colesoft™

ZXDC®

Z/XDC PRIMER

for Release z2.2 for z/OS

Bob Shimizu

ZXDC®

Contents

Preface **4**

Chapter 1: What is z/XDC? **1**

What is z/XDC's scope? 1

What makes z/XDC so useful? 1

Be careful with z/XDC 2

How to obtain Technical Support: 2

The z/XDC Story 3

Resources for this Primer and for z/XDC itself 4

Chapter 2: Invoking z/XDC and Understanding the Display. **5**

A Simple Way to Start z/XDC 5

Ending a z/XDC Session 8

Starting an APF-authorized session with z/XDC 8

The z/XDC Startup Panel 9

The Profile Facility 14

The z/XDC Session Log 17

Chapter 3: Looking Around **18**

Address Expressions 22

Shortcut Commands 23

Point-and-Shoot Capabilities 24

Back to Looking Around – the LIST Command 25

LIST PSW 26

LIST RWEGS 26

LIST FREGS 26

LIST AREGS 27

LIST CREGS 27

LIST VREGS 27

LIST TASKS 30

LIST RBS 30

LIST EQUATES 32

LIST SUBPOOLS 33

LIST PGMS 33

LIST RSA 34

LIST TIOT 35

LIST ESTAE 35
LIST LSTACK 36
LIST FEATURES 36
The FIND command 37
SCANLOG 39

Chapter 4: Mapping 40
Viewing XDCSYMED Without Any Map 40
Viewing XDCSYMED With an ADATA Map 41
Scripts, and z/XDC's READ command 47
The REXX interface 48
Mapping DSECTS - the DMAP and USING commands 48

Chapter 5: Breakpoints and Traces 55
Using X'00' or TRAP2 Instructions for Breakpointing/Tracing 55
The Breakpoint Commands 56
The Tracing Commands 60
Conditional Statements 62

Chapter 6: Changing Things 70

Chapter 7: Working with Other Address Spaces 80
Foreign Address Space Mode 80
How To Hook a Running Program with Foreign Address Space Mode 82
The Cross Domain Facility 85

Chapter 8: Help 89
Accessing z/XDC's Built-in Help 90
Useful Help Topics 93
Accessing z/XDC's Messages 97

Revision Date: 4th April, 2025 In this edition updated Preface

Preface

PROPRIETARY LEGEND

z/XDC® and its documentation (collectively, "Product"), including copies thereof, are the property of Mainchord, LLC 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 The Product may not be published, reproduced, distributed or made available to third parties for any purpose without the expressed written permission of Owner or Licensor.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:	**800-XDC-5150**
	540-456-8210
E-Mail:	sales@colesoft.com
Home Page:	http://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:	**800-XDC-5150**
E-Mail:	support@colesoft.com
Home Page:	www.colesoft.com

Our snail mail address is:

Address: **ColeSoft Marketing, Inc.**
414 3rd Street NE
Charlottesville, Virginia
22902
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 Mainchord, LLC.

Extended Debugging Controller®, **XDC®**, and **z/XDC®** are registered trademarks of Mainchord, LLC.

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.

How this booklet is structured

Throughout the discussion we will assume a correct installation of the product. If you run into a problem with the installation of the product, we will do all that we can to assist your systems personnel. I can be reached directly at (800) 932-5150. In general, our discussion will address the following topics.

I. What is z/XDC's Architecture?
II. Invoking z/XDC and Understanding the Display
III. Looking Around
IV. Mapping
V. Breakpoints and Traces
VI. Changing Things
VII. Working with other address spaces
VIII. Help
IX. Conclusion

That's where we're going. Let's get to work.

Chapter 1: What is z/XDC?

Briefly, z/XDC is a very large Recovery routine that gets control after any abend occurs in a target program. z/XDC can run as an ESTAI, an ESTAE, an ESTAEX, and ARR or even an FRR. Most often, however, you will employ z/XDC as an ESTAEX to your base code.

All the breakpoints and traces that you initiate in your program cause z/XDC to place a X'00' into the object code. This causes S0C1 abends in your program which z/XDC then processes. Alternatively, you may choose to employ TRAP2 instructions to cause z/XDC to get control.

The classic, time-honored way to investigate a broken piece of code is to trigger an abend in the program, generate the dump and read the dump. Instead of creating one huge monolithic dump (which might NOT capture the problem) z/XDC allows you to create as many “mini-dumps” as you like. You can navigate through your program with source statement displayed, watching as it executes and setting more traps with z/XDC’s trapping and tracing commands. You can interrogate the contents of memory using point and shoot commands to follow pointer chains or use the contents of registers to see where they point to.

What is z/XDC’s scope?

z/XDC works on raw object code, allowing you DISPLAY it in hexadecimal format or to FORMAT it as machine instructions, which invokes z/XDC’s on-board dis-assembler. z/XDC’s set of LIST commands allow you to quickly see z/OS-based internal structures. The TRAP and TRACE commands allow you to execute portions of or the entirety of a program. The ZAP command allows you to alter the contents of memory – making it possible to do unit testing on code sections – a what-if investigation of a program’s execution.

If you can understand the programming environment in terms of a disassembly, then you could work on ANY object code, including compiled high-level languages such as C or COBOL. You can even use z/XDC on code for which you have NO source code. This is particularly useful in understanding the flow of IBM’s z/OS operating system itself.

If you invoke z/XDC as an APF-authorized program, you can use it to debug system modifications and exit routines. You can use it to work on programs that create and use multiple address spaces. z/XDC can be used on multitasking programs, for the development and maintenance of SVCs, Program Call routines, Task-mode code, SRB-mode code - just about anything you can conceive of in the z/OS operating system whether it runs in 24-, 31- or 64-bit addressing mode.

z/XDC has exceptional scope, or range. If I had to write all of z/XDC’s capabilities in a single sentence I’d write this: *“Using z/XDC, a knowledgeable programmer may more quickly find and fix bugs in any arbitrary object code within any csect within any module running under any Request Block under any Task Control Block or SRB in any address space on the system, including much of the internals of the operating system itself.”*

What makes z/XDC so useful?

Among many other things, z/XDC allows you to drop your source code statements directly over object code. This way you can view your original opcodes and comments – a very useful resource for people

who are assigned to maintain code they didn't write.

z/XDC allows you to direclty SEE structures that before you had to deduce. This will save you the time and energy it takes to remember offset addresses, do conversions or hexadecimal arithmetic in your head. z/XDC saves you mental overhead, allowing you to think about a problem in a strategic manner.

z/XDC is a brain amplifier. It allows new Assembler language programmers to come up to speed much more quickly. z/XDC makes experienced Assembler language programmers MUCH faster!

z/XDC does not simulate execution, but lets your program execute its instructions within its normal running environment. This very generalized interface, along with the ability to execute in authorized environments, allows z/XDC to be used in a great many different situations. You can use z/XDC anywhere you can gain control via the ESTAI or ESTAE macros (or other abend handling mechanisms). For a much more detailed discussion of where and when you can use z/XDC, please refer to the z/XDC User's Guide (or the Help Subsystem) in the sections marked HELP DEBUGGING. That's a reference on how to deploy z/XDC in various environments.

Be careful with z/XDC

You'll find that z/XDC is amazingly versatile. When running z/XDC APF-authorized you can go nearly anywhere you like in the operating system (if your security system will authorize you to do so). Using z/XDC you can find and fix bugs in JES, CICS, VTAM, DB2, CPCS, – even in system modifications and exits. HOWEVER – please be careful, because z/XDC is the *ultimate power tool for serious coders*. If you set a breakpoint in a shared piece of code (assuming that your session is authorized to do so), then you may get exactly what you asked for – a breakpoint in EVERYBODY'S shared routine. That is not necessarily a good thing. So, be cautious, and start out working on code in your own TSO address space. Then you will only be able to destroy your own address space. Later, it might be a good idea to obtain your own local copy of a system routine if you want to work on it.

z/XDC for all its power, cannot do some things. Because z/XDC itself is a recovery routine, one can't generally debug z/OS retry structures (though even this could be possible with our help). If you are working with programs that utilize the display a lot, then z/XDC and this program might end up competing for the display (There are work-arounds for this, too).

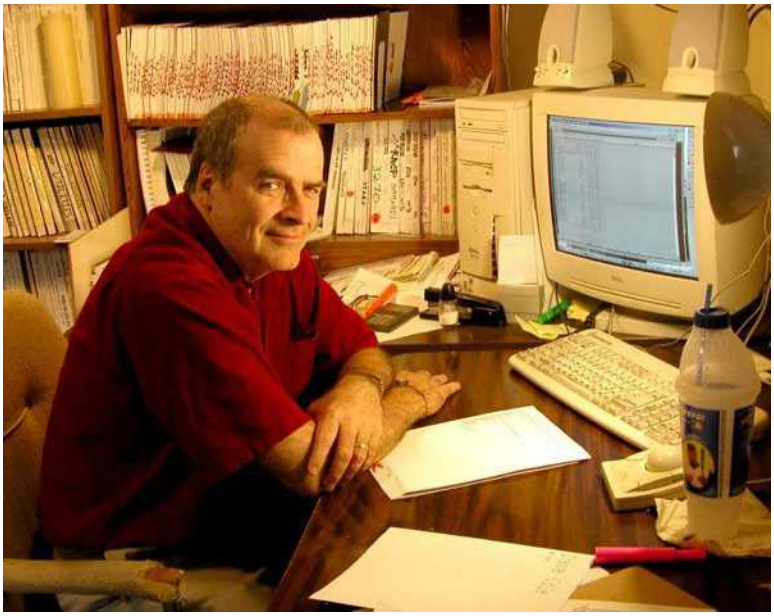
If you have esoteric problems to use z/XDC on, then we welcome your questions. Dave Cole and his team have put a great deal of thought into z/XDC over the decades, and we may know a couple of tricks that will help you in your own pursuits. That brings me to our support structure.

How to obtain Technical Support:

For Technical Support service, please contact us as follows:

- Our web site is available 24/7 at http://www.colesoft.com. Check out the z/XDC Support area.
- For Level One support and product training, please contact me (Bob Shimizu) at the following coordinates: bshimizu@colesoft.com (800)XDC-5150 or (928)771-2003.
- For Level Two support, please contact Frank Chu at frank@colesoft.com or (540) 456-6164.
- For level Three support, please contact David B Cole at dbcole@colesoft.com or (540)456-6518.

The z/XDC Story



The z/XDC story began with its author, David B. Cole, owner and founder of ColeSoft Marketing, Inc. Dave has nearly 50 years of programming experience in z/OS (and predecessor) systems with concentration in the Assembler and z/OS internals. During his career, Dave has had the opportunity to work on nearly every part of the z/OS operating system and has earned a reputation as an authority in this field.

It was during his employment at Yale University that Dave first conceived of what z/XDC is today - a powerful development tool for Assembler Language programmers. The product's design and coding began circa 1978.

The first predecessors to XDC were simple trapping and zapping utilities that evolved along with Dave's career in systems programming.

The first release of the DeBugging Controller (DBC) became available to selected users by word of mouth. Several prominent software development houses were among the product's early customers. It was the needs of these sophisticated users that helped drive the evolution of the product from its inception.

In 1988, Dave bought out the rights to XDC from Yale and formally started Cole Software®, now called ColeSoft Marketing, Inc. Thus, the product actually pre-dates the company that supports it today.

z/XDC's user community is small, but it is highly sophisticated and represents some of the best minds in the software industry. Our users are among the industry's foremost software development houses and Fortune 500 companies.

z/XDC's greatest strength is the continuous refinement it derives primarily from a single, dedicated programmer and his team. The product's users also have a great deal to do with z/XDC's future direction.

Dave built z/XDC on simple precepts: target a specific user group, continue to evolve to meet the needs of the users, and provide excellent support.

Dave loves to code, and it shows! He truly believes that the existence of z/XDC has helped contribute to the continued usefulness of Assembler Language as a modern program development tool. Just as MVS/SP has evolved over the years to become z/OS, so has ColeSoft Marketing's product evolved from the original DBC to today's z/XDC.

Dave Cole is the planet's leading authority on this product. He's able to answer authoritatively on other matters as well, such as the Assembler and the operating system. Dave serves as an informal clearing-house on these matters, and many people call him just to ask how the operating system behaves in a certain situation. Dave doesn't claim to have ALL the answers, but he may be able to point you towards a useful reference. We welcome this exchange of ideas, but please frame your questions carefully. Dave is always busy turning out the next release of the product!

We maintain a web site at https://www.colesoft.com. Our site will give you good overview information on the product. You may also download your own copy of our product's documentation. In an emergency you can create your own short-term activation codes for the product. There are links to Dave's youtube training films. You can find hard information at our website - not just marketing material.

Resources for this Primer and for z/XDC itself

Throughout the Primer we will run a debugging session on a "toy" program that we distribute with the product called XDCSYMED, which resides in z/XDC's XDCLINK library. This program has been assembled and link-edited with PARM TEST (for our SYMBOL record support) and also in ADATA format in order to demonstrate our full source statement support.

You will need to know the name of the dataset into which your systems programmer has put the XDCSYMED module. Typically, the low-level qualifier for that dataset is "XDCLINK". You will also need to know the name of the dataset into which your systems programmer has put our ADATA library. That dataset will be necessary in order to demonstrate our source statement support and for accessing the XDCMAPS module (used for demonstrating dsect mapping). Typically, the low-level qualifier for that dataset is XDCADATA. Check with your installing systems programmer for specifics.

As I present you with z/XDC's screens, I'll show you the syntax of each command in the command line AND its corresponding output. *That's different from z/XDC's true action* because, in normal operation, as soon as a command is processed the command line is cleared. I want you to see the command's syntax AND the result all in one place.

I will often abbreviate the displays by cropping them to show you only the relevant portion of a screen. There's no point in wasting space, especially if you decide to print this document (which is in color).

Finally, not all the commands are presented in a strict order. I tend to jump in and out of HELP, mostly to show you how easy it is to reference the product while you are in it. Also, and especially in the case of the LIST command, I'll introduce new commands in logical process order. If that offends your sense of organization, you can always look in the z/XDC Commands Reference for the commands in a strict, alphabetic progression.

Complete documentation for the product is available at the colesoft.com website, including the z/XDC Quick Reference which you may find useful.

In recent years, Dave Cole has recorded a series of useful training videos for z/XDC which you can access by going to www.youtube.com, and searching for "ColeSoft Marketing".

We have a lot of ground to cover. Let's get started.

Chapter 2: Invoking z/XDC and Understanding the Display.

A Simple Way to Start z/XDC

z/XDC can be invoked in several ways, but the simplest way is to invoke the XDCALL command from the TSO READY prompt. The command syntax is XDCCALL <program-name>. When you invoke z/XDC in this way, it will ATTACH your program as a sub-task, and specify itself as the ESTAI (the abend handler) for <program-name>. It will then invoke a #DIE macro, which is a stylized S0C1 abend with a WTO message. The program will go into a wait at the entry point of the program.

