence of sequence fields in scripts. Here are the details:

SEQUENCEFIELD=PRESENT
SEQFIELD=PRESENT
SEQF=PRESENT
- The sequence field is considered to be present in all records regardless of content. The field will be ignored in all records.

SEQUENCEFIELD=ABSENT
SEQFIELD=ABSENT
SEQF=ABSENT
- The sequence field is considered to be absent from all records. Whatever content is found in the sequence field position will be considered to be part of the data found in the rest of the record.

SEQUENCEFIELD=DETECT
SEQFIELD=DETECT
SEQF=DETECT

- Z/XDC itself will determine the presence or absence of sequence fields based upon its examination of the file's first record. The determination will be made according to the rules discussed in HELP SCRIPTS RYOSCRIPTS SEQUENCEFIELDS. (Note, since sequence fields can contain non-numeric data, the detection rules are probably a bit more complicated than you might expect.)

SEQFIELD=DETECT is the factory default.

ECHO=YES
ECHO=ON
ECHO
These are all synonyms of each other. They enable the **READ** command's **ECHO** feature: Normally, the displays and reports produced by a script's commands are delayed until the script finishes or fails. Sometimes, depending upon what the script is doing, this can take quite a long time, during which the terminal screen remains frozen, and the keyboard remains locked. This, of course, can cause the user to wonder whether or not Z/XDC has failed completely.

When debugging programs that, themselves, are running in TSO (e.g. options **1** or **2** of the **Z/XDC STARTUP PANEL** in ISPF), this **Read Echoing** facility causes the scripted commands and their resulting displays to be written to the user terminal in a **linemode** fashion immediately as the commands are being executed. Thus, the user is able to monitor his script's progress as it is running.

Unfortunately, this **Read Echoing** facility is **not** supported by cdf/XDC. So this setting has no effect when debugging background jobs via cdf/XDC (options **3** of the **Z/XDC STARTUP PANEL).**

ECHO=NO
ECHO=OFF
NOECHO
These are all synonyms of each other. They disable the **READ** command's **ECHO** feature.

ERROR=CONTINUE
ERROR=STOP
ERROR=FORCESTOP
These operands affect what a script does when a command it contains fails. The choices are to suspend the script or to continue processing the script as if nothing happened.

The various keywords of the **ERROR=** operand manage a nesting counter as follows:
- **ERROR=CONTINUE** increases the counter by one.
- **ERROR=STOP** decreases the counter by one.
- **ERROR=FORCESTOP** zeros the counter.

When the nesting counter is positive, command errors do not cause scripts to stop. They simply continue to process the next command.

When the nesting counter is zero, command errors cause scripts to suspend. You will then have the opportunity to manually issue a corrected command (if needed) and then resume the script with a **READ RESUME** command. (Or you can close the script or otherwise ignore the suspension, in which case it will be automatically closed in due course).

At the start of a debugging session, the nesting counter will be zero, thus the initial state will be to suspend scripts upon errors.

Comments:

All settings made by the **SET READ** command (except for ERROR=) are saved in your Debugging Session profile. But to make them permanent, you will have to issue a **PROFILE SAVE** command. (The **ERROR=** setting will not be saved.)

Because these **READ** command settings (except for ERROR=) are profiled, you can use the **Profile Menuing System** as an alternate way to display and change them. For more information, see HELP PROFILES MENU READZAPPARSE.

Help COmmands SEt READEcho

This command has been functionally replaced by the **SET READ** command. Please discontinue using this command and use that one instead.

Help COmmands SEt REFrprot

Starting with z/OS R1.9, MVS has a configuration setting, named REFRPROT, that affects how the System loads **refreshable** modules into storage. Previously, a load module's **REFR** attribute had no affect as to whether a module was loaded into user key storage (usually key 8) or key 0 storage. (It was the RENT attribute that affected this, not REFR). But starting with z/OS R1.9, if a data center decides to "turn on" the REFRPROT facility, then load modules and program objects having the REFR attribute will always be loaded into key 0 storage, **regardless** of whether or not it was loaded from an APF authorized, library, and **regardless** of whether or not the RENT attribute also is on.

Notes:

- In z/OS R1.9 (and newer systems) the REFRPROT facility has to be turned on by adding a REFRPROT statement to the PROGxx members of the System's PARMLIB libraries. See IBM's "z/OS MVS Initialization and Tuning Reference" manual (SA22-7592, -15 and newer) for details.

- z/OS R1.9 (and newer systems) supports both a system-wide REFRPROT setting and a task-level REFRPROT override. Z/XDC's **LIST REFRPROT** command can be used to display these settings and overrides. The **SET REFRPROT** command can be used to change the task-level overrides.

- If the SET REFRPROT command is issued on any system older than z/OS R1.9, then after parsing and checking all operands, the command simply reports that REFRPROT support is not present.

- Normally, the **SET REFRPROT** command can be used only when Z/XDC is running authorized. However, when the only TCB being targeted by the command is the current TCB, then authorization is not needed.

Syntax:

```
SET REFRPROT OFF      ALL          SUBTASKS  ASPACEREF
             SYSTEM   CURRENT      omitted   omitted
                      tcbaddress
                      omitted
```

Rules:

- One or the other of the OFF and SYSTEM operands is required. All other operands are optional. Varying defaults will be taken for omissions.

- Operands may be given in any order.

- Operands appearing above within the same column are mutually exclusive.

- SUBTASKS and ALL are mutually exclusive.

OFF
SYSTEM
These operands allow you to control the task-level REFRPROT setting for any TCB in any accessible address space. They do this as follows:

- **OFF**
  This operand causes Z/XDC to turn REFRPROT **off** for the targeted task(s). (It does this by turning the REFRPROT **override flag,** TCB_REFRPROT_OVERRIDE, on.)

- **SYSTEM**
  This operand causes Z/XDC to set the task-level REFRPROT setting(s) for the targeted task(s) to match the system-wide setting. It does this by turning the REFRPROT **override flag,** TCB_REFRPROT_OVERRIDE, off.)

aspaceref

This operand specifies the address space whose task-level REFRPROT overrides are to be altered. (Each task within an address space has its own override setting.) If omitted, then the space currently being targeted via Foreign/Local Address Space Mode (FASM/LASM) will be targeted. (See HELP VIRTMEM XDCACCESS FASM for more information.)

The aspaceref operand may be any of the following:
- A 1-4 digit hexadecimal number.
- A 1-8 character job name or TSO userid.

- An address space keyword (HOME, PASID, SASID, etc. See HELP COMMANDS SYNTAX ASIDS for a complete list of legal keywords.)
- CR3, CR4, ECR3 or ECR4.

Note, unlike other commands that accept aspaceref operands, this command will **not** accept general register names, address expressions, equate names and dsect names as address space references. For complete details, see HELP COMMANDS SYNTAX ASIDS.

CURRENT
This operand causes the SET REFRPROT command to target the home address space's current task. It may be used only:

- When the aspaceref operand is given and it resolves to the home address space.
- Or when the aspaceref operand is omitted and Z/XDC is not running in Foreign Address Space Mode (FASM).

When CURRENT is specified or implied (and SUBTASKS is not specified), The SET REFRPROT command can be used regardless of whether or not Z/XDC is running authorized. (In all other cases, the SET REFRPROT command will fail unless Z/XDC is running authorized.)

The CURRENT operand is mutually exclusive with the ALL and tcbaddress operands.

ALL
This operand causes the SET REFRPROT command to target "all" tasks in the targeted address space.

Well, **ALL** doesn't actually target all tasks. It really targets only the jobstep task (pointed to by ASCBXTCB) and all of its subtasks. Omitted are the system tasks (the Region Control Task, Started Task Control, and the System Dump Task).

This operand is mutually exclusive with the CURRENT, SUBTASKS and tcbaddress operands.

tcbaddress
This operand must be an address expression that resolves to the starting address of any TCB located in any accessible address space anywhere in the system. This operand causes the SET REFRPROT command to target the specified task. Notes:

- Since address expressions resolve to locations within address spaces, the aspaceref operand is not needed for specifying the address space within which the desired TCB resides.

- If both the tcbaddress and aspaceref operands are given, then they must resolve to the same space.

- This operand is mutually exclusive with the ALL and CURRENT operands.

tcbaddress, CURRENT and ALL all **omitted**
- When the home address space is targeted, **CURRENT** is the default.
- When a foreign address space is targeted, **ALL** is the default.

SUBTASKS
This operand causes the SET REFRPROT to target, not only the specified (or implied) task, but also all currently existing descendant tasks of the specified (or implied) task.

The SUBTASKS operand works in conjunction with the CURRENT and tcbaddress operands:

- When SUBTASKS is omitted, the only targeted task will be the specific task

identified by the CURRENT or tcbaddress operand.

- When SUBTASKS is given, the set of targeted tasks is expanded to include all descendant tasks of the specified tasks.

- With respect to new subtasks, the SUBTASKS operand as no affect; **however,** whenever an ATTACH occurs, the System automatically propagates the parent task's TCB_REFRPROT_OVERRIDE flag setting to the subtask being created.

SUBTASKS is mutually exclusive with the ALL operand.

Examples:

When REFRPROT support is installed, then the following examples will have the results indicated.

SET ASID
SET REFRPROT OFF

SET REFRPROT OFF HOME

SET REFRPROT OFF HOME CURRENT
Each of these command sequences accomplishes the same result: When the home address space is targeted, the default task that is targeted is the current task.

In each of these examples, the REFRPROT facility is turned off for the home address space's current task. (This is done by turning on the task's REFRPROT override flag, TCB_REFRPROT_OVERRIDE.)

SET REFRPROT SYSTEM ALL JES2

LIST TASKS JES2
SET REFRPROT SYSTEM TCB#3 SUBTASKS
Both of these command sequences accomplish the same result: In JES2's address space, for both the jobstep task and all of its descendant tasks, the REFRPROT facility is reset to match the system-wide setting, (This is done by turning off the task's REFRPROT override flag, TCB_REFRPROT_OVERRIDE.)

LIST TASKS
SET REFRPROT OFF TCB#1 SUBTASKS
This example shows how to change the REFRPROT setting for truly all tasks in an address space, including the system tasks:

- This LIST TASKS command creates TCB#n equates labeling the locations of all tasks (including the system tasks) in whatever address space is currently being targeted by Foreign/Local Address Space Mode.

- The TCB#1 equate labels the address space's top task's TCB.

- The SET REFRPROT OFF TCB#1 SUBTASKS command then turns off the REFRPROT facility both for that top task and all of its descendant tasks (i.e. for every task in the address space).

Help COmmands SEt REXx

This command allows you to change some processing options of the Z/XDC REXX interface.

Syntax:

SET REXX operand ...

Where each operand can be:

IO={DD | XDC | BOTH}	**[NOFORCE | FORCE]**
INPUT={DD | XDC}	**[NOFORCE | FORCE]**
OUTPUT={DD | XDC | BOTH}	**[NOFORCE | FORCE]**
NAME={PROC | CHECK}	**[NOFORCE | FORCE]**

Rules:

Operands can be in any order.
Operands must not be duplicated.
At least one operand must be specified.

IO={DD | XDC | BOTH} [FORCE | NOFORCE]
This operand may change a REXX procedure's input and output routing.

IO=DD can be used to indicate that a REXX procedure is to obtain its [PARSE] PULL and PARSE EXTERNAL input from a file (allocated to a DDname). and that a REXX procedure is to send the output from the SAY instruction and tracing to a file (allocated to a DDname).

IO=XDC can be used to indicate that a REXX procedure is to obtain its [PARSE] PULL and PARSE EXTERNAL input from Z/XDC, and that a REXX procedure is to send the output from the SAY instruction and tracing to Z/XDC.

IO=BOTH can be used to indicate that a REXX procedure is to obtain its [PARSE] PULL and PARSE EXTERNAL input from Z/XDC, and that a REXX procedure is to send the output from the SAY instruction and tracing to both a file (allocated to a DDname) and XDC.

NOFORCE (the default) indicates that the value set by the IO= operand is only honored by a REXX procedure if the **XDCROUTE** function has been used to set routing to SET.

FORCE indicates that the value set by the IO= operand is to be forced onto all REXX procedures, regardless of the use of the **XDCROUTE** function.

INPUT={DD | XDC} [FORCE | NOFORCE]
This operand may change a REXX procedure's input routing.
INPUT=DD can be used to indicate that a REXX procedure is to obtain its [PARSE] PULL and PARSE EXTERNAL input from a file (allocated to a DDname).
INPUT=XDC can be used to indicate that a REXX procedure is to obtain its [PARSE] PULL and PARSE EXTERNAL input from Z/XDC.
NOFORCE (the default) indicates that the value set by the INPUT= operand is only honored by a REXX procedure if the **XDCROUTE** function has been used to set input routing to SET.
FORCE indicates that the value set by the INPUT= operand is to be forced onto all REXX procedures, regardless of the use of the **XDCROUTE** function.

OUTPUT={DD | XDC | BOTH} [FORCE | NOFORCE]
This operand may change a REXX procedure's output routing.
OUTPUT=DD can be used to indicate that a REXX procedure is to send the output from the SAY instruction and tracing to a file (allocated to a DDname).
OUTPUT=XDC can be used to indicate that a REXX procedure is to send the output from the SAY instruction and tracing to Z/XDC.
OUTPUT=BOTH can be used to indicate that a REXX procedure is to send the output from the SAY instruction and tracing to both a file (allocated to a DDname) and XDC.
NOFORCE (the default) indicates that the value set by the OUTPUT= operand is only honored by a REXX procedure if the **XDCROUTE** function has been used to set output routing to SET.
FORCE indicates that the value set by the OUTPUT= operand is to be forced onto all REXX procedures, regardless of the use of the **XDCROUTE** function.

NAME={PROC | CHECK} [FORCE | NOFORCE]
This operand may change the meaning of a simple name when used by the REXX command (and its aliases) to indicate a procedure to be executed.
NAME=PROC can be used to indicate that if a simple name is specified on the REXX (etc.) command the name is always treated as a procedure name, unless immediately followed by an opening parenthesis, in which case the simple name is treated as a DDname.
NAME=CHECK can be used to indicate that if a simple name is specified on the REXX (etc.) command the name is used as a DDname if an allocation for that DDname is found, otherwise the name is treated as a procedure name, unless immediately followed by an opening parenthesis, in which case the simple name is treated as a DDname.
NOFORCE is recognized but ignored.
FORCE is recognized but ignored.

Notes

The current settings can be saved in your session profile, and so they can be both displayed and set via the **Profile Menuing System.** See **HELP PROFILES MENU** for more information.

If input and/or output is routed to a DD, records are truncated at the record length or 255 bytes, whichever is less.

Examples:

SET REXX IO=XDC FORCE
This command forces all REXX procedure input and output to be processed by XDC.

Help COmmands SEt SCreen

This command is an alias of the **SET WINDOW** command. See HELP COMMANDS SET WINDOW for more information.

Help COmmands SEt SECOndarysize

Some fullscreen terminals have a screen that may be formatted in two different ways. They have a primary set of dimensions (lines and columns) and a secondary (or alternate) set. Normally, a terminal's "primary" dimensions are 24x80 (24 lines by 80 columns), and its "secondary" dimensions are larger.

The **SET SECONDARYSIZE** command causes Z/XDC to format the screen using that terminal's secondary set of dimensions.

Syntax:

SET SECONDARYSIZE

This command accepts no operands.

The terminal's primary dimensions can be restored via the **SET PRIMARYSIZE** command.

The screen size choice can be saved in your profile dataset for automatic restoration every time you start a debugging session. For more information, please see:

HELP PROFILES
HELP COMMANDS PROFILE

The distributed version of Z/XDC has the default set to SECONDARYSIZE.

Help COmmands SEt SECUrity

The **SET SECURITY** command affects Z/XDC's system security interface in several ways:

- It can enable or disable a trace of all calls that Z/XDC makes to the security interface.
- It can force the immediate expiration of Z/XDC's internal security cache. (See

HELP SECURITY CACHE for more information about this.)

- It can reset the triggers for several security related one-time events. (See below.)

Syntax:

SET SECURITY TRACE FLUSHCACHE RESETCONTROLS
NOTRACE

Rules:
- At least one operand must be given.
- Operands may be given in any order.
- Operands occurring in the same column are mutually exclusive.

Operands:

TRACE
NOTRACE
This operand either enables or disables a security calls trace. When enabled, the trace messages are displayed at the user's terminal along with all of his other displays. Each trace report consists of several display lines showing (among other things):
- The reason for the security call.
- The resource name that Z/XDC constructs to represent the resource that is being protected.
- The specific security rule that the security system found that applies to the resource name.
- The final ruling that resulted from the security call.

For complete details regarding the trace reports, SEE HELP SECURITY TRACE.

Note, another way to activate a security calls trace is to preallocate a **//xxxTRSAF DD DUMMY** file (where **xxx** is Z/XDC's current clone name, usually **XDC).** At the start of a debugging session, if Z/XDC detects the presence of this ddname, then it will automatically activate the security calls trace. This allows the trace to be activated prior to it being possible to issue a SET SECURITY command.

FLUSHCACHE

This forces the immediate expiration of Z/XDC's internal security cache. (Note, the maximum lifetime of any cache entry is never more that two real-time minutes.) See HELP SECURITY CACHE for more information about this.

RESETCONTROLS
There are several security related events (warning messages, mostly) that Z/XDC normally allows to happen only one time per debugging session. This operand can be used to reset the flags that prevent reoccurrences, thus allowing those events to occur once again. Those events are:

- **DBC981W:** This message normally is issued only the first time during a debugging

session that Z/XDC makes a security ruling based upon an old format security rule instead of a new one. See HELP MESSAGES DBC981 for more information.

- **DBC953W:** This message normally is issued only the first time during a debugging session that Z/XDC discovers that it needs to resort to a referee rule (to resolve contradictory rulings from old format rules and new format rules) and it discovers that you do not have a READ permit to such a rule. See HELP MESSAGES DBC953 for more information.

- **Referee:** Normally, once Z/XDC determines that a referee rule exists and the current user has a READ permit for that rule, it remembers that ruling and does not make that check again. RESETCONTROLS clears the current debugging session's memory of the referee rule so that if a need for it arises again, then Z/XDC will call system security again.

Help COmmands SEt SIgnonwait

The **SET SIGNONWAIT** command relates to cdf/XDC.

When a background batch job or system task ABENDs, and when that job is properly set up for debugging via Z/XDC, cdf/XDC gains control and waits for an appropriate user to "signon" to a debugging session with the ABENDed program. (See **HELP XDCSRVER CDF** for more information.)

Generally in this situation, cdf/XDC will wait only a limited amount of time before ending the session and percolating the abend.

This time limit is controlled by the **SET SIGNONWAIT** command.

In order for the **SIGNONWAIT** setting to be effective, it must preexist in your default session profile. For details, see the following:

- A **PROFILE SAVE** command has to be issued to save this setting into the user's personal profile. And
- A **//ISPPROF** DD card pointing to the profile library must be included in the JCL for the job to be debugged via cdf/XDC.

Syntax:

```
SET SIGNONWAIT minutes
               0
```

minutes
This is a decimal number that defines how long cdf/XDC will wait for a user to signon to the debugging session. Accepted values range from 1 to 32767 minutes.
Example:

S SI,60
This causes a background debugging session to wait up to an hour for a user to signon to it.

If the time limit expires without a user signon, then the debugging session is ended, and the abend is percolated.

0
A time-limit of zero minutes indicates that no time limit should apply to the signon wait.

Signon time limits can be saved in your profile dataset for automatic restoration every time you start a debugging session. For more information, please see:

HELP PROFILES
HELP COMMANDS PROFILE

Z/XDC's default signon wait time limit is determined by your session profile. Z/XDC's factory default wait time limit is one hour.

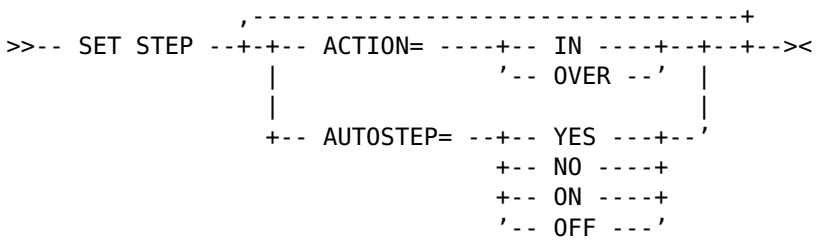
The factory defaults for all profiled values can be loaded at any time by the **PROFILE RESET** command.

During a pending signon, Z/XDC issues a **DBC640Q** WTOR that allows you some alternatives to signing on to the debugging session. You may, for example, tell Z/XDC to **just go,** i.e. just let the program start running without waiting for you to signon to the session. See **HELP COMMANDS SET DBC640Q** for more information.

Help COmmands SEt STep

The SET STEP command is used to change the default action of the STEP command:

Syntax:



ACTION
This is the default action of the STEP command when it is issued with no operands. Refer to HELP COMMANDS STEP for a complete description of ACTION=IN and ACTION=OVER. ACTION=OUT is not allowed here.

AUTOSTEP
The AUTOSTEP feature is designed to automatically skip stepping through function prolog and epilog code in C programs. The prolog and epilog code are the "pre-" and "post-" function routines that are generated by the compiler to implement register

save areas and automatic variable storage. The PROFILE RESET C command sets the initial value of SET STEP AUTOSTEP=ON. This has the effect of setting the breakpoint for C functions (including main()) at the first line of source code (after execution of the prolog) rather than at its prolog code. When a function returns to its caller then the next breakpoint will be in the caller rather than at the end of the called function. This is typically what you want to do.

If you do want to trace the prolog and epilog code of a function then specify SET STEP AUTOSTEP=OFF before issuing the STEP command.

There's a problem if you wish to trace the prolog to **main()** because when AUTOSTEP'ing is on, by the time Z/XDC is ready to accept a command from you, to turn it off, the prolog code has already been executed. But if you have a **//xxxNSTEP DD DUMMY** allocation, then AUTOSTEP'ing will be suppressed at Debugging Session startup time regardless of the AUTOSTEP setting saved in your Session Profile. For more information, see HELP DDNAMES NSTEP.

Help COmmands SEt TFs

The **SET TFS** command turns Z/XDC's Fullscreen Support on or off.

When Z/XDC's Fullscreen Support is **turned on:**

- Z/XDC displays are presented in fullscreen mode.
- Function keys are supported.
- Session scrolling is available.
- Session logging is available.
- Watch Windows are available.
- Session profile management is available.

Conversely, when Z/XDC's Fullscreen Support is **turned off:**

- Z/XDC sends all of its messages using linemode TPUTs.
- Fullscreen displays are not available
- Function keys are not available
- Scrolling is not available
- Session logging is not available
- Watch Windows are not available
- Profile management is not available

Syntax:

```
    SET TFS ON [or omitted]
            OFF
```

ON
omitted
This turns Z/XDC's Fullscreen Support on.

OFF
This turns Z/XDC's Fullscreen Support off.

For more information, please see:

HELP FULLSCREEN
HELP USERINTERFACES

Help COmmands SEt TImeout

The **SET TIMEOUT** command sets a timeout value that is useful in multi-tasking debugging situations.

Suppose that two or more tasks in an address space have ABENDed and are awaiting debugging services. Z/XDC has permitted one of those tasks to proceed with a debugging session, and it has caused the others to wait. Now suppose that the user has issued a "TRACE" or "GO" command to let the one task resume execution. That task (as in the case of the "TRACE" command) may return to debugging services very quickly or it may not (as in the case of a "GO" command with or without breakpoints set).

If such a task is resumed and does **not** return quickly to debugging services, then it is appropriate that another task that is awaiting debugging services be released so that it can be debugged.

The **SET TIMEOUT** command establishes the number of seconds that Z/XDC is to wait after losing control of the terminal under one task before it releases other tasks awaiting debugging services.

Syntax:

SET TIMEOUT seconds

seconds
This is a decimal number that specifies the length, in seconds, of the timeout interval. Its value may range from 1 to 3600 seconds (one hour).

The distributed version of Z/XDC has the default timeout value set to 5 seconds.

The timeout value can be saved in your profile dataset for automatic restoration every time you start a debugging session. For more information, please see:

HELP PROFILES
HELP COMMANDS PROFILE

Help COmmands SEt TPut

The **SET TPUT** command tells Z/XDC to paint its displays to your 3270 terminal using **fullscreen TPUTs.** If Z/XDC had been using ISPF Display Services, it will switch to using fullscreen TPUTs, instead.

Also, the display title line’s center caption will switch:

- From showing **z/XDC ISPF INTERFACE**
- To showing **z/XDC TPUT INTERFACE**

Syntax:

SET TPUT

This command accepts no operands.

For more information about communication interfaces, please see **HELP USERINTERFACES.**

The opposite of **SET TPUT** is **SET ISPF.** See **HELP COMMANDS SET ISPF** for more information.

The choice of which Z/XDC fullscreen interface to use can be saved in your session profile for automatic restoration every time you start a debugging session. For more information, please see:

HELP PROFILES
HELP COMMANDS PROFILE

Help COmmands SEt TRace

The **SET TRACE** command is used to set the following tracing related characteristics:

- Whether tracing displays **SCROLL** so as to maintain the current execution pointer at the top of the display, or remain stable so that the execution pointer **ROLLs** down the display.

- Whether or not execution tracing attempts to follow **BAL-type** instructions (BALR, BAL, BASR, BAS, BASSM and BRAS) into **"alien"** areas of storage.

- Whether **X'00'** opcodes or **TRAP2** machine instructions are used for breakpoints.

- Whether or not **T BY** and **T BNC** traces will stop at conditional **DEAD traps** that are about to be bypassed.

Syntax:

```
SET TRACE ROLL   LOCAL  ZERO  IGNORE
          SCROLL ALL    TRAP2 STOP
                 GLOBAL
```

Notes:

All operands are optional, but at least one operand must be given.

Operands may be given in any order.

Operands appearing above within the same column are mutually exclusive.

The current trace settings can be displayed via the **LIST TRACE** command.

The current settings can be saved in your session profile, and so they can be both displayed and set via the **Profile Menuing System.** See **HELP PROFILES MENU** for more information.

Subtopics Index

Specific operand descriptions can be selected below. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

IGNORE - Using the **IGNORE/STOP** operands
LOCAL - Using the **LOCAL/GLOBAL/ALL** operands
ROLL - Using the **ROLL/SCROLL** operands
TRAP2 - Using the **TRAP2/ZERO** operands

Help COmmands SEt TRace Ignore

When the **#DIE macro** creates a conditional DEAD trap, it usually will create that trap inline and preceded it with a conditional branch to be taken should the tested-for condition not be met.

When stepping through code using the **T BY** and **T BNC** commands, it is both rather tedious and quite uninformative for the trace to be stopping at these conditional DEAD traps when they are about to be bypassed.

So this setting has been implemented to control whether **(STOP)** or not **(IGNORE)** branch tracing is to stop at conditional DEAD traps that are about to be skipped around by execution.

For more information, see:
HELP BREAKPOINTS TRACING
HELP BREAKPOINTS DEADTRAPS
HELP COMMANDS TRACE

Syntax:

```
SET TRACE IGNORE
          STOP
```

Operands:

IGNORE [default]
Branch tracing will **not** stop at conditional DEAD traps that are about to be bypassed.

STOP
Branch tracing will stop at all DEAD traps, including those that are about to be bypassed.

The current trace settings can be displayed via the **LIST TRACE** command.

The current settings can be saved in your session profile, and so they can be both displayed and set via the **Profile Menuing System.** See **HELP PROFILES MENU** for more information.

Help COmmands SEt TRace Local

The **SET TRACE LOCAL/GLOBAL** command controls whether or not Z/XDC, while tracing, attempts to follow execution into **alien** storage. (For a discussion of **transfers into alien storage,** see **HELP BREAKPOINTS TRACING ALIEN.)**

The current trace settings can be displayed via the **LIST TRACE** command.

The current settings can be saved in your **session profile,** and so they can be both displayed and set via the **Profile Menuing System.** See **HELP PROFILES MENU** for more information.

For information about breakpoints in general, see **HELP BREAKPOINTS.**

Syntax:

SET TRACE LOCAL
GLOBAL

Default: **LOCAL**

Operands:

LOCAL
During traces using the **T, T B** and **T BY** commands, if Z/XDC encounters a **linkage** instruction (BASR, JAS, etc.), **and** if that instruction is about to transfer execution into **alien** storage, then Z/XDC will **not** attempt to set a breakpoint at the transfer target. Instead, it will set the next breakpoint on up to six instructions that sequentially follow the linkage instruction.

The **LOCAL** setting does not affect **T BNC, T SB** and **T SA** traces because they do not

attempt to follow call-with-return type linkage instructions. See **HELP BREAKPOINTS TRACING** for more information.

Note, when stepping over subroutines, Z/XDC assumes that the subroutine will return **only** to one of the first six machine instructions following the linkage instruction. If this assumption proves wrong, then control of execution will be lost. For more information, see **HELP BREAKPOINTS TRACING CONDITIONAL FAILURES.**

On the other hand, when the subroutine being stepped over is an **access method** that may exit to a **SYNAD** or **EODAD** routine, Z/XDC has special logic to find those exits and place recapture traps there as well. There is more information available at **HELP BREAKPOINTS TRACING ALIEN.**

GLOBAL [alias: **ALL**]
This operand nullifies the TRACE LOCAL setting.

During **T, T B** and **T BY** tracing, when Z/XDC encounters a linkage instruction, it will (as it would for any branching instruction) attempt to set the next breakpoint at the transfer target instead of at the assumed return address. That attempt may or may not succeed, as discussed below.

Linkage instructions are BAL, BALR, BAS, BASR, BASSM, BRAS, BAKR and JAS as well as EX/EXRL of any of these.

Alien Storage

If SET TRACE GLOBAL is in effect, and if Z/XDC determines that it is about to set a breakpoint in **alien** storage, then what happens next depends upon whether or not Z/XDC is running authorized:

- If Z/XDC is **not** authorized, then any attempt to set a breakpoint in alien storage fails.
- If Z/XDC is authorized, then it pauses and issues a message **(DBC803Q)** to ask the user if he really really wants Z/XDC to proceed with setting the breakpoint. If the user says yes, then Z/XDC does so. If the user says no, then Z/XDC aborts trace processing.

For a definition of alien storage, see **HELP BREAKPOINTS TRACING ALIEN.**

Help COmmands SEt TRace Roll

The **SET TRACE ROLL/SCROLL** command determines how execution trace processing displays storage. When execution is interrupted at a breakpoint, in the absence of an automatic commands string associated with the breakpoint (or with any WATCH), Z/XDC will generate a display of the current interrupt location.

When **SET TRACE SCROLL** is in effect, Z/XDC will always scroll the tracing displays

such that the execution pointer is maintained at the top of the display. It does this by internally executing a **FORMAT EPSW?** command.

On the other hand, when **SET TRACE ROLL** is in effect, Z/XDC will, when possible, maintain a static display with the execution pointer rolling down the display. It does this by internally executing an **EWHERE** command.

Generally, the **WHERE** and **EWHERE** commands will produce identical displays **except** when the **retry level** and **error level** environments are **different.** Then:

- The **WHERE** command will show the **retry level resume address,**
- While the **EWHERE** command (and, therefore, the **TRACE** command) will show the **error level ABEND address.**

For more information, see **HELP EXECUTIONLEVELS.**

Syntax:

```
SET TRACE ROLL
          SCROLL
```

Default: **SET TRACE ROLL**

SCROLL and **ROLL** are mutually exclusive.

The current trace settings can be displayed via the **LIST TRACE** command.

The current settings can be saved in your session profile, and so they can be both displayed and set via the **Profile Menuing System.** See **HELP PROFILES MENU** for more information.

Help COmmands SEt TRace Trap2

The **SET TRACE TRAP2|ZERO** command controls whether Z/XDC uses X’00’ opcodes or TRAP2 machine instructions (X’01FF’) for breakpoints.

Syntax:

```
SET TRACE TRAP2 INSTALL
                FORCE
                omitted
          ZERO
```

Default: **ZERO**

TRAP2
Breakpoints are implemented using TRAP2 machine instructions (X’01FF’) instead of

X’00’ opcodes. This causes Z/XDC to receive control via a Trap Exception instead of a program check ABEND. The advantage is that the System overhead necessary to process program checks is avoided.

The **INSTALL** option can be specified to cause the Z/XDC TRAP handler to be immediately installed for the current execution unit if it has not been installed already, and there is no other TRAP handler installed. In addition a TRAP handler will be installed for each execution unit that enters Z/XDC.

The **FORCE** option can be used to force the Z/XDC TRAP handler to be installed even if another TRAP handler is found. (The other TRAP handler will be chained to if Z/XDC determines that the TRAP2 or TRAP4 instruction is not because of a breakpoint that Z/XDC has set. In addition a TRAP handler will be installed for each execution unit that enters Z/XDC.

Note that the INSTALL and/or FORCE operands are not remembered in the profile but will be remembered for this Z/XDC session.

ZERO

Breakpoints are implemented using X’00’s. This is an illegal opcode that gives rise to a program check ABEND. When Z/XDC is properly set up, the ABEND will eventually cause the System to pass control to Z/XDC as an ESTAE-type (or FRR-type) ABEND recovery routine.

Any INSTALL or FORCE options pertaining to a previous **SET TRACE TRAP2** will be cleared (reset).

For more information, see HELP BREAKPOINTS TYPES.

The current trace settings can be displayed via the **LIST TRACE** command.