We'll pick a program you're likely to have on your system, like module IEFBR14. Try this command:

XDCCALL IEFBR14

z/XDC presents you with a screen that looks like this:

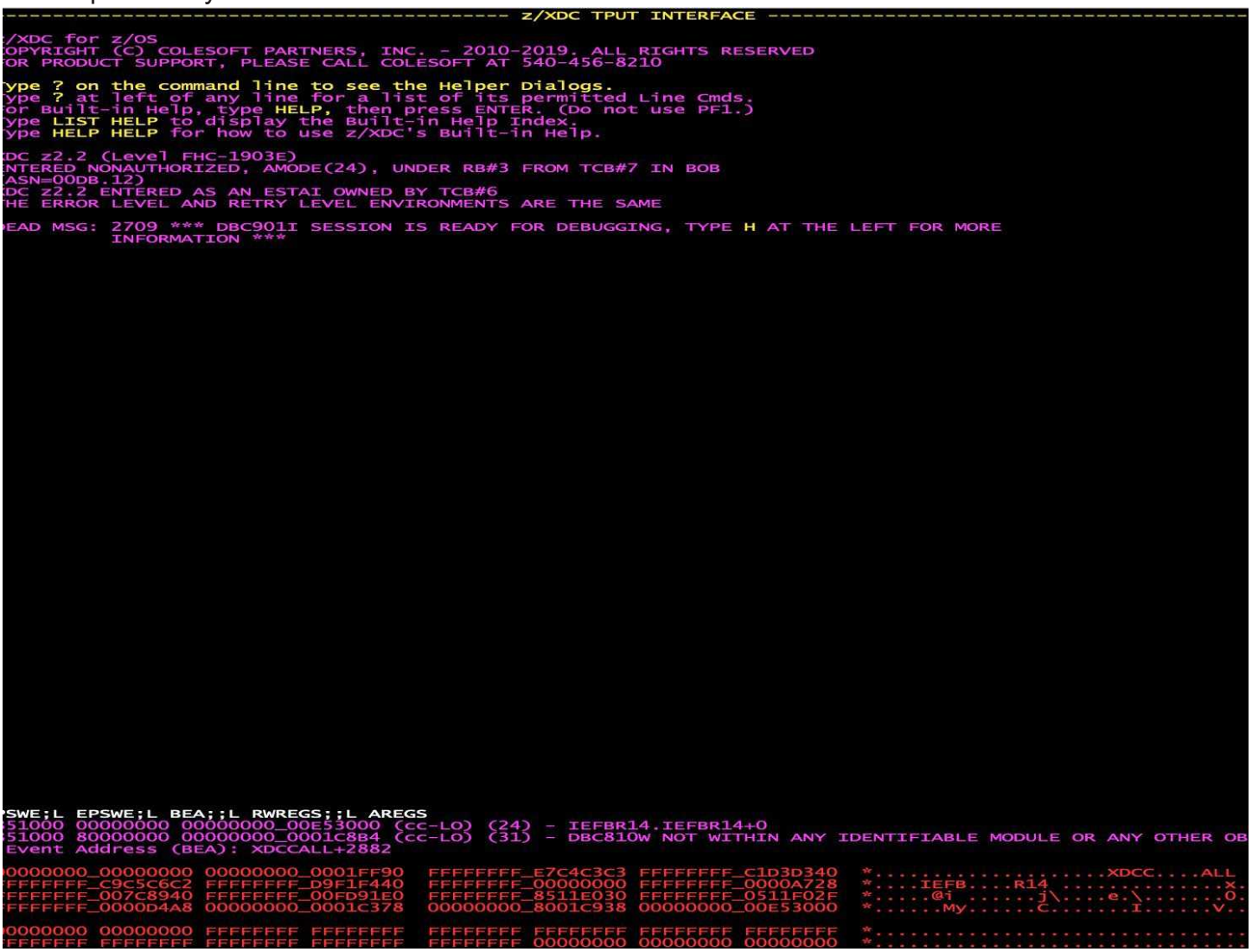


Figure 2-1.

OK, cool. We got a z/XDC session. Now, what the heck are we looking at?

Anatomy of a z/XDC Panel

On your 3270 display you have a line of information at the top. Right now z/XDC is using TSO TPUTs and TGETs to write the screens. If you were to run z/XDC under ISPF, you'd see z/XDC ISPF Interface.

- Under the ISPF Interface you have access to SPLIT and SWAP, but you may not execute APF-authorized. This is an ISPF restriction.
- Under the TPUT interface you do not have access to SPLIT and SWAP, but you may debug APF-authorized programs.
- If you use z/XDC in Batch mode under the Cross Domain Facility you will be able to use ISPF whether you're APF-authorized or not. Nice.

Anyway, moving down the panel you see one command line, some text and then another command line and more text. You're looking at two "windows".

The top window is the Working Window. You have to have that. It can't be deleted. You'll use the Working Window to enter the bulk of your commands, and here is where you will see the code you're investigating.

At the moment, z/XDC is showing you three paragraphs of information that you may learn to ignore. Don't do that. Embrace these messages and learn to understand them. I'm going to explain them, but you may always get definitive information by putting an "H" next to each of the messages, thereby obtaining Dave Cole's own explanation.

First paragraph: z/XDC tells about our copyright information, and almost BEGS you to call our phone number for support.

Second paragraph: z/XDC has "Helper dialogues" for some of the more complex commands such as DMAP, FIND, MAP and USING. The Helper dialogues allow you to fill in values rather than having to remember all the command parameters. They make things a lot easier.

z/XDC is also trying to urge you to use HELP or LIST HELP (sort of a HELP "site-map") in order to learn more about the product. We'll be in and out of the HELP system throughout this Primer.

Third paragraph: z/XDC tells you about your local environment:

You're running z/XDC Release z2.2 at maintenance level FHC-1903E in non-authorized mode, in 24-bit addressing under the third Request Block under the seventh Task Control Block in the BOB address space.

The XDCCALL command is running as the ESTAI for the IEFBR14 program and is located in Task number six in the address space.

Error Level vs. Retry Level Environments

You also see message "DBC83I THE ERROR LEVEL AND RETRY LEVEL ENVIRONMENTS ARE THE SAME", or "DBC954W THE ERROR LEVEL AND RETRY LEVELS ARE *DIFFERENT*!" What's that all about?

z/XDC maintains TWO environments for the user:

- The Error-level environment consists of the Request Blocks, Linkage Stack entries, PSW, general registers and the access registers that existed at the point the abend occurred. Sometimes this will be an IBM routine that was invoked by the user's program and that detected a problem severe enough that it forced an abend. The PSWE and EREGS constructs come from the errror-level environment.
- The Retry-level environment consists of the Request Blocks, Linkage Stack entries, PSW, general registers and the access registers that will exist when you resume the program via GO, GOT, GOX or TRACE commands. The PSW and REGS constructs from the retry-level environment.

Generally, when the error level and retry level environments are the same, that's a good thing. When they are different, that's often (but not always) a bad thing. Why? Because when they are different it is not possible to resume executing the same routine that abended.

When the error level and retry level environments are different you must take the time to understand why they are different and decide what to do about it. For more detailed information, start a z/XDC session and enter HELP EXECUTIONLEVELS.

z/XDC gives you the output of its own #DIE Macro (which is a stylized S0C1 abend with a Write-to-Operator message). What you see here is "DEAD MSG:" If you care to, you can enter an H next to this message to learn more about this message and what you might care to do next.

That was JUST the Working window. Now for the next window.

The bottom window is a Watch Window". You can have zero, or one or many of these kinds of windows. They're used to keep track of data areas, other sections of code - anything you like. Any command(s) you enter into a Watch Window are saved and re-evaluated as the code runs in your Working Window.

In our factory default settings in Figure 1 we see:

- A LIST PSWE command (the error level PSW; the one that existed at the time you got control);
- then a semi-colon (our command delimiter);
- then a LIST EPSWE (the extended PSW in it's full 16-byte format);
- another command delimiter (a semi-colon);
- the output of the LIST BEA command (the breaking event address - effectively "how we got here");
- two semi-colons (to create a blank line for readability);
- a LIST RWREGS command (to show the entirety of the 64-bit registers);
- two more semi-colons;
- and finally a LIST AREGS command (to show the Access Registers).

That is a lot of information to have without much effort on your part.

z/XDC Prefers Wide Terminal Geometries

Over the years, z/XDC (and the rest of the world) prefers wider terminal geometries. Now it's just *better* to define a 60 X132 terminal geometry in your 3270 emulator. However, those teensy characters make it difficult for me to render a legible screenshot in this booklet. So, for the purposes of this Primer, I've defined a terminal geometry of 43 X132, which is more legible but distinctly strange.

OK. We've given you the anatomy of a basic z/XDC panel in exquisite detail. However, IEFBR14 isn't all that interesting. Let's quit this session. So, how do you get out of a z/XDC session?

Ending a z/XDC Session

You have three options: END, END NOASK and END COMPLETELY. If you enter the END command without parameters, z/XDC will prompt you to confirm that choice. In effect, z/XDC is asking, "Do you really mean to end your session?" That's because you may have inadvertently pressed PF-3 (which is usually set to END). You may have maps defined, dsect maps defined, breakpoints all over the address space, equates, etc. In other words you do NOT want to end your session. In this situation you can enter Y to close the session or N to keep it.

If, on the other hand, you DO want to end the session you can force z/XDC to stop the session by using the END command with a NOASK operand. I call this option the "End, shut up, I know what I'm doing option".

You can also create debugging sessions within debugging sessions, because z/XDC dutifully will trap all the abends that it encounters. ENDing one session merely brings you back to the next most recently invoked session and next you must END that session. To bypass this sequence, you can issue END COMPLETELY. That will take you totally out of ALL your debugging sessions.

There are many more parameters on the END command. To learn more about them, enter HELP COMMANDS END.

Now that you know this distinction, issue END NOASK, and return to TSO READY.

Starting an APF-authorized session with z/XDC

If you wanted to, you COULD start z/XDC APF-authorized in your TSO region. To do that you issue:

XDCCALLA IEFBR14

If you do that, then the Working Window's third paragraph tells you that your session is AUTHORIZED. This is extremely powerful and cool - IF YOU KNOW WHAT YOU'RE DOING.

At first you may wonder why you'd want to run an authorized debugging session against boring old IEFBR14. Dig this, however: You're just using IEFBR14 for convenience. You could just as easily use IEBGENR instead. *IF your security system will let you debug in APF-authorized mode you can now go anywhere/do anything.* You could use Foreign Address Space mode to debug a system exit or an SVC, or place a HOOK in another address space and "hi-jack" it for debugging purposes.

We'll discuss using z/XDC in APF-authorized mode later on. *Don't try this yet* unless you have your own test-only system. We don't want you to limit your further career development by crashing your production system while discovering z/XDC's juicy wholesome goodness! And in any case, if you DO crash a system remember that WE didn't press the Enter key! **You did**.

The z/XDC Startup Panel

Now you've learned that you can run z/XDC from a TSO READY prompt in both non-authorized and APF-authorized mode. But there is something far more convenient. To make things even easier, we provide the z/XDC STARTUP PANEL, which should have been installed into your ISPF panel libraries. Ask your installation personnel how to access this panel. In my world it's "ISPF D".

The z/XDC Startup Panel is a generalized launching pad for using z/XDC in a number of ways and in a number of environments. Here is what the panel looks like.

Figure 2-2.

The Startup Panel allows you to do many things, some you'll use every time, and some that are more esoteric. We're going to limit our exposure to esoterica in this manual, so forgive us if we don't go into excruciating detail on each of them.

First, the command line: You can do four things here, but let's begin with the most obvious stuff. You can enter the integers 1, 2 or 3.

Option 1 sets up a debugging session for a TSO Command you may be writing. In this situation, z/XDC expects Register 1 to point to your command's parameter list.

Option 2 sets up a debugging session in your TSO region. Both Options 1 and 2 give you more parameter fields that you can fill in in the second "paragraph" of the panel.

The second paragraph is headed with, PARAMETERS FOR OPTIONS 1 OR 2, etc. It contains these fields, which pertain ONLY to debugging in your TSO Private Area (not a batch program):

- Does CMG/PGM use ISPF services? Use YES if they do, and NO if they don't.
- Dialog APPLID: all ISPF apps have SOME APPLID, and that is how they maintain their own environments (for PFKey settings and much more). Since you're working with the TFS component in z/XDC, leave this alone. This facility was implemented originally to allow people to work on ISPF dialogues with z/XDC. It has since been supplanted by Cross Domain Facility (which we'll get to in a bit), so for all intents and purposes, you may ignore it.
- Should CMD/PGM start APF auth (orized)? Use YES, if you want this and NO if you don't.
- Quick Start: The default is NO, but if you say YES, then z/XDC will allow your program to run until either it abends on its own, or executes a z/XDC HOOK.
- Should AUTOSTEP be allowed?: In c/XDC leaving this set to YES will allow c/XDC to skip the prologue, getting the user right to MAIN().

- Should RENT and REFR pgms be loaded into key-8 storage? Leave this YES if you want to more easily debug reentrant and refreshable programs.
- Upcase the parm? Leaving this set to yes will alter parameters to upper-case
- CMD/PGM Name, Tasklib DSN's and/or DDNAMES. Fill these values in as necessary.

Option 3 sets up a connection to the Cross Domain Facility, which is how you connect to a batch job to debug it. In reality, you'll use this Option far more often than Options 1 or 2.

When you input 3 on the command line and press Enter, you'll be directed to the Cross Domain Facility Router, which displays a list of batch jobs that are waiting to be debugged. We'll talk about Cross Domain Facility in a later chapter.

Here's the fourth thing you can do on the command line: To the right of the command line at the top of the Startup Panel, we have XDC's name = = =>. It's normally set to XDC. This field allows you to input a "clone" name for z/XDC. What the heck is a clone name? Glad you asked.

Your installation may have a few different versions of z/XDC available, and your installing Systems Programmer may choose different 3-character names for them. In our shop, z/XDC Release z1.13 is called Z1D. For z/XDC Release z1.10 we used Z1A. Our production z/XDC receives a clone name of XDC. The version of z/XDC that we're developing usually has a clone name of XXX. The clone name is a convenience. Most of the time you won't have to change its value.

You can change z/XDC's clone name either on the command line or in the XDC's name = = => field.

OK, that was your tour of the z/XDC Startup Panel. Now, I need you to fill this panel in with some values.

Viewing A Program That Has No Mapping Information

If z/XDC senses that it has arrived at a module/csect for the first time z/XDC now automatically attempts to map your code. That is a HUGE time-saver IF you have maps for your code. The relevant twiddly-bit to influence this behavior is the "SET AUTOMAP YES/NO" command. You can also influence the setting by entering PROFILE and selecting the bottom option (Display/Change AUTOMAP PRINT and Other Settings). The default behavior is SET AUTOMAP ON.

However, the AUTOMAP capability doesn't let you see z/XDC's abilities to disassemble arbitrary object code, so please bear with me. Before we get to debugging XDCSYMED, our toy program, I'm going to digress and ask you to debug IEBGENR instead.

On the z/XDC Startup Panel, enter a 2 in the top command line, and tab to the first instance of "CMD/PGM Name ===>". Enter IEBGENR, and press Enter. You should see a screen like the one below, in Figure 2:

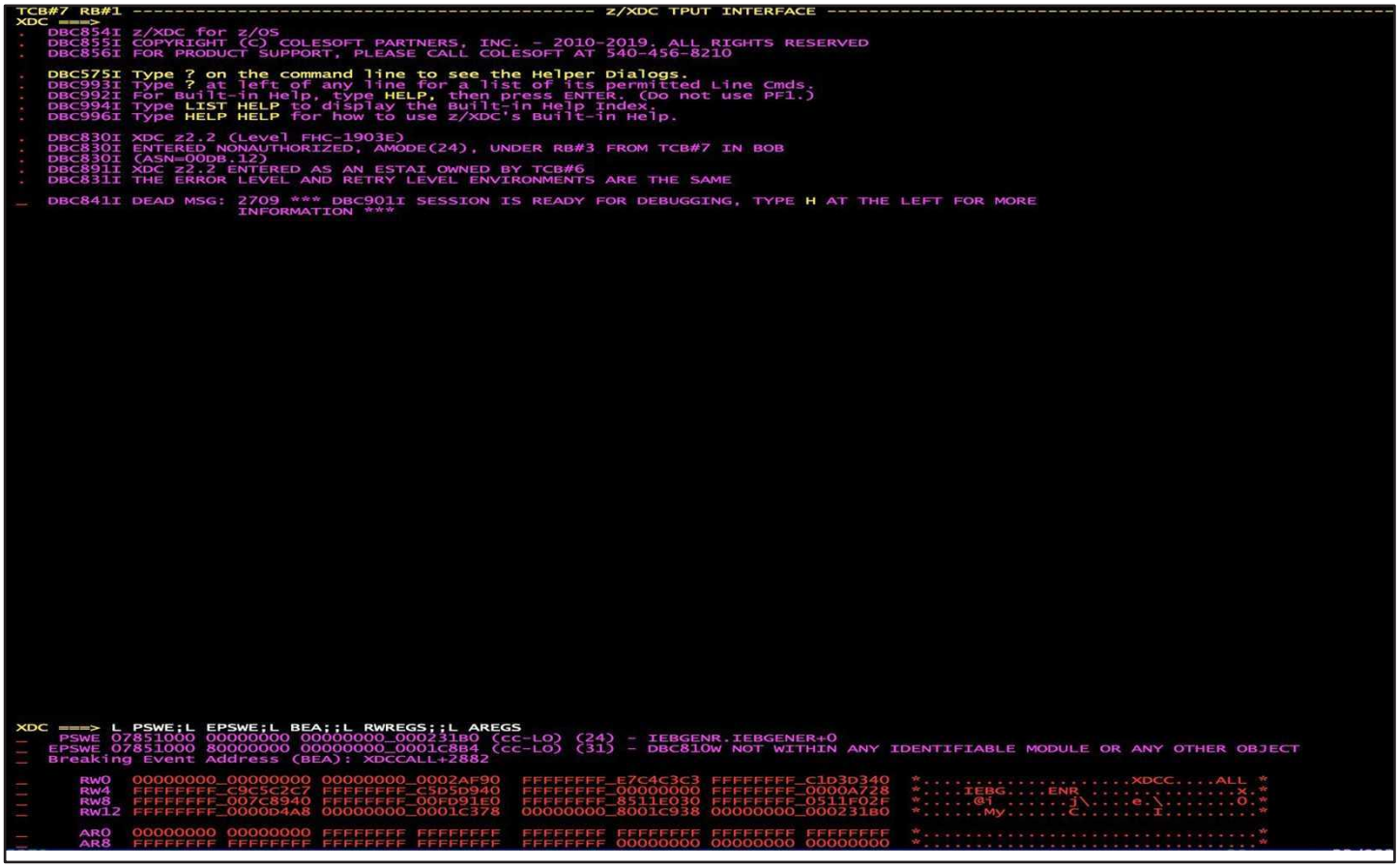


Figure 2-3.

I'm going to explain what we see later on in the Primer, but for now I want to show you z/XDC's utility when looking at un-mapped code. So, just follow along with me trustingly for now, and I'll explain it all further on.

Please enter WHERE in the command line, or merely “W” and press Enter. You’ll see something like Figure 4 below:

Figure 2-4.

In Figure 4, z/XDC automatically issues a FORMAT (meaning, “disassemble”) command for you and shows as much as it understands before it reverts back to DISPLAY mode (meaning, “dump”). Thus we can see a small portion of the beginning of IEBGENR. However, if you issue T BY (T-space-BY) and then T, you’ll see a good deal more of the raw object code within IEBGENR, disassembled nicely.

I've just done the T BY and the T commands for you. See Figure 5 below:

Figure 2-5.

This points out one of z/XDC's great strengths: You can work effectively without any map for the code you're interested in. This allows you to work on system routines and exits, or any program that comes to you in Object Code Only format. You can discover undocumented interfaces, and prove to yourself exactly what's going on.

That said, most people vastly prefer to work with some sort of mapping, and z/XDC can deliver that in two ways: You may use SYM records OR (and you'll probably prefer this) complete source statement support via ADATA.

Viewing a Program That Has Mapping information

If you've been following along with me you can now issue END NOASK to terminate the debugging session on IEBGENR, and you'll go back to the z/XDC Startup Panel.

For the rest of this Primer, we're going to debug an arbitrary toy program that we supply with the product called XDCSYMED.

Please put a 2" into the command line. Then, tab down to the first instance of the "CMD/PGM Name ===> field and enter XDCSYMED. If the XDCSYMED program, (typically the librry with a low-level qualifier of "XDCLINK") are NOT within your TSO region's search order, then you need to enter that dataset name into the "Tasklib DSNs" fields. I don't know what those datasets will be in YOUR system, but in mine, the XDCSYMED program comes from DBCOLE.XDCZ21.XDCLINK. Then, press Enter.

Well, we did it: We have you on the first panel of a z/XDC debugging session. We are going to debug XDCSYMED for the remainder of this Primer. Be advised that ALL of what we do in our own TSO region is equally applicable to a batch environment.

Here we are at Figure 6. Next, it's time to show you how you can configure your own screen layouts in z/XDC.

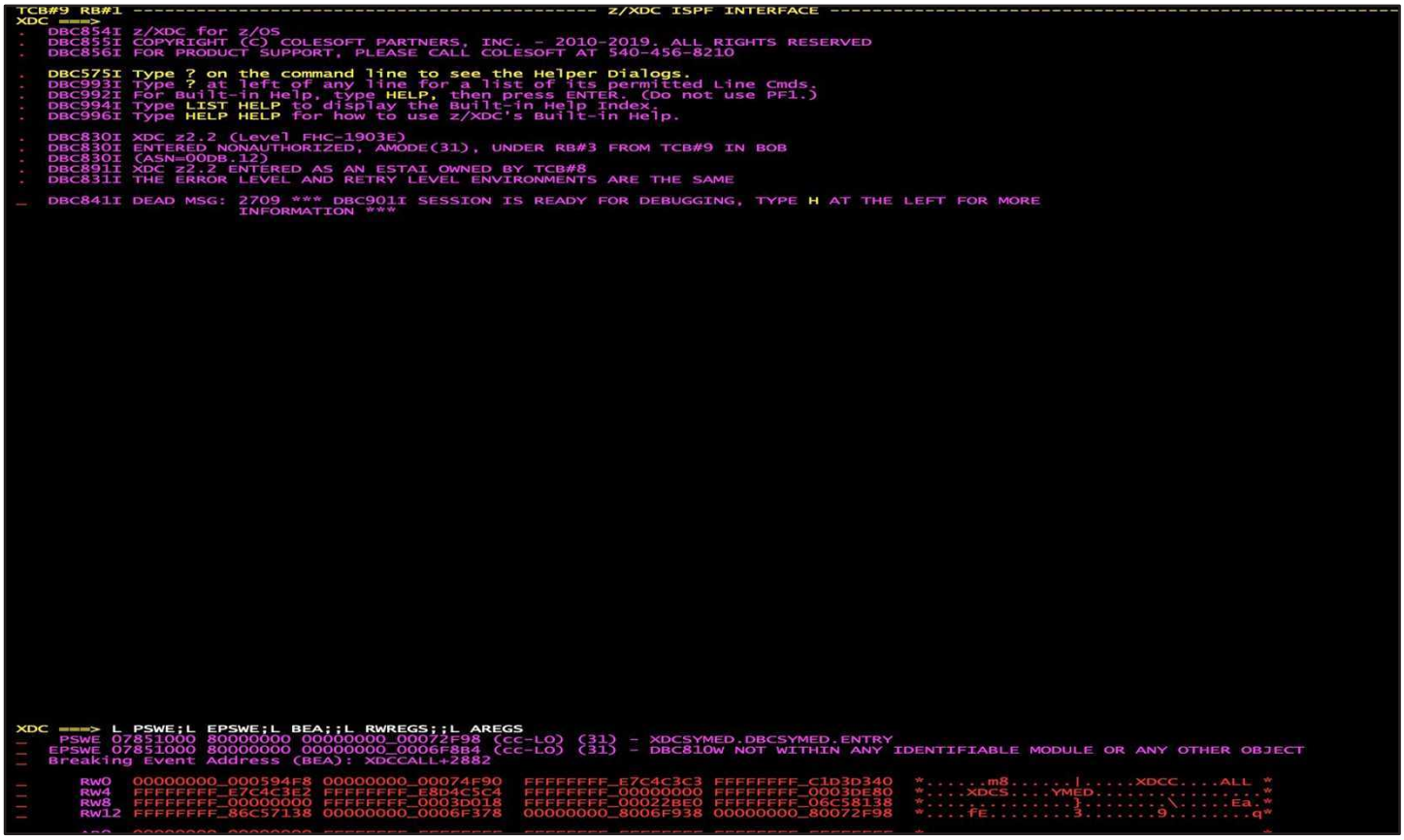


Figure 2-6.

The Profile Facility

This is the factory default panel you'll get if you haven't yet customized it. You are NOT going to be stuck looking at this panel layout if you don't like it. You can have as many world-views as you like. You can decide upon one profile as a default. You can have several for different situations.

z/XDC's Profile Facility allows you to configure your screen layouts, your screen colors, your session logging parameters, your debugging session PFKey settings, your HELP session PFKeys and a host of other parameters. You can access the Profile Facility by entering "PROFILE" on the command line, and putting an "S" next to any topic that you would like to select. You can drill down into the PROFILE Faclity settings and change things to your heart's content.

Go ahead and explore. You'll find plenty to keep you occupied. Among many other things, you can influence how you want z/XDC to interpret what it finds. Offsets in your displays can be hexadecimal or decimal values. z/XDC can be forced to assume EBCDIC, or ASCII information. You can bias z/XDC to showing you memory in data or instruction formats. There are a LOT of choices to make. However, gentle reader, it might be best not to go fooling with these settings until you understand them and are sure you know what you want. You can always get back to our default "factory settings" by reading the next few lines.

If you have customized your Profile settings, then you can issue the END command to get out of the Profile Facility and then SAVE your Profile (you have to leave the profile environment before running the commands below).

- If you issue PROFILE RESET, then your default profile will be restored to the “factory” settings.
- If you issue PROFILE SAVE XDC, then the current profile will become your default profile. The next time you start z/XDC you’ll see this layout.
- If you issue PROFILE SAVE <name> (Up to five characters are allowed for the name.), then it will be saved into your ISPF Profile Library under the name XDC-5-character-name. So, if I issue PROFILE SAVE FRED, it will be saved in my ISPF profile library under the name “XDCFRED”
- You may issue PROFILE READ <name> to recall a previously-saved PROFILE.
- To find out which profile you’re using at any given time, issue LIST PROFILE.

You can have as many named profiles as will fit into your ISPF Profile Library. Thus, you could have a set of profiles, each of which is just dandy for working on a particular sort of problem, or a particular suite of application programs. You could, for instance, set things just as you like them and issue PROFILE SAVE MYPGM. Then you’ll have that profile for use whenever you need to debug MYPGM.

The bottom line is that you’re not stuck with our screen layout. You can make your own, and have as many different profiles as you like.

To see the PF-key settings directly you can also issue the KEYS command, which gets you directly to that portion of the PROFILE system. By default, PF1 says, SET SCREEN DELETE, and PF13 says, SET SCREEN CREATE. These are the two PF-keys that you can use to add or delete new display windows. Let’s play with them!

Place your cursor anywhere in the display under a command line and press PF13. z/XDC will issue a SET SCREEN CREATE command to create a new display window and propagate a display command into it. To make the window go away, press PF1 anywhere within the window (This will issue a SET SCREEN DELETE command.).

When you create a new Display Window the new window inherits the settings of its predecessor Display Window, as it did below.

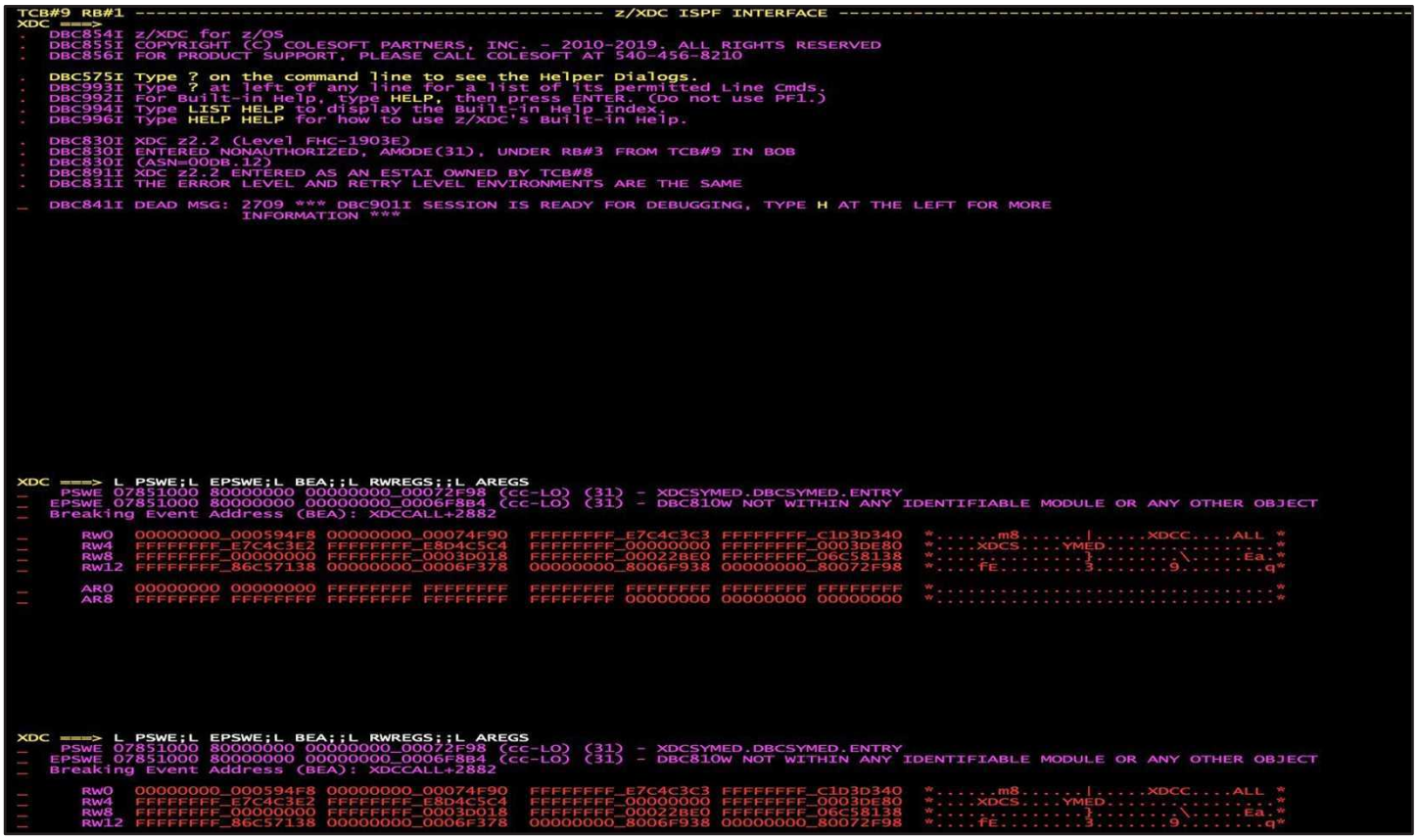


Figure 2-7.