The current settings can be saved in your session profile, and so they can be both displayed and set via the Profile Menuing System. See HELP PROFILES MENU for more information.

Help COmmands SEt USErid

The **SET USERID** command sets the TSO userid to be notified if and when a background program needs to be debugged via cdf/XDC. (See **HELP XDCSRVER CDF** for more information about connecting to background debugging sessions.)

Syntax:

```
SET USERID  userid
    LOGONID *
```

[blank]

Operands

userid
This must be the userid of the TSO user who is to be notified when a background program needs to be debugged via the Cross Domain Facility.

* [asterisk]
Notification is made to the user who owns the job to be debugged via cdf/XDC.

[blank]
When no operand is given, userid notification is suppressed. This is the factory default setting.

Notes

- The **USERID** setting can be saved in your session profile.
- It can be displayed by the **LIST USERID** command.
- It also can be displayed and changed by Z/XDC's **Profile Menuing System.**
- In order for this setting to be effective, it must preexist in your default session profile. For details, see the following:
  - A **PROFILE SAVE** command has to be issued to save this setting into the user's personal profile. And
  - A **//ISPPROF** (or **//XDCPROF)** DD card pointing to the profile library must be included in the JCL for the job to be debugged via cdf/XDC.

Help COmmands SEt USS

This command may be used to set the **USS** profile setting.

Syntax:

```
SET USS  NO
         YES
         IFUSS
```

NO
The current **USS** setting in the profile is set to NO.

YES

The current **USS** setting in the profile is set to YES.

IFUSS
The current **USS** setting in the profile is set to IFUSS.

The USS setting can be saved in your session profile. It can be displayed by the LIST USS command. It also can be displayed and changed by Z/XDC's "Profile Menuing System". For more information, please see:

    HELP COMMANDS LIST USS
    HELP PROFILES MENU

The initial factory default setting is **IFUSS.** If you wish to change this default, then you can use the following commands to do so:

    PROFILE READ (to ensure that no unwanted changes creep in)
    SET USS <whatever>
    PROFILE SAVE

Help COmmands SEt VARIables

This command is an alias of the **SET VSETTINGS** command. See HELP COMMANDS SET VSETTINGS for more information.

Help COmmands SEt VARS

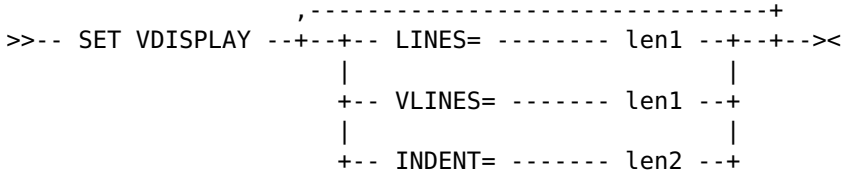
This command is an alias of the **SET VSETTINGS** command. See HELP COMMANDS SET VSETTINGS for more information.

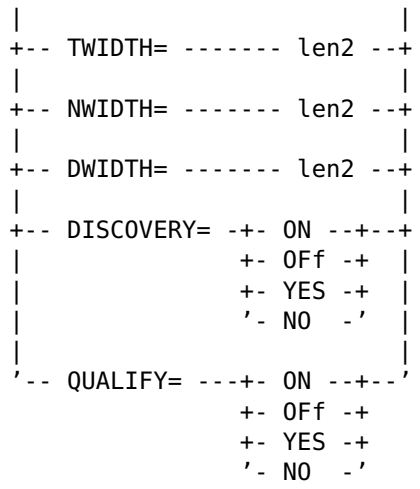
Help COmmands SEt VDisplay

SET VDISPLAY controls how LIST VARIABLES formats and displays information on the terminal.

Each line of the LIST VARIABLES display has type, name, and data columns, separated by three blank columns. The WIDTH of these are specified by TWIDTH, NWIDTH, and DWIDTH controls respectively. Further control of the Name column is achieved with the Indent control.

Syntax:





NOTE: len1 is value 1 to 32767
len2 is value 1 to 255

LINES=
Specifies the total number of output lines that the LIST VARIABLES command will produce.

VLINES=
Specifies the maximum number of members within a structure that are displayed by the LIST VARIABLES command.

INDENT=
Specifies the indentation for each level of a structure's display. Default=2. Indent shifts the Type and Name fields right the specified number of columns for each level.

TWIDTH=
Specifies the width of the "type" column in the display before wrapping occurs.

NWIDTH=
Specifies the width of the "name" column in the splay before wrapping occurs.

DWIDTH=
Specifies the width of the data column in the display before wrapping occurs.

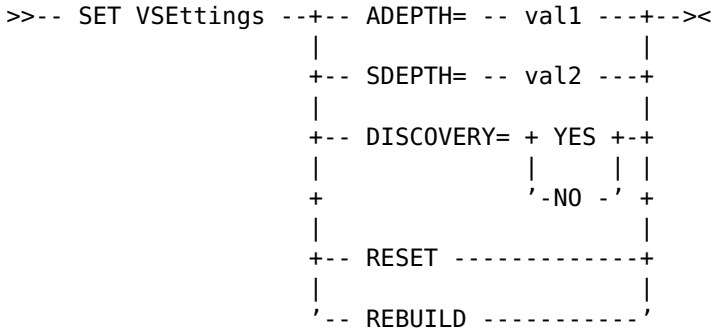
DISCOVERY=
This ON/OFF operand controls the automatic expansion of structures by the LIST VARIABLES command.

QUALIFY=
This ON/OFF operand controls the display of variable names. When set ON then fully qualified variable names are displayed. When set OFF then only the basic variable name is displayed.

Help COmmands SEt VSettings

This command is available only for customers who have Licensed Features for

debugging one or more of the supported **High Level Languages** (such as XL C/C++ and Metal C). For more information, see HELP SUPPORT FEATURESANDCAPS.



NOTE: val1 is a value from 1 to 64 or MAX
val2 is a value from 1 to 256 or MAX

ADEPTH=
Sets the maximum number of array dimensions that c/XDC will display. ADEPTH=MAX sets the limit to 64 dimensions.

SDEPTH=
Sets a limit on the depth to which stack frames can be listed by the LIST VSTACK command. SDEPTH=MAX sets the limit to 256 structures.

DISCOVERY=YES | NO
When **DISCOVERY=YES** c/XDC will examine all retry level environments looking for changes affecting the user program's defined C language variables and the storage locations assigned to them.

When **DISCOVERY=NO** c/XDC will not attempt to discover variables on its own. Instead, users must explicitly request an examination of the variable pools via use of the **SET VSETTINGS RESET** command. This setting can vastly improve c/XDC performance when debugging large, complex applications.

RESET
This causes c/XDC to reexamine its knowledge of variable pools, seeing if anything has changed. It should be used if c/XDC, for some reason, loses track of what the variable pools actually are. When new variables are found then message "DBC303I New C variables discovered" is displayed.

Frequently z/XDC issues the SET VSETTINGS RESET command internally and it is not typically used by the user.

REBUILD
The rebuild operation discards and rebuilds the variable pools.

Help COmmands SEt WIndow

The **SET WINDOW** command is used to create and destroy Watch Windows. It is also used to set various characteristics of a window. (See HELP FULLSCREEN WINDOWS for general information about windows.)

Syntax:

```
SET WINDOW omitted omitted omitted
           CREATE  row#    CMDS='string'
           CREATE=row#

SET WINDOW DELETE omitted
                  window#
           DELETE=window#
           DELETE=ALL

SET WINDOW RETRIEVE #ofcmds
           RETRIEVE=#ofcmds

SET WINDOW HORIZONTAL omitted
           HORIZONTAL #ofcolumns
           HORIZONTAL=#ofcolumns

SET WINDOW VERTICAL omitted
           VERTICAL #ofrows
           VERTICAL=#ofrows
```

Aliases:
- **SCREEN** is an alias of WINDOW.
- **RETRY** is an alias of RETRIEVE.
- **RECALL** is an alias of RETRIEVE.

The operands are mutually exclusive. If all operands are omitted, then **SET WINDOW CREATE** is assumed.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

CREATE - Creating Watch Windows
DELETE - Deleting Watch Windows
LAYOUTS - Creating, saving and restoring Window Layouts
RETRIEVE - Setting the number of retrievable commands
VERTICAL - Setting the window's default vertical scroll amount
HORIZONTAL - Setting the window's default horizontal scroll amount

The window characteristics can be saved in your profile dataset for automatic restoration every time you start a debugging session. For more information, please see:

HELP PROFILES
HELP COMMANDS PROFILE

The distributed version of Z/XDC has the following defaults:

```
SET WINDOW RETRIEVE 255
SET WINDOW VERTICAL CURSOR
```

```
SET WINDOW HORIZONTAL CURSOR
SET WINDOW HORIZONTAL NOAUTORESET
```

Help COmmands SEt WIndow Create

The **SET WINDOW CREATE** command (or just **SET WINDOW)** causes a Watch Window to be created. The location of the new window can either be explicitly stated or be implied by the location of the cursor.

The characteristics of newly created Watch Windows (retrieval list size and scroll amount) are copied from the screen's Primary Window (i.e. its top window).

Syntax:

```
SET WINDOW omitted omitted omitted
           CREATE  row#    CMDS='string'
           CREATE=row#
```

Aliases:

SCREEN is an alias of WINDOW.

Operands:

CREATE
omitted
These forms of this operand are identical. They indicate that a new Watch Window (i.e. a new command line) is to be created, starting at the terminal display line that contains **the cursor.** In other words, to use this form of the command, you have to:

- Type the command on an existing command line (any command line will do),
- Then move the cursor to the line at which you want the new window to start,
- And finally press the **ENTER** key.

Note: To simplify this process, the factory default setting for **shift-F1** is **SET WINDOW CREATE.** So to create a new window using that PF key,

- Just move the cursor to the desired display line,
- Then press shift-F1.

A new command line will appear on the row at which the cursor is located.

CREATE=row#
CREATE row#
row#
These forms of the command can be used to explicitly select the display line at which the new command line is to be created. **row#** must be a decimal number ranging

from 3 to the height of your display terminal.

CMDS='string'
This allows you to explicitly define the string of commands that is to be set onto the new command line. The string should be formed as follows:

- It may contain one or more Z/XDC commands separated from each other by one (or two) semicolons.
- When two consecutive semicolons are used, the displays generated by the separated commands will have a blank line between them (called "whitespace").
- The commands used should not be "action" commands. They should only be commands that produce displays. Examples include (without limitation) DISPLAY, FORMAT and the many many LIST commands.
- Action commands (if present):
  - Will be accepted and executed,
  - But they will be removed from the command line so that they will not be executed a 2nd time.

For more information, see **HELP FULLSCREEN WINDOWS WATCHWINDOWS.**

If you do not provide this **CMDS=** operand, a default commands string is selected as follows:

- If this new Watch Window is being suballocated out from another Watch Window (i.e. if there is another Watch Window command line above this on the screen), then that **higher** Watch Window's command string is copied into the new Window's command line.

- On the other hand, if this new Watch Window is being suballocated out from the Primary Window (i.e. if this new Watch Window will be higher on the screen than all other Watch Windows), then the next **lower** Watch Window's command string is copied into the new Window's command line.

- If there is no other Watch Window already on the screen, then the new Watch Window's command line is set to the comment string: *** **NEW WATCH WINDOW** *** .

The number and locations of Watch Windows can be saved in your profile dataset for automatic restoration every time you start a debugging session. Also, the actual commands that are automatically issued by the Watch Windows can be saved as well. For more information, please see the following:

HELP PROFILES
HELP COMMANDS PROFILE

Examples:

Suppose that you want to create a new window starting at your terminal's 15th line. To do so, you can simply move the cursor down to the 15th line and press shift-F1. A new command line will appear at the line containing the cursor.

set window 5 cmds='l estaes;map xdcmaps.scb;u scb scb#1 f;show .scbexit?'

This command does the following:
- It creates a new Watch Window starting at the display screen's 5th line.
- It populates that command line with the given commands string **including** the two action commands:
  - MAP XDCMAPS.SCB
  - USING SCB SCB#1 FLOAT

The first time you press ENTER, all the commands (including the action commands) will be processed, **but** then the action commands will be edited out from the commands string so they will not be processed more than once.

set win create=15 cmds='f psw?;l eregs;l bea'
set win 15 cmds='f psw?;l eregs;l bea'
Either of these will cause a new command line to appear at display line 15. Its initial command string will be **F PSW?;L EREGS;L BEA**

Help COmmands SEt WIndow Delete

The **SET WINDOW DELETE** command causes a Watch Window to be deleted.

Syntax:

```
SET WINDOW DELETE omitted
           DELETE window#
           DELETE=window#
           DELETE ALL
           DELETE=ALL
```

Aliases:

SCREEN is an alias of WINDOW.

Operands:

DELETE omitted
SET WINDOW DELETE (i.e. no further operands) may be typed on the command line of any existing Watch Window. Then you may move the cursor into any Watch Window that you wish to delete. (Or you may leave the cursor where it is to delete the current Watch Window.) Then when you press ENTER, the window that contains the cursor is deleted, and its space is added to the next above window.

Z/XDC's factory default PF keys has F1 set to **SET WINDOW DELETE.** So the easy way to delete a Watch Window is simply to move the cursor into it, and then press F1.

DELETE window#
DELETE=window#
This form of the command can be used to explicitly identify the window to be deleted. **window#** is the number of the window to be deleted. It may range in value from 2 (meaning the 2nd window on the screen) to the number of windows currently present.

Note: Any attempt to delete the screen's top window (window number 1) is rejected.

DELETE ALL
DELETE=ALL
This deletes all Watch Windows (if any) leaving just the Primary Window.

If you want to write a script to define a particular display layout, this **SET WINDOW DELETE ALL** command can be used to reset the display to a clean slate before using various **SET WINDOW CREATE** commands to build a new layout.

Example:

Suppose you have five windows defined on your screen and you want to delete the third one. Then to do so, move the cursor to any location within the 3rd window and press F1. That window will disappear, and its display space will be added to the 2nd window.

The number and locations of Watch Windows can be saved in your profile dataset for automatic restoration every time you start a debugging session. Also, the actual commands that are automatically issued by the Watch Windows can be saved as well. For more information, please see the following:

HELP PROFILES
HELP COMMANDS PROFILE

Help COmmands SEt WIndow Layouts

Saving/Restoring Window Layouts

The arrangement and content of command lines at your workstation is called your Windows Layout. You create Window Layouts through use of SET WINDOW commands:

- **SET WINDOW CREATE** allows to create Watch Window command lines (scattered throughout your display screen) and to populate them with command strings to be automatically processed upon every press of an action key (ENTER, PF keys, etc.).

- **SET WINDOW DELETE** allows you to remove Watch Window command lines you no longer want.

Window Layouts can be saved in your personal Session Profile. Just use a **PROFILE SAVE name** command.

To restore a profile that you've saved, Just use a **PROFILE READ name** command.

As you may have gathered from the above, profiles can be named. This means that you can save and later restore any number of unique, unrelated profiles. For a full discussion of this, see **HELP PROFILES NAMEDPROFILES.**

Help COmmands SEt WIndow Retrieve

Each window on your terminal's screen maintains a retrievable stack of the commands recently issued from that window's command line. The commands in this stack can be brought back to the window's command line via the use of the **RETRIEVE** command.

The **SET WINDOW RETRIEVE** command can be used to set, for any particular window, the maximum number of prior commands that can be saved in that window's commands retrieval stack.

Syntax:

```
SET WINDOW RETRIEVE #ofcmds
                   =#ofcmds
```

Aliases:
- **SCREEN** is an alias of WINDOW.
- **RECALL** is an alias for RETRIEVE.
- **RETRY** is an alias for RETRIEVE.

#ofcmds
This specifies the number of prior commands the retrieval stack can hold. **#ofcmds** may be any value from 1 to 255. The factory default is 255. For more information about command retrieval, please see:

HELP FULLSCREEN RETRIEVE
HELP COMMANDS RETRIEVE

The sizes of the retrieval stacks for all windows can be saved in your profile dataset for automatic restoration every time you start a debugging session. For more information, please see:

HELP PROFILES
HELP COMMANDS PROFILE

Help COmmands SEt WIndow Vertical

Each window that is displayed at your terminal can have its own default vertical

scroll amount (the number of lines by which the display is shifted when the **UP** and **DOWN** commands are used without operands). This value is controlled by the **SET WINDOW VERTICAL** command.

Syntax:

SET WINDOW VERTICAL omitted

CURSOR
=CURSOR

FULL
=FULL

HALF
=HALF

DATA
=DATA

#ofrows
=#ofrows

Aliases:
- **SCREEN** is an alias of WINDOW.
- **SCROLL** is an alias of VERTICAL.
- **PAGE** is an alias of FULL.

The factory default vertical scroll amount is CURSOR.

The meanings of these operands are the same as the meanings of similar parameters supported by ISPF's UP and DOWN commands.

CURSOR
When an UP or DOWN command is executed, the window containing the cursor is shifted so that the line containing the cursor is positioned at the top (DOWN) or the bottom (UP) of the window.

FULL
PAGE
When an UP or DOWN command is executed, the window containing the cursor is shifted so that the window displays data either following (DOWN) or preceding (UP) the previously shown data.

HALF
When an UP or DOWN command is executed, the data shown by the window containing the cursor is shifted up or down by a number of lines equal to half of the window's size.

DATA
When an UP or DOWN command is executed, the data shown by the window containing the cursor is shifted up or down so that the bottom line is moved to the top (DOWN) or the top line is moved to the bottom (UP) of the window.

#ofrows

This must be a decimal number specifying the number of lines the display area is to be shifted in response to an UP or DOWN command.

The default scroll amount is used only when the "UP" or "DOWN" command is given without operands. When "UP" or "DOWN" is given with operands, then the window's default scroll amount is ignored.

Each of the operands for defining the default scroll amount is also valid as an operand of the "UP" and "DOWN" commands. In addition, the "UP" and "DOWN" commands accept "CMD", "MAX", and "#n" as operands. These, however, cannot be made default scroll amounts. For more information, please see:

HELP COMMANDS UP
HELP COMMANDS DOWN

Help COmmands SEt WIndow Horizontal

Each window that is displayed at your terminal can have its own default horizontal scroll amount (the number of columns by which the display is shifted when the **RIGHT** and **LEFT** commands are used without operands). This value is controlled by the **SET WINDOW HORIZONTAL** command.

Syntax:

SET WINDOW HORIZONTAL omitted

CURSOR
=CURSOR

FULL
=FULL

HALF
=HALF

DATA
=DATA

#ofcolumns
=#ofcolumns

AUTORESET
=AUTORESET

NOAUTORESET
=NOAUTORESET

Aliases:
- **SCREEN** is an alias of WINDOW.
- **PAGE** is an alias of FULL.
- **RESET** is an alias of AUTORESET.
- **RETAIN** is an alias of AUTORESET.
- **NORESET** is an alias of NOAUTORESET.

- **NORETAIN** is an alias of NOAUTORESET.

The factory default horizontal scroll amount is CURSOR and NOAUTORESET.

The meanings of most of these operands are the same as the meanings of similar parameters supported by ISPF's RIGHT and LEFT commands.

CURSOR
When a RIGHT or LEFT command is executed, the window containing the cursor is shifted so that the column containing the cursor is positioned at the rightmost (LEFT) or the leftmost (RIGHT) edge of the window.

FULL
PAGE
When a RIGHT or LEFT command is executed, the window containing the cursor is shifted so that the window displays data either following (RIGHT) or preceding (LEFT) the previously shown data.

HALF
When a RIGHT or LEFT command is executed, the data shown by the window containing the cursor is shifted left or right by a number of columns equal to half of the window's width.

DATA
When a RIGHT or LEFT command is executed, the data shown by the window containing the cursor is shifted left or right so that the leftmost column is moved to the right edge (LEFT) or the rightmost column is moved to the left edge (RIGHT) of the window.

#ofcolumns
This must be a decimal number specifying the number of columns the display area is to be shifted in response to a RIGHT or LEFT command.

AUTORESET
RESET
RETAIN
When the window is shifted rightwards or leftwards pursuant to a LEFT or RIGHT command, the shift will persist until a subsequent LEFT or RIGHT command is issued.

NOAUTORESET
NORESET
NORETAIN
When the window is shifted rightwards or leftwards pursuant to a LEFT or RIGHT command, the shift will be canceled the next time a non-scrolling command is issued.

The default scroll amount is used only when the "RIGHT" or "LEFT" command is given without operands. When "RIGHT" or "LEFT" is given with operands, then the window's default scroll amount is ignored.

Each of the operands for defining the default scroll amount is also valid as an operand of the "RIGHT" and "LEFT" commands. In addition, the "RIGHT" and "LEFT" commands accept "MAX" as an operand. "MAX", however, cannot be made a default scroll amount. For more information, please see:

HELP COMMANDS RIGHT
HELP COMMANDS LEFT

Help COmmands SEt WTOR

This command has been replaced. Use **SET DBC640Q** instead.

Help COmmands SEt Zap

This command is effective only when Z/XDC is running authorized.

The **SET ZAP** command enables you to control whether or not Z/XDC will allow storing (zapping) into store protected storage.

Store protection is a hardware facility that controls access to virtual storage by using the store protection bit in each page-table entry. It provides protection against improper storing, even by key 0 programs. Generally, in order for a program to store into an area of storage, the execution key must match the storage key of the page to be stored into (or the execution key must be 0); otherwise, a code-4 program check (0C4 ABEND) occurs. But when storage is "Store Protected", a program cannot store into that page regardless of the program's execution key. (For more information, see HELP DEBUGGING REENTRANT STORAGEKEYS.)

Normally, when a program (such as Z/XDC) attempts to store into store protected storage, the attempt fails, the contents remain unchanged, and a program interrupt takes place. However, Z/XDC's SET ZAP SPROT command enables the logic necessary to allow Z/XDC's ZAP commands to store into store protected storage anyway (if you want to take the risk).

Syntax:

```
SET ZAP NORMAL
        SPROT
```

SET ZAP NORMAL
Z/XDC will not allow the ZAP command to store into store protected storage.

SET ZAP SPROT
Z/XDC will allow the ZAP command to store into store protected storage.

The factory default ZAP setting is SET ZAP NORMAL.

To display the current zap related settings use the LIST ZAP command. For more information on the LIST ZAP command, see HELP COMMANDS LIST ZAP.

Note that the SET ZAP NORMAL setting is not a security feature, it is an integrity feature. It exists simply to help users keep themselves from making common mistakes. For more information on security issues regarding the ZAP command, see HELP SECURITY ZAP.

You can save the ZAP setting in your Profile dataset for restoration every time you start a debugging session. For more information, please see:

- HELP PROFILES
- HELP COMMANDS PROFILE

Help COmmands SHow

The **SHOW** command provides a compact way to show an arbitrary list of data fields, showing one data field per terminal screen line. The SHOW command can also display Z/XDC messages by message id number. The SHOW command accepts multiple address expressions and/or message id numbers for operands.

For each **address expression** given, SHOW generates a formatted, object code display of up to 16 (NARROW) or 32 (WIDE) bytes of storage. Each such display takes exactly one display line at the user's terminal.

For each **message id** number given (DBCnnn or DBCnnnn), the SHOW command displays the associated message, but with null, zero, blank, or default values substituted for all variable fields.

All messages are shown in their entireties, regardless of how many display lines it takes to display them.

When displaying storage, the number of bytes displayed per line depends upon the nature of field labels that may have been defined at or near the locations being displayed. But the maximum number of bytes that can be displayed for each address is **16** for narrow displays and **32** for wide displays.

The **SHOW** command is the best way to efficiently display a series of scattered data fields. For example, a SHOW command can be used on the command line of a Watch Window, and its operands would be the names of several data fields. Then as you allowed execution to progress through your program, you would be able to very easily keep track of the values contained within those fields as they are changed by your program.

The **SHOW** command accepts several keyword operands that allow you to override various format control settings defined by the **SET FORMAT** command.

Syntax:

```
SHOW addressexpressions ... ADDRESSES  DECIMAL      ...
     DBCnnn                 OFFSETS    HEXADECIMAL
     DBCnnn_hh
     DBCnnnn
     DBCnnnn_hh
     ...
                        ... ASCII  WIDE   INSTRUCTION  POINTERS
                            EBCDIC NARROW DATA         NOPOINTERS
                                          NOBIAS
```

Rules:

- Except for the addressexpressions and the various DBC message ID operands, operands appearing above within columns are mutually exclusive.
- Multiple addressexpression and DBC message ID operands can be given, and they can be freely intermixed.
- All addressexpressions and DBC message ID operands must be given first. If any other operands are given, they may be given in any order, but they all must come after all addressexpressions and DBC message ID operands.
- At least one addressexpression or DBC message ID operand is required. All other operands are optional.

Operands

addressexpressions
These may be one or more address expressions that resolve to any locations in any accessible address spaces.

DBCnnn
DBCnnnn
DBCnnn_hh
DBCnnnn_hh
These identify one or more Z/XDC messages to be displayed:

- **nnn** or **nnnn** is any decimal number ranging from 001 to 9999. It identifies the particular message to be displayed. But be aware:
  - Not all message numbers are used. If you choose an unused number, then **DBC999E** is displayed instead.
  - Some message numbers, 2000 or greater, are used for unnumbered messages. So if you see something like that, that's why.
- **_hh** is optional. When present, it is any hexadecimal number ranging from **00** to **FF.** If the message being displayed might provide additional information, then **_hh** indicates the additional information to be displayed. (DBC005 is an example of a message that supports the display of additional information.)

For more information, see **HELP COMMANDS SHOW DBCNNN.**

...
Any number of address expressions and/or message id numbers may be given, separated from each other by blanks or commas.

Formatting Controls

The following operands may be given in any order. They must, however, follow the

given address expressions (if any). Also, all of the following operands serve to override the corresponding default values established via the **SET FORMAT** command.

ADDRESSES
OFFSETS
These operands control whether the far left column of the display (the "address" column) will show storage addresses or offsets:

ADDRESSES
Causes each line of data displayed to be prefixed by that data's virtual address.

OFFSETS
Causes each line of data displayed to be prefixed by that data's offset from the start of an appropriate including object (load module, csect, dsect, or equate).

DECIMAL
HEXADECIMAL
For disassembled machine instruction displays, these operands control whether the displacement fields are formatted as decimal or hexadecimal numbers:

DECIMAL
Causes the displacement fields to be displayed as decimal numbers.

HEXADECIMAL
Causes the displacement fields to be displayed as hexadecimal numbers.

ASCII
EBCDIC
For data displays, these operands control whether the text portions of data displays are to be interpreted using the **ASCII** or **EBCDIC** character set:

ASCII
Causes data displays to show the **ASCII** interpretation of the displayed storage. To visually indicate this, the text portion of each display line will be framed with **vertical bars (|)** instead of asterisks.

EBCDIC
Causes data displays to show the **EBCDIC** interpretation of the displayed storage. To visually indicate this, the text portion of each display line will be framed with **asterisks (*).**

WIDE
NARROW
These operands control whether data displays are to be wide or narrow:

WIDE
This causes data displays to show up to **32 bytes** of storage per line (not just 16). This option works best when your terminal's display is at least 136 columns

wide. For more information, see HELP FULLSCREEN TERMINALS.

NARROW
This causes data displays to show only **16 bytes** of storage per line. This width is appropriate for terminals with 80 character wide display lines.

INSTRUCTIONS
DATA
NOBIAS
The **SHOW** command generally attempts to format storage contents as instructions, but this bias can be influenced by many factors, such as whether or not a PSW points to the storage being formatted, as well as the attributes of maps, fields, and equates that label the storage being displayed. As a part of the management of this process, the **SHOW** command maintains a concept called the **formatting bias.**

The **INSTRUCTION, DATA,** and **NOBIAS** operands set the formatting bias for the **SHOW** command as follows:

INSTRUCTION
The **SHOW** command's formatting bias will be to interpret storage contents as being machine instructions. This will be done even if other factors might suggest that the storage contents should be interpreted as data. This will **not** be done, however, if the opcode being displayed is invalid.

DATA
The command's formatting bias will be to interpret storage contents as being data. This will be done even if other factors might suggest that the storage contents should be interpreted as machine instructions.

NOBIAS
The command's formatting bias will not be forced. Absent other factors, the storage contents will be interpreted as machine instructions, but if other factors suggest otherwise, then those factors won't be ignored.

The formatting bias set by the **INSTRUCTION, DATA,** and **NOBIAS** operands will be independently effective for every display line generated (not just the first). This is because the SHOW command's output is really just a collection of 1-line displays.

POINTERS
NOPOINTERS
When storage is being formatted via disassembly (i.e. when either **OBJECT** or **BOTH** is in effect), if a line of display is showing a snippet of data that is 3, 4 or 8 bytes long, then this **POINTERS** operand will cause Z/XDC to attempt to resolve that snippet as a storage pointer. Then it will display that resolution as a comment on the display line. Notes:

- The **NOPOINTERS** operand prevents these resolutions.
- The factory default is **POINTERS.**
- When very complex structures of **floating maps and equates** are present, the **POINTERS** setting may result in slower displays. In that case, you may wish to use **NOPOINTERS** instead. (For more information about floating maps and equates, see **HELP MAPS FLOATING.)**

More Information

Several subtopics are available providing examples of and tips about using the SHOW command. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

SIZES - Controlling the amount of data displayed.
CDP - SHOW's use of the Current Display Pointer.
DBCNNN - Displaying DBCnnn messages.

Other commands that can be used to display storage are:

DISPLAY - Displays an area of storage in a hex-text (dump like) format.
FORMAT - Formats an area of storage as either source code displays, machine instructions, or data fields, as appropriate.
WHERE - Formats an area of storage containing the current **retry level** PSW's resume execution address.
EWHERE - This is like the WHERE command except that its displays are centered on the **error level** PSW **(EPSW).**
FIND - Scans storage for a given string of data.

Help COmmands SHow Sizes

The **SHOW** command displays one line of data for each address expression given to the command. (See HELP COMMANDS SHOW for details.) The amount of data that can be displayed per line ranges from 1 to 16 bytes for narrow displays, and from 1 to 32 bytes for wide displays. This topic discusses the things that affect the size of a line of display.

For each address expression given, the data displayed begins at the location to which the address expression resolves. Generally (but not always), the displayed data ends:
- At the next location in storage to which a label has been assigned,
- Or at the end of a field,
- Or at the display line's maximum width (16 or 32 bytes),

whichever comes first.

Example #1:

```
show psatold .tcbtio .cvtabend .tiot +8 +10 data
 +21C PSATOLD  0079CC30          TCB.TCBRBP          *....*
   +C TCBTIO   007BD000          TIOT.TIOT           *.#}.*
 +108 CVTABEND 00FCDD10          IEANUC01.IEABEND+0  *....*
   +0 TIOT     C4C2C3D6 D3C54040                    *DBCOLE  *
   +8 TIOCSTPN C9E2D7C6 D7D9D6C3                    *ISPFPROC*
  +10 TIOCJSTN D3D6C7C4 C2C3D9C1                    *LOGDBCRA*
```

Example #2:

```
Consider the following piece of code:
    003BC100  ADDRAD64 DS  0AD
    003BC100  ADDR_HI  DS  A
    003BC104  ADDR_LO  DS  A
    003BC108           DS  XL10
    003BC112           ---
```

The following commands will have the following results:

- **SHOW .ADDR_LO** displays 4 bytes starting at location 3BC104.
- **SHOW .ADDR_HI+1** displays 3 bytes starting at location 3BC101.
- **SHOW 3BC100** displays 4 bytes starting at location 3BC100.
- **SHOW .ADDRAD64+0** displays 4 bytes starting at location 3BC100.
- **SHOW .ADDRAD64** displays **8** bytes starting at location 3BC100.

The last command is an exception. It displayed 8 bytes instead of 4. 8 bytes is what you would want, of course, but that's not explained by the rules described above. So there is an additional rule: If the address expression **ends** with a label, then the length represented by that label (8 bytes in this example) overrides the next-nearest-label rule.

Help COmmands SHow Cdp

Unlike other storage displaying commands, the SHOW command does not permanently change either the Current Display Pointer (CDP) or the Next Display Pointer (NDP). However, it does **temporarily change** these pointers, but for only the duration of the SHOW command's own processing. It does this according to the following rules:

- The SHOW command accepts any number of address expression operands, and it generates one display line for each expression given.

- If an address expression uses an explicit base term, the CDP will be temporarily changed by that expression to point to the expression's resolved address.

- If an address expression uses an implied base term, the CDP will **not** be changed by that expression.

- When the resolution of all of the command's target addresses is complete, the CDP will be restored to the value it had prior to the processing of the current command.

These rules have the following consequences:

- Commands that execute following a SHOW command will not see any change in the CDP caused by SHOW.

- A sequence of consecutive address expressions (within the SHOW command) that all have implied base terms will all be relative to the same CDP value.

- An address expression that has an explicit base term can be used to temporarily set the CDP for use by the following address expressions (if any) within the same command.

For more information about the Current Display Pointer. see HELP ADDRESSING IMPLICIT.

Some Examples:

SHOW R5? +10 +20 PSW? +4 +8
Six lines of display are produced. They are:
1st line - Storage pointed to by R5.
2nd line - Storage located at R5?+10
3rd line - Storage located at R5?+20
4th line - Storage pointed to by the retry level PSW.
5th line - Storage located at PSW?+4
6th line - Storage located at PSW?+8
An equivalent command would be: **SHOW R5? R5?+10 R5?+20 PSW? PSW?+4 PSW?+8**

T PSW?+4 +4 +8 +C;GOT
Assuming the PSW is pointing to a BAL or BAS instruction, and assuming the next four instructions following the BAL/BAS are each 4 bytes wide, then this T command makes use of the CDP to set a family of transient breakpoints onto the first four machine instructions following the BAL/BAS, and the GOT command causes user program execution to resume:
- The first address expression ("PSW?+4") sets a breakpoint at PSW?+4 and temporarily sets the CDP to point to PSW?+4.
- The second address expression ("+4") uses the temporary CDP to set the second breakpoint at PSW?+8. It does not, however, change the CDP.
- The third and fourth address expressions ("+8" and "+C") use the same temporary CDP to set breakpoints at PSW?+C and PSW?+10, respectively.

An equivalent command would be: **T PSW?+4 PSW?+8 PSW?+C PSW?+10;GOT**

If the BAL/BAS instruction jumps to a subroutine, and if that subroutine eventually returns to one of the four instructions following the BAL/BAS, then execution will be recaptured by Z/XDC when the subroutine completes.

Note, Z/XDC’s factory default definition for F11 uses a more complex TRAP command to accomplish a similar result while relying on fewer assumptions than are relied upon by this example. See HELP BREAKPOINTS TRACING SUBROUTINES for more information.

SH .PRNTFLGS .PRNTBUFR +20 +40 .PSAVER14? ADDRESS WIDE
This command generates five lines of display:
- The first line shows the contents of a field named PRNTFLGS.
- The second line shows the contents of the first 32 bytes of PRNTBUFR.
- The third line shows the second 32 bytes of PRNTBUFR (at PRNTBUFR+20).
- The fourth line shows the third 32 bytes of PRNTBUFR (at PRNTBUFR+40).
- And the last line shows the return address for the most recent caller of a printing subroutine.

(All assuming, of course, typical usages of the fields named in this example).

The **ADDRESS** operand near the end causes the locations of each of these displays to be shown as virtual addresses instead of as offsets.

The **WIDE** operand causes each displayed line to show up to 32 bytes of data. (This is most suitable if your terminal is set up to display at least 136 characters per line.)

If a **SHOW** command is issued from the command line of a Watch Window, then that command is automatically re-executed every time you press a PF key or the ENTER key. Thus, any changes that may occur to the contents of these fields are automatically displayed almost as soon as they occur.

Help COmmands SHow Dbcnnn

The **SHOW** command can be used to display most Z/XDC numbered messages as well as a random selection of unnumbered messages. If you just want to see what message is produced by a particular Z/XDC message number (without going to the books or into Built-in Help), then this one way to do that. (Another way is to issue **HELP MESSAGES DBCwhatever.)**

Below are various tips and examples about using the SHOW command to display Z/XDC messages. If you want syntactical details, then see **HELP COMMANDS SHOW.**

Z/XDC message IDs start with the characters **DBC** followed by 3 or 4 decimal digits.

The message ID may be optionally followed by a subselection code expressed as **_hh** appended to the ID. Some messages have subselections, some don't. The message that uses subselection codes the heaviest is **DBC005 INVALID SYNTAX -** . Example: **SHOW DBC005_1A**

The **SHOW** command displays DBC messages with null, zero, blank, or default values substituted for all variable fields.

If a DBC message has multiple display lines, then all lines are shown.

The **SHOW** command only shows basic copies of DBC messages. It does **not** give any description of those messages. Such descriptions are, however, available in Z/XDC's Built-in Help. To see these descriptions, simply type an **H** shortcut at the left of any displayed DBC message.

The **SHOW** command can display only those DBC messages that are generated by Z/XDC's internal messages generator. Messages that Z/XDC generates via older technologies cannot be displayed. (So when SHOW fails to display a message that you know exists, that's why.)

Example:

show dbc005 dbc005_03 dbc041 dbc1600
This example displays the following:
. DBC005 DBC005E INVALID SYNTAX

. DBC005 DBC005E INVALID SYNTAX - COMMA, BLANK, OR CLOSE PARENTHESIS EXPECTED
. DBC041 DBC041E THIS COMMAND IS SUPPORTED ONLY WHEN Z/XDC IS RUNNING AS AN FRR
OR SOMETIMES AS A TRAP HANDLER
. DBC1600 DBC1600W WHEN THE RITARGETS SCAN RANGE IS NOT WITHIN ANY OBJECT, RIxxx=
OPERANDS ARE REQUIRED

Help COmmands SPLIT

The **SPLIT** command functions in the same way it does in ISPF: It creates (or moves) a horizontal ISPF **split line** on the terminal's screen and transfers control from Z/XDC to another program running under ISPF. It also causes the cursor to move to a window belonging to that other work.

The **split line** is created at or moved to the line that contains the cursor at the time the ENTER key or PF key is pressed that executes the **SPLIT** command.

Syntax:

SPLIT
SPLIT NEW
SPLIT operands

Z/XDC does not check any operands that might be given. If any are given, then they are simply passed through for processing by ISPF. See IBM's **ISPF User's Guide, Vol. I** (SC34-4822) more information.

Help COmmands SPLITV

The **SPLITV** command functions in the same way it does in ISPF: It creates (or moves) a vertical ISPF **split column** on the terminal's screen and transfers control from Z/XDC to another program running under ISPF. It also causes the cursor to move to a window belonging to that other work.

The **split column** is created at or moved to the column that contains the cursor at the time the ENTER key or PF key is pressed that executes the **SPLITV** command.

Syntax:

SPLITV
SPLITV operands

Z/XDC does not check any operands that might be given. If any are given, then they are simply passed through for processing by ISPF. See IBM's **ISPF User's Guide, Vol.**

I (SC34-4822) more information.

Help COmmands STep

This command is available only for customers who have Licensed Features for debugging one or more of the supported **High Level Languages** (such as XL C/C++ and Metal C). For more information, see HELP SUPPORT FEATURESANDCAPS.

The **STEP** command is used to step execution through a High Level Language program (such as XL C/C++ and Metal C) one source statement at a time. It also can be used to step into, out of, or over a subroutine or function.

This command can be used only with High Level Language programs **(HLL** programs) that have been mapped. See HELP COMMANDS MAP for more information.

This **STEP** command does for HLL programs what the **TRACE** command does for Assembler code. It allows execution to proceed from one language statement to the next.

The **TRACE** command can still be used, however, within an HLL program to step execution one machine instruction at a time. Generally, **STEP** commands and **TRACE** commands can be freely intermixed.

Also, the **AT** and **TRAP** commands can be used to set breakpoints at (or between) HLL source statement boundaries. In fact, if you wish to allow execution to run to a particular point in an HLL program, then the **TRAP** or **AT** command (followed by a **GO** command) is the way to do that.

Syntax:

```
STEP omitted omitted
     INTO    ZERO
     OUT     TRAP2
     OVER
```

STEP INTO
The INTO operand causes the following:

- If execution is currently at a language statement that does not call a subroutine, then that statement is executed, and execution stops at the next following statement.

- If execution is currently at a language statement that calls a subroutine or invokes a function **and if** the called subroutine or function is written in a High Level Language **and if** a source image map has been loaded for that program, then **c/XDC** will follow execution into the subroutine or function, and execution will stop at that subroutine/function’s first executable statement.

- On the other hand, if the called function or subroutine either is not a High Level Language program or has not been mapped, then that subroutine/function will be allowed to execute, and execution will be recaptured by **Z/XDC** when the subroutine or function returns to the next following statement after the calling

statement.

STEP OVER
For the purposes of STEP'ing, the **OVER** operand causes calling statements to be treated like non-calling statements: If the current statement calls a subroutine or function, then that subroutine or function will be allowed to execute in its entirety, and execution will be recaptured by **Z/XDC** when the subroutine/function returns to the next following statement after the calling statement.

If the current statement is a non-calling statement, then (like STEP INTO) the current statement is executed, and execution stops at the next following statement.

STEP OUT
If execution is currently within an HLL subroutine (or function), then the OUT operand causes **Z/XDC** to allow the subroutine to run to completion. Execution is recaptured at the source statement following the statement that called the subroutine (or invoked the function).

STEP ZERO
The ZERO operand overrides any settings for a specific breakpoint type to force the use of an X'00' opcode which will cause a S0C1 exception.

STEP TRAP2
The TRAP2 operand overrides any settings for a specific breakpoint type to force the use of a TRAP2 instruction (X'01FF').

STEP
When the **STEP** command is given without operands (or only the ZERO or TRAP2 operands) it will default either to STEP **INTO** or to STEP **OVER,** according to the current **SET STEP** setting. The factory default for STEP is **STEP OVER.**

If you don't like a bare STEP being STEP OVER, it's easy enough for you to change...

For more information, see HELP DEBUGGING C.

Help COmmands SWAp

When ISPF is active **and** Z/XDC is communicating via its ISPF interface, the **SWAP** command functions in the same way it does in ISPF: If another program (in addition to the current debugging session) is running under ISPF, then **SWAP** transfers control from Z/XDC to that other program. It also causes the cursor to move to a window belonging to that other work.

Syntax:

SWAP
SWAP PREV
SWAP NEXT
SWAP LIST

SWAP name

Z/XDC does not check any operands that might be given. If any are given, then they are simply passed through for processing by ISPF. ISPF uses the operands to select an existing session to switch focus to. See IBM's **ISPF User's Guide, Vol. I** (SC34-4822) more information.

Help COmmands SWItch

The **SWITCH** command can be used in multi-tasking debugging situations when multiple tasks have either ABENDed or reached breakpoints and, therefore, are all awaiting use of debugging services.

Z/XDC permits the user to hold a debugging conversation with only one task at a time. When multiple tasks request debugging services, Z/XDC forces the additional tasks to wait. The SWITCH command can be used to cause Z/XDC to pass control of the debugging conversation from the current task to another task. The new task can be selected either explicitly or implicitly.

Syntax:

SWITCH tcbaddress
omitted

Shortcut: S [When used with LIST TASKS]

tcbaddress
This explicitly specifies the task to which control of the debugging conversation is to be switched. The specified task **must** currently be waiting for use of debugging services; otherwise, the switch is not made.

omitted
This specifies that control of the debugging conversation is to be passed to the "next" task that is awaiting use of debugging services. There must be at least one other task waiting for debugging services; otherwise, the switch is not made. Note, "next" is defined in terms of a queue whose ordering is essentially random. The queue is stable, however, so a series of SWITCH commands will follow a repeatable circuit through the queue.

When Z/XDC's Terminal Fullscreen Support is turned on, then a task switch can also be accomplished by:

- Issuing the **LIST TASKS** command to display the current space's tasks structure.
- Then placing an **S** shortcut at the left of the task to which you want control of the debugging conversation to be switched.

Note, this will work only for those tasks for which the caption **PENDING XDC** is displayed. For more information, see HELP SHORTCUTCMDS S.

Example:

LIST TASKS
SWITCH TCB#5
If task number 5 (as displayed by the LIST TASKS command) is currently waiting for use of debugging services, then control of the debugging conversation is passed to it. Task number 5 becomes the new current task, and the old current task waits until the debugging conversation is passed back to it.

Help COmmands TDeferred

The **TDEFERRED** command is **identical** to the **ADEFERRED** command except:
- The **TDEFERRED** command creates transient deferred breakpoints.
- The **ADEFERRED** command creates persistent deferred breakpoints.

For complete information, see HELP COMMANDS ADEFERRED.

Help COmmands THaw

The **THAW** command can be used to resolve a deadlocked address space if one or more tasks in that address space have passed control to Z/XDC and if one or more of those Z/XDC tasks are waiting in Conversation Management for control of the debugging conversation. For more information, see HELP MULTITASK CONVERSATIONMANAGEMENT.

This command can be issued by both authorized and non-authorized users. Authorized users, in addition, can use the LIST TASKS command while in Foreign Address Space Mode to determine whether or not any tasks in the Target Address Space are waiting in Conversation Management. Such tasks will be flagged "PENDING XDC".

Syntax:

THAW aspaceref

aspaceref
This identifies the address space that is to be THAW'd. Briefly, this can be any of the following:
- A jobname that uniquely identifies a specific address space. If there exists multiple spaces having the same jobname ("INIT", for example), then the THAW command fails.
- A keyword indicating the home space or the retry level or error level, primary, secondary, or instruction execution address space. Examples: HOME, PASID, ESASID, IASID, etc.
- An ASID number.
- An address expression pointing to an ASID number.
- A register containing an ASID number.
- The name of an equate or dsect that has been assigned to represent storage in an address space. (The assigned address space is THAW'd.)

For detailed information, see HELP COMMANDS SYNTAX ASIDS.

Example:

THAW DBCOLE7
If the TSO session named DBCOLE7 has become deadlocked, and if one or more of the tasks in DBCOLE7 are waiting in Z/XDC's Conversation Management, then issuing this command from a Z/XDC debugging session running in another address space immediately breaks the deadlock.

If, on the other hand, the Target Address Space was not deadlocked (i.e. if a non-Z/XDC task either is using or will use the terminal), and if there is a Z/XDC task waiting in conversation management, then this command will awaken that task and terminal conflict will result. Z/XDC's WAIT command will then have to be used to resolve the conflict.

Help COmmands TRACe

The **TRACE** command is used to step user program execution either one or a few machine instructions at a time. The size of the step depends upon the type of trace requested. For general details and background information, see **HELP BREAKPOINTS TRACING.**

Traces can be **conditional.** That means that they will run your program's execution until any one of one or more conditional expressions resolves as **TRUE.** See **HELP BREAKPOINTS TRACING CONDITIONAL** for more information.

Warning! The only type of trace that is suitable for conditional tracing is a **T BNC** trace.

Traces, like all other breakpoints, can be conducted using either **TRAP2** instructions or **X'00'** opcodes. See **HELP BREAKPOINTS TYPES** for the advantages and the pitfalls of each.

Syntax:

TRACE =familyid **tracetype** (conditional) 'command;command;...' FORCE TRAP2 ZERO

Notes:

The only operand described in this topic is the **tracetype** operand. For all other operands, see **HELP COMMANDS SYNTAX BREAKPOINTS.**

Only the **tracetype** operand is required. All other operands are optional.

The operands may be given in any order.

For comprehensive information about breakpoints in general, see **HELP BREAKPOINTS.**

Operands:

tracetype
This specifies the type of trace to be performed. It may be any of the following:

omitted (Example: **T**)
I (Example: **T I**)
These request an **Instruction Trace.** Control is returned to the user program so that it can execute the machine instruction that is currently pointed to by the **retry level** PSW. Z/XDC receives control again prior to the execution of the machine instruction reached next. That will be:

- Either the next sequential machine instruction,
- Or the machine instruction at the branch target if the current instruction is a branching instruction that actually branches.

For more information, see **HELP COMMANDS TRACE I.**

Note, **T** is the command that is set into Z/XDC's factory default PF key **F9.**

B (Example: **T B**)
This requests a **Branch Trace.** Control is returned to the user program. Z/XDC receives control again prior to the execution of the next branch type instruction or the next EX/EXRL of a branch type instruction. It also stops at all occurrences of SVC, PC, and certain other environment changing instructions. For more information, see **HELP COMMANDS TRACE B.**

Also see **HELP BREAKPOINTS TRACING BRANCHES.**

BY (Example: **T BY**)
This requests a **Successful Branch Trace.** This is a branch trace that stops only if the branch actually will occur. It also stops at all occurrences of SVC, PC, and certain other environment changing instructions. For more information, see **HELP COMMANDS TRACE BY.**

Note, **T BY** is the command that is set into Z/XDC's factory default PF key **F10.**

BNC (Examples: **T BNC** or just **T BN**)
This requests a **Branch-No-Call Trace.** This is a trace that is identical to **T BY** except that it will not follow execution into subroutines.

When the **T BNC** trace reaches a **linkage instruction** (BAL, BASR, JAS, BAKR, etc.), it might (or might not) follow the jump according to the particular way the linkage instruction is being used:

- For **point-and-jump-over** usages (for jumping over inline data), **T BNC** will follow the jump.

- For **call-with-return** usages (for calling subroutines), **T BNC** will **not** follow the jump. Instead, it will set recapture traps at the

subroutine's return points, and then let the subroutine run without being followed.

In order to handle linkage instructions correctly, **T BNC** processing has to distinguish these two usages. That is tricky but doable when the code containing the linkage instruction has been **mapped.** (ADATA or SYM data, either works.)

For all the gory details, see **HELP COMMANDS TRACE BNC.**

* (Example: **T ***)
This requests a **Loop Trace.** This trace allows the instruction currently pointed to by the retry level PSW to be executed **one time.** Then Z/XDC intercepts if the user program loops back and attempts to execute that same instruction a second time. For more information, see **HELP COMMANDS TRACE STAR.**

Note, this kind of trace is not a lot different from just setting a breakpoint at the user program's resume address using the **AT** command.

SB (Example: **T SB**)
This requests a **Store Before Trace.** This trace will not follow execution into subroutines.

The **T SB** trace scans forward looking for the next EX/EXRL, branch, jump or store-type instruction, and sets a recapture trap at that instruction. When that trap is reached, Z/XDC analyzes the now current instruction to determine if it is a store-type instruction. If it is, then the user program is interrupted (subject to conditional expression analysis, if any) **before** the store occurs. Otherwise, the trace silently continues by scanning forward again to the next EX/EXRL, branch, jump, or store-type instruction.

This cycle repeats until a store-type instruction is reached. For complete information, see:
- **HELP COMMANDS TRACE SB/SA**
- **HELP BREAKPOINTS TRACING STOREALTER**

SA (Example: **T SA**)
This requests a **Store After Trace.** This trace will not follow execution into subroutines.

The **T SA** trace scans forward looking for the next EX/EXRL, branch, jump or store-type instruction, and:
- Either sets a recapture trap on the EX/EXRL, branch or jump,
- Or sets a final capture trap just past the store.

When that trap is reached, Z/XDC:
- Either silently advances the recapture trap to the next recapture point,
- Or (subject to conditional expression analysis, if any) accepts the final trap, issues its storage display and awaits further commands.

For complete information, see:
- **HELP COMMANDS TRACE SB/SA**

- **HELP BREAKPOINTS TRACING STOREALTER**

W (Example: **T W**)
This creates a **WATCH.** Any number of WATCHs can be created.

T W does not set any breakpoints, nor does it cause execution to resume automatically. However, WATCHs have **conditional expressions** that are evaluated every time Z/XDC receives control from the user program due to any breakpoint (trap, trace or interim traps used by most traces):

- If **any** conditional evaluates **TRUE,** then program execution is intercepted, automatic commands (if any) are processed, and Z/XDC awaits further input from the user.
- On the other hand, if **all** conditionals evaluate as **FALSE,** then user program execution silently continues.

WATCHs **must** have conditional expressions; therefore, the **T W** command requires the **(conditional)** operand. For syntax information, see **HELP COMMANDS SYNTAX BREAKPOINTS CONDITIONS.**

For complete information about WATCHs, see:

- **HELP COMMANDS TRACE W**
- **HELP BREAKPOINTS CONDITIONAL WATCH**

=familyid
(conditional)
'command;command;...'
"command;command;..."
FORCE
TRAP2
ZERO

For detailed syntax information about all operands other than **tracetype,** see **HELP COMMANDS SYNTAX BREAKPOINTS.**

For information about **conditional traces,** see **HELP BREAKPOINTS TRACING CONDITIONAL.**

For other breakpoint related information, start with **HELP BREAKPOINTS.**

Subtopics Index

Detailed information about each **tracetype** is available in subtopics to this topic. Use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

Help COmmands TRACe I

Instruction Trace (T or T I)

(T and T I are synonyms.)

T is a trace that stops on **every** instruction. It is a terribly slow way to progress through execution. **T BNC** usually is a better choice, but if you want to examine what's happening closely, then **T** is your friend.

A **T** trace will follow execution into subroutines. It does not stay local to the current routine. Therefore, this trace **is not suitable** for conditional tracing. Use **T BNC** instead.

Syntax

In most cases, specifying just **T** is all you'll ever want to do. But for complete syntax details, see **HELP COMMANDS SYNTAX BREAKPOINTS.**

Recapture Traps

A **T** trace follows execution by setting **recapture traps** into the flow of execution and then letting the program run until its execution is recaptured by Z/XDC

In the case of **T** traces, recapture traps are set on each next instruction. For complete information about recapture traps, see **HELP BREAKPOINTS TRACING CONDITIONAL.**

Conditional Tracing

As mentioned above, **T** traces should **not** be used for conditional tracing. This is because these traces **do not** stay within the local code. Instead, they will **follow linkage instructions** into subroutines, and that's a rabbit hole that Z/XDC likely will not be able to climb out of! For more information, see **HELP BREAKPOINTS TRACING CONDITIONAL FAILURES.**

The best trace to use for conditional tracing is **T BNC.** See **HELP COMMANDS TRACE BNC** for more information.

Alien Storage

A **T** trace **will** follow all instructions **except those linkage instructions** that call subroutines and service routines located in **alien** storage. In those cases, **T** will, instead, **step over** them by setting **recapture traps** at the return address and than letting the service routine run without being followed. For more information about alien storage, see **HELP BREAKPOINTS TRACING ALIEN.**

Access Methods and EODAD and SYNAD Exits

Access methods are the z/OS routines that are called to service GET, PUT, READ, WRITE and CHECK requests. They always are located in alien storage, so **T** will always step over them. But access methods require special handling...

They require special handling because, while usually they will return to the code that called them, they also can instead **exit** to **EODAD** and **SYNAD** routines to handle end of data and I/O error conditions, respectively.

So Z/XDC has special logic to recognize access methods. Then when **T** sets up to step over them, additional recapture traps are set into the EODAD and SYNAD routines if they are present. (Z/XDC examines DCBs, DCBEs, DECBs and ACBs, as necessary, to find the exit routines.) This means that control of the trace is not **lost even** if the access method **exits** to the EODAD or SYNAD routines instead of returning to the calling GET, PUT, READ, WRITE or CHECK macro.

PF key F9

Factory default PF key **F9** is set to **T.** See **HELP FULLSCREEN PFKEYS DEFAULTKEYS SET-A** for more information.

Help COmmands TRACe B

Branch Trace (T B)

This is a trace that stops on **all** branching instructions. This is regardless of whether they will branch or fall thru. Other traces usually are better to use:

- If you wish to stop only on branches that will not fall thru, use **T BY.**

- If you wish to stop only on branches that will not fall thru **and** also not follow execution into subroutines, use **T BNC.** (T BNC is usually the best trace to use.)

A **T B** trace will follow execution into subroutines. It does not stay local to the current routine. Therefore, this trace **is not suitable** for conditional tracing. Use **T BNC** instead.

Syntax

In most cases, specifying just **T B** is all you'll ever want to do. But for complete syntax details, see **HELP COMMANDS SYNTAX BREAKPOINTS.**

Branching Instructions

A **branching** instruction is any instruction that has the potential to change the flow of execution. This includes:

- Branching instructions of all sorts:
  BC BCTR BRXH BSM Bwhatever LPSW PT
  BCR BRC BRXLE BXH JCT JXLE PR
  BCT BRCT BSA BXLE JXH Jwhatever

- Transfer instructions:
  LASP BSG SVC PC RP MC TRAP2 SIE
  SVCST DIAG SSM STOSM STNSM LCTL TRAP4
  Five of these (SSM, STOSM, STNSM, LCTL and LASP) are not really transfer instructions, but they are **state changing** instructions that can affect the debugging environment, so Z/XDC always stops when it encounters one during a branch trace.

- Linkage Instructions:
  BAKR BALR BASR BRAS BAL BAS BASSM JAS

- EX/EXRL of any of the above.

A **T B** trace will follow all of the above instructions except transfer instructions. **T B** will stop at transfer instructions but won't follow them. Instead, **T B** will resume following execution when it returns from the called service routine.

Recapture Traps

A **T B** trace follows execution by setting **recapture traps** into the flow of execution and then letting the program run until its execution is recaptured by Z/XDC.

In the case of **T B** traces, recapture traps are set on each branching instruction as it is about to be encountered. For complete information about recapture traps, see **HELP BREAKPOINTS TRACING CONDITIONAL.**

Conditional Tracing

As mentioned above, **T B** traces should **not** be used for conditional tracing. This is because these traces **do not** stay within the local code. Instead, they will **follow linkage instructions** into subroutines, and that's a rabbit hole that Z/XDC likely will not be able to climb out of! For more information, see **HELP BREAKPOINTS TRACING CONDITIONAL FAILURES.**

The best trace to use for conditional tracing is **T BNC.** See **HELP COMMANDS TRACE BNC** for more information.

Alien Storage

A **T B** trace **will** follow linkage instructions **except** those that call subroutines and service routines located in **alien** storage. In those cases, **T B** will, instead, **step over** them by setting **recapture traps** at the return address and than letting the service routine run without being followed. For more information about alien storage, see **HELP BREAKPOINTS TRACING ALIEN.**

Access Methods and EODAD and SYNAD Exits

Access methods are the z/OS routines that are called to service GET, PUT, READ, WRITE and CHECK requests. They always are located in alien storage, so **T B** will always step over them. But access methods require special handling...

They require special handling because, while usually they will return to the code that called them, they also can instead **exit** to **EODAD** and **SYNAD** routines to handle end of data and I/O error conditions, respectively.

So Z/XDC has special logic to recognize access methods. Then when **T B** sets up to step over them, additional recapture traps are set into the EODAD and SYNAD routines if they are present. (Z/XDC examines DCBs, DCBEs, DECBs and ACBs, as necessary, to find the exit routines.) This means that control of the trace is not **lost even** if the access method **exits** to the EODAD or SYNAD routines instead of returning to the calling GET, PUT, READ, WRITE or CHECK macro.

Help COmmands TRACe BY

Successful Branch Trace (T BY)

This is a trace that stops on all branching instructions that will, in fact, branch.

A **T BY** trace will follow execution into subroutines. It does not stay local to the current routine. Therefore, this trace **is not suitable** for conditional tracing. Use **T BNC** instead.

Syntax

In most cases, specifying just **T BY** is all you'll ever want to do. But for complete syntax details, see **HELP COMMANDS SYNTAX BREAKPOINTS.**

Branching Instructions

A **branching** instruction is any instruction that has the potential to change the flow of execution. This includes:

- Branching instructions of all sorts:
  BC BCTR BRXH BSM Bwhatever LPSW PT
  BCR BRC BRXLE BXH JCT JXLE PR
  BCT BRCT BSA BXLE JXH Jwhatever

- Transfer instructions:
  LASP BSG SVC PC RP MC TRAP2 SIE

```
    SVCST DIAG  SSM   STOSM STNSM LCTL  TRAP4
```

Five of these (SSM, STOSM, STNSM, LCTL and LASP) are not really transfer instructions, but they are **state changing** instructions that can affect the debugging environment, so Z/XDC always stops when it encounters one during a branch trace.

- Linkage Instructions:

```
    BAKR  BALR  BASR  BRAS  BAL   BAS   BASSM JAS
```

- EX/EXRL of any of the above.

A **T BY** trace will follow all of the above instructions except transfer instructions. **T BY** will stop at transfer instructions but won't follow them. Instead, **T BY** will resume following execution when it returns from the called service routine.

Recapture Traps

A **T BY** trace follows execution by setting **recapture traps** into the flow of execution and then letting the program run until its execution is recaptured by Z/XDC, at which point Z/XDC determines whether or not the requirements of the trace are met. If not, then a next recapture trap is set, and the program is resumed.

In the case of **T BY** traces, recapture traps are set on each branching instruction as it is about to be encountered. Then when the trap is reached, Z/XDC determines whether or not the branch will, in fact, branch:

- If so, then the requirements of the trace are met, and program execution is interrupted.
- If not, then the requirements are not yet met, and so program execution is silently resumed.

For complete information about recapture traps, see **HELP BREAKPOINTS TRACING CONDITIONAL.**

Conditional Tracing

As mentioned above, **T BY** traces should **not** be used for conditional tracing. This is because these traces **do not** stay within the local code. Instead, they will **follow linkage instructions** into subroutines, and that's a rabbit hole that Z/XDC likely will not be able to climb out of! For more information, see **HELP BREAKPOINTS TRACING CONDITIONAL FAILURES.**

The best trace to use for conditional tracing is **T BNC.** See **HELP COMMANDS TRACE BNC** for more information.

Alien Storage

A **T BY** trace **will** follow linkage instructions **except** those that call subroutines and service routines located in **alien** storage. In those cases, **T BY** will, instead, **step over** them by setting **recapture traps** at the return address and than letting the service routine run without being followed. For more information about alien

storage, see **HELP BREAKPOINTS TRACING ALIEN.**

Access Methods and EODAD and SYNAD Exits

Access methods are the z/OS routines that are called to service GET, PUT, READ, WRITE and CHECK requests. They always are located in alien storage, so **T BY** will always step over them. But access methods require special handling...

They require special handling because, while usually they will return to the code that called them, they also can instead **exit** to **EODAD** and **SYNAD** routines to handle end of data and I/O error conditions, respectively.

So Z/XDC has special logic to recognize access methods. Then when **T BY** sets up to step over them, additional recapture traps are set into the EODAD and SYNAD routines if they are present. (Z/XDC examines DCBs, DCBEs, DECBs and ACBs, as necessary, to find the exit routines.) This means that control of the trace is not **lost even** if the access method **exits** to the EODAD or SYNAD routines instead of returning to the calling GET, PUT, READ, WRITE or CHECK macro.

PF key F10

Factory default PF key **F10** is set to **T BY.** See **HELP FULLSCREEN PFKEYS DEFAULTKEYS SET-A** for more information.

Help COmmands TRACe BNc

Trace Branch-No-Call (T BNC or Just T BN)

This is a trace that is like **T BY** except that **call-with-return** linkage instructions (BAL, BAS, JAS, etc.) are **not** followed. Instead, **recapture traps** are placed at the called subroutine’s return points, and the subroutine is allowed to run. This, effectively, is a **STEP OVER** action.

A **T BNC** trace attempts to stay local to the currently running routine or subroutine. Therefore, this trace **is suitable** for conditional tracing. See **HELP BREAKPOINTS TRACING CONDITIONAL** for more information.

Syntax

In most cases, specifying just **T BNC** or **T BNC (FALSE)** is all you’ll want to do. But for complete syntax details, see **HELP COMMANDS SYNTAX BREAKPOINTS.**

Return Vectors

Typically, when a subroutine has **multiple return points,** those returns are all into a **branch vector** that directly follows the linkage instruction. Example:

```
        JAS   R14,MYSUB
        J     ERROR        +0 Error; Go handle
        J     SPECIAL      +4 Ok but special; Go handle
******* FALL  THRU         +8 Normal; Continue processing
```

The **T BNC** trace will detect and set recapture traps on return vectors up to **six** slots in size.

Linkage Instructions - Branching and Non-Branching Forms

The following are all linkage instructions:

```
    BAKR   BAL   BALR   BAS  BASR
    BASSM  BRAS  BRASL  JAS  JASL
```

They all set a register to point to a return location, and then they all jump somewhere.

Some of them (BAKR, BALR, BASR and BASSM) have **non-branching** forms that Z/XDC checks for. When a non-branching form is recognized, Z/XDC processes it just like any other non-branching instruction: The T B, T BY and T BNC traces all ignore it just as they would ignore a LA instruction, for example.

Linkage Instruction Usages

Effort is made to distinguish linkage instructions used for **call-with-return** purposes from those used for **point-and-jump-over** purposes. Here is a common point-and-jump-over example generated by z/OS's **OPEN** macro:

```
         OPEN  (SYSPRINT,OUTPUT),  Open SYSPRINT            *
               MODE=31             31-bit plist
+        CNOP  0,4          Align list to word
+        BAS   1,*+12 Load reg 1 with list address
+        DC    AL1(143)     Option byte
+        DC    AL3(0)       Reserved
+        DC    A(SYSPRINT)  DCB or ACB address
+        LR    0,1          Set parameter list addr
+        SR    1,1          Set indicator of 31-bit
+        SVC   19           Issue OPEN SVC
```

In this example, a **T BNC** trace must **follow** the jump, not try and capture a "return" that's never going to happen. Keep that in mind.

- For **point-and-jump-over** linkage instructions, (just like for all other branching instructions), **T BNC** follows the jump (just like **T BY** does).

- For **call-with-return** linkage instructions, **T BNC** places recapture traps at the called subroutine's return points, and then lets the subroutine run without trying to follow it.

Distinguishing Call-with-Returns vs. Point-and-Jump-Overs

It's very important that Z/XDC be able to accurately distinguish point-and-jump-over usages from call-with-return! In the example above, the BAS is not calling a subroutine. It's jumping over a parameter list and setting R1 to point to it. If Z/XDC were to misunderstand this and, as a result, attempted to do a step over by setting recapture traps onto the plist, then several severe problems would result:

- The plist would be corrupted!
- So when OPEN tried to use the parameters, it would fail!
- The recapture traps would be in the wrong place, so Z/XDC would lose control of the trace.
- **BOOM!**

Because of this danger, **T BNC** takes a conservative approach:

- If Z/XDC can accurately determine the nature of the usage, it will follow or step over the linkage, as appropriate.
- On the other hand, if an accurate determination cannot be made, attempting a step over action is simply too dangerous, so Z/XDC will follow the linkage.

Identifying Point-and-Jump-Over Usages via Maps

When a **csect map** of your code is present, **T BNC** has all the information it needs to make an accurate Call-with-Return vs. Point-and-Jump-Over determination.