You can put whatever command(s) you like into the new Display Window's command line and it will remain until you decide to change it, or you delete the Display Window entirely. Try putting this in: LIST RW12. Now the contents of General Register 12 in wide format will be shown in this Display window until you no longer need to see them.

To make that extra display window go away, just put your cursor anywhere in it and press PF-1. That's what I've done on my own for subsequent screens in this Primer. Go ahead. Try it.

If you become concerned about saving real estate on the display panel and want to see only a small amount of information in a display panel, you can issue SHOW <address> and z/XDC will display only two lines on the screen. SHOW is a variant of the DISPLAY command, and it uses less terminal "real-estate".

Remember, if you ever mess up your screen, you can always issue the PROFILE RESET command and reload the default settings. I'll do that myself, now.

The z/XDC Session Log

Everything you do in the Working window (only) is saved to a session log, which is typically assigned as SPOOL CLASS X HOLD. Your installation personnel may have changed this. In any case, you can scroll through the log looking backward and forward. Commands for this are the UP and DOWN commands. Page Up and Page Down work for this as well. You can look these up in the Quick Reference or issue HELP COMMANDS <UP> or <DOWN>.

- You can use UP C (for up a command's worth), UP MAX, or DOWN with the same operands.
- Each command in your session log receives a unique number, so if you want to go to a particular command you could enter UP #nn or DOWN #nn.
- If you happen to page to an earlier place in the log, your actions on these historical displays are every bit as active as when they were first generated.
- If you issue a new command, then you are bumped to the end of the log and that command and its output are logged at the end of the session log file.

If, rather than saving your log as a SPOOL file, you'd like to save it into a dataset, you can do that. Have another look into the Profile Facility. The choice you're interested in is under Display/Change Session Logging Parameters.

Now you know what z/XDC's Working and Display windows are. You can add display windows, and put commands into them. You now know that the PROFILE facility allows you to have as many different named world-views as you'd like and that you can call them with the PROFILE READ <name> command.

Chapter 3: Looking Around

In this chapter we're going to show you how z/XDC can make raw object code more legible to a human being.

z/XDC works perfectly fine with pure object code, and its FORMAT command will disassemble object code no matter what the source language actually is. So if you have the proper sort of listing for your COBOL program (one that shows your COBOL source and your object code) you COULD debug it. It wouldn't be pretty but it would work.

Until we introduced c/XDC, C programmers would debug their code with z/XDC just by looking at the disassembled object code. It wasn't easy, but they made it work anyway.

z/XDC also works fine on code for which you have no map, such as "Object Code Only" programs. These require more understanding from the user, but it's still possible to make sense of such programs.

Today, with z/XDC (and now c/XDC), if you have access to your source files you're "golden"! You can place your ADATA or DWARF data right over storage and see your program as it was originally written.

z/XDC can do this trick in two ways for Assembler programs. It can read both SYM records and ADATA.

SYM records are created when you assemble with PARM=TEST and then link-edit (excuse me, "BIND") the program with PARM=TEST. SYM records that map the labels in your program are created and stored in the module on DASD. When you issue the MAP command z/XDC finds your SYM records and lays it over storage, creating a far more legible display than pure object code.

This is the way z/XDC used to work for decades. The downside is that you don't have access to your comments, which most people really like to see.

When the High Level Assembler came out, Dave Cole exploited ADATA, incorporating it into our product. So, with a couple of commands that I'll show you soon, you can get ALL of your source code laid over storage. The downside is that ADATA consumes VAST amounts of storage when they are loaded. So, most shops have to make a strategic decision on which modules they will keep ADATA for. Perhaps you decide to keep ADATA only for active projects - that's up to your shop.

That said, we DO offer an ADATA filter program. You can run this against your ADATA and it will remove all duplicate records achieving up to a 90% compression ratio. To learn about this start a z/XDC session and enter "Help Maps AData Excessadata Xdcadata".

So the decision to use SYM or ADATA is yours to make. Dave Cole has a personal bias towards SYM records, because of the storage issue and XDCSYMED comes with both SYM records and ADATA available. However, because Dave prefers SYM records he's set z/XDC up to use them first. Hey, he's the boss, OK?

Anyway, if you were to issue the MAP command at this point, and if AUTOMAP is set to YES in your profile (which it probably is), then z/XDC will show you the SYM records, not the ADATA. If this is your preference you're happy, but let me assume that you'd rather see your comments.

To do this we need to issue the DELETE MAPS and DELETE MAPLIBs commands (just for good measure). Now we have a clean slate.

Next we'll establish a linkage between XDCSYMED's object code and the ADATA file. Thjs is the SET MAPLIBS command. Your libraries will be different from my own so ask your friendly neighborhood systems programmer for your installation-specific magic "incantation".

For me, the syntax is:

SET MAPLIBS DBCOLE.XDCZ21.XDCADATA.

If this command succeeds, z/XDC will tell you something like "MAPPING INFORMATION IS AVAILABLE FROM THE FOLLOWING LIBRARIES AND DATASETS", and will show you the name of the dataset, confirming your choice. Next, issue MAP PSW? and z/XDC will load the ADATA for the XDCSYMED program.

The WHERE command

z/XDC allows you to examine the entirety of an address space, or multiple address spaces. That's a LOT of territory! If you want to "go home" to where your base code is executing, just enter the WHERE command. The minimum abbreviation is a "W". What the WHERE command does is issue a "FORMAT PSW?" command in the background for you. WHERE is much more convenient. *Psst:* You can also put a "W" in any of the underscored spaces in column one of the display. "W" is a "Shortcut command", which we'll discuss later. Try it, and you should see something like Figure 3-1 below. You've SET MAPLIBS to the proper library, and MAP'd it. Now you issue Where and you see this screen. Nice.

Figure 3-1.

Notice several things:

z/XDC's third line of the display gives you an absolutely unambiguious statement about where you "are" at any moment. You get the virtual address (00072F98), the "storage key" (Key 8, fetch-protected), the address space name (BOB), your location within the current module/csect (XDCSYMED.DBCSYMED with an offset of zero), any EQUATEs that map to this location ("@R15+0") "XDCSYMED+0" and your location within memory (the PRIVATE Area+70F98). The actual addresses shown may vary in your environment.

Let's step through to the next instruction. Issue a single step trace by entering "T" on the command line and pressing Enter. You just did a single-step Trace command. Now we should be at the STM that begins a standard OS entry linkage.

Figure 3-2.

If you've kept the factory "default" screen layout (with a Display window at the bottom), then you've noticed that the PSW and register values shifted as you traced in the Working window.

The FORMAT and DISPLAY commands

FORMAT is how you can disassemble code at arbitrary instruction addresses. You have to be a bit careful with this command, because FORMAT is perfectly capable of mis-interpreting any raw memory as instructions. So, make sure you tell FORMAT to do its thing in the proper place(s).

If you issue FORMAT <address-expression>, then z/XDC will disassemble memory from <address-expression> onward (assuming that the address-expression points to a viable opcode). Since the STM instruction appears near the bottom of the working window, let's issue "F +0" (F-space-plus-zero). This will force z/XDC to put the current instruction at the top of the working window. If you already did that, no problem.

The DISPLAY command gives you a hex-EBCDIC dump of memory with a little reminder to the left showing you what kind of memory it is (in this case Key 8, Fetch-protected). The DISPLAY command's syntax is "DISPLAY <address-expression>". I'll issue "D_+0" (D-space-plus-0) so that we can see a hex-EBCDIC "dump" of memory where the PSW currently points to:

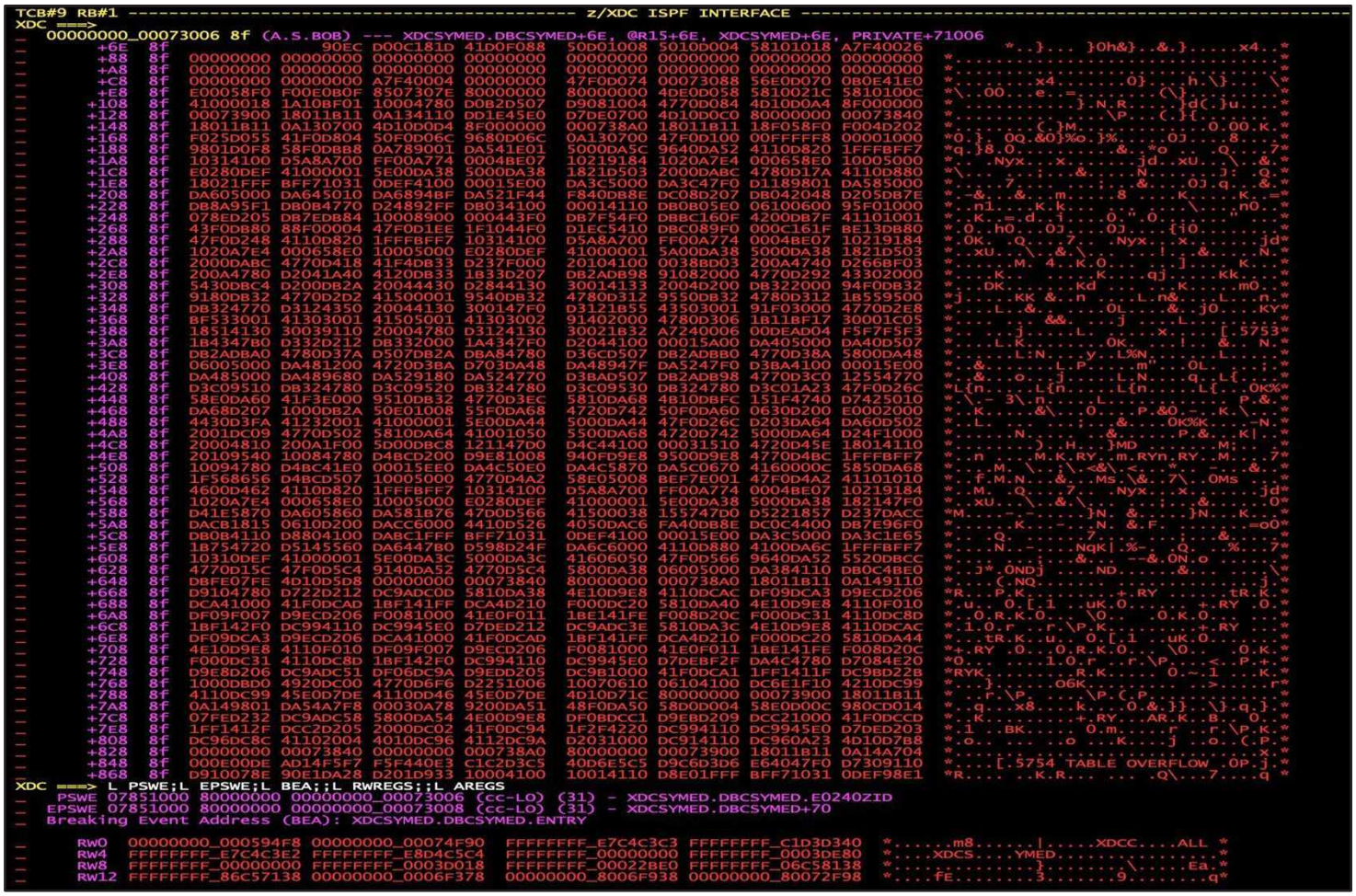


Figure 3-3.

Let's review. You'll be using FORMAT and DISPLAY a LOT in z/XDC.

- FORMAT <address-expression> - disassembles object code.
- DISPLAY <address-expression> - gives you a "dump" of memory.
- Pressing the Page Down or PF8 key will "Page forward" in memory.
- You can press Page Up or PF7 as well to page back up, but you can't "Display back" in memory from an arbitrary location. If you try that sooner or later you'll be looking "earlier" in your session log.

You can type HELP COMMANDS FORMAT or HELP COMMANDS DISPLAY for more information.

I'm aware that I just threw you a curve with the cryptic commands "F-space-plus-zero", and "D-space-plus-zero". It's time for a little overview on address expressions (of which "space-plus-zero" is a special case). The concept of address expressions is CRUCIAL to your success with z/XDC. Let's dive in.

Address Expressions

Most of z/XDC's commands take an address-expression as a parameter. Think of the command as a "tool", and the address-expression as where you want the tool to be applied.

Address expressions are composed of three elements: bases, operators and offsets.

The base value can be a virtual address, the Instruction Address portion of the Program Status Word, the name of a label in our program (.LOOPSTART, for example), the name of an EQUATE (like "FRED"), the contents of a general or access register. It can even be a pure hexadecimal virtual address. Yes, a pure virtual address IS an address expression all by itself. Play along, wise-guy.

The operator can be a plus (+), minus (-), multiply (*) or divide (/). So, you can do simple arithmetic in address-expressions, perhaps to directly address the nth element of a three dimensional table.

z/XDC also makes use of indirect operators. These are:

- Percent (%) for 24 bit mode;
- Question mark (?) for 31-bit mode;
- Exclamation point (!) for 64-bit mode.

Offsets are either hex values or more complex computations. Offsets are used to "count out" from a base value to another place in memory. Essentially, they are displacements off of the base.

The following are examples of various address-expressions you might use (to save my fingers I'll be using "<address>" for address-expression from here on):

"**R15%**" gives us the location pointed to by the contents of General Register 15 in 24-bit mode.

"**FRED+2**" gives the location two bytes higher in memory than where the EQUATE FRED currently points to. "**FRED-2**" is equally valid as an address-expression.

"**PSW!**" shows us where the PSW points to in 64-bit mode.

"**mymod.mycsect.mylabel**" points to a specific label within a csect within a module somewhere in memory.

Here's an interesting variant:

"**R15%%%+C**" means: "Take the contents of R15 and go there in 24-bit mode. When you get there, resolve those 4 bytes in 24-bit mode and go there. When you get there, resolve THESE 4 bytes in 24-bit mode and go there. When you arrive at this place, add a decimal 12 to the result. That is what I want to see." Thus, you can stack your indirect operators and save time when chasing down pointer chains.