All maps contain information that explicitly identifies all DC and DS statements (whether they're named or unnamed). This is true for both **ADATA** maps and **SYM data** maps.

So when a T BNC trace encounters a linkage instruction, it finds the map (if any) and sees if the instruction is followed by a DC/DS:

- If so, then Z/XDC presumes a point-and-jump-over usage, so it follows the jump.
- If not, then Z/XDC presumes a call-with-return, so it sets return capture traps on up to six machine instructions following the linkage instruction.

This is not foolproof, but we believe that it is good enough to make **T BNC** usable for conditional tracing in most code. See **HELP BREAKPOINTS TRACING CONDITIONAL** for a comprehensive discussion.

Identifying Point-and-Jump-Over Usages Without Maps

When csect maps are not available, T BNC logic cannot reliably determine whether a linkage instruction is followed by other machine instructions or by data.

It is tempting to think that it could be helpful to simply inspect the 1st byte following the linkage instruction. But there are only 256 possible byte values, and as of this writing, **214** of them are value opcode first-byte values. That's a bit over **83%!** So it is highly likely for data to be mistakable for machine instructions.

In fact, take another look at the example above. In the **DC AL1(143)** statement, the **AL1(143)** is **X'8F'** in hex which is the opcode for **SDLA.** Thus absent a map, T BNC has no way to know that the AL1(143) is not an SDLA.

So when suitable CSECT maps are not available, T BNC logic has no choice but to follow some linkage instructions that it might otherwise not follow.

But it does not follow all linkage instructions. There are some judgements that it still can make:

1) When just a **Binder Map** is available, T BNC logic can recognize when a linkage target is located within the current or a different csect. When different, T BNC can safely presume the usage is call-with-return.

2) When even a Binder Map is **unavailable,** T BNC logic can still recognize when a linkage target is located within the current or a different load module. When different, T BNC can safely presume the usage is call-with-return.

3) Otherwise, the only **safe** thing for T BNC to do is **follow** the jump.

So when maps are not present, T BNC will follow linkage instructions that it might not otherwise follow.

Conditional Tracing

Fundamentally, there are two typical ways to trace through code: **Manually** and **automatically.**

While **T BNC** can be used to step through code manually, it is not particularly better that using the three tracing PF keys: **F9, F10** and **F11:**

- **F9** lets you step one instruction at a time.
- **F10** lets you step from taken branch to next taken branch.
- **F11** lets you step over subroutines using your own good judgement for when to do or not do that.

For more information, see **HELP BREAKPOINTS TRACING.**

The place where **T BNC** becomes **invaluable** is in **conditional tracing!** This is because T BNC tries to keep the tracing **local** to the current main routine or subroutine (or its callers). This is very important for performance reasons and for reducing the possibility of tracing failures. For a detailed discussion, see **HELP BREAKPOINTS TRACING CONDITIONAL.**

The problem with trying to use other conditional traces (T BY, for example) is that the trace **will not stay local!** T BY will attempt to follow the trace into all subroutines no matter what. That can be a rabbit hole from which Z/XDC will never return. For the ugly details, take a look at **HELP BREAKPOINTS TRACING CONDITIONAL FAILURES.**

Access Methods and EODAD and SYNAD Exits

Access methods are the z/OS routines that are called to service GET, PUT, READ, WRITE and CHECK requests. They require special handling because, while usually they will return to the code that called them, they also can instead **exit** to **EODAD** and **SYNAD** routines to handle end of data and I/O error conditions, respectively.

Z/XDC has special logic to recognize access methods, so that when **T BNC** sets up to step over them, additional recapture traps are set into the EODAD and SYNAD routines if they are present. (Z/XDC examines DCBs, DCBEs, DECBs and ACBs, as necessary, to find the exit routines.) This means that control of the trace is not **lost even** if the access method **exits** to the EODAD or SYNAD routines instead of returning to the calling GET, PUT, READ, WRITE or CHECK macro.

Help COmmands TRACe STar

Loop Trace (T *)

This is a trace that allows the instruction currently pointed to by the retry level PSW to be executed **one time.** Then Z/XDC intercepts if the user program loops back and attempts to execute that same instruction a second time.

This kind of trace is not a lot different from just setting a breakpoint at the user program's resume address using the **AT** command.

Syntax

In most cases, specifying just **T** * or **T** * **(FALSE)** is all you'll want to do. But for complete syntax details, see **HELP COMMANDS SYNTAX BREAKPOINTS.**

Bypass Traps

T * works by setting a temporary trap (a "bypass trap") on the next instruction to be executed and then allowing execution to resume. The current instruction is executed, and Z/XDC receives control back at the next instruction. Then Z/XDC sets a recapture trap on where execution had been, then allows execution to resume again.

When execution loops around and reaches the recapture trap, program execution is again intercepted.

Alien Storage

When **T** * is issued when execution is at a linkage instruction, Z/XDC attempts to place the bypass trap at the target address. However, if the target is in **alien** storage, the bypass trap has to be set at the presumed return address instead. See **HELP BREAKPOINTS TRACING ALIEN** for more information.

Stuck Instructions

When a branching instruction branches to itself, then execution is stuck either temporarily or permanently. Processing traps placed on stuck instructions requires special recognition and handling.

Permanently stuck branches (such as **J ∗**) are recognized and cause the trace to abort.

On the other hand, temporarily stuck branches (such as **JCT R5,∗**) are also recognized and simulated.

Conditional T ∗

A **T ∗** trace, by its very nature, stays within the local code. Therefore, it is perfectly suitable and reasonable to use conditional expressions with **T ∗** traces. The behavior is exactly the same as using a conditional expression on a **TRAP** command. See **HELP BREAKPOINTS TRACING CONDITIONAL** for more information.

Help COmmands TRACe SB/sa

Store Before Trace (T SB)

The **T SB** trace scans forward looking for the next EX/EXRL, branch-type, linkage-type, jump or store-type instruction, and sets a recapture trap at that instruction. When that trap is reached, Z/XDC analyzes the now current instruction to determine if it is a store-type or linkage-type instruction. If it is, then the user program is interrupted (subject to conditional expression analysis, if any) **before** the store or linkage occurs. Otherwise, the trace silently continues by scanning forward again to the next EX/EXRL, branch, linkage, jump, or store-type instruction.

This cycle repeats until a store-type or linkage-type instruction is reached.

T SB stops on all linkage instructions (BAS BALR JAS etc.) on the presumption that the subroutine about to be called will include store-type instructions.

Store After Trace (T SA)

The **T SA** trace scans forward looking for the next EX/EXRL, branch-type, linkage-type, jump or store-type instruction, and:

- Either sets a recapture trap on the EX/EXRL, branch or jump,
- Or sets a final capture trap just past the store or just past the linkage instruction.

When that trap is reached, Z/XDC:

- Either silently advances the recapture trap to the next recapture point,
- Or (subject to conditional expression analysis, if any) accepts the final trap, issues its storage display and awaits further commands.

T SA stops just past linkage instructions (BAS BALR JAS etc.) on the presumption that the subroutine that has just returned included store-type instructions.

Syntax

In most cases, specifying just **T SA** or **T SA (FALSE)** is all you'll want to do. But for complete syntax details, see **HELP COMMANDS SYNTAX BREAKPOINTS.**

Conditional Tracing

The **T SB** and **T SA** traces will not follow execution into subroutines. Consequently, they are good candidates for conditional tracing. See **HELP BREAKPOINTS TRACING CONDITIONAL.**

More Information

See **HELP BREAKPOINTS TRACING STOREALTER** for a thorough discussion of storage alteration tracing.

Help COmmands TRACe SA

The **T SA** allows user program execution to advance to the next store type instruction, and stops **after** that instruction has executed.

T SB is a similar trace that stops **before** the store type instruction is executed. For complete details about both traces, see **HELP COMMANDS TRACE SB/SA.**

Help COmmands TRACe W

T W creates a **WATCH.** Any number of WATCHs can be created. **T W** does not set any breakpoints, nor will it cause execution to resume automatically. It just creates WATCHs.

A **WATCH** is essentially a conditional expression that is not associated with any particular breakpoint. Normally, a conditional expression would be evaluated only when the breakpoint to which it is assigned is reached by user program execution. But when WATCHs exist, they **all** are evaluated every time Z/XDC receives control from

the user program for any trap or trace that also has a conditional expression.

They also are evaluated at all interim trap points. (See below.)

Every time Z/XDC receives control as just described, all available conditional expressions are evaluated. Then:
- If **any** conditional evaluates **TRUE,** then execution is interrupted.
- If **all** conditionals evaluate **FALSE,** then execution will silently resume.
- (Note, if Z/XDC receives control due to an **unconditional** trap, the WATCHs are ignored, and user program is interrupted. See below for further details.)

Syntax:

TRACE =familyid **W** **(conditional)** 'command;command;...' TRAP2 ZERO

Notes:

The operands may be given in any order.

Only the **(conditional)** operand is required. All other operands are optional. For details about the **(conditional)** operand, see **HELP COMMANDS SYNTAX BREAKPOINTS CONDITIONS.**

In fact, none of the operands are described in this topic. For that information, see **HELP COMMANDS SYNTAX BREAKPOINTS.**

For comprehensive information about breakpoints in general, see **HELP BREAKPOINTS.**

When do WATCHs do Their Magic?

WATCHs are evaluated only when Z/XDC receives control from the user's program. But they do not cause Z/XDC to receive control. Z/XDC must receive control by some other mechanism.

WATCHs are not **SLIP traps.** They do not have the ability to watch for storage alterations globally. Instead, they can only do their evaluations when Z/XDC receives control for some other reasons.

Further, WATCHs are not evaluated for all mechanisms that cause Z/XDC to receive control, only for some.

Reasons Why Z/XDC Receives Control and What Happens When it Does

The mechanisms by which Z/XDC can receive control and how WATCHs are handled for each are as follows:

- **ABEND:** If the user program ABENDs, then its execution is unconditionally

interrupted, and WATCHs are ignored.

- **Unconditional TRACE, TRAP or AT:** If the user program reaches an unconditional trap, then execution is interrupted unconditionally, and WATCHs are ignored.

- **Conditional TRACE, TRAP or AT:** If the user program reaches a conditional trap, then **all** conditional expressions are evaluated. This includes both the trap's conditional plus all WATCH conditionals.

  User program execution is interrupted if **any** conditional evaluates **TRUE.** All must evaluate **FALSE** in order for execution to silently resume.

- **FALSE TRACE, TRAP or AT:** A **FALSE Trap** is a trap whose conditional expression is the **FALSE** constant. This prevents the trap from ever being a reason for execution being interrupted. Its primary usefulness is to shift the entire interrupt/resume decision process over to WATCHs. For more information, See **HELP COMMANDS SYNTAX BREAKPOINTS CONDITIONS TRUEFALSE.**

  So, when Z/XDC receives control due to a FALSE Trap, all WATCH conditionals are evaluated. Execution is interrupted if **any** WATCH evaluates **TRUE.** All must evaluate **FALSE** in order for execution to silently resume.

- **Interim Traps:** These are traps that are used by Z/XDC when searching for a interrupt point. They give Z/XDC the opportunity to determine whether or not a particular **TRACE's** requirement has been met.

  For example, the **T BY** trace places interim traps onto branching instructions located in the user program's execution path. This is so that when execution reaches them, Z/XDC can decide whether or not the branch will fall thru.

  Normally, if the interim trap decides that the trace's requirement has not been met, it will silently cause the user program to resume.

  But if WATCHs are present, they are also evaluated. Then if any happen to resolve **TRUE,** the silent resumption is aborted, and program execution is interrupted after all.

Automatic Commands

WATCHs may have **automatic commands** associated with them. See **HELP COMMANDS SYNTAX BREAKPOINTS AUTOCMDS** for the syntax details.

When a WATCH evaluates TRUE, and it has automatic commands assigned, then those commands are queued for execution.

When **multiple** WATCHs evaluate TRUE, and some or all have automatic commands assigned, then **all** the commands are queued for all WATCHs that evaluate TRUE.

Personally, I often find it useful for a WATCH's automatic command to be an **invalid command!** Why? Because when the WATCH condition is met:
- The command is queued for execution.
- Its execution fails with a parse error.
- This causes Z/XDC to abort all incoming command streams from a sources.
- In other words, all things stop!
- The invalid command is displayed.
- And the workstation's **bell rings!**

That generally gets my attention... Especially when it's been awhile since I created the WATCH, and I've gotten all wrapped up in tracing through the code waiting for the WATCH to trigger.

My favorite invalid command is **PING.**

When I have multiple WATCHs defined, I will assign differing PING "commands" to each of them...
- **ping this** for one,
- **ping that** for another,
- **ping something else** for etc. Then when one of them triggers, I'll know which one.

In other words, a WATCH is sort of the **fire and forget** weapon for debugging. You can setup a WATCH to monitor a condition of interest, and associate with that WATCH an PING "command" that will alert you whenever that condition occurs. You can then turn your attention to other issues, and when Z/XDC eventually sees that the condition has finally occurred, you will be notified.

WATCHs vs. SLIP Traps

A WATCH is **not** an equivalent to a storage alteration SLIP trap! Unlike SLIP traps, a WATCH will not report an alteration at the **moment it happens.** Further, a WATCH will never cause Z/XDC to receive control.

WATCHs will report their results only when Z/XDC receives control for **other reasons.** The frequency with which Z/XDC does receive control is related to the number of traps you have set, and the nature of conditional traces you may have running.

For detailed information about the differences between SLIP traps and conditional expression driven trapping and tracing (as well as the advantages of each over the other), see **HELP BREAKPOINTS TRACING CONDITIONAL SLIPVSXDC.**

Automatic Commands

You can create as many WATCHs as you need. Whenever Z/XDC receives control due to a conditional trap,
- All currently existing and enabled WATCH definitions are scanned for an accepting condition.
- The automatic commands associated with all accepted WATCH definitions are stacked and processed.

- Eventually when all of the automatic commands have been processed, the keyboard unlocks, and the user can then resume his debugging efforts. See **HELP**

BREAKPOINTS CONDITIONAL WATCH for more information.

FALSE Traps and Traces

One way good way to use WATCHs is in conjunction with **FALSE Traps/Traces.** These are conditional traps (AT, TRAP and ATX) or traces whose conditional expressions will always resolve as FALSE. Such traps and (especially) traces will cause Z/XDC to receive control frequently, but in and of themselves, they will never cause Z/XDC to interrupt user program execution. However, each time Z/XDC receives control, all WATCHs are evaluated, and if any of them evaluate TRUE, then user program execution will be interrupted after all.

One good way to set a **FALSE Trap** or run a **FALSE Trace** is to use the **FALSE** constant as the conditional expression. Example: **TRACE BNC (FALSE)**

Warning! If you wish to run a **FALSE TRACE:** (or any conditional trace):
- Do **not** use T, T I, T B or T BY.
- **Do** use **T BNC, T SB** or **T SA.**

This is because T, T I, T B and T BY will follow subroutines, while **T BNC, T SB** or **T SA** will not. See **HELP BREAKPOINTS TRACING CONDITIONAL** for the reasons why that matters.

Further Information

For additional information about WATCHs, see **HELP BREAKPOINTS CONDITIONAL WATCH.**

Help COmmands TRAP

The **AT, ATX** and **TRAP** commands are all **trapping** commands. They set **trapping breakpoints** into programs.

(Hooks are different from traps. Hooks are used to create debugging sessions where none yet exist, while traps need a debugging session to already exist before they can be effective. For information about hooks, see **HELP HOOKS.)**

Traps - The AT Command

The **AT** command is used to set **persistent** traps. Such traps can be cleared only by the OFF command; they are never cleared automatically.

In most cases, it is better to set breakpoints with the **TRAP** command, not the AT command. See more below.

Traps - The ATX Command

The **ATX** command is used to set persistent traps that are **super persistent.** They can be removed only by:

- **OFF** commands that specifically name them. Example: **OFF ATX0003C**
- **OFF** commands that identify them by type. Example: **OFF ATX**
- **OFF** commands that identify them by address, but only when no other kinds or breakpoints exist at said address.
- **O** (off) and **X** (toggle) shortcuts, but only when no other kinds or breakpoints exist at the targeted address.

ATX breakpoints cannot be removed by **categorical** OFF commands. Examples:

```
OFF ALL
OFF ENABLED
OFF #3
```

Neither of these will remove **ATX** traps.

Traps - The TRAP Command

The **TRAP** command is used to set **transient** traps. Such traps are cleared automatically when they (or any other member of the same family) are reached by user program execution **and accepted** by Z/XDC. (A conditional expression, for example, might keep a breakpoint from being accepted.)

In most cases, breakpoints should be set with this **TRAP** command, not the AT command. The AT command should be used only when you expect the breakpoint to be reached by execution multiple times, not just once.

T is an abbreviation for **both** the **TRAP** command and the **TRACE** command. Z/XDC resolves this ambiguity via inspection of the given operands.

General Syntax:

The complete syntax of all breakpointing commands can be found at **HELP COMMANDS SYNTAX BREAKPOINTS.** But briefly, the syntax of trapping commands is this:

```
AT   =familyid addressexpressions (conditional) 'cmd;cmd;...'  ...
ATX
TRAP


... SAVE=YES  REMOVAL=PURGE    ZERO
         =NO          =DISABLE  TRAP2
     omitted   omitted

    Shortcuts: A
               T
```

Notes:

The above is just a summary. The complete syntax of all breakpointing commands can be found at **HELP COMMANDS SYNTAX BREAKPOINTS.**

For the trapping commands, addressexpressions are a required operand. All other operands may be omitted.

Operands

addressexpressions
This must be a list of one or more address expressions separated from each other by commas or blanks. Here are the rules:
- The address expressions must resolve to halfword boundaries in storable areas of virtual storage.
- A separate breakpoint is created for each given address.
- All of the breakpoints created by a single issuance of a breakpoint command are assigned to the same family.
- The opcode at a breakpoint address need not be valid.
- Other breakpoints can already exist at a given address.
- A breakpoint target address can be either an **EX** instruction or the **target** of an EX instruction, or both. (Ditto for **EXRLs.)**

=familyid
(conditional)
'command;command;...'
"command;command;..."
SAVE=NO [or =YES]
REMOVAL=PURGE [or =DISABLE]
ZERO
TRAP2
For detailed syntax information about all operands other than addressexpressions, see **HELP COMMANDS SYNTAX BREAKPOINTS.**

Transient CDP Changes within Multiple Address Expressions

Like address expressions for any command, the address expressions for the AT/ATX/TRAP commands may make use of Z/XDC's **Current Display Pointer (CDP)** to reduce the amount of typing necessary. This is particularly useful when the commands are given **multiple** address expressions. In this case:

- Those address expressions that **do not** reference the CDP temporarily change the CDP.

- While those address expressions that **do reference** the CDP do not change it.

- Once all address expressions are parsed, the CDP is restored to its value from before the command was issued.

(Weird, huh?)

Well, these rules have the following useful consequences:
- Commands that execute following an AT/ATX/TRAP command will not see any change in the CDP temporarily caused by the AT/ATX/TRAP command.
- A sequence of consecutive address expressions (within the AT/ATX/TRAP command)

that all have implied base terms will all use the same CDP value.
- An address expression that has an explicit base term can be used to temporarily set the CDP for use by immediately following expressions that use implied base terms. (Example: **R14? +4 +8 +C**)

For more information, see **HELP ADDRESSING IMPLICIT.**

Examples:

AT R14? +4 +8 .JOBTOP1 +2 .PRNTSUB 'L REGS;F R15?'
- This command sets six persistent breakpoints at the six given locations: R14?, R14?+4, R14?+8, .JOBTOP1, .JOBTOP1+2, and .PRNTSUB
- They are all assigned to the same breakpoint family.
- The breakpoint family identifier is chosen by Z/XDC. (No =nnn" operand is present.)
- They are all unconditional.
- They all have two automatic commands assigned to them: "L REGS" and "F R15?".

Note that three of the address expressions above ("+4", "+8", and "+2") made reference to temporary CDPs set by preceding address expressions ("R14?" and ".JOBTOP1"). See **HELP ADDRESSING IMPLICIT** for more information.

T = R13? (R1,GT,7FFFFFFF) ''
- This command sets one transient breakpoint located at the address pointed to by R13.
- It is assigned to the same family as the family used by the preceding breakpoint command (indicated by the "=" not followed by a decimal number).
- It is conditional: When reached by user program execution, it won't be accepted by Z/XDC unless the 4-byte value in R1 is negative.
- It has associated with it the same automatic commands that were associated with the preceding breakpoint command (indicated by the presence of the null commands string).

ATX =3 .+4 ()
- This command sets one super persistent breakpoint.

- It is located at +4 bytes past the zero point for the default csect (as established by the SET QUALIFIER command).
- It is assigned to breakpoint family #3.
- It has the same conditional expression that was assigned by the preceding breakpoint command (indicated by the "()"),
- It does not have any automatic commands assigned to it.

F PSW? 0;T +4 +8 +C
T PSW?+4 +4 +8 [not a typo]
- Believe it or not, these two command strings do basically the same thing. They both set a family of transient breakpoints at +4, +8 and +C past the address pointed to by the retry level PSW.

- (Notice that the F PSW? command sets the CDP to one location, while the T PSW?+4

sets a temporary CDP to a different location. Hence, The T command's operands PSW?+4 +4 +8 are correct for setting breakpoints at +4, +8 and +C past the resume execution location.)

- If, during subsequent user program execution, any of these breakpoints is reached, then all of them are automatically cleared.

- This sort of thing is useful in the cases where:
  - The PSW points to a JAS instruction (for example),
  - And the subroutine being branched to returns to various points within a branch vector table following the JAS,
  - And you want to let the subroutine run without stepping through it,
  - But you need to recapture execution when it returns somewhere into the return vector.

  A command like one of these will do the trick. See **HELP BREAKPOINTS TRACING SUBROUTINES** for related information.

- (Notice that the default definition for **F11** does something similar to this example. That PF key is very useful both for stepping over subroutines, and for letting certain loops run to completion before falling through the bottom.)

For related information, see **HELP BREAKPOINTS.**

A Nomenclature Issue

For historical reasons, we have gotten into sort of a bind over exactly what is meant by the word **trap:**

- On the one hand, it has a functional meaning concerning the differences between breakpoints that are use to assist in the stepping of execution through your code, vs. those that are intended to trap program execution at a specific location.

  - I will refer to the former as **tracing breakpoints** because they are created, originally by the **TRACE** command (but more recently also by c/XDC's **STEP** command).

  - I will refer to the latter as **trapping breakpoints** because they are used to trap execution at specific locations. They are created by the **TRAP AT ATX TDEFERRED** and **HDEFERRED** commands.

- While on the other hand, the word **trap** has an implementation meaning regarding the physical nature of the breakpoint itself. Specifically, breakpoints may be built from:
  - Illegal opcodes **(X'00')** which I'll refer to herein as **0C1-type** breakpoints,
  - Or TRAP2 instructions **(X'01FF')** which I'll refer to herein as **TRAP2-type** breakpoints.

Both tracing and trapping breakpoints can be implemented either as 0C1-type breakpoints or TRAP2-type breakpoints. So to recap:

- **TRAP2-type breakpoint** refers to any breakpoint (for tracing or for trapping) that is implemented via a TRAP2 instruction.

- **Trapping breakpoint** refers to any breakpoint (regardless of implementation) that is intended to trap program execution at a specific location. Trapping breakpoints can be built from either 0C1-type breakpoints or TRAP2-type breakpoints.

Help COmmands TSo

This command may not be used when using cdf/XDC.

Z/XDC's **TSO** command permits the user to issue arbitrary TSO commands and CLISTs without having first to get out of the debugging session. Z/XDC executes the requested command as a subtask of the task currently being debugged. Z/XDC waits for the attached task to complete before allowing the current task and debugging session to proceed.

If the attached TSO command should ABEND, then Z/XDC gives the user the option of debugging or ignoring the ABEND.

Z/XDC's TSO command can be used only when Z/XDC is running from within a TSO environment.

Syntax:

TSO tso-command operands

tso-command
This is the name of the TSO command or CLIST to be executed.

operands
These are the operands, if any, needed by the requested command. Their syntax is dependent upon the requested command.

Note, No upcasing of any data is performed by Z/XDC.

Examples:

TSO LISTD 'SYS1.LPALIB'
The LISTD command is executed. Debugging of the current task resumes when the LISTD command completes.

TSO XDCCMD FREEALL EXCLUDE(ISPPROF)
The XDCCMD command is executed to debug a locally written command named FREEALL. When the debugging of FREEALL is completed, debugging of the current task resumes.

TSO ISPF
The current debugging session is suspended and an ISPF session is started. When the ISPF session ends, the current debugging session is resumed.

Note, ISPF can be invoked this way only if Z/XDC is not already running in an ISPF environment. Z/XDC's Fullscreen Support, when turned on, provides a preferable way to invoke ISPF: Z/XDC's "ISPF" command can reinvoke ISPF even when ISPF is already running. See HELP COMMANDS ISPF for more information.

Help COmmands UP

The **UP** command causes a display window to be scrolled upwards (usually). Prior messages are brought into view. The window that is scrolled is always the one that contains the cursor at the time the command is issued.

UP Commands and PF keys

F7's factory default value is **UP -.** The factory default setting for shift-F7 is **UP M.**

In the definition of F7, the dash is important: It permits you to optionally place an operand on a Window's command line and have that operand merged with the PF key's definition. For example, If you placed an **8** on the command line and then pressed **F7,** then an **UP 8** command would occur.

If the dash were not included in the PF key's definition, then any potential operand placed on the command line would be ignored. For more information about this, see HELP FULLSCREEN PFKEYS.

UP Commands and the Primary Window

The Primary Window has associated with it a Scroll Area that keeps a history of your debugging session. This typically is up to **10,000** (or so) lines long. The **UP DOWN LOCATE** and **SCANLOG** commands can be used to navigate freely through this recent history. For more information, see HELP FULLSCREEN WINDOWS PRIMARYWINDOW.

As you scroll around through the history, you will quickly see that every command string that you have issued from the command line has been given a sequence number and has been saved into the Scroll Area. Following each command string, you will see all of the messages generated by that string's commands.

You may also see command strings that you did not type in. Usually, these are commands that Z/XDC has generated internally in order to effect some Point-and-Shoot command (or other fullscreen action) that you have issued. **ProTip:** You should take the time to examine and understand these generated commands. They sometimes can be quite instructive...

Unlike Watch Windows (see below), the concept of display stability does not apply to the Primary Window. Every time a command string that generates messages is executed, the Window is always repositioned to show the first of the newly generated messages

at the top of its display area.

Also, the command line is erased in preparation for you to type in your next commands.

Dual Use - Scrolling Thru Storage

Unlike the DOWN command (see HELP COMMANDS DOWN), the UP command cannot be used to scroll through storage. In can only scroll through a Window's Scroll Area.

UP Commands and Watch Windows

Unlike the Primary Window, Watch Windows do not keep histories of their activities. The only messages available for scrolling are those that have been generated by the most recent issuances of their command strings. So there is no opportunity to use scrolling commands to peruse the Window's recent history. You can scroll only within the current instance of the display generated by the Window's current command string.

But another difference vis-a-vis the Primary Window is that the command line's contents are retained so that its displaying-type commands can be re-executed each time you press the ENTER key (or a PF key). This means that the general form of the display remains reasonably stable while the data itself can change in near real time.

If you use scrolling commands to reposition into the display, this scroll position will be retained either until you issue more scrolling commands or until you change the content of the command line. But see HELP COMMANDS DOWN for more information.

Syntax:

```
UP CURSOR+nnn-nnn...
   FULL  |   |
   PAGE  |   |
   DATA  |   |
   HALF  |   |
   MAX   |   |
   CMD   |   |
   TRACE |   |
   nnn   |   |
   #nnn  |   |
   omitted V   V
```

Notes:

- Every operand **(including the omitted operand)** can optionally be appended by one or more plus or minus adjustments that adjust the display upwards or downwards by some number of lines. Examples:
  - UP 10+5 [Works like UP 15]

```
      - UP +15 [Ditto]
      - UP -15           [Works like DOWN 15]
      - UP MAX-5
      - UP HALF+12-3+4   [Works like UP HALF+13]
      - UP #10+3
    See below for more information.
```

- It is possible to form UP command operands that would cause positioning to change in an downwards direction (like a DOWN command). This occurs when operand resolution results in a negative scroll distance.

- Command operands that result in a positive scroll distance will scroll upwards. Negative distances result in downwards scrolling.

- Any scrolling that would result in positioning above the top of the Scroll Area will be converted into a simple **UP MAX** command.

- Similarly, any scrolling that would result in positioning completely below the bottom of the Scroll Area will be converted (effectively) into a simple **DOWN MAX** command.

UP **CURSOR**
If the cursor is within a window’s display area, then that window is scrolled upwards so as to move the line containing the cursor to the bottom of the window. If the cursor is located on the window’s command line, then **UP CURSOR** functions like **UP FULL:** The window is scrolled upwards so as to bring the previous full window’s worth of information into view.

UP **FULL**
UP **PAGE**
The window containing the cursor is scrolled upwards so as to bring the previous full window’s worth of information into view.

UP **DATA**
The window containing the cursor is scrolled upwards so as to move the previously top display line to the bottom of the window.

UP **HALF**
The window containing the cursor is scrolled upwards half a window’s worth.

UP **MAX**
The window containing the cursor is scrolled upwards to the top of its scroll area. The first message currently still saved in the scroll area is positioned at the first line of the window’s display area.

UP **CMD**
The window containing the cursor is scrolled upwards so as to bring into view the previously logged command string and its generated display. The command string is shown at the top of the window's display area.

UP **TRACE**
This command is valid only when issued within the Primary Window. It functions like UP MAX when issued from a Watch Window.

This command scrolls the Primary Window upwards to the first display generated by the last prior time that Z/XDC received control from the user program. Usually, this will be a display generated by an internally issued **WHERE** command.

Usually, Z/XDC receives control as a result either of a Hook, a Trap or a Trace action. Usually, those commands will cause a **WHERE** command to be issued in order to display the logic at the location where user program execution reached the trap or hook.

Well if, for example, you have used a **T BY** command repeatedly to step through your code, you can repeatedly use an **UP T;RETRIEVE** command to **replay execution backwards** just is if you were watching a movie!

Even easier, if you have the default PF keys in effect, then you can run your movie just by repeatedly typing **T** on the command line and pressing **F7** (or the **Page Up** key, depending upon your keyboard setup).

UP **nnn** [Example: UP 10]
The window containing the cursor is scrolled upwards the specified number of lines. This number may range from 0 to 32,767.

UP **#nnn** [Example: UP #10]
As you scroll around through the Primary Window's scroll area, you will notice that the recorded copies of the Z/XDC commands that you have issued have been assigned sequence numbers. This form of the UP command scrolls the Primary Window **either** upwards or downwards so as to bring the desired command string into view at the top of the window's display area.

Before you ask, the following three command forms all function **identically:**
- UP #nnn
- DOWN #nnn
- LOCATE #nnn

This form of the UP command is valid only for the debugging session's Primary Window. It cannot be issued from any Watch Window.

omitted [Example: UP]

The window containing the cursor is scrolled upwards by that window's default scroll amount. This default can be set by the SET WINDOW VERTICAL command. It can be set to CURSOR, FULL, DATA, or HALF. It **cannot** be set to MAX, CMD, nnn or #nnn. For more

information, see HELP COMMANDS SET WINDOW VERTICAL.

+nnn
-nnn
Every one of the above described operands can be modified by appending one **or more** plus or minus adjustments to it. They work by adjusting the scroll position determined so far either upwards (for +nnn) or downwards (for -nnn) the specified distance.

Examples of this usage can be found above.

Help COmmands UNdub

The **UNDUB** command can be used to remove USS dubbing from the current task, providing:
- it is not a formal USS process.
- it has been 'dubbed'.

You cannot undub a formal USS process or a task that has not been dubbed.

Syntax:

UNDUB

This command has no operands.

Examples:

UNDUB
If the task being debugged is not a formal USS process and has been dubbed then dubbing will be removed.

Help COmmands USing

The **USING** command assigns or reassigns a base value to a given dsect map. The base may be an address that is either fixed or floating. A "fixed" base address is resolved at the time that the USING command is issued, and that resolved value is used each time that the dsect is referenced. A "floating" base address is saved in an unresolved form so that it can be recomputed each time that the dsect is referenced; consequently, the address may resolve to different locations at different times.

Dsects whose bases resolve to a location in common storage are considered to be **global.** They are valid for formatting displays regardless of what the current Target Address Space is. On the other hand, dsects whose bases resolve to locations in a private area are considered to be **local.** They are not used to format displays for other address spaces. **A dsect's LOCAL/GLOBAL attribute can be overridden.**

Whenever a base is assigned to a dsect map, the map is placed at the top of the dsect map search order. This is important because it affects the resolution of an incompletely qualified label when that label exists in more than one dsect.

A special form of the **USING** command can be used to force a given dsect to the top of the search order without changing its base address.

When the USING command is used to assign to storage a dsect version of a Module Map, the rest of Z/XDC is made aware of the module represented by the map. This is particularly useful for **Privately Loaded** load modules whose existence is not recorded in system control blocks (CDEs, LPDEs and CICS APEs). For detailed information, check out HELP MAPS PRIVATELYLOADED.

Like any other dsect map, dsect versions of Module Maps can be assgned either fixed or floating bases. When their bases float, the underlying module's location can appear to change automatically.