There is one finger-saving "short-cut" that I'll introduce here. Your may remember the equate "@CDP" in a previous display. z/XDC maps the most recently displayed address into an equate called the "Current Display Pointer", or "@CDP". You can imply the CDP as a base in your commands by using the character string "_+0" (Space-plus-zero). That is equivalent to telling z/XDC "do the command

where I am right now", or "Use the value of the CDP with an offset of zero".

So, if I want to disassemble code "right here", I can type "F_+0" and use the CDP as the "BASE" for my address expression. This will help protect you (a little bit) from Carpal Tunnel Syndrome….

Remember, "space-plus-zero" is your good friend. It's a handy address expression! It means, "Perform the command here."

Keep in mind that these are only basic address expressions. With the advent of MVS/ESA, z/XDC's address expression underwent major re-engineering in order to accommodate ALETs, Access Registers and dataspaces. In effect, IBM had gone horizontal with memory. And when z/OS arrived, IBM had gone vertical with memory as well. Address Functions will help you work with advanced address expressions that refer to dataspaces, and other multi-address space concepts.

Addressing Functions

You can now use the new address expression functions to refer to real memory addresses or virtual memory addresses in other address spaces. I will not go into this area here, but you can read up on them by entering HELP ADDRESSING FUNCTIONS.

In practice, you'll state an address expression, then a tilde character ("~"), then you'll invoke the Address Function. The tilde character tells the z/XDC address expression parser to expect a function.

So, you could specify a virtual address and have it resolved not in your address space, but in another address space.

Programmers who work with Access Registers and in multiple address spaces would do VERY WELL TO READ Help ADdressing Functions. You'll find it very useful.

Shortcut Commands

z/XDC's command language is subject to abbreviation wherever possible, perhaps because we at Cole Software are such poor typists! We also want to save time in our work. z/XDC's displays often show an underscore character out to the left of each line in Column one of the terminal screen. You can use this field to enter "shortcut commands". These commands will save you a lot of time.

If you are ever unsure of what Shortcut Command you can put in the underscored area, you can enter a question mark ("?") and z/XDC will read you the list of available commands, which can vary with context. Thus, you may receive a list of allowed commands that looks like this: "A D E F H J K L M N O P S T W X Z ?". That's clear as mud, isn't it? Here is a table of the Line Commands and what they can do for you:

A	Sets a permanent breakpoint at "this" line in the program by way of the "AT" command.
CU	In dump/XDC, this forces the relocation of the Current eXecution Unit.
D	Performs a context-sensitive DISPLAY command.
E	Performs an "EWHERE" command (display the user's Error level abend address).

F	Performs a FORMAT command (disassemble object code beginning at this point in the pro gram).
H	Enter this next to any generated z/XDC Message, and you'll follow a "hyperlink" directly into the Help system for this message.
J	Performs a "JUMP" command, forcing the ILC portion of the PSW to point to "this" instruction.
K	Sets a HOOK at this point. A Hook automatically subverts other, prior ESTAEs, "forcing" a z/XDC session "here".
L	Performs a context-sensitive LIST command.
LO	"LO" Performs a LIST OPERANDS command. Results will be "questionable" unless applied against the current, or prior instruction.
M	Performs a MAP command.
N	Sets a "NOTE" command. Creates a "book-mark" in memory that you can see later with the "LIST NOTES" command. The "NOTE <text-string>" variant creates a "book-mark-with-text" in memory.
O	Performs an "OFF" command, removing any breakpoint set at this location.
P	Purges a previously defined "NOTE". Can only be used from the "LIST NOTES" command dis- play.
Q	Issues "Set Qualifier" to make this mod.csect the current default.
S	Performs a SELECT operation in the Help and PROFILE Subsystems, OR performs a "SWITCH" to another task when in the List TAsks display.
SH	Performs a "SHOW" operation, yielding one line of memory display.
T	Sets a transient breakpoint at "this" instruction in the program.
W	Performs a "WHERE" command (FORMATs the current execution location).
X	Toggles a defined breakpoint on or off without deleting the breakpoint definition.
Z	Indicates that any overtyping on "this" line will be taken as a ZAP command, changing memory or storage at "this" location. Wherever the TAB key stops, you can overwrite existing fields.
?	Not all line commands are available everywhere. If you use "?", then z/XDC will show you which line commands may be used in the current context.
*	For c/XDC: Show a variable in EBCDIC format
|	For c/XDC: Show a variable in ASCII format
\	For c/XDC: Show a variable in null-terminated format
/	For c/XDC: Show a variable in length-terminated format.

Not all line commands are available in every situation. For example, S (for SWITCH) is valid only within the context of the LIST TASKS command. P (for "PURGE) is only valid within the context of the "LIST NOTES' command. The Question mark is always available. Use it and z/XDC will tell you what's valid wherever you are, depending on the context of the display you're in.

Point-and-Shoot Capabilities

z/XDC has a lot of point-and-shoot capabilities that will save you LOADS of time. These are context-sensitive displays that utilize the cursor, and the "%", "?" and "!" characters.

For general reference, please remember that:

- The percent character ("%") denotes 24-bit mode;
- The question mark character ("?) denotes 31-bit mode;

- The exclamation point (“!”) denotes 64-bit mode.
- The F character will produce a disassembly.
- The Enter key will also produce a disassembly.
- The D character will produce a DISPLAY (a dump).

Things to explore:

1. Put the cursor on any word of memory in the working or display window(s) that looks like code and press Enter. z/XDC will FORMAT (disassemble) memory at that point. Cool!
2. Put the cursor over any part of the display, enter a D-space and press Enter. The memory pointed to under the “D-space” is DISPLAYED for you in hex-EBCDIC format.
3. Put the cursor over any part of the display, enter an F-space and press Enter. The memory pointed to under the “F-space” is FORMATed for you.
4. One can merely type %, or ? or ! as well. z/XDC will establish whether you want a Display command or a Format command to be processed, based on whether the screen you’re working from resulted from a Display or Format command in the first place.
5. One can also use D or F together with %, ? and ! in any combination.
6. Put the cursor on any word of memory in a window, enter % and press Enter. z/XDC will display that word as a 24-bit pointer. This works in a register display as well.
7. Put the cursor on any word of memory in a window, enter ” and press Enter. z/XDC will display that word as a 31-bit pointer. This works in a register display as well.
8. Put the cursor on any word of memory in a window, enter ! and press Enter. z/XDC will display that word as a 64-bit pointer. This works in a register display as well. Be advised that in Foreign Address Space Mode, z/XDC will default to 24-bit mode because it cannot tell which PSW you may refer to. In this case, use D or F with a % or ? or ! as appropriate.
9. Put the cursor on the first digit of a machine instruction’s base/displacement field, type in a %, a ? or a ! and press Enter. The memory used by the instruction is interpreted according to the width specified (24-, 31-, or 64-bits).

For a REAL time-saver, try these point-and-shoot commands on registers! Hey, that’s neat!

You can also use “D_“ and “F_“ anywhere on the screen as well as on the one-byte area at the left of the display (“D-space” and “F-space”). If you put the cursor on any interesting area of storage (or a register in the lower display window), and type “D_“ or “F_“ then z/XDC will take the address over which you typed, and issue a Display or Format command for that address. This makes following pointer chains much easier. Try it!

Caveat: In playing with the above commands, you may receive a message such as DBC045E STORAGE ACCESS VIOLATION. This is because the XDCSYMED program has a storage key that may not match the storage key of the memory you inadvertently wanted to look at. That’s as it should be, don’t you think? If you want to get a better explanation for this message, tab over the period at the left of the error message (the “.”) type in an “H” and press enter. You may as well start using our time-saving features now.

Back to Looking Around – the LIST Command

Playing around with the DISPLAY command, we can say things like DISPLAY R15? and see the calling

program in hex-EBCDIC mode (that's where R15 points, doesn't it?). You can say "FORMAT system-module-name" and z/XDC will go there and disassemble as much as it can (assuming you are allowed to look there). Or you can look at XDCSYMED in hex mode by saying DISPLAY PSW" If you want, you can issue FORMAT R14! and see if there is any code there. If you've been experimenting with these commands, then you may be FAR away from your program by now.

Let's get back to looking around our address space. Please issue a WHERE" command. An exact equivalent would be to issue F PSW? (remember the indirect operator ? denotes 31-bit addressing).

LIST PSW

ANY debugger should show you the Program Status Word, but there's a cool parameter for this command you NEED to know about. Try LIST PSWE FORMAT, or LIST PSWE F.

Figure 3-4.

That is NICE. You no longer have to deduce the components of the PSW. You just go LOOK.

LIST RWEGS

Here's the output of a "vanilla" LIST RWEGS command. Remember you can use your point-and-shoot commands directly IN the register fields, as well as the underscore out to the left for Line commands:

Figure 3-5.

There are two more forms of the LIST RWREGS command. Plain old LIST REGS will give you the low-order halves of the 64-bit registers (or the General registers as they appeared in 24-bit and 31-bit programs). LIST RHREGS will give you the high-order halves of the 64-bit registers.

"What about Floating Point registers?"

LIST FREGS

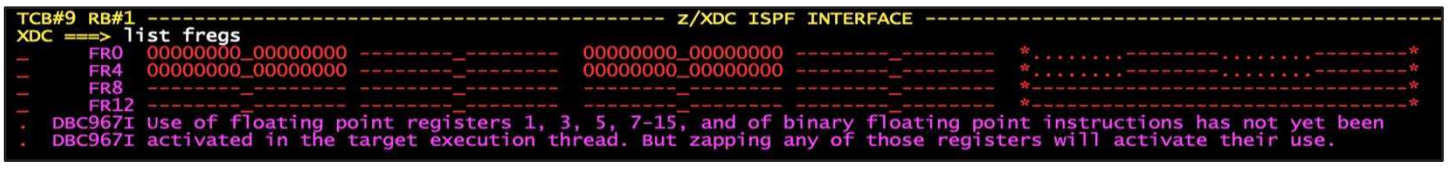


Figure 3-6.

“OK... Can you show us Access Registers?”

LIST AREGS

Figure 3-7.

“OK, here’s a toughie: Show us the Control Registers!”

LIST CREGS

Figure 3-8.

IBM recently introduced the SIMD instruction set (for in-core sorting) and the Vector Registers they manipulate. If you don’t have this feature on your processor, you may issue LIST VREGS PRETEND and z/XDC will show them as if they were available.

LIST VREGS

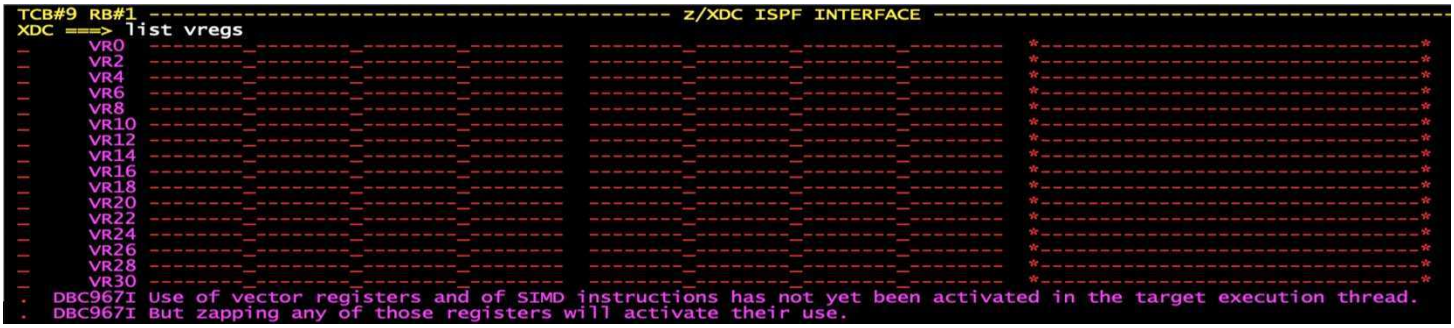


Figure 3-9.

These are all the register sets that you can view in z/XDC. But it does get better. If you want to “drill down” and interrogate a single register, then z/XDC will show you as much information about that individual register as it can. Once again, z/XDC removes the need for you to deduce content. Instead you just LOOK, and make strategic decisions, as a human should.

LIST a Single General Register

Figure 3-10.

LIST a Single Floating Point Register

Figure 3-11.

LIST a Single Access Register

Figure 3-12.

LIST a Single Control Register

Figure 3-13.

LIST FIXED or LIST FLOAT

You can also issue a LIST FIXED <address> or LIST FLOAT <address> and z/XDC will show you the contents of memory at <address> as a fixed, or floating point number. This is a handy opportunity to demonstrate the L line command (a short-cut for LIST).

Below, in Figure 22, you can see where I have issued LIST FIXED R12?. That is, I want z/XDC to go to the memory referenced by R12 in 31-bit mode, and show me that address interpreted as a fixed-point number. You also may notice that I've put an "L" next to the output of the command. Now, I'll press enter.

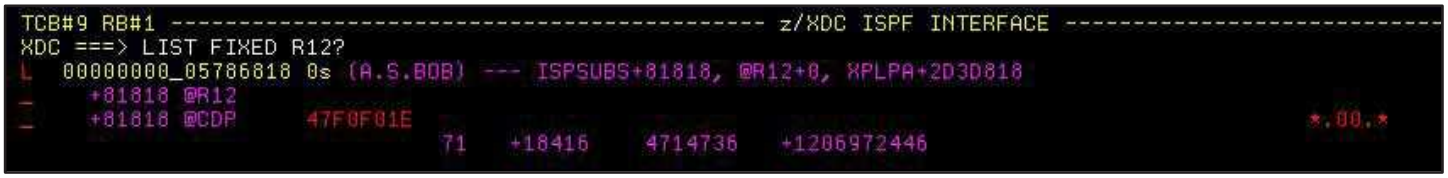


Figure 3-14.

In the next Figure, z/XDC has toggled the value, and is now showing you the same address interpreted both as a floating AND fixed point number. In this display you can see the Binary Floating Point (BFP), the Decimal Floating Point (DFP) and Hexadecimal Floating Point (HFP) interpretations.

You can try the "L" line command out in many other z/XDC displays. To find out if you CAN use "L" in a given situation, just put a question mark in the underscored space in column one of your display, and press Enter. If "L" appears in the list of allowed commands, give it a try.]

Figure 3-15.

So far, so good. However, z/XDC's LIST command is FAR more powerful than we've seen to this point. Let's try out some more interesting LIST commands.

LIST TASKS

When presented with a dump, you can - with some effort - build a mental image of the task structure in your address space. Indeed you may have done this many times over the years. What if we could save you that time? LIST TASKS is truly one of z/XDC's coolest commands:

Figure 3-16.

The LIST Tasks command deserves a bit of study. Firstly, all tasks in the address space have been shown, with the hierarchy of the tasks shown by indentation out to the right. Now we can prove to you that our earlier assertion is true: XDCCALL is in Task number 8 and has attached XDCSYMED (our test program) as a sub-task. XDCSYMED is the current task and has abended with an S0C1 abend. This is no surprise, because that's how z/XDC works: by putting X'00's into the execution stream.