Dsect maps can be **autocloned.** An entire series of maps can be automatically generated to represent multiple entries in a table or on a chain. For tables (and of course chains), the entries can be either fixed length or variable.

Like other dsect maps, autocloned maps can be either fixed or floating. However, care must be taken not to use autocloning to create "too many" floating maps! This is because the resolution of floating maps is extremely CPU intensive, so when too many floating equates and maps exist, Z/XDC's responsiveness will become unacceptably sluggish.

A Command Dialog is available for assisting with constructing and issuing USING commands. Just issue the command either with no operands or with a lone question mark as its sole operand.

Syntax:

?
USING
USING ?

[Invokes a Command Dialog]

```
    USING dsect-reference addressexpression PRIVATE FLOAT    ...
                                            LOCAL   omitted
                                            COMMON
                                            SHARED
                                            GLOBAL
                                            omitted

      ... AUTOCLONE CFOFFSET=offset LFOFFSET=offset MAXCLONE=number
                    CFWIDTH=width   LFWIDTH=width
    (autocloning)   CFMASK=mask     TESIZE=size
     (operands)     ZCVALUE=address ZTVALUE=address
                                    ZTPTR=address
```

- The dsect-reference and addressexpression operands are required, the rest are optional.

- The dsect-reference and addressexpression must be given in the order shown. The optional operands can be given in any order.

- Operands within columns are mutually exclusive.

- Autocloning occurs only when one or more of the autocloning operands are given. When none of them are given, then the referenced map is assigned a base, and no additional maps are generated. The autocloning operands are described in HELP COMMANDS USING AUTOCLONING.

dsect-reference
This refers to the dsect whose base is to be assigned or reassigned. It can simply be the name of the dsect, or it can refer (in various ways) to a field within a dsect, or it can be more complex than that. So for a detailed description, See HELP COMMANDS USING DSECTREF.

For autocloning, the resolution of the dsect reference provides the name of the dsect map that is to be cloned. The name of each generated map will consist of this map's name appended by a sequence number.

addressexpression
This must be an expression that defines the desired base address. This address may be either a common storage address, or an address that is located in a private area of any accessible address space, or a data space address, or an address in real storage. (For autocloning, addressexpression provides the location of the first table or chain entry to be mapped.)

PRIVATE or **LOCAL**
These operands are aliases of each other. They override Z/XDC's determination of the scope of the dsect, forcing the dsect to be treated as a private area dsect even if it actually is assigned to a common area location.

Normally when common storage is displayed that has an dsect assigned to it, the display will be formatted by the dsect regardless of the address space that is being displayed. However, when an dsect in common storage has been assigned the **PRIVATE** attribute, it will show up in the display only when that display is of the address space to which the dsect has been assigned.

COMMON or **SHARED**
These operands are aliases of each other. They override Z/XDC's determination of the scope of the dsect. They cause the dsect to be treated as being owned by all address

spaces even if it resolves to a private storage location. This means that when a FORMAT command displays the location to which the dsect is assigned, the dsect will be displayed regardless of what address space is being displayed.

A shared dsect will **not** be displayed when real storage or data space storage is being displayed.

GLOBAL
A global dsect is owned by ALL spaces: address spaces, data spaces, and real storage. A global dsect that's been assigned to a particular address will be displayed in any space for which that address is being displayed. It will be displayed regardless of whether the space is an address space, a data space, or real storage.

GLOBAL, COMMON, SHARED, PRIVATE, and **LOCAL** all omitted
If the dsect resolves to a common storage location, then it is treated as being COMMON. Otherwise, it is treated as being PRIVATE.

FLOAT
This causes the dsect's base to "float". The base address expression is saved in an unresolved form and is recomputed whenever the dsect is referenced. As a result, the dsect can resolve to different locations at different times, depending upon the possibly changing values of pointers that the basing address expression might reference.

When **FLOAT** is omitted, the dsect map's defining address expression is computed immediately upon being parsed. Accordingly, the map's location remains fixed to a specific address.

Warning! When creating a large number of dsect maps through autocloning, care must be taken not to create "too many" floating maps! This is because the resolution of floating maps is extremely CPU intensive, so when too many floating equates and maps exist, Z/XDC's responsiveness will become unacceptably sluggish.

autocloning
For descriptions of the autocloning related operands, see HELP COMMANDS USING AUTOCLONING.

Alternate Syntax:

USING dsectname samedsectname

This form of the USING command forces the given dsect to the top of the dsect search order without altering its base address or other characteristics.

dsectname
This must be the name of a currently active dsect.

samedsectname
This must be exactly the same dsect name.

See HELP COMMANDS USING EXAMPLES for examples of the USING command.

Help COmmands USing Dsectref

The first operand of the **USING** command is a dsect map reference. The syntax of this reference is quite flexible and powerful. It is designed to allow you to assign a base address to a dsect by reference to any of the following:

- The dsect's zero point. (See HELP COMMANDS DMAP for information about zero points.)
- Any offset (positive or negative) from that zero point.
- Any field within the map.
- Any offset from any field within the map.

An offset can resolve to any relative location either within or outside of the map. **Example:**

USING CVTFIX.CVTTCBP 10?
USING .CVTTCBP-200 10?-200
The above two commands are equivalent. The second is a mildly strained example to illustrate the following:

- The actual name of the map does not have to be given. It is sufficient to refer to any field within the map. Z/XDC will automatically search all available dsect maps for the first one containing the given name.

- The map's relative location being assigned to storage is 200 hex bytes preceding the location of the CVTTCBP field. But that relative location is being assigned to the storage location that is 200 hex bytes preceding zero point of the CVT control block. Consequently, the dsect map will be correctly position over the CVT control block.

For autocloning, the resolution of the dsect reference provides the name of the dsect map that is to be cloned. The name of each cloned map will consist of this map's name appended by a sequence number.

Dsect Reference Syntax:

dsectname.fieldname+offset

dsectname
This names a particular dsect. Either it or ".fieldname" or both must be given. (dsectname and .fieldname may not **both** be omitted.) If dsectname is omitted and fieldname is given, then Z/XDC will search all available dsect maps until it finds the first one that contains the named field. See HELP COMMANDS USING for information about the search order used.

.fieldname
This names a particular field within a dsect. Either it or "dsectname" or both must be given. (dsectname and .fieldname may not **both** be omitted.)

+offset
-offset
This is optional. If given, then it must be a hexadecimal value or an n-trailed decimal value to be included in the calculation for finding the desired map-relative location to be assigned to a storage location. (Example: **+10** and **+16N** both mean plus sixteen.)

Either "dsectname" or ".fieldname" or both must be given.

- If only "dsectname" is given, then the dsect is based such that its zero point coincides with the given base location.
- If ".fieldname" is given, then the dsect is based such that the given label coincides with the given base location.
- If "+offset or -offset" is given, then the base location is adjusted by the value of the offset.

If both "dsectname" and ".fieldname" are given, then the named field must be contained within the given dsect.

If only ".fieldname" is given, then all dsects (based and unbased) are searched until one is found that contains the name. This then becomes the map whose base is set. See HELP COMMANDS USING For information about the search order used.

Help COmmands USing Autocloning

Autocloning is a method by which you can automatically generate an entire series of equates or dsects maps to represent either each entry in a table or each control block on a chain. The equates or dsects that are created all have unique sequence numbers but share a common root name that you must provide.

This topic describes those operands of the **EQUATE** and **USING** commands that specifically relate to autocloning. The remaining operands are described in the parent topic **(HELP COMMANDS EQUATE** or **HELP COMMANDS USING).**

Chains

The **EQUATE** and **USING** commands have operands that let you define the following about chains:

- Where the chain field is.
- How wide the chain field is (3, 4, or 8 bytes).
- A mask for selecting a chain field’s significant bits.
- The chain field value that signifies end-of-chain (zero, usually).
- The maximum number of chain entries to label or map.

Tables

Z/XDC’s autocloning supports contiguous tables of either fixed length or variable length entries. When variable, the table entries will (presumedly) have a length field. This length field can be of any reasonable width, and it may define either the table entry’s entire length or only the length of the entry’s variable portion (in which case, all entries in the table are presumed to consist partly of common fields of a fixed size). In support of all this, the **EQUATE** and **USING** commands have operands that let you define the following about tables:

- For fixed length table entries, or for variable table entries containing a fixed length root section, you can specify the length either of the entries or of the root section.
- For variable length table entries, you can specify where the length field is and how wide it is (1 through 8 bytes).

- You can specify either the length field value (if any) that identifies the end of the table or the address at which the table ends.
- Finally, you can specify the maximum number of table entries to label or map.

Whenever a series of equates or maps is created via autocloning, Z/XDC first checks to see whether or not a series of previously created equates or maps, having the same root name, already exists. If so, then that series is purged in its entirety prior to the creation of the current series.

The equates or dsects that are generated through autocloning can all have the same attributes and characteristics that any other equate/dsect might have. In particular, they can be assigned either fixed bases or **floating bases,** thus the equates/dsects can be created to represent either particular instances of or generic templates for table or chain entries.

Command Dialog

A Command Dialog is available for assisting with constructing and issuing USING commands (but not EQUATE commands yet). Just issue the command either with no operands or with a lone question mark as its sole operand.

Syntax:

?
USING
USING ?
[Invokes a Command Dialog]

```
EQUATE name  address  length  PRIVATE  FLOAT    INSTRUCTION ...
USING  COMMON   omitted  DATA
                              GLOBAL            NOBIAS
                              omitted           omitted

        autocloning
        operands        chain-related    table-related
    ... AUTOCLONE       CFOFFSET=offset  LFOFFSET=offset
        1STCLONE=nnn    CFWIDTH=width    LFWIDTH=width
        MAXCLONE=count  CFMASK=mask      TESIZE=size
                        ZCVALUE=address  ZTVALUE=address
                        ZTPTR=address
```

Notes:

- The name and address operands are required, the rest are all optional.
- The name and address expression must be given in the order shown. The optional operands can be given in any order.
- All **chain** related operands are **mutually exclusive** with **table** related operands.

- Autocloning occurs only when one or more of the autocloning operands are given. When none of them are given, only a single equate is generated.

Autocloning Operands Related to Both Chains and Tables

1STCLONE=number
This sets the start of the range of sequence number to be assigned to the generated equates. The default first sequence# is **1.**

The given value may range from **0** to **999999.**

The value set for this operand affects the maximum value that can be used for the **MAXCLONE** operand.

MAXCLONE=number
This sets a limit on the maximum number of equates or dsect maps to be created through autocloning. **Number** may range from 1 to **1000000,** but adjusted for whatever you might provide with the **1STCLONE=** operand. More precisely, the maximum allowed value for **MAXCLONE=** is **(1000000-1STCLONE).** Example: -
When **1STCLONE=1,** then **MAXCLONE=** may range from 1 up to **999999.**

Notes:
- The default limit is **100.**

- The number of significant digits in the highest permitted clone# determines the width of the sequence number that is appended to the equate's or dsect map's root name in order to create uniquely named equates/maps.

- Example, if **1STCLONE=10** and **MAXCLONE=99,** then the highest possible clone# will be 109, which is 3 digits wide. So all generated sequence numbers will have 3 digits.

- The highest permitted clone# is **999999** This limits the sequence# field width to never more than six digits.

- When **FLOAT is given,** the number of equates or maps that will be created will **always** be the **MAXCLONE=** number. (So be careful of large numbers!)

- On the other hand, when **FLOAT is NOT given,** the number of equates/maps created will be limited to the number of chain elements (or table entries) that actually exist.

AUTOCLONE
This activates autocloning. Normally, it is not needed since specifying any of the other autocloning operands will also activate autocloning. But due to defaults, this operand can be used to activate autocloning without providing any of the other autocloning operands. When this happens, the following defaults will be in effect:
CFOFFSET=0

```
CFWIDTH=4
CFMASK=7FFFFFFF
ZCVALUE=0
1STCLONE=1
MAXCLONE=100
```

In other words, specifying **AUTOCLONE** without providing any of the other autoclone operands will generate equates or dsects for labeling/mapping up to the first 100 instances of any chain whose chain field is 4 bytes wide and located at the start of each chain entry.

Chain-Related Autocloning Operands

CFOFFSET=offset
This provides the offset, from the start of each chain entry, of the entry's chain field. This must be a hex number or an N-trailed decimal number whose value ranges from 0 to 7FFFFFFF. (Example: **10** and **16N** both mean sixteen.) The default offset is 0.

CFWIDTH=width
This defines the width of the chain field. Permitted values are **3, 4,** or **8.** The default width depends upon whether or not the CFMASK= operand is given. If it is given, then the default width is the width of the mask provided by CFMASK=. If CFMASK= is not given, then the default width is 4.

CFMASK=mask
This defines a selection mask that is AND'd against the chain field's value to eliminate bits that are not part of the chain pointer. The width of the mask must match the width of the chain field (as defined by the CFWIDTH= operand). The default mask depends upon the width of the chain field as follows:

```
WIDTH  DEFAULT MASK
  3    FFFFFF
  4    7FFFFFFF
  8    FFFFFFFFFFFFFFFF
```

ZCVALUE=address
ZCVALUE=NEGATIVE
ZCVALUE=NOTPOSITIVE
Z/XDC **always** considers that a chain has ended when it encounters a zeroed chain field. But some chains (request block queues for example) actually have a different end-of-chain value. This ZCVALUE= operand can be used to provide such an alternate value. The following ZCVALUE= values are supported:

ZCVALUE=address
The given address expression is resolved, and the resulting address is compared to the contents of each chain entry's chain field. A match identifies the chain's last entry.

ZCVALUE=NEGATIVE

ZCVALUE=NOTPOSITIVE
Both of these keywords cause any zero or negative value to signify end of chain. **Important!** When testing for a negative value, the CFMASK= mask is ignored. In other words, the **field's** hi-order bit is checked. If it is on, then the chain is ended.

The default ZCVALUE= value is 0.

Table-Related Autocloning Operands

LFOFFSET=offset
If the table entries have a length field, then this operand provides the offset of that field from the start of each table entry. **Offset** must be a hex number or an N-trailed decimal number whose value ranges from 0 to 7FFFFFFF.
(Example: **10** and **16N** both mean sixteen.)

The default is to presume that the table entries do not have a length field, and therefore, that all of the table entries have the same size. (See TESIZE= below.)

Notes:
- If **LFOFFSET=** is given, then **LFWIDTH=** also must be given.
- The table entry's length field, if present, can provide either the entire length of each table entry or just the length of a variable portion of each table entry:
  - In the first case, be sure to also specify (or default to) **TESIZE=0** (see below).
  - In the second case, you will have to use the **TESIZE=** operand to provide the length of the nonvariable part of each table entry.

LFWIDTH=width
If the table entries have a length field, then this operand provides the width of that field. **Width** may range in value from 1 to 8.

There is no default for LFWIDTH=, since if LFOFFSET= is given, then LFWIDTH= also must be given (and vice-versa).

TESIZE=size
If the table to be labeled/mapped has entries that are all the same size, or if the entries are variable but also have a fixed section of a constant size, then this operand provides the length of the fixed entries or of the fixed section. **Size** must be a hexadecimal number or an N-trailed decimal number having a value that ranges from 0 to 7FFFFFFF. (Example: **10** and **16N** both mean sixteen.) The default value is 0.

ZTVALUE=address
ZTVALUE=NEGATIVE
ZTVALUE=NOTPOSITIVE
Either **ZTVALUE=** or **ZTPTR=** or **MAXCLONE=** can be used to provide end-of-table information. **ZTVALUE=** and **ZTPTR=** are mutually exclusive. **(MAXCLONE=** is not.)

ZTVALUE= can be used when the table entries lengths are defined by a length field. It causes Z/XDC to examine the contents of the length fields to look for a specified end-of-table condition.

When **TESIZE=0,** then a length field must be present, and a length field containing a zero will always end a table. However, when **TESIZE>0** and length fields also are present, then a zeroed length field does **not** necessarily end a table.

There is no default value for **ZTVALUE=.** When **ZTVALUE=** is omitted, the length field's value does not participate in end-of-table determination (except as noted in the preceding paragraph).

ZTVALUE= may be specified as follows:

ZTVALUE=address
The given address expression is resolved and the resulting address is compared with the contents of the length field. A match identifies the last entry in the table. (The resolved address expression may not be wider than the length field.)

ZTVALUE=NEGATIVE
This keyword notifies Z/XDC that the length field for the last entry in the table will contain a negative value (i.e. the field's hi-order bit will be on).

ZTVALUE=NOTPOSITIVE
This keyword notifies Z/XDC that the length field for the last entry in the table will contain a zero or negative value.

ZTPTR=address
Either **ZTPTR=** or **ZTVALUE=** or **MAXCLONE=** can be used to provide end-of-table information. **ZTPTR=** and **ZTVALUE=** are mutually exclusive. **(MAXCLONE=** is not.)

ZTPTR= can be used to provide a table's ending address. Z/XDC will recognize the end of the table when it attempts to create an equate or dsect map that is located at or beyond the ending address.

There is no default value for **ZTPTR=.** When **ZTPTR=** is omitted, an ending address check is not performed.

Selected Non-Autocloning Related Operands

All of the remaining operands are not autocloning operands, so they are not described here. (They are described in the parent topic, **HELP COMMANDS EQUATE** or **HELP COMMANDS USING).** Nevertheless, some of them require a comment or two with respect to autocloning.

FLOAT
This causes the generated equates or maps to be assigned floating bases instead of fixed bases. This is a very powerful and useful capability, but you need to be aware that the resolution of floating bases is extremely **CPU intensive** and has to be done

repeatedly during typical Z/XDC processing. Therefore, if the number of equates and maps having floating bases grows "too large", then **Z/XDC's performance can be impacted severely!**

Be aware that when **FLOAT** is specified, the autocloning process will **always** create the number of equates/maps equal to the **MAXCLONE** value. This means that if **MAXCLONE** is too high, Z/XDC's responsiveness will become terrible!

Usually, specifying **MAXCLONE=10** will give you enough floating equates/maps to meet your needs without impacting performance noticeably.

name
For autocloning, **name** gives the root name to be used for the generated equates or dsect maps. Sequence numbers will be appended to this name to form the names of the individual equates or maps.

address
For autocloning, **address** provides the location of the first table or chain entry to be labeled or mapped.

All remaining operands
All operands that can be given with normal **EQUATE** and **USING** commands can also be given with the autocloning forms of these commands. The attributes are assigned to all equates or maps created by the autocloning process.

Examples:

EQ RB 21C%% CFO=1C CFM=00FFFFFF ZCV=21C% MAX=100
This causes a series of equates (named RB001 through RBnnn) to be created to label the current task's request blocks. The RBs will be labeled from newest (RB001) to oldest. (Note, this is the opposite sequence by which the LIST RBS command labels request blocks.) Not more than 100 equates will be created. The last labeled RB will be the one whose link field points back to the TCB.

DMAP XDCMAPS.RB
USING RB 21C%% CFO=1C CFM=00FFFFFF ZCV=21C% MAX=99
This sequence of commands accomplishes the same thing as the preceding example, but uses dsect maps instead of equates. Here are the details:

DMAP XDCMAPS.RB
This loads a copy of a request block's dsect map from Z/XDC's XDCMAPS module. (See **HELP MAPS XDCMAPS.)**

USING RB 21C%% CFO=1C CFM=00FFFFFF ZCV=21C% MAX=99
This command first assigns the RB dsect map to represent the newest request block. It then generates up to 99 copies of the RB map, with names ranging from RB01 to RBnn. It then assigns the first generated map (RB01) to represent the

very same request block to which the RB map was assigned. It then generates additional copies (named RB02 through RBnn) to represent the remaining request blocks queued from the current task. The **ZCVALUE=21C%** operand lets Z/XDC recognize that the last queued request block will have a link field that points back to the TCB.

EQUATE TIOE 21C%+C%+18 LFO=0 LFW=1 MAX=1000
This command creates a series of equates (named TIOE0001 through TIOEnnnn) to label the DD entries in the current task's TIOT (Task I/O Table). Not more than 1000 equates will be created. The last labeled DD entry will be the one whose length field contains X'00'. (Note, LFVALUE=0 did not have to be specified because a zeroed length field is the default end-of-table condition for tables that contain variable length entries and for which the length fields specify the **entire** length of each table entry.)

LIST TASKS
DMAP XDCMAPS.TCB
USING TCB TCB#3 FLOAT
EQUATE RSA .TCBFSA? DATA 48 CFO=8 MAX=100 FLOAT
This series of commands creates a set of 100 floating equates that will map up to 100 standard 72-byte register save areas that are chained from a task's TCBFSA field.

- Because the **FLOAT** operand is given, the full maximum number of equates permitted by the **MAXCLONE=** operand are created. This has the following results:
  - When the chain being mapped has 100 or more elements, the first 100 elements will be labeled by the generated equates.
  - When the chain has fewer than 100 element, the excess labels simply won't resolve.
  - When the length of the chain changes, the labels will automatically resolve or not so that the then-current elements on the chain will always be labeled.

- The **LIST TASKS** command, as a side effect, automatically creates equates named **TCB#1, TCB#2,** [...] to label all of the Task Control Blocks that are displayed by that command. (See **HELP COMMANDS LIST TASKS** for more information.) So the **USING** command takes advantage of that fact to assign the TCB map to the third listed Task Control Block.

- If you wish to map the RSA chain for a different task, simply issue another **USING** command to reassign the TCB map to a different TCB. Example:
  USING TCB TCB#7 FLOAT

- Because the TCB map floats, you can automatically map the RSA chain of the nth task in any other address space simply by issuing a **SET ASID asname** command (security and authority permitting, of course) followed by another **LIST TASKS** command (to rebuilt the TCB#n equates). Example:

```
SET ASID JES2
LIST TASKS
```

DMAP XDCMAPS.TCB

USING TCB 21C%
DMAP .SCB
USING SCB .TCBSTAB% AUTOCLONE FLOAT
This series of commands first loads a dsect map for the TCB and positions it to represent the current task's Task Control Block. It then loads an SCB map (STAE Control Block), and then autoclones it to represent all SCBs that are currently queued from the current task's TCB.

The **USING ... AUTOCLONE** command uses the following defaults, all of which are appropriate for mapping the STAE Control Block queue:

CFOFFSET=0
The SCB's chain field (SCBCHAIN) is located at the start of the control block.

CFWIDTH=4
The width of the SCBCHAIN field is 4 bytes.

CFMASK=7FFFFFFF
The SCBCHAIN field contains 31-bit addresses.

ZCVALUE=0
The end of the SCB queue is signaled when SCBCHAIN is zeroed.

MAXCLONE=100
Since I am creating floating maps, The default **MAXCLONE** value of 100 is a bit high. Z/XDC response time is likely to be affected. So next time, you'll probably want to stipulate a lower number. **MAXCLONE=10** or **=20** might be good choices.

Since I am satisfied with the defaults for **all other** autoclone related operands, I am not using them in my command string. But I still need something to notify Z/XDC that I am intending autocloning. So I have to use this **AUTOCLONE** operand to provide that notification.

Help COmmands USing Examples

Using Command Examples:

U .TCBRBP,21C?
All dsect maps are searched until one containing the field named TCBRBP is found. This map is then based such that TCBRBP refers to the start of the current task's TCB's basic section. (This result is valid only if Z/XDC is not running in Foreign Address Space Mode.)

U TCBFIX,21C?-20
This accomplishes the same result as the previous example. The standard IBM mapping macro for the TCB is actually named TCBFIX and actually maps from the start of the TCB prefix which is X'20' bytes prior to the TCBRBP field.

U TCBFIX+20,21C?
This accomplishes the same result as the preceding examples.

DM TCBCRNT,TCBFIX.TCBRBP

U TCBCRNT,21C?
The DMAP command creates a copy of the TCBFIX map and names it TCBCRNT and defines its zero point to be at the TCBRBP label. (Subsequent references to the name TCBCRNT refer to this zero point.) The USING command sets TCBCRNT's base such that the map's zero point (at TCBRBP) coincides with the start of the current task's TCB's basic section.

U .RBBASIC,R1! F
The first map found to contain the label RBBASIC is based such that the RBBASIC field coincides with whatever address R1 points to at the time that the map is referenced in subsequent Z/XDC commands. In other words, the map "floats" according to the value in R1 and, in this case, according to the program's current addressing mode.

U .RBBASIC,TCBFIX.TCBRBP? F
The first map found to contain the label RBBASIC is based such that the RBBASIC field coincides with whatever address the TCBRBP field of the TCBFIX map points to at the time that the map is referenced in subsequent Z/XDC commands. In other words, the map "floats" according to the values both of TCBFIX's base address and of the contents of the TCBRBP field.

U TCBCRNT,TCBCRNT
The effect of this is to force the TCBCRNT map to the top of the dsect search order without changing either its base address or whether or not it floats.

U ASCB @ASCB F
U ASXB .ASCBASXB F G
The first USING command assigns a floating map to the ASCB pointed to by the built-in equate named "@ASCB". "@ASCB" always points to the ASCB that is correct for the current Target Address Space (see HELP EQUATES BUILTIN for more information), so the ASCB map always floats to the appropriate ASCB whenever the Target Address Space is changed by the SET ASID command. The ASCB map is considered to have a "global" base because the ASCBs that it maps are all located in common storage.

The second USING command assigns a floating map for the ASXB that is correct for the Target Address Space. ASXBs are always located in private storage, so normally Z/XDC would have considered the ASXB map to have a base that was local to the Target Address Space that was set at the time the USING command was issued. However, the above USING commands are structured such that the ASXB map always floats to the correct ASXB every time the SET ASID command is used. Therefore, in this case it is appropriate to override Z/XDC and force it to treat the ASXB map as having a global base.

Help COmmands Verify

The **VERIFY** command is used to verify the contents of:
- Accessible virtual storage. This can include:
  - The private area
  - Common storage
  - Dataspaces
  - Foreign address spaces
  - Real storage
- The error level and retry level general registers.
- The error level and retry level access registers.

- The error level and retry level control registers.
- The floating point registers.
- The error level and retry level PSWs.
- Computed values returned by certain numeric built-in functions.

If the **VERIFY** fails, then an error message is issued and the current command input source is aborted. Accordingly, this command is most useful when placed in Z/XDC command scripts (to be executed by Z/XDC's READ command) to check that the current environment is as expected by the remaining commands in the file. See HELP COMMANDS READ for more information.

Like the **ZAP** command, when the **VERIFY** command is used to verify data that is located in virtual storage, it also sets both the Current Display Pointer and the Next Display Pointer to point to the location being tested. Thus, a subsequent ZAP command need not repeat the targeting address expression used by the VERIFY command. Instead, "ZAP +0..." can be used to refer to the verified location.

Please note, however, that the Current and Next Display Pointers do **not** apply to registers, PSWs, and computed values. Thus the "ZAP +0..." shortcut cannot be used when VERIFY'ing and ZAP'ing registers and PSWs. For more information about the CDP and NDP, see HELP ADDRESSING IMPLICIT.

The syntax for the **VERIFY** command is essentially the same as it is for the ZAP command. There are two forms. One verifies raw string data at the target location. The other resolves a given address expression to a 24-bit, 31-bit, or 64-bit value and verifies that against the target location.

General Syntax:

```
VERIFY target address-data
            ,address-data
            =string,string,...
```

target

This specifies the location (or number) whose contents (or value) is to be verified. See HELP COMMANDS VERIFY TARGETS for specific information.

address-data
,address-data

This is an address expression whose value is to be resolved and verified against the target. See HELP COMMANDS VERIFY ADDRESS for specific information.

=string,string,...

This is string data (hexadecimal, decimal, and/or character) that is to be verified against the target. See HELP COMMANDS VERIFY STRING for specific information.

Note, for string data, the delimiter must be an equals sign (=). For address data, the delimiter must be a comma (,) or blank ( ).

The following specific syntax information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

TARGETS - Locations and values that can be verified.
ADDRESS - Converting an address expression into hex data and comparing it to a target.
STRING - Comparing a hexadecimal string, a character string, or a decimal number to a target.

The following differences exist between the **VERIFY** command and the **ZAP** command:

- The **ZAP** command can change the contents of storage, the registers, and the PSW. The VERIFY command cannot.

- The **VERIFY** command can reference the error level registers and PSW. The ZAP command cannot. (ZAP can reference only the retry level registers and PSW.)

- The **VERIFY** command can test against computed numeric values (from certain built-in functions). The ZAP command cannot.

- The **VERIFY** command can compare any part of a PSW against both string data and address data. The ZAP command can zap only the instruction address portion of only the retry level PSW and only with address data. (Use the SET PSW command to alter other fields in the retry level PSW.)

- The **ZAP** command can perform AND, OR, and XOR logical operations when zapping data; the VERIFY command cannot.

For more information, see HELP COMMANDS ZAP.

Help COmmands Verify Targets

The **VERIFY** command can be used to verify:
- Accessible virtual storage. This can include:
  - The private area
  - Common storage
  - Dataspaces
  - Foreign address spaces
  - Real storage
- The error level and retry level general registers.
- The error level and retry level access registers.
- The error level and retry level control registers.
- The floating point registers.
- The error level and retry level PSWs.
- Computed values returned by certain numeric built-in functions.

Syntax:

```
VERIFY target address-data
             ,address-data
             =string,string,...
```

target
This specifies the location (or number) whose contents (or value) is to be verified.

Target may be any of the following:

addressexpression
This is an address expression that resolves to some accessible virtual storage location. The location may be in the private area, in common storage, in a dataspace, in a foreign address space, or in real storage (security permitting, of course). The length of accessible storage (starting at this location) must be as least as long as (or longer than) the data to be verified. For more information, see HELP ADDRESSING.

register-reference
This must be the name of a retry level or error level register (possibly qualified by a starting offset). For string data verifies, the register may be:
- A general register (Rn or ERn etc.)
- An access register (ARn or EARn)
- A control register (CRn or ECRn etc.)
- A floating point register (FRn)

The register name may be followed by a simple numeric offset for selecting the first byte to be verified. Example:

VERIFY R5+1=34BC
This verifies that the middle two bytes of general register R5 contains the value X'34BC'.

For address data verifies, the target register may only be a general register (Rn).

PSW-reference
This must be either "EPSW" or "PSW" (representing the error level or retry level instance, respectively, of the user program's PSW.) For string data verifies, the PSW name may be followed by a simple numeric offset for selecting the first byte to be verified. Example:

VERIFY EPSW+1=8D
This verifies that the second byte of the error level PSW contains the value X'8D' (i.e. is a typical problem mode PSW).

For address data verifies, Z/XDC presumes that the user's intent is to compare the address data to the instruction address field of the PSW (excluding the AMODE flag). Consequently, it is neither required nor permitted to use an offset value for address data verifies. Example:

VERIFY PSW,10?+50
This verifies that the retry level PSW points to the location represented by "10?+50", in other words that the next instruction to be executed by the user program will be the "SVC 3" instruction located at the CVTEXIT field of z/OS's CVT.

computed numeric value
Z/XDC permits the following numeric built-in functions to be used for generating a numeric value to be verified.

XADDR(...) - (Alias: XADR) Returns the numeric value of an address expression (as opposed to the contents of the storage pointed to by that expression). The returned value is 4 bytes wide.

XALET(...) - Returns the numeric value of an ALET. The returned value is 4

bytes wide.

XASID(...) - (Alias: XASN) Returns the numeric value of an ASID. The returned value is 2 bytes wide.

For detailed information about these and other built-in functions, see HELP FUNCTIONS.

address-data
,address-data
=string,string,...

These operands provide the address data or string data to be tested for. They are described in detail by subsequent topics. Type HELP *NEXT for more information.

Examples:

VERIFY R5?+1=C'ABC'
This verifies that the storage at one byte past the location pointed to by general register R5 contains X'C1C2C3'.

VERIFY R5+1=C'ABC'
This verifies that general register R5 contains X'C1C2C3' starting in its 2nd byte.

VERIFY PSW,CVT.CVTEXIT
This verifies that the retry level PSW points to the CVTEXIT field of z/OS's Communications Vector Table.

VERIFY PSW?=0A03
This verifies that the retry level PSW points to a location that contains an SVC 3 instruction.

VERIFY EPSW+1=0C
This verifies that the error level PSW is in supervisor state and key 0.

VERIFY XASID(PASID)=0001
This verifies that the user program's retry level Primary Address Space is the Master Scheduler's address space.

VERIFY R9,.BUFFER2
VERIFY XADDR(R9?),.BUFFER2
Both of these commands verify that general register R9 contain the address of a location labeled "BUFFER2".

Help COmmands Verify Address

The address-data form of the **ZAP** and **VERIFY** commands resolves a given address expression into a 24-bit, 31-bit, 32-bit or 64-bit value and verifies or stores that value at a target location. The target must be 4 or 8 bytes wide. The value is verified or stored at the low-order 24, 31, 32 or 64 bits of the target as follows:

24-bit value: - The value is stored into the low-order 3 bytes of a 4-byte wide target. The high-order byte is not altered.
31-bit value: - The value is stored into the low-order 31 bits of a 4-byte wide target. The high-order bit is not altered.
32-bit value: - The value is stored into all 32 bits of a 4-byte wide target.
64-bit value: - The value is stored into all 64 bits of an 8-byte wide target.

In all cases, the target location need not be aligned.

Syntax:

```
VERIFY target address-data
ZAP          ,address-data
```

target

This identifies the target at which the data is to be verified or stored. It may be a storage location, a register reference, or a PSW reference. For the VERIFY command, it may also be any of certain numeric built-in functions. This operand is fully described by the preceding topic. See HELP *PREVIOUS for more information.

, (comma)
(blank)

The comma or blank delimiter distinguishes this "address data" form of the ZAP and VERIFY commands from the "string data" form (which uses an "=" instead). The comma or blank indicates that the following operand is an address expression, not a hex or character string.

address-data

This is an arbitrary virtual address expression. It is resolved to a 24-bit, 31-bit, 32-bit or 64-bit value before being verified or stored. The width of the value is derived in either of two ways. If the address-data contains a WIDTH(...) built-in function call, then the final width of the address data is derived from the value specified by the function. On the other hand, if **no** WIDTH(...) function call occurs within the address-data, then the final width of the address data is derived from the AMODE setting in either the retry level or error level PSW. More specifically:

- **When WIDTH(...) is present:** The resulting address data's final width will be set as follows:

WIDTH FUNCTION VALUE	FINAL BIT WIDTH	FINAL BYTE WIDTH	COMMENT
1-24	24	4	- The address data is zapped into the 24 bits following the target's first 8 bits. The target's first 8 bits are **preserved.**

25-31	31	4	- The address data is zapped into the 31 bits following the target's first bit. (The target's first bit is **preserved.)**
32	32	4	- The address data is zapped into all 32 bits at the target address.
33-64	64	8	- The address data is zapped into all 64 bits at the target address.

- **When WIDTH(...) is absent:** Then the width of the value will depend upon a PSW's current addressing mode (which can be displayed by the LIST AMODE command and changed by the SET PSW command).
  - If the current AMODE is 24-bit but the resolved address is greater than 16 megabytes, then the high-order bits of the address are truncated.
  - Similarly, if the current AMODE is 31-bit but the resolved address is greater than 2 gigabytes, then the high-order bytes of the address are truncated.

    In other words, the size of the address verified or stored depends upon the address mode as follows (when WIDTH(...) is absent):

    - **AMODE(24):** The address is resolved to a 3-byte value, and is then verified against or stored into the lo-order 3 bytes of the 4 bytes pointed to by the target expression. The hi-order byte is neither checked nor altered.

    - **AMODE(31):** The address is resolved to a 31-bit value, and is then verified against or stored into the lo-order 31 bits of the 4 bytes pointed to by the target expression. The hi-order bit is neither checked nor altered.

    - **AMODE(64):** The address is resolved to an 8-byte value, and is then verified against or stored into the 8 bytes pointed to by the target expression.

Examples:

Z R5,C
The address expression "C" is resolved to a 24-bit, 31-bit, or 64-bit value (by prefixing high-order zeros) and then stored into the low-order portion of retry level general register R5.
- In AMODE(24), R5's hi-order byte is not altered.
- In AMODE(31), R5's hi-order bit is not altered.
- In AMODE(64), all 32 bits of R5 are set. (RH5 is not altered.)

If you want to set just the hi-byte of R5 to X'0C', then try "Z R5=0C" (i.e. the string data form of the ZAP command).

Z R5,C~WIDTH(32)
In this case, the address "C" is padded out to 32 bits and stored into all 32 bits of register R5. (RH5 is not altered.)

Z RW5,C

When the target is a wide register, then the address data is resolved to 64 bits regardless of all other considerations. So in this case, RW5 is set to X'000000000000000C'.

LOAD CDAEED CEE.SCEERUN2
Z R5,CDAEED
Note that **CDAEED** could be either a hex address or a module name. In this case, it happens to be the name of an actual LE module. So when **CDAEED** is present in storage, the address expression **CDAEED** does not resolve to location X'00CDAEED'. Instead, it resolves to the location of the CDAEEE load module, and that is the address that is stored into R5.

To avoid ambiguity, try either of the following:

Z R5,0CDAEED
This zaps the address 00CDAEED into the lo-order 24 or 31 bits of R5.

Z R5,'CDAEED'
Z R5,CDAEED.
Either of these zaps the address of the CDAEED load module into the lo-order 24 or 31 bits of R5.

V R9,10%;Z R10,21C%
Here's how to do **a conditional zap.** The VERIFY command first checks to see if general register R9 points to the CVT. If so, then R10 is zapped to point to the current TCB. On the other hand, if R9 does not point to the CVT, then the VERIFY fails, and so Z/XDC aborts the command string, so the ZAP then is not executed.

Z 10R,1F9234CB+ADC6
The value of the address expression is computed: X'1F92E291'. Then:
- If the current AMODE is 24, then the value is truncated to 24 bits (X'92E291') and stored into the low-order 3 bytes of R10. The hi-order 8 bits of R10 remain unchanged.
- If the current AMODE is 31, then the value is stored into the low-order 31 bits of retry level general register R10. The hi-order bit of R10 remains unchanged.
- If the current AMODE is 64 (z/OS only), then the value is first padded out to 64 bits (X'000000001F92E291'). Then it is truncated back down to 32 bits (the width of R10) and store into all 32 bits of R10.

Z RW10,1F9234CB+ADC6
(z/OS only) The value of the address expression is computed, to 64 bits (X'000000001F92E291'). Then it is stored into all 8 bytes of RW10. Thus, RH10 (the high half of RW10) is set to X'00000000', and R10 (the low half or RW10) is set to X'1F92E291'.

Z PSW,PSW?-4
This backs up the retry level PSW by 4 bytes. The address expression "PSW?-4" is

resolved to a 24-bit or 31-bit value and "stored" into the instruction address field of the retry level PSW. The addressing mode flag is not altered.

Z .MYTCBP,21C?
If the current AMODE (in the retry level PSW) is 24-bit, then the address of the current TCB is stored into the low-order three bytes of the 4 bytes starting at the address referenced by the label MYTCBP. The hi-order 1 byte remains unchanged.

On the other hand, if the current AMODE is 31-bit, then the address of the current TCB is stored into the low-order 31 bits of the target address (MYTCBP), and the target's hi-order bit remains unchanged.

If AMODE is 64 bits, then the current TCB's current address is padded out to 64 bits and stored into the **eight** bytes starting at MYTCBP. (Is this what you would want to happen?)

Z .MYTCBP,21C?~WIDTH(32)
The address of the current TCB is padded out to 32 bits and stored at MYTCBP. All 32 bits are stored. No bits in the target are preserved. The width of the stored address is set to 32 bits **regardless** of the current addressing mode.

S PSW 31
Z R2,XCSA
D
The SET PSW command sets the user program's addressing mode to 31-bit. The ZAP command then stores the starting address of the extended CSA into R2. **(XCSA** is a built-in equate that labels the start of extended CSA storage. See HELP EQUATES BUILTIN for more information.)

The ZAP command causes both the **Current Display Pointer** and the **Next Display Pointer** to be set to point to MYTCBP. The **DISPLAY** command, in this case, displays the storage pointed to by the Next Display Pointer. In other words, the zapped storage is immediately redisplayed.

Help COmmands Verify String

The string form of the **VERIFY** and **ZAP** commands verifies or stores string data at the target location.

Syntax:

```
VERIFY target=string,string,...
ZAP          =&string,string,...
             =|string,string,...
             =#string,string,...
```

target
This identifies the target at which the data is to be verified or stored. It may be any of the following:

addressexpression
This is an address expression that resolves to some accessible virtual storage location. The location may be in the private area, in common storage, in a dataspace, in a foreign address space, or in real storage (security permitting, of course). For more information, see HELP ADDRESSING.

register+offset
This refers to a particular byte within either a retry level general register, a retry level access register, or a floating point register. For the VERIFY command, this may also be any of the error level registers: general, access, or control.

The offset must be in the range of 0 to 3 for 4-byte wide registers and 0 to 7 for 8-byte wide registers. It identifies a specific starting byte **within** the register.

PSW+offset
EPSW+offset
This is permitted only for the VERIFY command. The ZAP command may only use the address data (not string data) form of the command. See HELP *UP ADDRESS for more information.

"PSW+offset" refers to a particular byte within the retry level PSW. "EPSW+offset" refers to a particular byte within the error level PSW. The offset must be in the range of 0 to 7. It identifies a specific starting byte **within** the PSW (or EPSW).

=
The equals sign distinguishes this "string data" form of the ZAP or VERIFY command from the "address data" form (which uses a blank or a comma as the delimiter). The equals sign indicates that the following operand is string data. See HELP *UP ADDRESS for information about address data.

=&
=|
=#
These operators are valid only with the ZAP command. (They cannot be used with the VERIFY command.) They indicate that the following string data is to be combined into the target locating using a boolean operation instead of a simple store. The supported operations are AND (=&), OR (=|), and XOR (=#). See HELP COMMANDS ZAP BOOLEAN for more information.

string,string,...
This gives a string of data to be verified against or stored into the target location. The string is converted to internal form and either verified or stored using as many bytes as are necessary to hold the value; no padding is performed.

As many strings may be given as desired. When multiple strings are given, they are each converted to binary according to their types and then concatenated together before being used for verify or zap purposes. See "Examples" below for a usage example.

The strings may be given as either hexadecimal digits, EBCDIC characters, or as

decimal integer numbers. **Brief examples:**

```
    ZAP ...=F0D93B72      a hex string
    VER ...='DBCOLE'      a character string
    FIND C4'BCOL'C5,...   a mixture
    ZAP ...=F'-1234557'   a 4-byte wide integer
    VER ...=H'32767'      a 2-byte wide integer
    ZAP ...=1'139'        a 1-byte wide integer
```

For more complete information, See "HELP COMMANDS SYNTAX STRINGDATA".

Examples:

V ER5=153D
The value X'153D' is verified against the 1st and 2nd bytes of error level general register R5. The 3rd and 4th bytes are not checked.

Z 3R+2=01
The value X'01' is stored into the 3rd byte of retry level general register R3. The 1st, 2nd, and 4th bytes are unaltered.

Z R7+1=H'1024'
The value X'0400' is stored into the 2nd and 3rd bytes of retry level general register R7. The 1st and 4th bytes are unaltered.

Z FR6=F3,7539
The value X'F37539' is stored into the first three bytes of floating point register FR6. The remaining five bytes are unaltered.

ZAP PSW?=12FF,078E,00DEAD,1'6','ERROR!'
This example illustrates the use of multiple strings of various types in one zap. This zap stores data that is equivalent to the following Assembler code:

```
    LTR   R15,R15
    BZR   R14
    DC    X'00DEAD',AL1(6)
    DC    C'ERROR!'
```

Note, the string "1'6'" means "convert the decimal number "6" to binary and store it into a 1-byte wide field.

F PSW?
V +4=47F0CBDE;Z +0=47000000;F -4;DOWN M
The first FORMAT command displays storage and sets the "Current Display Pointer" to point to the storage just displayed.

The VERIFY command ensures that the current contents of 4 bytes past the displayed location are X'47F0CBDE'. If the verification fails, then the VERIFY command aborts with an error message and the following ZAP command is not executed. If the

verification succeeds, then the VERIFY command sets the "Next Display Pointer" (as well as the "Current Display Pointer") to point to the verified location.

The ZAP command stores the value X'47000000' (a NOP) at 4 bytes past the previously displayed location and resets the "Next Display Pointer" (as well as the "Current Display Pointer") to point to the zapped location.

The second FORMAT command redisplays the same location that was displayed by the first FORMAT command. Note that "-4" has to be specified here in order to get back to that location because of the effects of the intervening VERIFY and ZAP commands.

Finally, the DOWN M command brings the redisplayed storage into view. Without it, the display is left showing the results of the first FORMAT command, and you would then have to scroll down to see the results of the remaining commands in this string.

Z PSW?=0700,0700;F
Here, the value X'07000700' (two NOPRs) is stored and then displayed. (The original contents of the target location are not verified or displayed.) Note that in this case the FORMAT command is given without operands. This causes it to reference the Next Display Pointer instead of the Current Display Pointer. But since the ZAP command has set both the NDP and CDP to point to the same location, in this case "F" and "F +0" are equivalent.

V PSW+1=8D;Z PSW,R5?
The VERIFY command ensures that the retry level PSW is set to problem state and execution key 8. If so, then the ZAP command changes the PSW's next instruction address to point to the location pointed to by retry level general register R5. Note that the VERIFY command (in this example) uses string data while the ZAP command uses address data.

Z .DATA='C'D6D3'E'
This stores the character string "COLE" at the location labeled DATA. Notice that the string is given as a mixture of EBCDIC characters and hexadecimal digits.

Help COmmands WAit

The **WAIT** command can be used to resolve a conflict over use of the terminal between Z/XDC and a user or system task. For more information, see HELP MULTITASK CONVERSATIONMANAGEMENT.

In the event of a terminal control conflict, you should type the WAIT command repeatedly until it is received by the Z/XDC task. Z/XDC will then immediately cease trying to use the terminal. Instead, it will resume waiting in Conversation Management for control to be passed to it again. You then should be able to use the **PA1** or **PA2** key to restore the user task's display without further interference from the Z/XDC task.

Syntax:

WAIT

This command accepts no operands.

Help COmmands WHEre

The **WHERE** command generates a formatted display of storage showing the current **retry** level PSW's resume execution location.

The **EWHERE** command generates a formatted display of storage showing the current **error** level PSW's **(EPSW's)** ABEND location.

Display Positioning

The **WHERE** and **EWHERE** commands will always produce a display that shows the **PSW's** or **EPSW's** execution location; however, that location generally will **not** be at the top of the display. Instead, it will most often be embedded several lines down into the display.

But the display line showing the execution location will be both flagged and hilighted, so it will be easy to spot.

When **SET TRACE ROLL** is in effect, the **TRACE** command internally uses the **WHERE** command to produce its displays such that the resume address (which, of course, advances during tracing) has the appearance of rolling down the display screen.

The **WHERE** and **EWHERE** commands accept an address expression for an operand. This can be used to change the start of the generated display relative to the **[E]PSW's** execution address.

But only addresses which result in a display that **includes** that resume address are valid. If this rule is violated, then the given address expression is silently **ignored,** so that Z/XDC can still generate a display that will contain the appropriate execution address.

The CDP and NDP

Both the **WHERE** and **EWHERE** commands set the **CDP (Current Display Pointer)** to point to the **[E]PSW's** execution address (as flagged and hilighted in the **[E]WHERE** command's display). Note, **unlike** all other storage displaying commands, this CDP setting usually is **not** at the start of the display.

Personally, I hate this exception, but I had to create it because it

simplifies references by subsequent commands to the **[E]PSW's** resume address.

Example: Suppose a WHERE command display is up, and somewhere down the middle of the display it shows that the next instruction to be executed is a **BAS** to subroutine. you might want to issue **T +4;GOT** to allow that subroutine to run without your having to step through it. The fact that the **CDP** points to the BAS in this example (and not to the start of the display) is what makes the **T +4** command possible.

The **NDP** (the **Next Display Pointer)** is more normal. It is set to point to the **end** of the display.

WHERE and EWHERE Display Similarities and Differences

Generally, the **WHERE** and **EWHERE** commands will produce similar displays **except** when the retry level and error level environments are **different.** Then the **WHERE** command will show the **retry level resume address,** while the **EWHERE** command will show the **error level ABEND address.** For more information, see HELP EXECUTIONLEVELS.

When the Retry level and Error Level execution environments are the same, the **WHERE** and **EWHERE** commands may still show resume addresses that **differ** (usually by two bytes). This difference arises from the fact that for most program checks, the hardware leaves the PSW pointing **past** the failing instruction. This is what will be retained in the Error level PSW and what will be shown by the **EWHERE** command.

On the other hand, for the Retry Level PSW, Z/XDC will adjust the resume instruction address to point to the **start** of the failing instruction, hence the difference. For more information, see HELP EXECUTIONLEVELS.

When the Error and Retry Levels are the Same

Notice: Even when the error and retry environments are the same, the **WHERE** and **EWHERE** commands will usually still produce differing displays when Z/XDC has received control due to a program check. This is because for most program checks, the hardware will leave the PSW pointing **past** the failing instruction, not at it. This has the following effects:

- The **error level** PSW (represented within Z/XDC as **EPSW** or **EPSWE)** will be set to the actual PSW, exactly as the hardware left it at program check time or ABEND time.

- The **retry level** PSW, on the other hand, will be adjusted by Z/XDC to point to the **start** of the failing instruction (not its end).

Examples: [when the error level and retry level environments are the same]

- **WHERE** produces a display guaranteed to include your program's resume address.

- While **EWHERE** does not. Instead, the display will be guaranteed to include the program check or ABEND location, not the resume address.

- **WHERE PSWE?** will produce a display that **starts** at the resume address.

- To produce the same display with the **EWHERE** command, you would have to use

either of the following:
- **EWHERE EPSWE?-2** [usually]
- **EWHERE PSWE?**

(Note, the presumptions here are:
- That the error level and retry level environments are the same,
- And Z/XDC received control as a result of a program check in which the failing instruction was 2 bytes long,
- And the hardware left the error time PSW pointing past that instruction.
- Note that the hardware perceives all Z/XDC breakpoints as being 2-byte wide machine instructions. Thus, whenever Z/XDC receives control due to a breakpoint, the EPSW will always be set to 2 bytes past the start of the machine instruction even when that instruction is a 4-byte or 6-byte instruction.)

When Z/XDC Receives Control from a Breakpoint

Breakpoints can be constructed either using an illegal opcode (specifically, **X'00'**) or TRAP2 instructions **(X'01FF'):**

- In the former case, a code 0001 program check will occur which eventually will cause Z/XDC to received control as an ABEND Recovery routine. In this case the error and retry levels **may be the same** or they **may be different** (which may be OK, but usually is not).

- In the latter case, a Trap Exception will occur which causes Z/XDC to receive control as a Trap Handler routine. In this case, the error and retry level environments **will always be the same.**

For more information, see HELP BREAKPOINTS TYPES.

For both types of breakpoints, when the error and retry levels are the **same,** there will **always be a 2-byte difference** between the locations pointed to by **PSWE** and **EPSWE.** This will result in the **EWHERE** issues discussed above in the preceding section.

The 2-byte difference occurs **regardless** of the length of the instruction upon which the breakpoint is placed. This is because the hardware doesn't see that instruction. Instead, it sees:
- Either the illegal opcode **X'00'** which the hardware regards as a 2-byte wide instruction,
- Or a **TRAP2** instruction **(X'01FF')** which, in fact, is a 2-byte wide instruction.

Local Overrides of SET FORMAT Options

The **[E]WHERE** commands accept several keyword operands that allow you to do one-time overrides of the various format control settings defined by the **SET FORMAT** command.

Syntax:

[E]WHERE addressexpression lines SOURCE ADDRESSES DECIMAL ...

```
EW or W  omitted          LINES=lines     OBJECT  OFFSETS    HEXADECIMAL
         ,                BYTES=hexnumber BOTH

     ... ASCII   WIDE   INSTRUCTION SHOWMCODE   POINTERS
         EBCDIC  NARROW DATA        HIDEMCODE   NOPOINTERS
                        NOBIAS      CURRENTMCODE
```

Shortcuts: **E**
W

Notes:

Operands appearing above within columns are mutually exclusive.

All operands are optional.

If the addressexpression operand is given, then it must be first. All other operands may be given in any order.

If the addressexpression operand is omitted but other operands are given, then a **comma** must be used to indicate the omission.

Operands:

addressexpression
This operand is optional. If given, then it must resolve to a location that is **at or preceding** the execution address for either the **PSW** (for the **WHERE** command) or the **EPSW** (for the **EWHERE** command). **But** the address must still be close enough to the **[E]PSW's** execution address so that the resulting display will include that address. (Note, the size of the display usually is determined by the size of the window from which the **[E]WHERE** command was issued.) If the given address expression fails to follow these rules, then it is silently ignored, and the **[E]WHERE** command is processed as if the address expression had been omitted.

omitted
If the **addressexpression** is omitted, then the **[E]WHERE** command chooses a starting display address automatically.

When **SET TRACE ROLL** is in effect, then when possible, Z/XDC will choose the same address as was used by the last preceding [E]WHERE display, but if execution tracing has resulting in the [E]PSW moving beyond the range of the display, then the command will instead choose an address that repositions the [E]PSW to the top of the display.

,
If the **addressexpression** is omitted and other operands are given, then a **comma** must be used to show the omission.

Format Controls

The following operands may be given in any order. They must, however, follow the given address expression (if any).

Also, all of the following operands (except **lines, LINES=** and **BYTES=)** serve to override the corresponding default values established via the **SET FORMAT** command.

lines
LINES=lines
BYTES=hexnumber
These control the size of the display. The **lines** and **LINES=** operands do so in terms of the number of lines (rows) of instructions, data and comments the display is to contain. The **BYTES=** operand does so in terms of the amount of storage that is to be displayed:

lines
LINES=lines
These two operands function identically; however the **LINES=lines** form is preferred because eventually support for the **lines** form may be discontinued.

The **lines** value must be a decimal number specifying the number of display lines to generate containing instructions, data and comments (i.e. not including the Location Interpretation header lines). This value must be in the range of 0 to 65,535 (64K-1). When omitted, either the default value set by the **SET LINES** command is used (for nonfullscreen displays), or the length of the current window is used (for fullscreen mode displays).

The **bytes** value must be a number giving the amount of storage to be displayed. It can be either of the following:

- A hexadecimal number (Example: **BYTES=C8)**
- A decimal number followed by the letter **N** (Example: **BYTES=200N)**
- A scaled decimal number (Example: **4K)**

Regardless of the number's form, it must resolve to a value between **0** to **1FFFE0:**

- The very strange X'1FFFE0' limit is (64K-1)*32:
  - 64K-1 is the LINES= limit.
  - 32 is because one line can display up to 32 bytes of storage.

The number of display lines needed to produce a BYTES= driven display can vary.

The **lines, LINES=** and **BYTES=** operands are mutually exclusive.

SOURCE
OBJECT
BOTH
These operands control the use of maps that might be available for controlling the formatting of the storage being displayed:

SOURCE

Causes storage to be formatted under the control of a source level map, if available.

OBJECT
Causes storage to be formatted by disassembly. Any labels generated by equates, dsect maps, or csect maps will also be displayed where applicable. Information from source level maps will **not** be displayed.

BOTH
Causes storage to be formatted by both SOURCE and OBJECT; source level statements will be interspersed with statements generated by disassembly.

Note, when **DATA** is explicitly given, both **SOURCE** and **BOTH** are ignored in favor of **OBJECT.**

POINTERS
NOPOINTERS
When storage is being formatted via disassembly (i.e. when either **OBJECT** or **BOTH** is in effect), if a line of display is showing a snippet of data that is 3, 4 or 8 bytes wide, then this **POINTERS** operand will cause Z/XDC to attempt to resolve that snippet as a storage location. Then it will display that resolution as a comment on the display line. Notes:

- The **NOPOINTERS** operand prevents these resolutions.
- The factory default is **POINTERS.**
- When very complex structures of **floating maps and equates** are present, the **POINTERS** setting may result in slower displays. In that case, you may wish to use **NOPOINTERS** instead. (For more information about floating maps and equates, see **HELP MAPS FLOATING.)**

SHOWMCODE
HIDEMCODE
CURRENTMCODE
These operands control whether or not source image displays formatted via **ADATA** maps will include source code create by certain macro expansions:

- **SHOWMCODE:** The expansions of all macros are shown.
- **HIDEMCODE:** The expansions of all code generating macros are suppressed. (Data generating macros, such as control block mapping macros, are unaffected.)
- **CURRENTMCODE:** The expansions of all code generating macros are suppressed **except** if the macro expansion currently being displayed includes the error level or retry level PSW address. In this case, the expansion is shown. (Data generating macros, such as control block mapping macros, are unaffected. They are never suppressed.)

Note the following:

- The intent of the **CURRENTMCODE** operand is to improve the display of code written using structured programming macros.

- Also, the hiding of macro expansions is effective only for storage formatted via **ADATA** maps. Storage formatted via SYM data maps is unaffected.

- Finally, the hiding of macro expansions does not occur for dsect maps that map

data areas and control blocks.

For detailed information about what happens when macro expansions are suppressed, see HELP MAPS ADATA MACROS.

ADDRESSES
OFFSETS
These operands control whether the far left column of the display (the "address" column) will show storage addresses or offsets:

ADDRESSES
Causes each line of data displayed to be prefixed by that data's virtual address.

OFFSETS
Causes each line of data displayed to be prefixed by that data's offset from the start of an appropriate including object (load module, csect, dsect, or equate).

DECIMAL
HEXADECIMAL
For disassembled machine instruction displays, these operands control whether the displacement fields are formatted as decimal or hexadecimal numbers:

DECIMAL
Causes the displacement fields to be displayed as decimal numbers.

HEXADECIMAL
Causes the displacement fields to be displayed as hexadecimal numbers.

ASCII
EBCDIC
For data displays, these operands control whether the text portions of data displays are to be interpreted using the **ASCII** or **EBCDIC** character set:

ASCII
Causes data displays to show the **ASCII** interpretation of the displayed storage. To visually indicate this, the text portion of each display line will be framed with **vertical bars (|)** instead of asterisks.

EBCDIC
Causes data displays to show the **EBCDIC** interpretation of the displayed storage. To visually indicate this, the text portion of each display line will be framed with **asterisks (*).**

More information on EBCDIC and ASCII can be found at **HELP CHARACTERSETS**

WIDE
NARROW
These operands control whether data displays are to be wide or narrow:

WIDE
This causes data displays to show up to **32 bytes** of storage per line (not just 16). This option works best when your terminal's display is at least 136 columns wide. For more information, see HELP FULLSCREEN TERMINALS.

NARROW
This causes data displays to show only **16 bytes** of storage per line. This width is appropriate for terminals with 80 character wide display lines.

INSTRUCTIONS
DATA
NOBIAS
The **WHERE** and **EWHERE** commands generally attempt to format storage contents as instructions, but this bias can be influenced by many factors, such as whether or not a PSW **(PSW** or **EPSW)** points to the storage being formatted, as well as the attributes of maps, fields and equates that label the storage being displayed. As a part of the management of this process, the **[E]WHERE** commands maintain a concept called the **formatting bias** which, for a given displayed line, retains information about how the preceding line was formatted and, therefore, contributes information about how the current displayed line should be formatted. In other words, if a display line was formatted as data, then the **[E]WHERE** commands will be biased towards formatting the following display line also as data, unless new information (such as label attributes) suggests otherwise.

The **INSTRUCTION, DATA,** and **NOBIAS** operands set the initial formatting bias for the **[E]WHERE** commands as follows:

INSTRUCTION
The **[E]WHERE** commands' initial formatting bias will be to interpret storage contents as being machine instructions. This will be done even if other factors might suggest that the storage contents should be interpreted as data. This will **not** be done, however, if the initial opcode is invalid.

DATA
The commands' initial formatting bias will be to interpret storage contents as being data. This will be done even if other factors might suggest that the storage contents should be interpreted as machine instructions. Note, when **DATA** is explicitly given, both **SOURCE** and **BOTH** are ignored in favor of **OBJECT.**

NOBIAS
The commands' initial formatting bias will not be forced. Absent other factors, the storage contents will initially be interpreted as machine instructions, but if other factors suggest otherwise, then those factors won't be ignored.

These **INSTRUCTION, DATA,** and **NOBIAS** operands set only the **initial** formatting bias. Once the display gets going, the formatting bias for the second and subsequent display lines will be influenced only by the normal factors discussed above.

Storage Protect Key Display

The **[E]WHERE** commands show the storage protection key as a single hexadecimal digit

appearing in the display's header line, just to the right of the address field. Following the storage protect key may be additional characters:

f - The 'f' character indicates that the storage is fetch protected.
s - The 's' character indicates that the storage is store protected. (there are additional comments about store-protected storage below).
r - The 'r' character indicates that the storage may be redacted. (this character can only be seen when processing a redacted dump in dump/XDC). A redacted area is indicated by the storage being all binary zero. Note that it is impossible to distinguish storage that has a value of binary 0 from storage that has been redacted.

The "store protected" indicator, **s,** is displayed whenever Z/XDC detects that the storage being displayed has one or another of the following special protections:

(Note than when processing a dump (using dump/XDC), many store-protection areas are not indicated. This is because in many cases the store-protection flag is in the page table and that information is not available to dump/XDC).

- **Low Address Protection** (LAP): This is a special kind of protection that the Operating System uses to specifically protect the first 512 bytes of both real and virtual storage. It prevents the storage from being altered by any program, regardless of that program's state or key. (LAP does not apply to data space storage.)

- **Page Protection:** This is a kind of protection that applies to the following areas of storage:
  - The Pageable Link Pack Area (PLPA)
  - The read-only areas of the System Nucleus (IEASYS0x)
  - Any area of storage that has been set to "read-only" by the PGSER or IARVSERV macros.

  When a page of storage has been marked as being "read-only", that storage cannot be directly altered by any program, regardless of that program's state or key. (Note, "Page Protection" does not apply to real storage. Thus, it cannot protect a virtual storage page when that page is accessed via its real address.)

- **Access List Entry Protection:** When an address space or a data space is accessed via an ALET that "points to" an Access List Entry (ALE) having its "fetch only" flag on, then accesses via that ALE are store protected. Note, accesses to that same space via a different ALE might not be store protected.

Examples:

W
A formatted display is generated that shows the current retry level PSW's resume execution location somewhere within the display. The display will be either wide or narrow according to the current SET FORMAT command setting. Note, if you then start (or resume) using Z/XDC's TRACE command to step through program execution, Z/XDC attempts to continue using this positioning for as long as it can for the displays generated by the TRACE commands that you issue.

W ,WIDE
This causes a wide display to be generated without specifying a new display address. Note, the WIDE/NARROW operand won't have any effect on storage being formatted as

machine instructions. It primarily affects displays that include large data fields.

EW EPSW?
A formatted display is generated that shows the current error level PSW's ABEND location at the top of the display.

F EPSW?
Like the preceding example, a formatted display is generated that shows the current error level PSW's ABEND location at the top of the display. **However,** unlike the preceding example, the **EWHERE** command's display position information is **not** changed. Thus, a subsequent **EW** command, without operands, probably would not show the execution location at the top of the display.

W PSW?+10
This is not valid. The given address expression resolves to a location that **follows** the resume PSW's address, and so the display that would be generated cannot possibly include the resume address. Accordingly, Z/XDC ignores the address expression and proceeds to process the **WHERE** command as if no address expression had been given.

W
W +0
The first **WHERE** command generates a display that includes the retry level PSW address somewhere within that display. It also sets the CDP (Current Display Pointer) to point to that retry level address.

The second **WHERE** command references the CDP ("+0") to reposition the display so that the retry level PSW address is shifted to the top of the display. In other words, this sequence of two commands has basically the same effect as the "W PSW?" command.

W ,OFFSETS SOURCE
This generates a display that (a) shows offsets instead of virtual addresses and (b) formats the display using a source level map instead of disassembly (assuming that a source level map has been loaded and positioned to include the retry level PSW address).

Other commands that can be used to display storage are:

- **DISPLAY** - Displays a contiguous chunk of storage in a hex-text (dump like) format.
- **FORMAT** - Formats a contiguous chunk of storage as either machine instructions or data fields, as appropriate.
- **SHOW** - Accepts a list of address expressions and formats a 1-line display for

each expression given.

FIND - Scans storage for a given string of data.

Help COmmands WHItespace

A **whitespace** command causes Z/XDC to display a single blank line. Z/XDC recognizes a whitespace command when a command string either starts with a semicolon (;) or contains an embedded pair of semicolons (;;).

A **whitespace** command is most useful for inserting a line of "white space" between the outputs of two other report producing commands. For some people, such white space helps to make the overall display more readable. Example:

```
XDC ===> L TASKS;;L RBS;;L REGS
   TCB#  PROGRAM NAME (ASID 0039, DBCOLE9)
_  1     IEAVAR00
_  2     . IEESB605
_  3     . . IKJEFT01, JOBSTEP (STEPLIB)
_  4     . . . IKJEFT02
_  5     . . . . IKJEFT09
_  6     . . . . . ISPF
_  7     . . . . . . ISPTASK (ISPLLIB)
_  8     . . . . . . . S20CALL
_ 9     . . . . . . . . "XXXCALL" (TASKLIB144)
_ 10    . . . . . . . . . "Z12CALL" CURRENT (ABEND: S0C1)
_  11    . IEAVTSDT

   RB#  TYPE   CREATED BY   NAME           CURRENT EXECUTION LOCATION
_  1    PRB    ATTACH       "Z12CALL"      PRIVATE+65CFC (RETRY LEVEL)
_  2    SVRB   DEAD TRAP    ABEND-0C1      IGC0101C+79D6 (ARTIFACT)
_  3    PRB    SYNCH        ESTAE/I        XXXTFS.X#1+23DB4 (ARTIFACT)

_      R0  00009B88 0006BF90 E7C4C3C3 C1D3D340 *...h....XDCCALL *
_      R4  E9F1F2C3 C1D3D340 80FDA906 000650CD *Z12CALL ..z...&.*
_      R8  00000000 0000BFE0 0001BBF0 00067718 *..........0....*
_      R12 000640CE 0005ECA8 80067CFE 800690A0 *.. ....y..@.....*
```

Help COmmands Zap

The **ZAP** command is used to change the contents of:
- Accessible virtual storage. This can include:
  - The private area
  - Common storage
  - "Read-only" storage
  - Dataspaces
  - Foreign address spaces
  - Real storage
- The retry level general registers

- The retry level access registers
- The floating point registers
- The floating point control register
- The vector registers
- The guarded storage registers
- The resume address portion of the retry level PSW

Note, when Z/XDC is running authorized, it runs in key 0; consequently, in this case **any** defined storage in the Target Address Space can be altered (security permitting. See HELP SECURITY.)