Further, all tasks in the address space have been assigned a unique equate name, TCB#1 through TCB#n where n is the number of tasks.

Also, notice that you have a familiar underscore in column one of the display, which invites you to try some line commands. If you put a question mark in any of these underscores z/XDC will return this:

_ DBC732I PERMITTED LINE CMDS - C D F L M N Q S

LIST RBS

On the List Tasks command output, the Shortcut command I use most often is L, and since we're deep into the LIST command, let's try that next to Task 9, the XDCSYMED task:

Figure 3-17.

Now we see the Request Block structure for the current task. I could also have issued LIST RBS and I would have gotten here as well. Now I know some things:

There are three Request Blocks under Task 9.

1. The first is a Program Request Block (the PRB) that was set up to attach XDCSYMED. I see XDCSYMED's current execution location - at XDCSYMED+6E (where we're getting ready to execute the STM instruction.
2. However, z/XDC stopped execution at that STM instruction by placing a X'00' on that instruction which caused a S0C1 abend. That caused a Supervisor Request Block (the SVRB) to be created to handle the abend. This program is currently stopped in IGC0101C+816A. IGC0101C ran the Retry chain to find out if an ESTAE was set up to handle this abend. There was one - z/XDC.
3. So, IGC0101A created a Program Request Block for z/XDC and has SYNC'D to it. That is a lot of knowledge to have without much effort.

What happens if you put an F next to any of these lines? Try it. Cool, isn't it? You've done a"point-and-disassemble by wiggling only a few fingers ("Tab", "F" and "Enter").

One more thing: Once you have issued LIST TASKS, you can issue LIST RBS TCB#n and z/XDC will show you the RB structure for that task. Or, you could be lazy as I am and just put an L Next any task in the LIST TASKS display.

References: HELP COMMANDS LIST TASKS, and HELP COMMANDS LIST RBS.

Using LIST TASKS and LIST RBS Together

In practice you'll use LIST TASKS and LIST RBS a good deal, ESPECIALLY if you encounter an unexpected abend in your code. This command pair gives you definitive information on where you are and why you got here. In that situation, it'd be a good idea to issue this character string:

LIST TASKS;;LIST RBS. Remember, the semi-colon is our command delimiter. I've actually used TWO semi-colons, which creates a (morereadable) blank line between the output of the command pair:

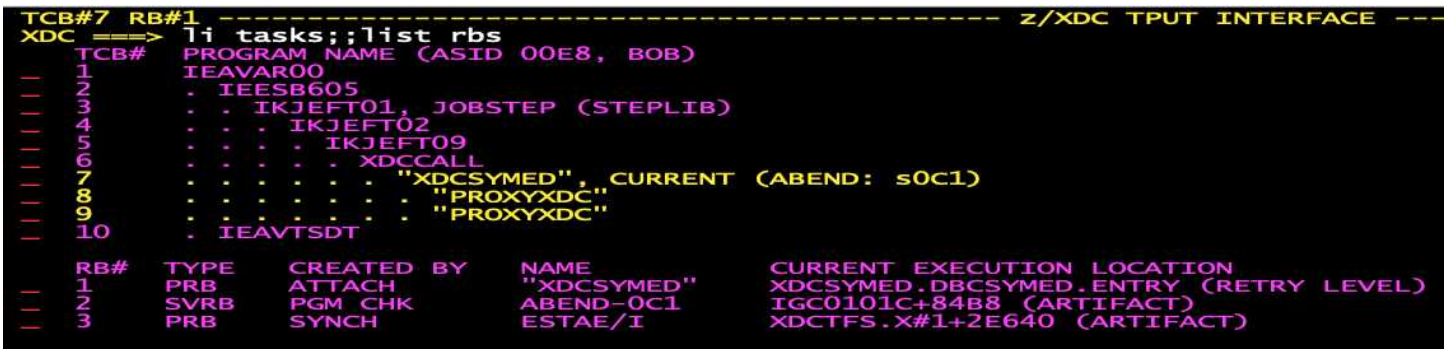


Figure 3-18.

[Note: You may combine many of z/XDC's commands to give you the maximum amount of information without having to work very hard. That's the entire point of the product - to free your mind from minutiae so that you can think strategically about the problem. Just use the construct "<command>;<other-

command>;<yet-another-command>"]

Another shortcut: If you are in a multitasking environment and if you issue LIST TASKS, and if some of the tasks are under your z/XDC's ESTAE control, then you can put an S next to any of these tasks and SWITCH or SWAP to another task. That's a very quick way to suspend your debugging session in one task and open up a new one in another task.

While we're on the subject, let's introduce a design concept. **If you can make a thing in z/XDC, you can also LIST it or DELETE it.** This applies to the special EQUATES like TCB#n as well.

LIST EQUATES

Please issue the LIST EQUATES command. At the top of the list, you will see entries named RB#1-n. Below that you'll see your TCB#n equates, and then a bunch of special equates which say (BUILT-IN) out to the right. These equates refer to system areas that you may find useful. They are immutable and are retained from session to session. Here's what I get when I enter "LIST EQUATES":

Figure 3-19.

The LIST Equates command gives you a complete display of all equates defined for your debugging session. Many of these are "built-in", and there for your convenience. You can page down to see the rest of the list if you like, but I'll save space by not doing so in this booklet.

Note that some equates are built for you dynamically. Thus, you'll see the equates RB#1, RB#2 and RB#3. These were created the last time we issued LIST RBS. You can also see the equates for the Task Control Blocks ("TCB#1", "TCB#3", etc.).

The display is organized in columns, and the columns have these meanings:

- The first column is, of course, the name of the equate.

- In the next column you'll see either "(D)" or "(I)" This means "data" or "instruction", respectively.
- The next field is the virtual address.
- The next column contains the length of your equate in hexadecimal.
- The next column is the length of your equate in decimal.
- The final column describes "where" the equate exists. The RB and TCB equates are in the Private Area. Others are "built in", meaning z/XDC maintains them for you. Others are "common" which means they're in the Common Area, or "Global", which means they exists for all address spaces.

Many of the Equates in LIST EQUATES may be useful to you in your work. Having an Equate name for the beginning of the PLPA or the SQA could save you a lot of time.

Later in this book we'll discuss the Equate command, which allows you to create your own equates in memory. If you're curious about this right now, enter the character string, HELP COMMANDS EQUATE and read ahead.

LIST SUBPOOLS

Let's see what the organization of memory is. Issue LIST SUBPOOLS and you will get a complete display of how memory is allocated for the current tas in this address space. If you are interested in only one subpool, issue LIST SUBPOOLS 78 (or any other subpool ID).

If you would like to find out from which subpool the memory at some address comes from, then you can issue LIST SUBPOOLS <address> and z/XDC will tell you. Keep in mind that <address> can be an address expression including PSW?. In this way, you could find out which memory subpool your program is allocated from.

```
TCB#7 RB#1 ------------------------------------------- z/XDC TPUT INTERFACE
XDC ===> list subpools
          SPID      START          END    LENGTH      USED     UNUSED   KEY  TCB#
_          TOTALS FOR SUBPOOL 0:            8k     3,168      5,024
_          TOTALS FOR SUBPOOL 1:            4k       112      3,984
_          TOTALS FOR SUBPOOL 57:         11m+      11m+       33k+
_          TOTALS FOR SUBPOOL 58:           4k        32      4,064
_          TOTALS FOR SUBPOOL 59:          16k       16k         0k
_          TOTALS FOR SUBPOOL 78:          96k       89k+     6,360
_          TOTALS FOR SUBPOOL 229:          4k     2,112      1,984
_          TOTALS FOR SUBPOOL 230:         12k     1,496       10k+
_          TOTALS FOR SUBPOOL 236:        284k      273k+      10k+
```

Figure 3-20.

However, LIST SUBPOOLS, like a lot of z/XDC commands have many more parameters than I can show you here. LIST SUBPOOLS COMMON show all common memory subpool allocations. LIST SUBPOOLS COMMON DETAILED would display extent-by-extent allocations. It would be a great idea to read Help COmmands LISt SUbpools, and really check it out.

I should remind you that all of z/XDC's commands can be abbreviated. In each Help Frame, the minimum abbreviation to navigate directly to that page appears in capital letters. So, to research the FORMAT command, all you need to enter is H CO FO. In this book I'll use the full names of each command and Help panel (or maybe not. I'm a lazy typist). You will discover the abbreviations over time.

LIST PGMS

Another useful List command is LIST PGMS. You will get a display of all the modules running on behalf of the current task, along with their aliases, if any, and other attributes as well. Again, there are a lot of parameters for all our List commands that do really interesting things. Try LIST PGMS ALL, or

LIST PGMS CICS and see what the possibilities are.

Figure 3-21.

Whenever you have a question and you can't call us it's always best to consult the ultimate authority - the z/XDC product itself. In the case of this command, Help COmmands LISt PGms Report will give you the complete story in our Built-in Help system.

This is a LOT of information to get without much effort on your part. That's the heart-and-soul of z/XDC's usefulness. Instead of deducing complex structures, *you just go look.*

LIST RSA

If you're curious about the savearea chain, you can issue LIST RSA. If you were interested in going to another program in this chain you could type a D or an F next to the savearea desired, and z/XDC would DISPLAY the savearea for you.

Figure 3-22.

LIST RSA shows you the savearea chain for your program. This XDCSYMED environment is very simple, so in working with your own programs you will probably see a far more extensive display.

LIST TIOT

Here's how you get a look at the Task I/O Table. Once you've issued the LIST TASKS command you can use the TCB#<n> equates as a parameter for many of our LIST commands. Here's an example of LIST TIOT for the third Task Control Block in my address space (TCB#3). Press Page Down to see more on your display (I've only captured a portion of a single screen here).

```
TCB#9 RB#1 ------------------------------------------- z/XDC ISPF
XDC ===> LIST TIOT TCB#3
_   JOB=BOB, STEP=LOGRWS, PROC=ISPFPROC

     DDNAME     VOLSER   DSNAME
_    SYSUADS    A3CFG1   SYS1.UADS
_    SYSLBC     A3SYS1   SYS1.BRODCAST
_    SYSEXEC    A3RES1   ISP.SISPEXEC
_               A3SYS1   ADCD.Z23A.SYSEXEC
_               A3RES1   ISF.SISFEXEC
_               A3RES1   SYS1.SBPXEXEC
_               A3PRD1   CSQ800.SCSQEXEC
_               A3PRD2   CSQ901.SCSQEXEC
_               A3RES1   IOE.SIOEEXEC
_               A3PRD1   FAN140.SEAGCMD
_               A3RES1   EOY.SEOYCLIB
_    ISPEXEC    A3RES1   ISP.SISPEXEC
_               A3RES1   SYS1.SBPXEXEC
_               A3PRD1   CSQ800.SCSQEXEC
_               A3PRD2   CSQ901.SCSQEXEC
_    ISPSLIB    A3RES1   ISP.SISPSLIB
_               A3RES1   GIM.SGIMSENU
_               A3RES1   ISP.SISPSENU
_               A3RES1   ISF.SISFSLIB
_               A3RES1   SYS1.DGTSLIB
_               A3RES1   SYS1.HRFSKEL
_               A3RES1   SYS1.SBLSKEL0
_               A3PRD1   FAN140.SFANSKL
_               A3PRD1   FMNE10.SFMNSLIB
_               A3PRD1   AUT350.SINGISKL
_    DITPLIB    A3PRD1   DIT130.SDITPLIB
_    SDSFMENU   A3RES1   ISF.SISFPLIB
_    ISPPROF    CSW00H   BOB.ISPF.ISPPROF
_    SMPTABL    A3CFG1   SYS1.SMP.OTABLES
_    AOFTABL    A3PRD2   AUT350.AOFTABL
_    IHVCONF    A3PRD2   AUT350.IHVCONF
_    AOFPRINT   (JES2)   BOB.BOB.TSU02301.D0000101.?
_    ISPCTL1    CSWL03   SYS19093.T120743.RA000.BOB.R0155313
_    ISPCTL2    CSW00N   SYS19093.T120743.RA000.BOB.R0155314
_    ISPLST1    CSW00Q   SYS19093.T120743.RA000.BOB.R0155315
_    ISPLST2    CSW00E   SYS19093.T120743.RA000.BOB.R0155316
```

Figure 3-23.

I should point out that the LIST TIOT command isn't actually a z/XDC command. Instead, it is implemented via a sample User Commands Exit that we distribute with the product. In most cases, a properly installed z/XDC will automagically make LIST TIOT available. See Help COmmands LISt TIOt for complete information.

LIST ESTAE

Since z/XDC itself often runs as an ESTAE, it's useful to know what other ESTAEs may exist. When entered without any parameters, z/XDC shows the list of STAE Control Blocks for the current task. You may issue this command against any other TCB that you like (once you have issued LIST TASKS, that is). If you prefer to type less (I do) a synomym for this command is LIST SCBS.

```
TCB#9 RB#1 ------------------------------------- z/XDC ISPF INTERFACE -
XDC ===> list estae
                                          XDC     FLAGSn
    SCB#  TYPE    OWNER    STATUS         REL.    1, 2, 3   EXIT ADDRESS
_   1     ESTAI   TCB#5    PENDING                96,80,00  IKJEFT04+2C
_   2     ESTAI   TCB#6    PENDING                96,88,40  ISPMAIN.X#1+41E88
_   3     ESTAI   TCB#7    PENDING                96,88,00  ISPSUBX+138
_   4     ESTAI   TCB#8    ACTIVE-RB#3   z2.2     96,98,48  XDCCALL+19A2
_   5     ESTAEX  RB#3     PENDING                10,04,00  XDC.X#1+21560 (ARTIFACT)
```

Figure 3-24.

LIST LSTACK

You can also query the Linkage Stack by entering either LIST LSTACK or LIST LSES (for Linkage Stack Entries). When given without parameters, z/XDC shows you the Linkage Stack entries for the current task. You may also see the Linkage stack for any TCB or RB in this or any accessible address space, or the linkage stack for a Suspended Service Request Block.

Figure 3-25.

LIST FEATURES

Here in one screen is all the information z/XDC can gather about the current processor.

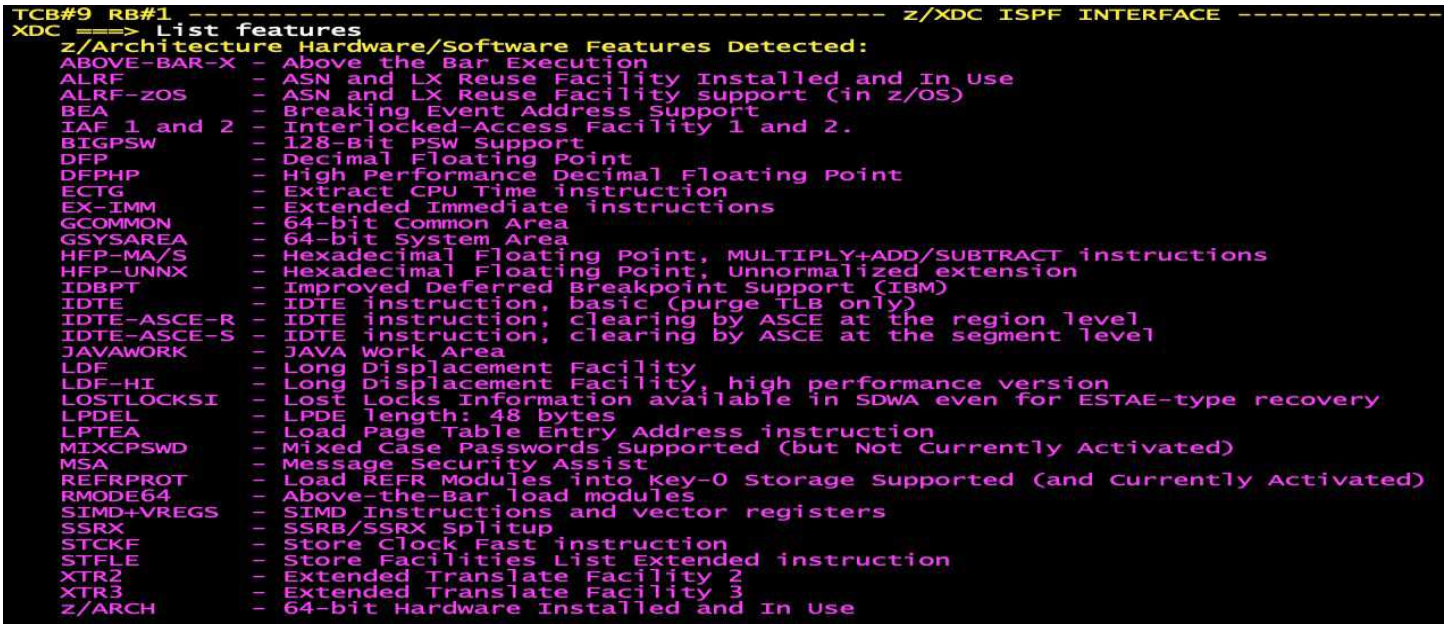


Figure 3-26.

To find out what features aren't available, try LIST FEATURES NOT.

As you may have surmised, there are a LOT of LIST commands available to you. They all have the same philosophy behind them: Show the user a maximum of information with a minimum of effort, saving mental overhead. In other words, let z/XDC do the hard work and let the human make strategic decisions.

Here are some useful LIST commands that I just don't have space to show you. Check them out your self if you're curious:

- LIST SSCT <== Query the Subsystem Control Table
- LIST ENQ <== Display a list of system and user enqueues
- LIST MOBJECTS <== see above-the-bar memory objects

- LIST XMS <== Display the cross-memory execution environment for any program
- LIST LOCKS (and SET LOCKS) <== for SRB debugging

The most useful LIST command of all is this: **HELP COMMANDS LIST.** If you try this, you can page down a list of all of z/XDC's LIST commands. We suggest you try this command each day for a couple of weeks, and select a new LIST command each time. You're going to be pleasantly surprised at your new capabilities!

So far we've been looking around the address space. We've used DISPLAY <address> to see a dump of memory. We've used FORMAT <address> to disassemble object code. We've used a LOT of LIST commands to see z/OS-specific structures. The next command in this unit is FIND.

The FIND command

FIND has gotten VERY sophisticated over the years, but the simplest form is this: FIND <string> <starting-address>. Your string can be pure hexadecimal characters or EBCDIC characters surrounded by single quotations. I'll do a FIND for the character string 'COLE" starting from where the program is now.

Please perform a Where command. A W in column one of your display or on the command line will do nicely.

Now I'll issue my FIND command:

Figure 3-27.

If at this point I want to search for the same thing, I can merely issue the FIND command without operands, and z/XDC will keep searching memory sequentially. Just enter FI into your command line and press Enter. You may issue FIND for anything else in memory.

This was a VERY simple case of using Find. There are far more sophisticated uses you can put FIND to. Check it out. Issue HELP COMMANDS FIND.

The FIND Helper Dialogue

Our Frank Chu has written Helper dialogues for some of z/XDC's more complex commands. FIND is one of them. If you enter FIND ? you will get the panel below. Now you can fill in all the parms you like and execute the command far more easily. Try it. Page down for more parameters.

Figure 3-28.

When you have filled in all relevant fields, press Enter to run the command, or PF4 to abandon it.

Currently there are four such Helper dialogs (most of which run to more than a single page). You access them by typing the command's name, a space and a question mark, as below:

- FIND ?
- MAP ?
- DMAP ?
- USING ?

SCANLOG

z/XDC's FIND command is only for looking around in memory, and perhaps instead you'd like to see some text from your source code instead. To do this trick, first issue FORMAT <address> <nnn> where <nnn> is the number of lines of code you want to see. This stuffs that number of lines into the session log, and then z/XDC's SCANLOG command will search for your text in the session log.

You may also use the PREVIOUS parameter to search "upward" in the results of your Format command.

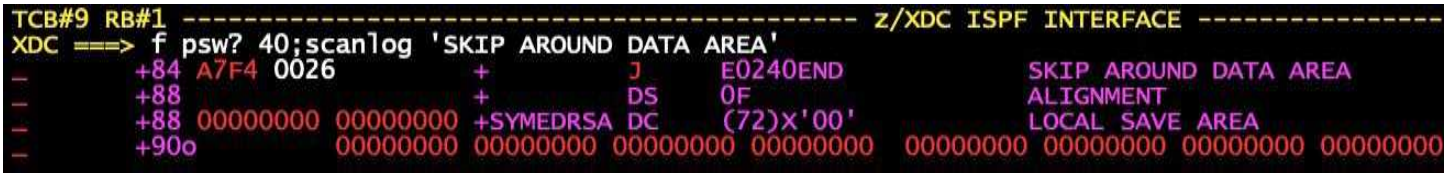


Figure 3-29.

The appropriate resource is Help COmmands SCAnLog.

Chapter 4: Mapping

This chapter is concerned with mapping, or “Symbol Manipulation” (as Dave Cole puts it). Mapping is the process of making raw object code more intelligible to a human being. z/XDC offers the MAP command for csect mapping, and the DMAP command for dsect mapping. You can see the source code for your programs, and you can see the names and values in your data sections.

Remember, *z/XDC works just fine on raw object code for which there IS NO MAP*. That is one of the original design points. A LOT of great work has been done with the product in just this way because z/XDC will happily disassemble all that it understands, following program execution, branches - everything. It’s a bit hard on the eyes *but it works.*

That said, z/XDC’s Format command quits when it encounters memory contents that are not valid opcodes. It can also misinterpret EBCDIC characters as object code. It cannot always decode the branch mnemonics that were originally in the source (Branch Low, or Branch Not High?). None of this will affect z/XDC’s performance; the displayed code looks a bit wierd, but it’s *worlds better than poring over a dump!*

All the uncertainty goes away if you have ADATA for z/XDC to use. You see the code exactly as the programmer coded it.

Viewing XDCSYMED Without Any Map

For this panel, I explicitly issued DELETE MAPLIBS and DELETE MAPS to arrive at an Object Code Only view of XDCSYMED. The code is parked at an unconditional Branch to the STM instruction that actually starts the program, so for fun I created an EQUATE called START at that instruction, for clarity. Have a look at how I strung those commands together with semi-colons:

- DELETE MAPLIBS
- DEL MAPS;
- EQU START XDCSYMED+6E I (the I parameter tells z/XDC to treat this equate as an instruction boundary)
- WHERE

You too can stack commands to do many things at once. It’s a useful technique. You could equate stacks of commands to a PFKey setting if you like.

Here's an image of z/XDC's resulting actions in Figure 4-1 below:

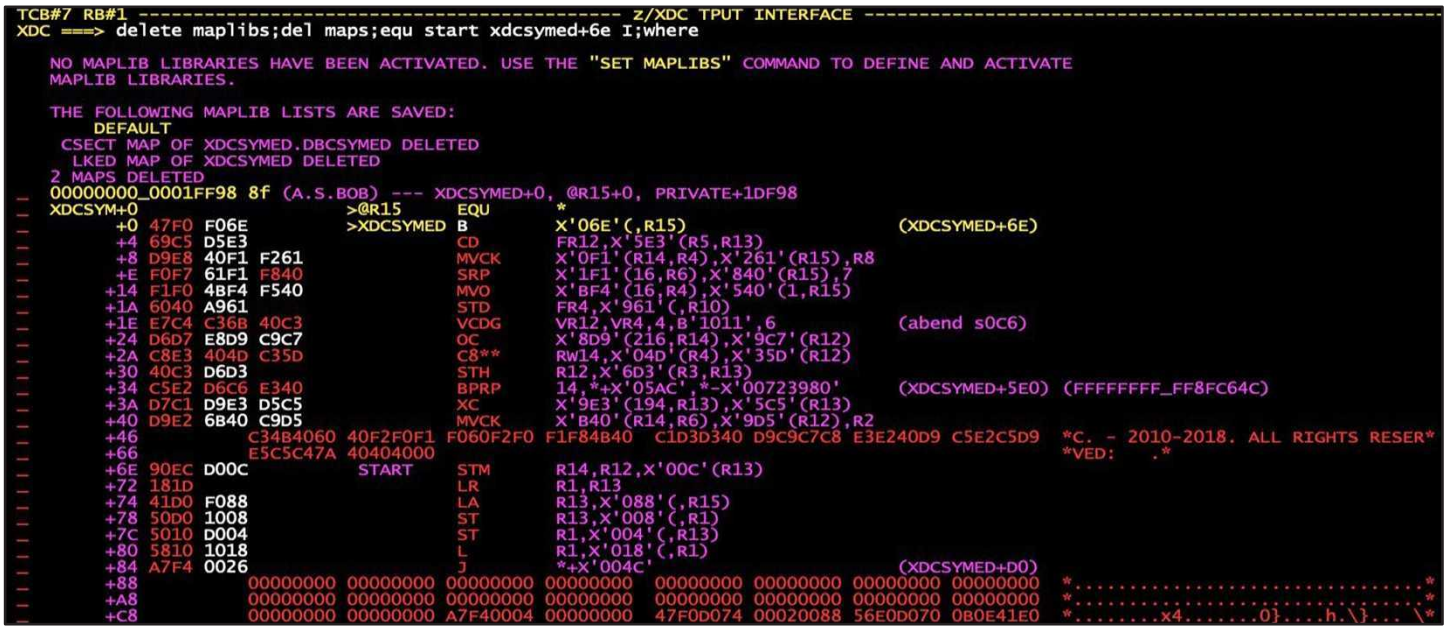


Figure 4-1.

If you had NO maps available you could still do useful work, because z/XDC would follow the branch at XDCSYMED+0 to the STM at XDCSYMED+6E. You could trace your way to the Branch at XDC-SYMED+84, and z/XDC would skip the zeroed-out patch area between XDCSYMED+88 to the branch target (another Branch) at XDCSYMED+D0. The program would behave normally with or without a map. NOW would you be happier working on an undocumented system exit?

Note, however, that z/XDC has given up trying to format at XDCSYMED+88. It saw X'00' and quit. You see the rest of the code beyond this point in "DISPLAY-mode".

Point made. I'm going to end the debugging session and bring myself back in. You stay where you are.

Viewing XDCSYMED With an ADATA Map

Preparing your code for Source Statement Mapping

- Assemble your program under the High Level Assembler, specifing PARM='ADATA' .
- Include a //SYSADATA DD card. Here you specify the name of the dataset to contain the ADATA generated during the Assembly.
- Later, in your debugging session, you'll use the SET MAPLIBS command to identify the dataset you specified in the SYSADATA DD statement during the assembly step.
- Here's the definitive Built-in Help subject, which you can enter from within a z/XDC debugging session: Help MAps AData CReating. In the preceding sentence, I've capitalized the minimum abbreviation for the command.
- You could also use z/XDC's LIST HELP command to give yourself an index into this topic. That syntax would be List Help MAps ADAta. Then put an "H" next to any subtopic that catches your eye, and press Enter.

We issued SET MAPLIBS earlier in this session when I asked you to find your installation-specific data-set name for the XDCSYMED ADATA file. Again, in my world that's, SET MAPLIBS DBCOLE.XDCZ21.XDCADATA. You can probably count on the low-level qualifier, but the rest is up to your installing systems programmer. Go ask.

In order to set up this screenshot, I restarted my session on XDCSYMED. I then used PFK1 to issue "SET WINDOW DELETE" to get rid of the watch window. Then I issued:

- SET MAPLIBS DBCOLE.XDCZ21.XDCADATA;
- MAP+0;
- WHERE.

Here's the result in Figure 37 below:

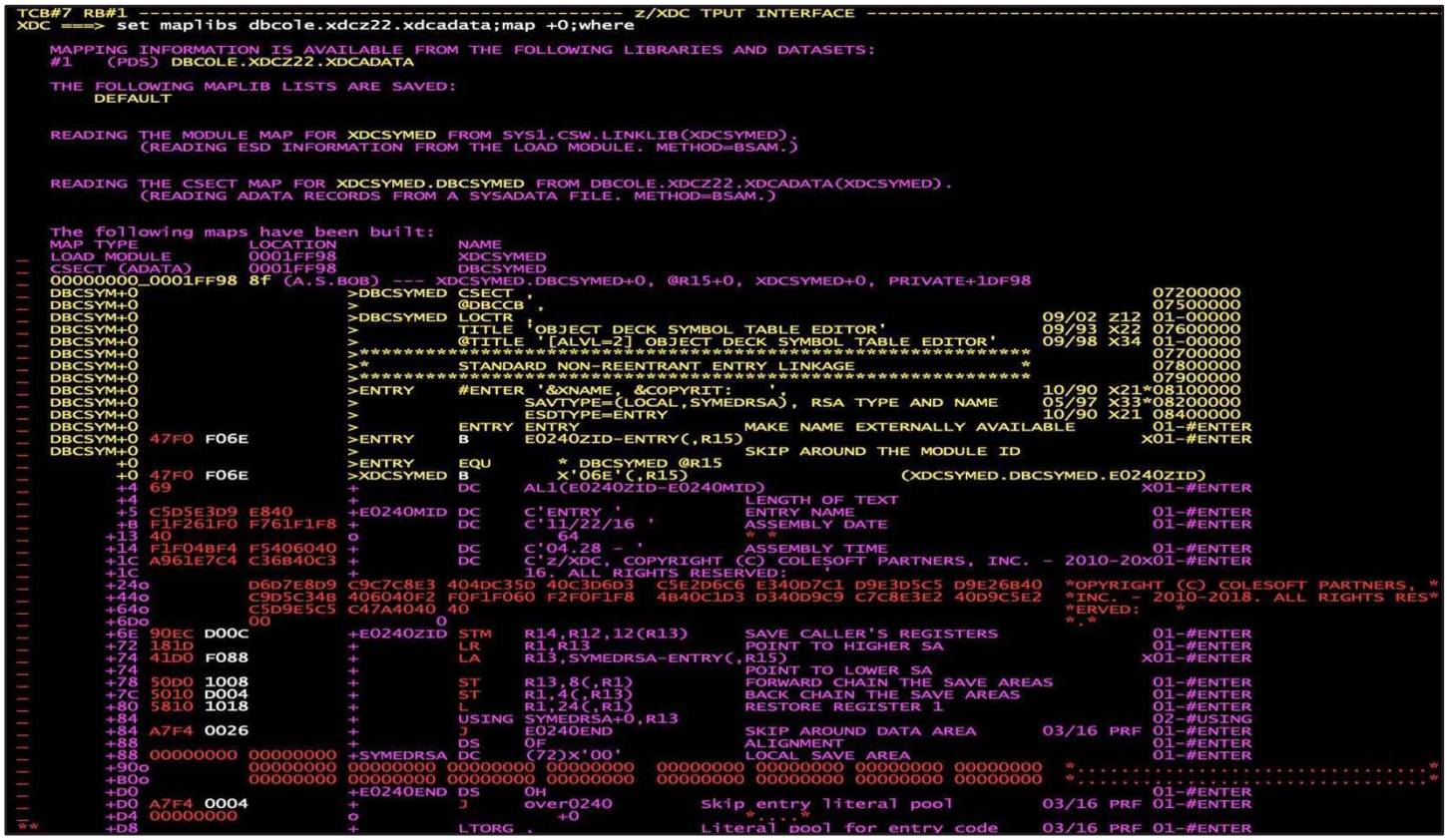


Figure 4-2.