Further, when Z/XDC runs authorized, even "protected" pages in the PLPA and the "read-only" areas of the System Nucleus can be zapped. In this case the altered page or pages are permanently page-fixed. Thus the zap is preserved until (but not beyond) the next IPL.

Also, if an authorized user has requested and has been given access to a foreign address space (via the SET ASID command), then the ZAP command can be used to alter the private areas of that address space. See HELP VIRTMEM XDCACCESS FASM for more information.

Also, subject to suitable access and security permits, the ZAP command can even alter store protected storage located essentially anywhere:

- In common storage
- In the private area of the Home Address Space
- In the private area of any accessible Foreign Address Space

Note, Z/XDC **cannot** alter storage protected by the System's **Low Core Protection** feature. This would be the first **512 bytes** of each of the first two pages of both virtual and real address space storage. (Dataspace storage is not affected by Low Core Protection.)

All ZAPs to virtual storage are carried out subject to:

- The user program's authorization level.
- Security rules.
- The current **SET ZAP SPROT/NORMAL** setting.

For specific information, see HELP SECURITY ZAP.

The **ZAP** command has two forms. One stores raw string data into the target location. The other resolves a given address expression to a 24-bit, 31-bit, or 64-bit value and stores that into the target location.

The following rules are true for both forms of the ZAP command.

- All operands are thoroughly checked for validity, accessibility, security, integrity and consistency before **any** data is actually stored. In other words the ZAP command does not store any data until it is sure that it can store all of the given data successfully.
- After zaps to virtual or real storage, both the **Current Display Pointer** and the **Next Display Pointer** are set to point to the start of the target location. This facilitates the redisplaying of the altered storage should you wish to do so. Note, these display pointers are **set to zero** when the target location is a register or PSW.

General Syntax:

```
    ZAP target address-data
             ,address-data
             =string,string,...
             =&string,string,...
             =|string,string,...
             =#string,string,...
```

target

This specifies the location (storage, register or PSW) whose contents are to be zapped. See HELP COMMANDS ZAP TARGETS for specific information.

When the zap target is a **vector register,** and the computer on which you are running does not support vector registers, the zap command will fail with a DBC123E message.

However, if you want to experiment anyway with zapping vector registers, you can add a **PRETEND** operand, and Z/XDC will pretend that they exist after all. But of course, whenever Z/XDC releases control back to your program (i.e. you issue a GO or TRACE command), all zapped changes will be lost.

For more information, see HELP COMMANDS LIST VECTORREGISTERS.

address-data
,address-data

This is an address expression whose value is to be resolved and zapped into the target. See HELP COMMANDS ZAP ADDRESS for specific information.

=string,string,...

This is string data (hexadecimal, decimal, and/or character) that is to be zapped into the target. See HELP COMMANDS ZAP STRING for specific information.

=&string,string,...
=|string,string,...
=#string,string,...

This is string data that is to be AND'd (&), OR'd (|), or exclusive OR'd (#) into the target. See HELP COMMANDS ZAP BOOLEAN for more information.

The following specific syntax information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

TARGETS - Locations that can be zapped.
ADDRESS - Converting an address expression into hex data and zapping it into a target.
STRING - Zapping hexadecimal strings, character strings, and/or decimal numbers into a target.

BOOLEAN - Zapping individual bits by AND'ing, OR'ing, or XOR'ing a string into a target.

Zapping can also be performed simply by overtyping displayed data. For more information, see HELP SHORTCUTCMDS Z.

Help COmmands Zap Targets

The **ZAP** command can be used to alter the contents of:
- Accessible virtual storage. This can include:
  - The private area
  - Common storage
  - Dataspaces
  - Foreign address spaces
  - Real storage
  - Store protected storage
- The retry level general registers
- The retry level access registers
- The floating point registers
- The floating point control register
- The vector registers
- The guarded storage registers
- The retry level PSW (resume address only - Use the **SET PSW** command for altering other fields of the retry level PSW)

Syntax:

ZAP target address-data
,address-data
=string,string,...
=&string,string,...
=|string,string,...
=#string,string,...

Shortcuts: J [to change the Retry Level PSW's resume execution address]
Z [to change data in registers or storage]

target
This specifies the location whose contents are to be zapped. Target may be any of the following:

addressexpression
This is an address expression that resolves to some accessible virtual storage location. The location may be in the private area, in common storage, in a dataspace, in a foreign address space, or in real storage (security permitting, of course). The length of accessible storage (starting at this location) must be as least as long as (or longer than) the data to be zapped. For more information, see **HELP ADDRESSING.**

register-reference
This must be the name of a retry level register (possibly qualified by a starting offset). For string data zaps, the register may be:

- A general register (Rn, RHn, or RWn)

- An access register (ARn)

- A floating point register (FRn)

- The Floating Point Control Register (FPC)

- A vector register (VRn)

- A Guarded Storage register (GSn)

The register name may be followed by a simple numeric offset for selecting the first byte to be zapped.

Example: **ZAP R5+1=34BC**
This changes the middle two bytes of general register R5 to be the value X’34BC’. (The first and last byte remain unchanged.)

For address data zaps, the target register may only be a general register (Rn, RHn, or RWn). For access and floating point registers, the data must be string data, address data may not be used.

For vector registers, if the computer you are using does not support vector registers, then the command is failed. (Vector registers are supported on **Z13** and newer processors.)

However, If you want to play with displaying and zapping vector registers anyway, you can add a **PRETEND** operand to make Z/XDC pretend that vector registers exist after all. But of course, whenever Z/XDC releases control back to your program (i.e. you issue a GO or TRACE command), all zapped changes will be lost.

For guarded storage registers, if the computer you are using does not support guarded storage, then the command is failed. (Guarded Storage is supported on **Z14** and newer processors.)

However, If you want to play with displaying and zapping guarded storage registers anyway, you can add a **PRETEND** operand to make Z/XDC pretend that guarded storage exists after all. But of course, whenever Z/XDC releases control back to your program (i.e. you issue a GO or TRACE command), all zapped changes will be lost.

PSW-reference
This must be a reference to the **retry level** PSW. (Error level PSWs may not be zapped.) It must be either (and exactly) **PSW** or **PSWE,** it doesn’t really matter which. Technically:
- **PSW** refers to the 8-byte ("scrunched") view of the PSW.
- **PSWE** refers to the 16-byte ("wide") view of the PSW.

But in truth, it does not matter which you use. Attempts to zap a 64-bit address into **PSW** will be honored as if **PSWE** had been specified. And vice-versa.

Only the instruction address portion of the PSW may be zapped; therefore, only address data zaps are permitted. (Use the **SET PSWE** command to "zap" other areas of the PSW.)

String data zaps are not supported for the PSW.

Example: **ZAP PSW,10?+50**
This zaps the retry level PSW to point to the location represented by **10?+50.** This changes the PSW so that the next instruction to be executed by the user program will be the **SVC 3** instruction located at the **CVTEXIT** field of z/OS's CVT.

,address-data
=string,string,...
=&string,string,...
=|string,string,...
=#string,string,...
These operands provide the address data or string data to be zapped into the target location. They are described in detail by subsequent topics. Type **HELP *NEXT** (F11) for more information.

Examples:

ZAP R5?+1=C'ABC'
This zaps the storage at one byte past the location pointed to by general register R5 to contain X'C1C2C3'.

ZAP R5+1=C'ABC'
This zaps general register R5 to contain X'C1C2C3' starting in its 2nd byte. The hi-order byte is not changed.

ZAP PSW,10?+50
ZAP PSWE,CVT.CVTEXIT
These both do exactly the same thing: They zap the retry level PSW to point to the **CVTEXIT** field of z/OS's Communications Vector Table.

SET PSW AMODE=64
ZAP PSW,FEDCBA9876543210
ZAP PSWE,FEDCBA9876543210
The **SET PSW** command sets the retry level PSW's Addressing Mode to 64-bit. The two **ZAP** commands then do exactly the same thing: They zap the retry level to a rather high address.

ZAP R9,.BUFFER2
This zaps the retry level general register R9 to contain the address of a location labeled "BUFFER2". The hi-order bit or byte (depending upon the AMODE of the retry level PSW) remains unchanged.

Help COmmands Zap Address

The address-data form of the **ZAP** and **VERIFY** commands resolves a given address expression into a 24-bit, 31-bit, 32-bit or 64-bit value and verifies or stores that value at a target location. The target must be 4 or 8 bytes wide. The value is verified or stored at the low-order 24, 31, 32 or 64 bits of the target as follows:

24-bit value: - The value is stored into the low-order 3 bytes of a 4-byte wide target. The high-order byte is not altered.
31-bit value: - The value is stored into the low-order 31 bits of a 4-byte wide target. The high-order bit is not altered.
32-bit value: - The value is stored into all 32 bits of a 4-byte wide target.
64-bit value: - The value is stored into all 64 bits of an 8-byte wide target.

In all cases, the target location need not be aligned.

Syntax:

```
VERIFY target address-data
ZAP          ,address-data
```

target

This identifies the target at which the data is to be verified or stored. It may be a storage location, a register reference, or a PSW reference. For the VERIFY command, it may also be any of certain numeric built-in functions. This operand is fully described by the preceding topic. See HELP ∗PREVIOUS for more information.

, (comma)
(blank)

The comma or blank delimiter distinguishes this "address data" form of the ZAP and VERIFY commands from the "string data" form (which uses an "=" instead). The comma or blank indicates that the following operand is an address expression, not a hex or character string.

address-data

This is an arbitrary virtual address expression. It is resolved to a 24-bit, 31-bit, 32-bit or 64-bit value before being verified or stored. The width of the value is derived in either of two ways. If the address-data contains a WIDTH(...) built-in function call, then the final width of the address data is derived from the value specified by the function. On the other hand, if **no** WIDTH(...) function call occurs within the address-data, then the final width of the address data is derived from the AMODE setting in either the retry level or error level PSW. More specifically:

- **When WIDTH(...) is present:** The resulting address data's final width will be set as follows:

WIDTH FUNCTION VALUE	FINAL BIT WIDTH	FINAL BYTE WIDTH	COMMENT
1-24	24	4	- The address data is zapped into the 24 bits following the target's first 8 bits. The target's first 8 bits are **preserved.**

25-31	31	4	- The address data is zapped into the 31 bits following the target's first bit. (The target's first bit is **preserved.)**
32	32	4	- The address data is zapped into all 32 bits at the target address.
33-64	64	8	- The address data is zapped into all 64 bits at the target address.

- **When WIDTH(...) is absent:** Then the width of the value will depend upon a PSW's current addressing mode (which can be displayed by the LIST AMODE command and changed by the SET PSW command).
  - If the current AMODE is 24-bit but the resolved address is greater than 16 megabytes, then the high-order bits of the address are truncated.
  - Similarly, if the current AMODE is 31-bit but the resolved address is greater than 2 gigabytes, then the high-order bytes of the address are truncated.

  In other words, the size of the address verified or stored depends upon the address mode as follows (when WIDTH(...) is absent):

    - **AMODE(24):** The address is resolved to a 3-byte value, and is then verified against or stored into the lo-order 3 bytes of the 4 bytes pointed to by the target expression. The hi-order byte is neither checked nor altered.

    - **AMODE(31):** The address is resolved to a 31-bit value, and is then verified against or stored into the lo-order 31 bits of the 4 bytes pointed to by the target expression. The hi-order bit is neither checked nor altered.

    - **AMODE(64):** The address is resolved to an 8-byte value, and is then verified against or stored into the 8 bytes pointed to by the target expression.

Examples:

Z R5,C
The address expression "C" is resolved to a 24-bit, 31-bit, or 64-bit value (by prefixing high-order zeros) and then stored into the low-order portion of retry level general register R5.
 - In AMODE(24), R5's hi-order byte is not altered.
 - In AMODE(31), R5's hi-order bit is not altered.
 - In AMODE(64), all 32 bits of R5 are set. (RH5 is not altered.)

If you want to set just the hi-byte of R5 to X'0C', then try "Z R5=0C" (i.e. the string data form of the ZAP command).

Z R5,C~WIDTH(32)
In this case, the address "C" is padded out to 32 bits and stored into all 32 bits of register R5. (RH5 is not altered.)

Z RW5,C

When the target is a wide register, then the address data is resolved to 64 bits regardless of all other considerations. So in this case, RW5 is set to X'000000000000000C'.

LOAD CDAEED CEE.SCEERUN2
Z R5,CDAEED
Note that **CDAEED** could be either a hex address or a module name. In this case, it happens to be the name of an actual LE module. So when **CDAEED** is present in storage, the address expression **CDAEED** does not resolve to location X'00CDAEED'. Instead, it resolves to the location of the CDAEEE load module, and that is the address that is stored into R5.

To avoid ambiguity, try either of the following:

Z R5,0CDAEED
This zaps the address 00CDAEED into the lo-order 24 or 31 bits of R5.

Z R5,'CDAEED'
Z R5,CDAEED.
Either of these zaps the address of the CDAEED load module into the lo-order 24 or 31 bits of R5.

V R9,10%;Z R10,21C%
Here's how to do **a conditional zap.** The VERIFY command first checks to see if general register R9 points to the CVT. If so, then R10 is zapped to point to the current TCB. On the other hand, if R9 does not point to the CVT, then the VERIFY fails, and so Z/XDC aborts the command string, so the ZAP then is not executed.

Z 10R,1F9234CB+ADC6
The value of the address expression is computed: X'1F92E291'. Then:
- If the current AMODE is 24, then the value is truncated to 24 bits (X'92E291') and stored into the low-order 3 bytes of R10. The hi-order 8 bits of R10 remain unchanged.
- If the current AMODE is 31, then the value is stored into the low-order 31 bits of retry level general register R10. The hi-order bit of R10 remains unchanged.
- If the current AMODE is 64 (z/OS only), then the value is first padded out to 64 bits (X'000000001F92E291'). Then it is truncated back down to 32 bits (the width of R10) and store into all 32 bits of R10.

Z RW10,1F9234CB+ADC6
(z/OS only) The value of the address expression is computed, to 64 bits (X'000000001F92E291'). Then it is stored into all 8 bytes of RW10. Thus, RH10 (the high half of RW10) is set to X'00000000', and R10 (the low half or RW10) is set to X'1F92E291'.

Z PSW,PSW?-4
This backs up the retry level PSW by 4 bytes. The address expression "PSW?-4" is

resolved to a 24-bit or 31-bit value and "stored" into the instruction address field of the retry level PSW. The addressing mode flag is not altered.

Z .MYTCBP,21C?
If the current AMODE (in the retry level PSW) is 24-bit, then the address of the current TCB is stored into the low-order three bytes of the 4 bytes starting at the address referenced by the label MYTCBP. The hi-order 1 byte remains unchanged.

On the other hand, if the current AMODE is 31-bit, then the address of the current TCB is stored into the low-order 31 bits of the target address (MYTCBP), and the target's hi-order bit remains unchanged.

If AMODE is 64 bits, then the current TCB's current address is padded out to 64 bits and stored into the **eight** bytes starting at MYTCBP. (Is this what you would want to happen?)

Z .MYTCBP,21C?~WIDTH(32)
The address of the current TCB is padded out to 32 bits and stored at MYTCBP. All 32 bits are stored. No bits in the target are preserved. The width of the stored address is set to 32 bits **regardless** of the current addressing mode.

S PSW 31
Z R2,XCSA
D
The SET PSW command sets the user program's addressing mode to 31-bit. The ZAP command then stores the starting address of the extended CSA into R2. **(XCSA** is a built-in equate that labels the start of extended CSA storage. See HELP EQUATES BUILTIN for more information.)

The ZAP command causes both the **Current Display Pointer** and the **Next Display Pointer** to be set to point to MYTCBP. The **DISPLAY** command, in this case, displays the storage pointed to by the Next Display Pointer. In other words, the zapped storage is immediately redisplayed.

Help COmmands Zap String

The string form of the **VERIFY** and **ZAP** commands verifies or stores string data at the target location.

Syntax:

```
VERIFY target=string,string,...
ZAP          =&string,string,...
             =|string,string,...
             =#string,string,...
```

target
This identifies the target at which the data is to be verified or stored. It may be any of the following:

addressexpression
This is an address expression that resolves to some accessible virtual storage location. The location may be in the private area, in common storage, in a dataspace, in a foreign address space, or in real storage (security permitting, of course). For more information, see HELP ADDRESSING.

register+offset
This refers to a particular byte within either a retry level general register, a retry level access register, or a floating point register. For the VERIFY command, this may also be any of the error level registers: general, access, or control.

The offset must be in the range of 0 to 3 for 4-byte wide registers and 0 to 7 for 8-byte wide registers. It identifies a specific starting byte **within** the register.

PSW+offset
EPSW+offset
This is permitted only for the VERIFY command. The ZAP command may only use the address data (not string data) form of the command. See HELP *UP ADDRESS for more information.

"PSW+offset" refers to a particular byte within the retry level PSW. "EPSW+offset" refers to a particular byte within the error level PSW. The offset must be in the range of 0 to 7. It identifies a specific starting byte **within** the PSW (or EPSW).

=
The equals sign distinguishes this "string data" form of the ZAP or VERIFY command from the "address data" form (which uses a blank or a comma as the delimiter). The equals sign indicates that the following operand is string data. See HELP *UP ADDRESS for information about address data.

=&
=|
=#
These operators are valid only with the ZAP command. (They cannot be used with the VERIFY command.) They indicate that the following string data is to be combined into the target locating using a boolean operation instead of a simple store. The supported operations are AND (=&), OR (=|), and XOR (=#). See HELP COMMANDS ZAP BOOLEAN for more information.

string,string,...
This gives a string of data to be verified against or stored into the target location. The string is converted to internal form and either verified or stored using as many bytes as are necessary to hold the value; no padding is performed.

As many strings may be given as desired. When multiple strings are given, they are each converted to binary according to their types and then concatenated together before being used for verify or zap purposes. See "Examples" below for a usage example.

The strings may be given as either hexadecimal digits, EBCDIC characters, or as

```
decimal integer numbers. Brief examples:
    ZAP ...=F0D93B72      a hex string
    VER ...='DBCOLE'      a character string
    FIND C4'BCOL'C5,...   a mixture
    ZAP ...=F'-1234557'   a 4-byte wide integer
    VER ...=H'32767'      a 2-byte wide integer
    ZAP ...=1'139'        a 1-byte wide integer
```

For more complete information, See "HELP COMMANDS SYNTAX STRINGDATA".

Examples:

V ER5=153D
The value X'153D' is verified against the 1st and 2nd bytes of error level general register R5. The 3rd and 4th bytes are not checked.

Z 3R+2=01
The value X'01' is stored into the 3rd byte of retry level general register R3. The 1st, 2nd, and 4th bytes are unaltered.

Z R7+1=H'1024'
The value X'0400' is stored into the 2nd and 3rd bytes of retry level general register R7. The 1st and 4th bytes are unaltered.

Z FR6=F3,7539
The value X'F37539' is stored into the first three bytes of floating point register FR6. The remaining five bytes are unaltered.

ZAP PSW?=12FF,078E,00DEAD,1'6','ERROR!'
This example illustrates the use of multiple strings of various types in one zap. This zap stores data that is equivalent to the following Assembler code:

```
    LTR   R15,R15
    BZR   R14
    DC    X'00DEAD',AL1(6)
    DC    C'ERROR!'
```

Note, the string "1'6'" means "convert the decimal number "6" to binary and store it into a 1-byte wide field.

F PSW?
V +4=47F0CBDE;Z +0=47000000;F -4;DOWN M
The first FORMAT command displays storage and sets the "Current Display Pointer" to point to the storage just displayed.

The VERIFY command ensures that the current contents of 4 bytes past the displayed location are X'47F0CBDE'. If the verification fails, then the VERIFY command aborts with an error message and the following ZAP command is not executed. If the

verification succeeds, then the VERIFY command sets the "Next Display Pointer" (as well as the "Current Display Pointer") to point to the verified location.

- The ZAP command stores the value X'47000000' (a NOP) at 4 bytes past the previously displayed location and resets the "Next Display Pointer" (as well as the "Current Display Pointer") to point to the zapped location.

- The second FORMAT command redisplays the same location that was displayed by the first FORMAT command. Note that "-4" has to be specified here in order to get back to that location because of the effects of the intervening VERIFY and ZAP commands.

- Finally, the DOWN M command brings the redisplayed storage into view. Without it, the display is left showing the results of the first FORMAT command, and you would then have to scroll down to see the results of the remaining commands in this string.

Z PSW?=0700,0700;F
Here, the value X'07000700' (two NOPRs) is stored and then displayed. (The original contents of the target location are not verified or displayed.) Note that in this case the FORMAT command is given without operands. This causes it to reference the Next Display Pointer instead of the Current Display Pointer. But since the ZAP command has set both the NDP and CDP to point to the same location, in this case "F" and "F +0" are equivalent.

V PSW+1=8D;Z PSW,R5?
The VERIFY command ensures that the retry level PSW is set to problem state and execution key 8. If so, then the ZAP command changes the PSW's next instruction address to point to the location pointed to by retry level general register R5. Note that the VERIFY command (in this example) uses string data while the ZAP command uses address data.

Z .DATA='C'D6D3'E'
This stores the character string "COLE" at the location labeled DATA. Notice that the string is given as a mixture of EBCDIC characters and hexadecimal digits.

Help COmmands Zap Boolean

The **ZAP** command, as expected, can be used to replace data in storage or registers on a byte by byte basis. Alternatively, it can also be used to **AND, OR,** or **XOR** (exclusive OR) a given string of data into storage. This allows you to select and manipulate individual bits in a target without the necessity of knowing the values of bits not being zapped.

Syntax:

ZAP target=|string,string,...

=&string,string,...
=#string,string,...

target
This identifies the target at which the data is to be stored. It may be any of the following:

addressexpression
This is an address expression that resolves to some accessible virtual storage location. For more information, see HELP ADDRESSING.

register+offset
This refers to a particular byte within either a retry level general register, a retry level access register, or a floating point register.

The offset must be in the range of 0 to 3 for 4-byte wide registers and 0 to 7 for 8-byte wide registers. It identifies a specific starting byte **within** the register.

=| or **=?**
These special characters indicate that the string data is to be **OR'd,** byte for byte, into the target location. The string data can be used to turn on individual bits without regard for the settings of other bits that are not of interest. Note, either a solid vertical bar ("=|") or a broken vertical bar ("=?") can be used to signal the **OR** operation.

=#
This special character indicates that the string data is to be **XOR'd,** byte for byte, into the target location. The string data can be used to toggle individual bits without regard for the settings of other bits that are not of interest.

=&
This special character indicates that the string data is to be **AND'd,** byte for byte, into the target location. The string data can be used to turn off individual bits without regard for the settings of other bits that are not of interest.

string,string,...
This gives a string of data to be **OR'd, XOR'd,** or **AND'd** at the target location. The string is converted to internal form, then **OR'd, XOR'd,** or **AND'd** into the target object using as many bytes as are necessary to hold the value; no padding is performed.

As many strings may be given as desired. When multiple strings are given, they are each converted to binary according to their types and then concatenated together before being **OR'd, XOR'd,** or **AND'd** into the target location.

The strings may be given as either hexadecimal digits, EBCDIC characters, ASCII characters, or decimal integer numbers. For more information, see HELP COMMANDS SYNTAX STRINGDATA.

Examples:

Z R15?+3C=|01
Z R15?+3C=?01
Both of these commands turn on the lo-order bit of the byte located at +X'3C' past

the location pointed to by register R15. All other bits at that location remain unaltered.

Z R15+3=&FD
This command turns off the second from last (second from right-most) bit of register R15. All other bits in the register remain unaltered.

Z .INBUFFER=|' ';F
Assuming that the data field named INBUFFER contains text data, the ZAP command upcases the first eight characters of the field. The FORMAT command then redisplays that field after the upcasing has occurred.

Help COmmands =jump

The ISPF style **=jump** commands (=X, =6, =3.3, etc.) are available only when Z/XDC's Fullscreen Support is turned on and its ISPF communications interface is active. (See **HELP USERINTERFACES** for information about communication interfaces.)

The **=jump** commands function in the same way they do in ISPF: They exit from the current program and transfer control to another program in ISPF, thus they cause Z/XDC to terminate as if an **END** command had been given.

Syntax:

=jumpcodes

jumpcodes
These are the ISPF function codes that identify the work to be transferred to.

Example:

=3.3
This causes the Z/XDC debugging session to terminate and to transfer control to ISPF's Move/Copy utility.

For more information, look up "Jump Functions" in IBM's **ISPF User's Guide, Vol. I** (SC34-4822).

Help COmmands %

% is a command that allows the user to run user-written subcommands written in REXX.

Syntax:

```
%    execname argumentstring
%execname argumentstring
```

Note that the **execname** may immediately follow the % - there is no need to have any intervening blanks.

execname
This is the name of the REXX exec to be run.
the following formats are recognized:
  name
  name(member)
if **name** has at least 1 fullstop within it, the value is always regarded as a dataset name.
if **name** has no fullstops within it, the value is regarded as **simple** and the value could be:
  1: A procedure name.
  2: A DDname.
The rules used to decide which applies are:
  - If immediately followed by an open parenthesis (’(’) then the value is treated as a ddname.
  - If **SET REXX NAME=PROCEDURE** is in effect then the value is treated as a procedure name.
  - Otherwise, the value is checked to see if it is an allocation:
    - If a current allocation is **not found** the value is treated as a procedure name.
    - If a current allocation is **found** the value is treated as a ddname.

Based on the above, an execname could be:
  1: A procedure name - naming a member in the xxxREXX library
  2: A ddname - defining a sequential file allocated to that ddname.
  3: A dsname - defining a sequential file by dataset name.
  4: A member of a library defined by a specific ddname.
  5: A member of a named PDS(E) library.

Note that, in all cases, the REXX execs library must be allocated (ddname: xxxREXX, see below).

argumentstring
This is the argument string (if any) that may be required by the REXX exec. The user’s exec can receive this string via an **ARG** or **PARSE ARG** instruction.

Note, due to Z/XDC’s own syntax rules, argumentstring may not contain semicolons or colons unless the string is enclosed within quotes (’). See COMMANDS SYNTAX CHARACTERSTRINGS for more information.

Z/XDC Services Available to User-Written Execs

The rexx/XDC Interface provides several built-in functions by which user-written execs can call a selection of Z/XDC’s internal services. Generally, these services

do such things as:
- Control the source of REXX input and output
- Parse address expressions
- Extract or zap data in storage
- Extract or zap data in registers
- Extract or alter the retry level PSW
- Send messages to Z/XDC's Display Management for display at the user's terminal.
- Request data from a user

For complete and detailed information, see HELP REXX XDCSERVICES.

For more information on the use of REXX procedures, see **HELP COMMANDS REXX.**

Help SCripts

What are Scripts

A script is simply a sequential list of Z/XDC commands that are to be executed one after another...
- Either until an end of file is reached, in which case all commands have executed successfully.
- Or until a command fails, in which case the script is suspended but remains positioned at the next record following the record that contained the failing command.

Several examples of scripts can be found in the **XDCCMDS** library that is distributed as a part of Z/XDC. Its factory default name is **hlq.XDCV31.XDCCMDS,** but you will have to ask your Systems Programmer for its actual name at your Data Center.

Scripts are executed by the **READ** command. **READ** can be used to:
- Open a new script and process it from beginning to end.
- Resume a suspended script.
- Close a suspended script.
- Intentionally suspend a script.

For more information, see **HELP COMMANDS READ.**

Scripts are Primitive

Scripts are simple lists of commands that are processed one after another until either end of file is reached or and error occurs. There is no support for:
- DO/END constructs
- IF/THEN/ELSE constructs
- User defined variable substitutions

If you require something more sophisticated, try Z/XDC's **REXX** support. There, you will find the infrastructure for generating and processing Z/XDC commands under the control of whatever logic you care to write.

When Scripts Fail

As mentioned above, scripts process sequentially either until end of file is reached or until a command being processed fails. When a command failure occurs, the following happens:

- The failing command issues its error messages and aborts.
- The abort causes the script to be suspended, not closed. It remains opened and positioned. A **LIST READ** command will report this state.
- Individual logical records within a script may contain multiple commands separated from each other by semicolons (;).
- When a command aborts:
  - All remaining commands from the same logical record as the failing command are discarded.
  - The scripts file is **not closed.**
  - Instead, it is **suspended** and left positioned to the start of the next logical record.
  - Z/XDC then unlocks the keyboard and waits for further command input from the user.
- This gives you the opportunity:
  - To investigate the reason for the failure,
  - Issue a corrected command (if needed),
  - Then resume the script starting at the next logical record.
- The script will remain suspended until one of the following occurs:
  - You issue a **READ RESUME** command (or just a **READ** command) to resume the script.
  - You issue a **READ CLOSE** command to close the script.
  - You issue a **READ someotherscript** command to start another script.
  - You issue a **TRACE** or **GO** command to return control to the user program. (Scripts are always closed when that occurs, but see **Propagating Scripts ...** below for a suggestion.)

Suggestion: When a script contains a command that might fail, followed by other commands that depend upon the success of that prior command, then place those commands onto a single logical record. Then to resume the script, you need to issue **READ RESUME** only once (not twice) to move past the problem. Example:

DMAP .CVT;U CVT 10%
If for some reason the **DMAP** command should fail, then the **USING** command is discarded, and the script file is positioned to the next following logical record. This allows the script to be resumed with only a single **READ RESUME** command.

If the two commands had been placed on separate lines, then it would have taken two **RESUMEs** to get past them.

Variables Support

Scripts do not have generalized support for variable substitutions within script records.

However, you do need to be aware that there is some extremely limited variables support for Colesoft's in-house use. So if you're unlucky enough to happen to use certain strings within your script, you will see the following unexpected results:

C - All occurrences will be replaced by **XDC** (most likely).

X - All occurrences will be replaced by the first three characters of the name of the load module that contains the retry level resume address.

Propagating Scripts Across GO and TRACE Commands

As noted above, when a **GO/GOT/GOX** command or a **TRACE** command (other than TRACE W) is encountered within a script, prior to processing that command, the script file is CLOSE'd. The reason this happens is that those commands cause Z/XDC to relinquish control back to the user program, and Z/XDC does not have the logic necessary to keep datasets OPEN'd in that circumstance.

A consequence of **GO/GOT/GOX** and **TRACE** causing file closure is that all scripted commands following the **GO/GOT/GOX** or **TRACE** command will be ignored and lost. However, there is a workaround.

Notice that all breakpointing commands (AT, TRAP, TRACE, TDEFERRED, etc.) can have one or more arbitrary **automatic commands** assigned to them. There is no reason why such an automatic command can't be a **READ** command. In other words, you can associate a **READ** command to a breakpoint to cause successor scripts to be processed! (See **HELP COMMANDS SYNTAX BREAKPOINTS AUTOCMDS** for syntax details.)

So if your script has a **GO/GOT/GOX** or **TRACE** command (to return execution to the user program), and if you want additional scripted commands be executed when Z/XDC receives control back from the user program, then:

- Place those additional commands into a successor script,
- And then include a trapping command **(AT, ATX, TRAP, ADEFERRED** or **TDEFERRED)** in your original script,
- And associate a **READ** command with that trap that causes the successor script to be processed.

Similarly, if your script has a **TRACE** command (other than TRACE W), then associate a **READ** command with the TRACE command to pass control to a successor script so that when Z/XDC resumes control after the trace step is taken, the successor script will be processed.

For an interesting example, check out the **AUTOTBNC** script located in the distributed hlq.XDCV31.XDCCMDS library. (You will have to ask your Systems Programmer for its actual name at your Data Center.)

Subtopics Index

Additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

OURSCRIPTS - Prewritten scripts that we provide.
RYOSCRIPTS - About writing your own scripts.

Help SCripts Ourscripts

Scripts are simple sequential lists of Z/XDC commands that are to be executed one after another. You may, of course, write your own scripts (see **HELP SCRIPTS RYOSCRIPTS** for details), but we also provide a number of prewritten scripts that hopefully have a general usefulness.

Scripts are processed by the **READ** command. See **HELP COMMANDS READ** for details.

Factory Provided Scripts

The scripts that we provide can be found in the **XDCCMDS** library. Its factory default name is **hlq.XDCV31.XDCCMDS,** but you will have to ask your Systems Programmer for its actual name at your Data Center.

Our scripts can be used to perform a variety of services that will be useful to many customers. They also serve as good examples of how to do some perhaps unexpected things with Z/XDC. Please feel free to lift whatever you find interesting or useful.

Our scripts serve as models showing many interesting tricks about using Z/XDC. However, if a particular script does not do exactly what you want and in the way you want it done, then feel free to make your own copy and modify it any way you like.

The **XDCCMDS** library is a standard **FB-80** PDS. The scripts within the library all have a sequence field in the standard columns, **73-80.** Some scripts have non-numeric data in the sequence fields, but Z/XDC needs to ignore the field nonetheless:

- If **SET READ SEQF=PRESENT** or **SET READ SEQF=DETECT** is in effect, then Z/XDC will ignore the sequence fields, and everybody will be happy.

- On the other hand, if **SET READ SEQF=ABSENT** is in effect, then the scripts will fail when you attempt to run them unless you add **SETF=DETECT** to your **READ** command, then everybody will be sorta happy again.

- The current **SEQF=** setting can be displayed with the **LIST READ** command.

Subtopics Index

The **XDCCMDS** library contains the following scripts. For detailed information, type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

- **AUTOTBNC** - Automatically steps through execution of the current subroutine and logs the PSW, the BEA and the registers at each successful branch.
- **DCBMAPS** - Loads maps for all DEBs and DCBs currently OPEN'd for a given TCB. Also labels the corresponding DD entries in the TIOT.
- **EPMAPS** - Can be used at the start of a debugging session to load maps and define a default qualifier for the user program's primary load module.
- **ISPSETUP** - Can be used along with XDCCALL to build an ISPF internal parameter list for ISPxxxxx and ISRxxxxx modules.
- **RSAMAPS** - Defines labels both for the queue of RSAs pointed to by R13 and for the subroutine return points pointed to by those RSAs.
- **SDWAMAPS** - Loads maps for the RTM2WA, the SDWA, and some of the SDWA secondary control blocks.
- **SPIEMAPS** - Loads maps for the PIE, EPIE, and other control blocks that control SPIE and ESPIE routines.
- **STAXMAPS** - Loads TAXE maps and defines STAX labels to map the queue of control blocks that control pending attention exit routines and the exit routines themselves.
- **SVCTABLE** - Defines equates labeling all SVC routines defined in z/OS.
- **SYSINFO** - Displays information about both Z/XDC's and the Operating System's level and features.
- **TCBMAPS** - Loads maps for a TCB and its STCB ("Secondary TCB").
- **TIOTMAP** - Creates a series of equates labeling up to 999 DD entries in the current task's TIOT.
- **TRAPSTAE** - Builds an SVC screen to trap and nullify SPIE and ESTAE SVCs issued from the user program.
- **TSBMAPS** - Loads maps for various terminal related control blocks in TSO.
- **TSOMAPS** - Loads maps for various TSO related control blocks.
- **XSBMAPS** - Maps the XSBs that are linked to the RB's most recently displayed by Z/XDC's LIST RBS command.
- **XSVCMAPS** - Defines pointers to slots in the System's SVC table that may have been placed into use by Z/XDC.

Help SCripts Ourscripts AUTOTBnc

What AUTOTBNC is and Does

The **AUTOTBNC** script automatically steps through the execution of the current subroutine and logs the PSW, the BEA and the registers at each successful branch. The information is written to the Primary Window's scroll area and, eventually, to the session log.

AUTOTBNC works by setting up a closed loop that repeatedly issues a **T BNC** command. This is a trace that stops on all successful branches and all calls to and returns from subroutines. It does **not** trace into the subroutines. For more information about both the use of and **limitations** of **T BNC,** see **HELP COMMANDS TRACE BNC.**

WARNING! Auto-tracing's underlying **T BNC** command follows program execution, but it is not perfect. There are certain situations where unusual code could trick the **T BNC** logic, causing it to lose control of execution. These are hopefully rare, but

they are possible. For the details, see **HELP BREAKPOINTS TRACING CONDITIONAL FAILURES.**

Ending the AUTOTBNC Trace

As just noted, **AUTOTBNC** runs a **closed loop.** It will run forever unless it is interrupted by some external event. If you set a trap **(AT** or **TRAP** command) at the one or more places where you want the trace to end, that would be a well suited external event.

If the code that you are auto-tracing through has **multiple** possible exit points, then you have to take care to place your traps at all of them. Otherwise, your auto-trace could run forever.

Other ways to stop an auto-trace include:
- An ABEND of any sort
- A **#DIE** macro
- The **ATTN** key

Once the auto-trace has been stopped, a **LIST BREAKPOINTS** command will reveal that there still exist one or more tracing breakpoints by which the auto-trace followed execution. Use an **OFF TRACES** command to get rid of them.

Logging the Registers, the PSW and the BEA (or Whatever You Like)

The logging is done via Z/XDC's **Latent Commands Facility** which, by default, records register, PSW and BEA information automatically. But you can change the Latent Commands String to log anything you want. For more information, see **HELP FULLSCREEN LOGGING LATENTCOMMANDS.**

The logged information is written both to the Primary Window's scroll area and (eventually) to your debugging session's session log.

By default, the session log is written to SYSOUT class **X** on spool. But you can changed where (and whether) the log gets written by the **SET LOG** command.

If the session log is written to spool, you can, of course, use IBM's **SDSF** or Triangle System's **IOF** to view it and extract it out to your own dataset where you can use an editor to massage it into whatever form is of use to you.

Viewing with the LIST BRANCHES Command

The trace information is also recorded by Z/XDC into an internal **Branch History Table.** This is a table that records user program branches that Z/XDC detects. The table records:
- The branching instruction's mnemonic,
- Where that instruction is,
- Whether or not the branch occurred,
- And if it occurred, then where it branched to.

This table can be displayed by the **LIST BRANCHES** command.

Of course, once you issue this command, its report will be recorded into the session log, which you can later extract and massage as noted above.

Viewing the Logged Information in the Scroll Area

As noted above, the logged registers, PSW and BEA (or whatever else you have set up) are saved in your Primary Window's scroll area, but by default, you won't be able to see them. They are, after all, **"latent".**

But don't worry. Making them visible is quite easy. First, issue **SET LOG LATENT=SHOW.** You can do this before or after you run your auto-trace, it doesn't matter.

Then to see them, just scroll up.

Where to Find the AUTOTBNC Script

The **AUTOTBNC** script is distributed in the **XDCCMDS** library. The library's factory default name is **hlq.XDCV31.XDCCMDS.** Ask your Systems Programmer for its actual name at your Data Center.

Help SCripts Ourscripts Dcbmaps

The DCBMAPS script defines a structure of floating maps and labels for the following:

- The queue of DEBs for datasets OPEN'd under a given task. This queue is anchored from the TCBDEB field.
- The DCBs that are linked to those DEBs via the DEBDCBAD field.
- The TIOT DD entries that are associated with those DCBs (via the DCBTIOT field).

Once you have used the READ command to process the DCBMAPS commands, you might then want to use the "LIST MAPS" and "LIST EQUATES" commands to behold the result. Note the following points:

- Enough maps and labels are created to label up to nine DEBs, DCBs, and TIOT DD entries.
- You can determine the OPEN'd ddnames just by displaying the labeled TIOT DD entries. Example: "FORMAT DD1".
- Because the maps and labels form a floating structure, they will react instantly to the OPEN'ing and CLOSE'ing of datasets.

Initially, the script assigns the structure of maps to the currently active task. However, the entire structure can be moved at any time to label the DEBs and DCBs that are queued from any task in either the Home Address Space or any accessible foreign address space. Simply issue a USING command to reassign the base address for the "TCB" map to the address of any desired TCB. Example: "USING TCB TCB#3".

The DCBMAPS script is distributed in the XDCCMDS library. The library's factory default name is hlq.XDCV31.XDCCMDS. Ask your Systems Programmer for its actual name at your Data Center.

Help SCripts Ourscripts Epmaps

The EPMAPS script can be used at the beginning of a debugging session that has been started via Z/XDC's Startup Panel in ISPF, or via the XDCCALL command, or via any of its aliases (XDCCALLA, XDCCMD, and XDCCMDA). EPMAPS does the following:

- It loads at least a module map of the program to be debugged.
- If one is available, then it also loads a map of the csect that contains the program's entry address.
- It then issues a SET QUALIFIER command to define the entry csect as the debugging session's default csect. (This gives meaning to address expressions that start with a dot. Examples: ".+3BC", ".TOPLOOP", and just ".".)
- Finally, it produces a formatted display of the program's entry point code.

In other words, this script simply issues the first few commands that you might typically want to issue anyway at the start of almost any debugging session.

The EPMAPS script is distributed in the XDCCMDS library. The library's factory default name is hlq.XDCV31.XDCCMDS. Ask your Systems Programmer for its actual name at your Data Center.

Help SCripts Ourscripts Ispsetup

The ISPSETUP script is intended to be used in conjunction with the XDCCALL command to help set up a special input parameter list for ISPF and PDF load modules (i.e. load modules whose names start with "ISR" or "ISP"). Such modules expect R1 to point to a parameter list consisting of at least two pointers. The first always points to the current "ISPTASK"'s "TLD". The second pointer (and subsequent pointers, if any) points to miscellaneous data for the routine being called.

This script sets up a 2-word plist pointed to by R1. The 1st word points to the TLD. The 2nd word points to a standard OS-type PARM field containing the parameters, if any, provided on the XDCCALL command.

To use this file, first invoke the desired ISP... or ISR... module via XDCCALL as follows:

```
XDCCALL ISP... parameters
XDCCALL ISR... parameters
```

Or via Z/XDC's Startup Panel in ISPF.

This causes XDCCALL to load the ISR... (or ISP...) program into storage with R1 pointing to a 1-entry plist that points to a standard OS-style PARM field. XDCCALL then passes control to Z/XDC so that the user can issue further commands before the program actually starts.

Next, issue the Z/XDC READ command necessary to run this script. Example: "READ hlq.XDCV31.XDCCMDS(ISPSETUP)".

The script will pause from time to time and give you instructions about what to do next. Just follow the instructions.

The ISPSETUP script is distributed in the XDCCMDS library. The library's factory default name is hlq.XDCV31.XDCCMDS. Ask your Systems Programmer for its actual name at your Data Center.

Help SCripts Ourscripts Rsamaps

The RSAMAPS script defines a structure of floating labels for the following:

- The queue of standard 72-byte register save areas ("RSA"s) pointed to by retry level register R13.
- The return addresses associated with those RSAs.

The RSA labels are named "RSA0", "RSA1", etc. with "RSA0" labeling the RSA currently pointed to by R13, and "RSA1" labeling the next older RSA, etc. The corresponding return addresses are labeled "RETURN0", "RETURN1", etc.

Once you have used the READ command to process the RSAMAPS commands, you might then want to use the "LIST EQUATES" command to see the result. Note the following points:

- Enough labels are created to label up to ten RSAs, and their associated return addresses.

- Because the labels form a floating structure, they will react instantly when new subroutines are called and when currently called subroutines return.

- Initially, the script assigns the structure to the currently active RSA queue pointed to by retry level R13. However, the entire structure of labels can be moved at any time to label the RSAs located on any queue in either the Home Address Space or any accessible foreign address space. Simply issue an EQUATE command to reassign the address of the RSA0 label. Example:

```
    EQUATE RSA0 TCB#3+30+D*4? F D
```

  This labels the queue of RSAs currently associated with the programs running under TCB#3.

Note that the labels created by the RSAMAPS script label RSAs from a given starting point and continuing from there backwards to older RSAs. If you want to label RSAs moving forward from a starting point to newer RSAs, then you may want to create a script of your own that does that. Such a file would be useful for labeling RSAs queued from the TCBFSA field of a Task Control Block.

Z/XDC's **LIST RSA** command can be used to display a report of RSAs located on any desired queue.

The RSAMAPS script is distributed in the XDCCMDS library. The library's factory default name is hlq.XDCV31.XDCCMDS. Ask your Systems Programmer for its actual name at your Data Center.

Help SCripts Ourscripts SDwamaps

The SDWAMAPS script defines a structure of floating maps and labels for the following:

- The SDWA ("System Diagnostic Work Area") for the current ABEND.
- The SDWAPTRS block that contains pointers to the various SDWA sub-blocks.
- All of the sub-blocks pointed to by SDWAPTRS:

  SDWARC1 SDWANRC1
  SDWARC2 SDWANRC2
  SDWARC3 SDWANRC3
  SDWARC4 (z/OS only)
- The RTM2WA ("Recovery/Termination Manager 2nd level Work Area") describing the current ABEND (or breakpoint) that has caused Z/XDC to receive control.

Once you have used the READ command to process the SDWAMAPS script, you might then want to use the **LIST MAPS** and LIST EQUATES commands to see the result.

The SDWAMAPS script is distributed in the XDCCMDS library. The library's factory default name is hlq.XDCV31.XDCCMDS. Ask your Systems Programmer for its actual name at your Data Center.

Help SCripts Ourscripts SPiemaps

The SPIEMAPS script defines a structure of floating maps for the control blocks that identify the existence and characteristics of "Program Interrupt Exit" routines (SPIE and ESPIE routines). Specifically, the script defines floating maps and labels for the following:

- The SCA ("SPIE Control Area") queued from the current TCB.
- The PIE ("Program Interrupt Element") queued from the SCA.
- The EPIE ("Extended PIE") queued from the PIE.
- RPP ("Recovery PIE/PICA") queued from the SCA.

Once you have used the READ command to process the SPIEMAPS commands, you might then want to use the "LIST MAPS" and "LIST EQUATES" commands to see the result. Note that the RPP is not a map. Instead, it is built from a series of related floating equates.

Initially, the script assigns the structure to the currently active task. However, the entire structure can be moved at any time to label the SPIE/ESPIE definition blocks that are queued from any task in either the Home Address Space or any accessible foreign address space. Simply issue a USING command to reassign the base address for the "TCB" map to the address of any desired TCB. Example: "USING TCB TCB#3".

The SPIEMAPS script is distributed in the XDCCMDS library. The library's factory default name is hlq.XDCV31.XDCCMDS. Ask your Systems Programmer for its actual name at your Data Center.

Help SCripts Ourscripts STaxmaps

The STAXMAPS script defines a structure of floating maps for the control blocks that

identify the existence and characteristics of **Attention Exit** routines. Specifically, the script defines floating maps and labels for the following:

- The ASCB (Address Space Control Block).
- The ASXB (Address Space Extension Block) that is linked to the ASCB.
- The RCTD (Region Control Task Data Area) that is pointed to by the ASXB.
- The queue of TAXEs (Terminal Attention eXit Element) that is chained from the RCTD.
- The attention exits themselves (known as STAX routines) that are pointed to by the various TAXEs.

Once you have used the READ command to process the STAXMAPS commands, you might then want to use the **LIST MAPS** and **LIST EQUATES** commands to see the result. Note the following points:

- The labels and maps created by the script are defined in sets of two: **TAXEn,** and **STAXn.**

- Enough sets are created to label up to nine TAXEs and their associated exits. These sets are numbered 1 through 9 with TAXE1 mapping the newest TAXE, TAXE2 mapping the next older, etc.

- Because the maps and labels form a floating structure, they will react instantly to the issuance of SPIE and ESPIE macros.

Initially, the script assigns the structure to the Home Address Space. However, the entire structure can be moved at any time to label the TAXEs and attention exits that are queued for any accessible foreign address space. Simply issue a SET ASID command to display a different address space. The whole structure will automatically move to the new address space.

The STAXMAPS script is distributed in the XDCCMDS library. The library’s factory default name is hlq.XDCV31.XDCCMDS. Ask your Systems Programmer for its actual name at your Data Center.

Help SCripts Ourscripts SVctable

The SVCTABLE script defines equates that label all SVC routines defined in z/OS ranging from SVC 0 thru SVC 199.

As a side effect, the script loads a map of the system nucleus (IEANUC01).

For SVC numbers that are not defined in z/OS, the generated equates label IEANUC01.IGCERROR.

After it has defined the equates, the script finishes with a **FORMAT IEANUC01.IGCERROR LINES=10** command. The resulting display shows all SVC numbers that are **not** defined in z/OS.

Help SCripts Ourscripts SYsinfo

The SYSINFO script issues a series of commands that display both Z/XDC's and the Operating System's current release, maintenance level, and/or features. Generally, when reporting a problem to ColeSoft, including this script's output into the session log would be helpful.

The script itself contains commentary that provides more information. The script is distributed in the XDCCMDS library. The library's factory default name is hlq.XDCV31.XDCCMDS. Ask your Systems Programmer for its actual name at your Data Center.

See HELP SUPPORT CONTACTUS for ColeSoft's contact information.

Help SCripts Ourscripts TCbmaps

The TCBMAPS script defines a structure of floating maps for the following:
- The currently active TCB ("Task Control Block").
- The STCB ("Secondary TCB") that is linked to the TCB.

Once you have used the READ command to process the TCBMAPS commands, you might then want to use the "LIST MAPS" command to see the result.

If you are debugging a multi-tasking program, the TCB/STCB maps will automatically float to whichever TCB/STCB is current each time Z/XDC receives control.

The TCBMAPS script is distributed in the XDCCMDS library. The library's factory default name is hlq.XDCV31.XDCCMDS. Ask your Systems Programmer for its actual name at your Data Center.

Help SCripts Ourscripts TIotmap

The TIOTMAP script creates (and then displays) a series of equates that label up to 999 DD entries in the current task's Task I/O Table (TIOT).

The equates created by this script are created using the **EQUATE** command's **AUTOCLONE** operands. Feel free to take a look at the script and then check out HELP COMMANDS EQUATE AUTOCLONING.

All the equates created by this script are **floating** equates anchored from a root equate named **TIOT.** The entire structure can be automagically moved to label any TIOT located in any accessible address space in the system simply by changing the location of the **TIOT** equate. For more information, see HELP EQUATES FLOATING.

This TIOTMAP script complements the displays produced by the **LIST TIOT** command:
- The **LIST TIOT** command displays information built from the TIOT DD entries.
- The **TIOTMAP** script labels and displays the DD entries themselves.

The LIST TIOT command is implemented as a sample **User Commands Exit.** As a result, the command is not always available for use, while the script is.

The TIOTMAP script is distributed in the XDCCMDS library. The library's factory default name is hlq.XDCV31.XDCCMDS. Ask your Systems Programmer for its actual name at your Data Center.

Help SCripts Ourscripts TRapstae

The TRAPSTAE script creates an SVC screen on the current TCB that traps and nullifies all SPIE, and STAE/ESTAE SVCs that might be issued by programs running under that TCB.

This script can help prevent the user program from creating ESTAE, STAE, and SPIE routines that interfere with Z/XDC. However, this script is not a cure-all for this kind of problem. The SVC screen does not trap ESPIE macros (SVC #109-28), ESTAI= parameters for ATTACH and ATTACHX macros, and ESTAEX macros (a PC call).

The TRAPSTAE script can be used only when Z/XDC is running authorized.

The TRAPSTAE script contains commentary giving additional information that may be of interest, so read that file before proceeding.

The TRAPSTAE script is distributed in the XDCCMDS library. The library's factory default name is hlq.XDCV31.XDCCMDS. Ask your Systems Programmer for its actual name at your Data Center.

Help SCripts Ourscripts TSBmaps

The TSBMAPS script builds a structure of floating maps for certain system control blocks that exist in TSO address spaces and that describe the current characteristics and status of the user's TSO terminal. These control blocks are:

- The **TSB** ("Terminal Status Block") pointed to by the ASCB.
- The **TSBX** ("terminal Status Block Extension") pointed to by the TSB.
- The **TVWA** ("TSO/VTAM Work Area") pointed to by the TSBX.
- The terminal's "bind image" (ISTDBIND) contained within the TVWA.

This script also loads a dsect map for the ASCB.

Unfortunately, the System builds these control blocks in fetch protected key 6 storage; therefore, Z/XDC must be running authorized in order for these control blocks to be viewable.

Initially, the script assigns the structure of maps to the Home Address Space. However, the entire structure can be moved at any time to label the corresponding control blocks of any accessible foreign address space. Simply issue the following command:

SET ASID userid

- This command gains access to another TSO address space. Also, it automatically reassigns the **@ASCB** built-in equate to label that address space's ASCB. This automatically causes the floating maps that are based on **@ASCB** to move to the new address space as well.

The TSBMAPS script is distributed in the XDCCMDS library. The library's factory

default name is hlq.XDCV31.XDCCMDS. Ask your Systems Programmer for its actual name at your Data Center.

Help SCripts Ourscripts TSOmaps

The TSOMAPS script builds a structure of floating maps for a number of control blocks that exist only in TSO address spaces. Specifically, the script defines floating maps for the following:

- The **PSCB** ("Protected Step Control Block") pointed to by the address space's JSCB.
- The **UPT** ("User Profile Table") pointed to by the PSCB.
- The **RLGB** ("Relogon Buffer") pointed to by the PSCB.
- The **LWA** ("Logon Work Area") pointed to by the ASXB.
- The **ECT** ("Environment Control Table") pointed to by the relogon buffer.
- The **ECT** ("Environment Control Table") pointed to by the LWA.

Maps for the following control blocks also are loaded: **ASCB, ASXB, TCB** and **JSCB.**

Once you have used the READ command to process the TSOMAPS commands, you might then want to use the **LIST MAPS** command to see the result. Please note the following:

- **PECT** is the name of the map for the ECT pointed to by the PSCB.
- **LECT** is the name of the map for the ECT pointed to by the LWA.
- **PUPT** is the name of the map for the UPT pointed to by the PSCB.

Initially, the script assigns the structure to the Home Address Space. However, the entire structure can be moved at any time to label the TSO control blocks of any accessible foreign address space. Simply issue the following commands:

SET ASID userid

- Gains access to another TSO address space. Also, automatically reassigns the **@ASCB** built-in equate to label that address space's ASCB. This automatically causes the floating maps that are based on **@ASCB** to move to the new address space as well.

LIST TASKS

- Displays the new address space's subtask tree. Also, as a byproduct, redefines the **TCB#n** equates to label the TCBs in the new address space.

USING TCB TCB#3 F

- Reassigns the TCB map to the new address space's jobstep TCB. This automatically causes all of the TSO control block maps that are anchored out of the TCB map to move to the new address space as well.

The TSOMAPS script is distributed in the XDCCMDS library. The library's factory default name is hlq.XDCV31.XDCCMDS. Ask your Systems Programmer for its actual name at your Data Center.

Help SCripts Ourscripts XSBmaps

The XSBMAPS script defines a structure of floating maps for the XSBs ("Extended Status Blocks") that are linked to the RBs ("Request Blocks") that are queued for the current task. XSBs basically are nothing more than extensions to the RBs to

which they are connected. XSBs contain status information about the current program's Cross Memory Mode state as well as a linkage stack checkpoint pointer.

Once you have used the READ command to process the XSBMAPS commands, you might then want to use the **LIST MAPS** command to see the result. Note that enough maps are created to label up to nine XSBs.

Initially, the script assigns the XSB maps to the XSBs that are associated with the current task. However, the entire structure can be moved at any time to label the XSBs associated with any other task in either the Home Address Space or any accessible foreign address space. Simply issue a **LIST RBS tcb-addr** command directed towards the desired TCB. Example: **LIST RBS TCB#3**

The XSBMAPS script is distributed in the XDCCMDS library. The library's factory default name is hlq.XDCV31.XDCCMDS. Ask your Systems Programmer for its actual name at your Data Center.

Help SCripts Ourscripts XSVcmaps

The XSVCMAPS script defines a set of labels for the System's **SVC table** entries for SVC numbers 196, 197, 198, and 199. The script also builds a module map of the System Nucleus.

The labels that are created are named **S196PTR,** ..., **S199PTR.** A **FORMAT** command can be used to display the SVC routines themselves that are pointed to by these SVC table entries. Example: **FORMAT S199PTR?**

If Z/XDC is properly and typically installed, then **S199PTR?** will point to Z/XDC's hook support SVC routine, and **S198PTR?** will point to Z/XDC's services SVC routine.

If multiple releases of Z/XDC or multiple clones of Z/XDC are installed, then the additional SVC table slots also will point to various additional versions and copies of Z/XDC's SVCs.

There is, of course, no reason why you couldn't create a modified copy of this script with additional equates labeling whatever SVC routines are of particular interest to you.

The XSVCMAPS script is distributed in the XDCCMDS library. The library's factory default name is hlq.XDCV31.XDCCMDS. Ask your Systems Programmer for its actual name at your Data Center.

Help SCripts Ryoscripts

Scripts are easy to write. Just type them into any sequential file. Each record may contain...
- Either a **single** command,
- Or a string of **multiple** commands separated from each other by semicolons (;).

The following sections provide rules and information you need to be aware of when

writing scripts. More complex topics can be found in subtopics following this topic.

Dataset Types

As just noted, scripts are simple sequential files, so they can reside in pretty much any kind of dataset that permits QSAM I/O. This includes:
- Sequential datasets (DSORG=PS)
- Individual members of partitioned data sets (DSORG=PO)
- Individual members of extended partitioned data sets (PDSEs)

Generally, it's most convenient to gather your scripts together into libraries.

DCB Attributes

The DCB attributes of the dataset or library can be anything reasonable:

- **RECFM:** Pretty much anything is supported:
  - Fixed, variable, undefined
  - Blocked or unblocked

  It's all good.

- **LRECL:** Anything is supported that is long enough to hold the command strings that you want to use.

  Note, individual commands cannot be longer than 255 characters, but command strings (containing multiple commands) can be as long as you want.

- **BLKSIZE:** Anything that the System considers legal.

Command Abbreviations

There are exceptions, but Z/XDC's command abbreviation rules generally are that you must give as much of a command's name as is necessary to distinguish it from other command names. So for example, **G** could be either **GO** or **GETMAIN,** so at a minimum, you would need to use either either **GO** or **GE.**

Notable exceptions are:
- **F** for **FORMAT** not **FIND,**
- **L** for **LIST** not **LOAD,**
- But I digress.

However, when the abbreviation rules do apply, there is a risk that as we add new commands to Z/XDC, abbreviations that used to be legal, will no longer be legal.

So a good rule of thumb to follow would be to **avoid using abbreviations within scripts.**

Continuation Syntax

Script processing does not have any support for continuing long command strings onto multiple records. However, script files may have any LRECL, so long command strings can be accommodated.

Pro Tip

If you place a **DOWN M** command as a script's very last record, then when you run that script, you will be able to see immediately whether that script has succeeded or failed.

- For a successfully completed run, the **DOWN M** command will cause the display to be positioned at the **bottom** of all of the messages generated by the script.

- On the other hand, for an unsuccessful run, script processing will be suspended, so the **DOWN M** command will not have been executed, so the display will remain positioned to the **top** of all the script-generated messages.

An example of where the presence of a **DOWN M** command is particularly helpful can be found in our **SYSINFO** script. Take a look. It has commentary that explains everything.

Quirks Using C and X Strings in Your Scripts

Scripts do not have a generalized variables substitution support; but nevertheless, we have hardcoded a couple of quick and dirty substitutions for our own, in-house use.

If you happen to stumble upon these, you will be in for a surprise. To avoid surprises, here's what not to use in your scripts:

C - This will be replaced with Z/XDC's current clone name. At customer shops, that will almost always be **XDC.**

X - In house, we frequently use one clone of Z/XDC to debug another. So this variable will be replaced by the target clone's name.

In customer shops, on the other hand, this will most likely be replaced with the first three characters of the name of the program that you are debugging.

Sample Scripts

The Z/XDC product contains a large number of prewritten scripts ranging from very simple to mind numbingly complex. They serve as comprehensive examples of what can be done with sophisticated use of scripts as well as of Z/XDC commands. We strongly suggest you given them a look-see.

The scripts are each documented in the subtopics of **HELP SCRIPTS OURSCRIPTS.**

The scripts themselves can be found in the **hlq.XDCV31.XDCCMDS** library that is distributed as a part of Z/XDC. You will have to ask your Systems Programmer for its actual name at your Data Center.

Subtopics Index

Additional information regarding writing your own scripts is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially.
Use **HELP *FORWARD (F5)** to skip.

SEQUENCEFIELDS - Z/XDC's support for sequence number fields and the commands for managing that support.
CONDITIONALLOGIC - There is limited support for conditional logic within scripts.
LIMITATION - A limitation regarding GO, TRACE and END commands.

Help SCripts Ryoscripts Sequencefields

Where Sequence Number Fields Occur (within Scripts)

Sequence number fields are **8-character** fields that Z/XDC **needs to ignore.** They most commonly occur only in **FB-80** files; however, TSO ISPF's support is broader than that, so Z/XDC follows ISPF's rules:

- For RECFM=F files, they occur at the end of each record.
- For RECFM=V (and =U) files, they occur at the beginning of each record.
- Sequence fields are optional. They may or may not be present completely at the discretion of the person who coded the file.
- When **not** present, their content are simply considered to be a part of the rest of the record's data, and should be processed as such.
- When present, they are to be ignored.

What Sequence Number Fields Contain

When sequence fields are present, they usually contain 8 decimal digits, but nowadays with the advent of **git** and the decline of the importance of sequence numbers in modern Software Change Management methodologies, they can easily contain non-numeric and even arbitrary characters that the coder **might still** want to have ignored.

Sequence Field Detection

Z/XDC can be directed to attempt to automatically determine the presence or absence of a sequence number field, but when the field might contain non-numeric characters, making such a determination becomes programmatically hard.

When Z/XDC does attempt an automatic determination, it does so by examining the file's **first** record to see whether or not it meets any of the following criteria:

- First, the position of a possible sequence number field is determined according to the file's **RECFM** and **record length** as described above.

- If the first record's sequence field contains **8 decimal digits,** then the sequence number field is declared present and will be ignored on all records.

- If the file is a **member of a PDS** and the first record's field's upcased content **matches the member name,** then the sequence number field is declared present and will be ignored on all records.

- If the file is a **sequential dataset,** and the first record's sequence field's upcased content matches the dsname's **2nd-to-last** qualifier, then the sequence number field is declared present and will be ignored on all records.

- If the first six characters of the first record's sequence field's upcased content match **CL6'IGNORE',** then the sequence number field is declared present and will be ignored on all records.

- **Otherwise,** sequence number fields are considered **absent** from all records, and all content in the sequence field position are processed as part of the record's normal data.

Who Determines Whether Sequence Fields are Present - You or Z/XDC?

Z/XDC does the best it can to help you out with sequence fields, but in the end there may be cases where it gets it wrong. So we provide a couple of ways for you to override Z/XDC's automatic detection process and simply tell Z/XDC when sequence fields are present or absent.

Both the **READ** and **SET READ** commands accept a **SEQFIELD=** operand that allows you to manage Z/XDC's sequence field detection policy:

- **SET READ SEQFIELD=** allows you to set a default policy that will be saved in your personal Session Profile.

- **READ scriptname SEQFIELD=** allows you to override your default policy for individual scripts.

- The default policy can also be displayed and set via:
  - The **Profile Menuing System,**
  - The **LIST READ** and **SET READ** commands.

The SEQFIELD= Operand Syntax

```
SET READ [...] SEQUENCEFIELD=PRESENT
               SEQFIELD=     ABSENT
                             DETECT
```

SEQFIELD=PRESENT
SEQUENCEFIELD=PRESENT
The sequence field is considered to be present in all records regardless of content. The field will be ignored in all records.

SEQFIELD=ABSENT
SEQUENCEFIELD=ABSENT
The sequence field is considered to be absent from all records. Whatever content is found in the sequence field position will be considered to be part of the data found in the rest of the record.

SEQFIELD=DETECT
SEQUENCEFIELD=DETECT
Z/XDC itself will determine the presence or absence of sequence fields based upon its examination of the file's first record. The determination will be made according to the rules discussed above.

SEQFIELD=DETECT is the factory default.

Help SCripts Ryoscripts Conditionallogic

Scripts do not have support for Conditional Logic. There is no support for...
- Symbolic Substitution
- For IF-THEN-ELSE
- For DO-END
- Not even for GOTO

Scripts are nothing more than simple sequential files that are processed from beginning to end.

Leveraging Error Processing into Conditional Processing

What little conditional control there is arises from the way in which command errors are processed. Here's how it works:

- Each record within a script contains a command string which may consist of one **or more** individual commands (separated from each other by **semicolons** [;]).

- When a command fails, the following things happen:
  - Execution of the script is suspended.
  - The script file remains open and positioned.
  - But Z/XDC's command handler reverts to accepting commands from your terminal.
  - The command string that was being processed is discarded.
  - And the script file is positioned to the next following record.

This means that all commands that follow the failing command **and are in the same record** are discarded. So if you should decide to resume the script, resumption commences with the file's next following record.

Example

Here's an example of where this is useful:

- Suppose a script contains a **ZAP** command.
- But you need to verify the contents of the zap target before actually doing the zap.

- So you need to add a **VERIFY** command to the script.

If you combine the **VERIFY** command and the **ZAP** command into one command string so that it can be placed into a single record, then should the **VERIFY** fail...

- The remainder of the command string (i.e. the **ZAP** command) would be discarded,
- And error messages would be displayed,
- And the keyboard would be unlocked so that you could type in new commands,
- **But** the script file would remain OPEN'd and positioned to the next record following the **VERIFY;ZAP** record.

You could then look around and manually enter a corrected **ZAP** command, and then issue a **READ** command (without operands) to resume execution of the remainder of the script.

Help SCripts Ryoscripts Limitation

A Limitation Regarding GO, TRACE and END Commands

One limitation about scripts is that the script file is always forced **closed** whenever Z/XDC is about to lose control, i.e. whenever you issue a **GO, TRACE** or **END** command.

Basically this means that if a script contains a **GO, TRACE** or **END** command, said command must be in the last command string in the file. All following records are discarded. (They are not saved for processing later.)

Daisy-Chaining Multiple Scripts Together

There is a workaround: **GO, TRACE** and **END** commands that are embedded within **Automatic Command Strings** associated with **traces** and **breakpoints** are not parsed at the time the script is executed. Their parsings occur later when the breakpoints to which they are assigned are reached by program execution.

It is perfectly OK for one script to end with a **READ** command that causes processing to continue in another script. That's OK... but boring.

But if you bury **READ** commands within a breakpoint's Automatic Commands, you can daisy-chain together one script after another in far more creative ways. For example, you can have different scripts run depending upon which breakpoints are

reached by execution.

Further, using **conditional** breakpoints, you can even place multiple breakpoints at a single address...

- Each with a different **READ** command as its Automatic Command String,
- And each with a different conditional expression.

Then when execution reaches the breakpoints, some READs will run, and others will not, depending upon which conditional expressions resolve TRUE and which resolve FALSE.