Now, THAT's more like it. Now I see my comments and my code. All branch mnemonics will now be properly seen - life is wonderful.

It's also a bit too cluttered for Dave Cole. So, he's suggested that you have the option of using the FORMAT command with the OBJECT parameter. This will remove all the comments and clutter, leaving you with all the (long) names in your Assembly - without truncating them. It's the net effect of using our old SYM record support. I'm not going to bother to show that, but you can try this for yourself: FORMAT +0 OBJECT. Once you've seen what's there, issue a WHERE command and all the comments will be restored.

Sharp-eyed guys and gals will have noticed, “Hey, there’s TWO copies of the current instruction at XDCSYMED +0. What’s up with that?”

That’s a good question. z/XDC keeps TWO representations of each instruction when it stops at any instruction:

1. The first representation is the source line.
2. The second is the object code that it disassembled there.

If you were to zap the instruction to another opcode or change the operands (both of which you CAN do), you’d see “**CHANGED**” in the left portion of the display for that instruction. z/XDC is trying to point out that there CAN be differences between what the source code said at Assembly time, and what the object code is right now.

What if you mistakenly issued SET MAPLIBS for an out-of-date ADATA map for the object code you’re debugging? Sooner or later you’d see so many CHANGED messages that you’d have a clue as to what was wrong. It’s all part of our great service(!).

Managing Maplibs

MAPLIBS are cumbersome things, and take space. z/XDC has to provide for them in its own housekeeping which is why you’ll need a larger region size if you intend to make heavy use of mapping in z/XDC.

If you start getting very strange errors when you’re debugging with z/XDC, problems that go away when the program runs alone, then it’s probably because z/XDC didn’t have enough space to do its thing. Try setting REGION=0M, and this problem may disappear.

Remember, whatever you can create in z/XDC you may also LIST, and DELETE. Thus you can LIST MAPLIBS to see what’s currently in storage.

- You can also create lists of MAPLIBs for convenience. See Help MAps AData MAPlibs for more information.
- Maplibs are managed through a caching mechanism. So, you can also LIST and DELETE CACHE. See Help COmmands LIST CACHe for more information.
- ADATA wastes a LOT of storage, so recently Dave Cole wrote an ADATA filter that can radically reduce the size of these files. See Help MAps AData Excessadata for more information.

This is a Primer, not the User’s Guide, so I’m being purposely brief here. There’s one convenience I should not leave out. I have my own DEFAULT MAPLIB. You can have one too.

Creating A Default MAPLIB

In my simple world, I work entirely with XDCSYMED. It is, in effect, my current project. So, it’s convenient for me to have the my MAPLIB defined in my default profile so that AUTOMAP does its thing in the background.

The command sequence to do that is this:

1. SET MAPLIBS <dsn> <== Establish your MAPLIB.
2. SET MAPLIBS SAVE DEFAULT <== Make it the default.
3. PROFILE SAVE <== “Harden” the default profile to include the MAPLIBS setting.

Unfortunately, there isn’t (yet) a menu item in the PROFILE Facility to do this sort of thing. You have to learn this command sequence. Now you know it!

Once you create a default maplib setting (and you can include lists of maplibs in your environment), thereafter AUTOMAP will do it’s thing and you’ll see your source statements without any additional effort.

You can also get AUTOMAP to work for you in batch mode debugging. Just include a DD card naming “XDCMAPLB” pointing to a dataset containing your ADATA. Nice.

The SET QUALIFIER Command

z/XDC allows you to set a default module/csect name so that you can refer to names and symbols in your assembly by <.label-name> (that’s dot-label-name. The dot becomes significant).

As a painful illustration, you can see a label within XDCSYMED in the current panel named, E0133MID. Try this: Format E0133MID.

Instead of what you want (a Format at that line in the display), you get a nastygram: DBC028E NOTHING NAMED “E0133MID” WAS FOUND IN THE CURRENT ADDRESS SPACE (BOB).

Point of astonishment: E0133MID is RIGHT THERE on the page. What’s up?

You’re astonished because you haven’t given z/XDC a complete address expression yet. The COMPLETE address expression would have been FORMAT XDCSYMED.DBCSYMED.E0133MID. I’m not going to type all that, and you don’t have to either.

Instead, try the command, SET QUALIFIER +0. If you’re lazy like me, issue S Q +0. If you’re lazier than I am, put a Q Shortcut command in column one next to any line in the Working Window.

No matter how you accomplish it, you have established a default module, and a default csect. You can now issue Format **.**E0133MID and get what you expect.

Are you still astonished? *Remember to use the leading dot*. Now try it again. That’s better.

The MAP Helper Dialogue

To help you negotiate z/XDC's MAP command, there's a helper panel" If you issue MAP ?, you'll get that panel as in Figure 4-3 below.

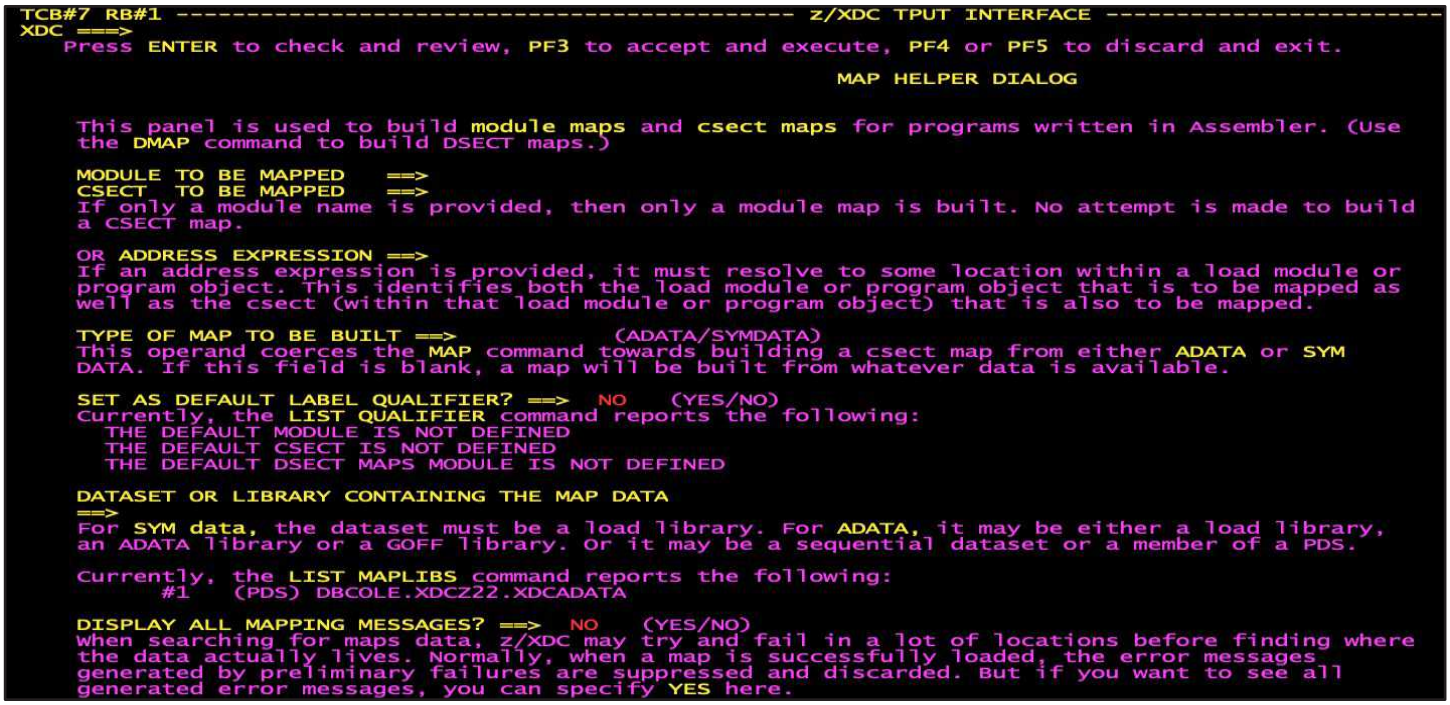


Figure 4-3.

Fill in the panel with the values you want and press Enter (or press PF4 to abandon the dialogue). Nice.

Creating Equates

If you count across to XDCSYMED+84 you'll see the hex characters 47F0. Does that look like a Branch to you? It does to me. Let's put an equate there.

I'll enter this: EQU FRED XDCSYMED+84 I; W

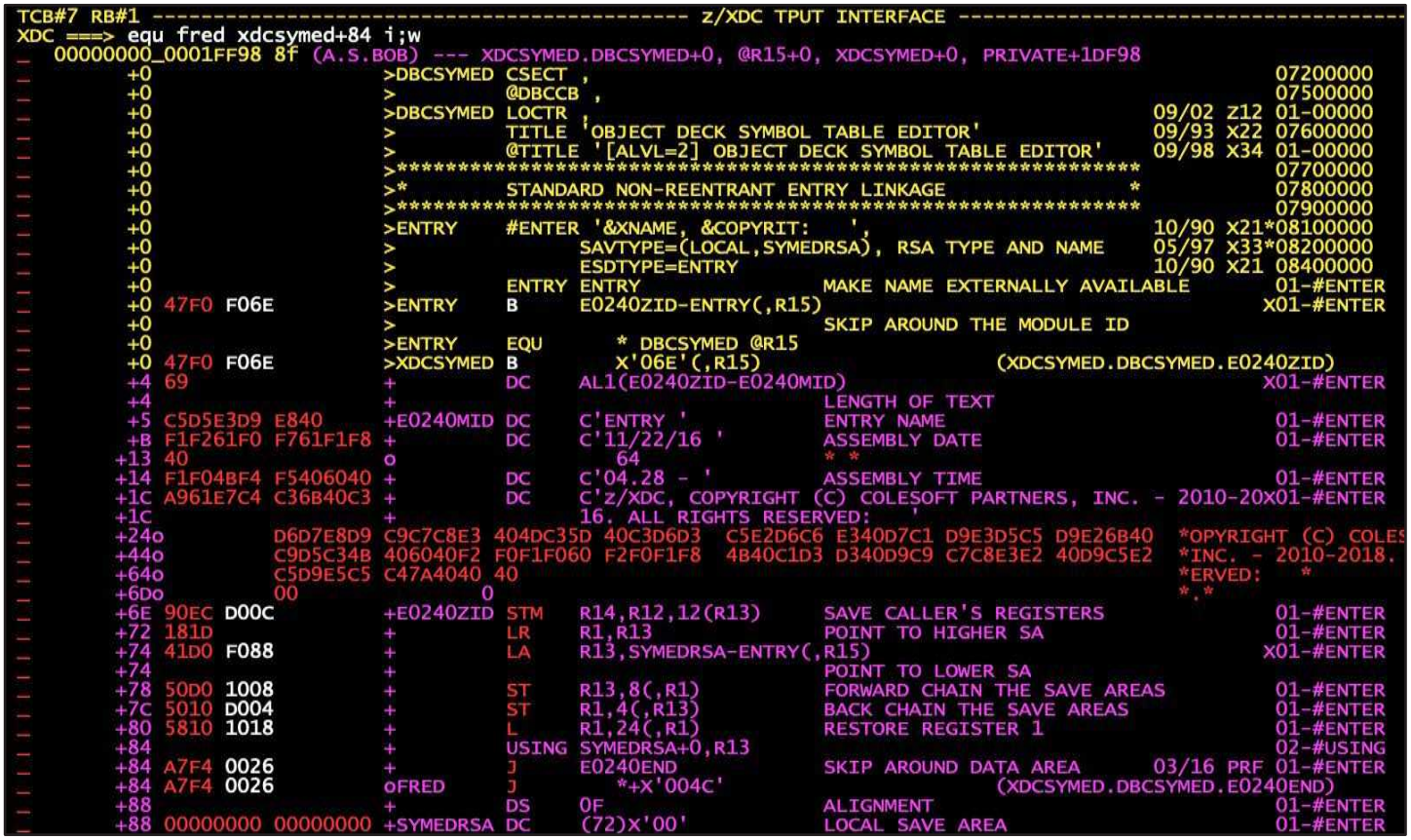


Figure 4-4.

Look way down in the screenshot below and you'll notice that the equate FRED has been placed in memory. The I parameter forces z/XDC to consider the equate to be at an instruction boundary so that z/XDC disassembles what it finds at the equate. If you used the D parameter z/XDC would be biased towards showing this as data. See Help COmmands EQUATE for more information.

Little Golden Book level syntax: EQUate Name <address-expression> I (or) D. That's all you need to know for now.

Remember, you also have the LIST EQUATE and DELETE EQUATE commands at your disposal.

At this point it's a good idea to digress just a bit, and talk about scripts.

Scripts, and z/XDC's READ command

z/XDC is deep and rich in functionality because it has to be. The downside is that it is a "verbose" tool - VERY "commandy". If you find yourself typing the same series of commands all the time in your debugging sessions, don't do that! Your time is way too valuable. Instead, consider writing a script containing those commands and all of them will be "batch processed" for you.

Create a sequential file full of z/XDC commands separated by semi-colons or Carriage Returns and save it. Then, z/XDC's READ command will open that sequential file for input, and will execute the z/XDC commands it finds there until the end of your script is reached (or some error is encountered).

This macro function allows you to do a lot of interesting things in z/XDC with a minimum of effort. For example, if you REALLY needed to create a bunch of EQUATE commands to fake a map of some object code, a script would be a very good way to do that.

Limitations of scripts:

1. At the time of this writing, scripts cannot be nested. We may expand this in future.
2. Scripts do not survive a "GO" command. So, if you issue a GO in a script that's effectively the end of your script. If you want more things to happen after the GO, create a second script.
3. Scripts are lists of simple declaratives. You don't have access to IF/THEN/ELSE.

Even with these limitations, Dave Cole says, "There are a LOT of neat things you can do with lists of declaratives. Have a look at HELP SCRIPTS." Please do so.

Pre-built Scripts That Dave Cole Made for You

The HELP SCRIPTS OURSCRIPTS panel shows you a list of pre-built SCRIPTS that you can use or adapt for your own work. If you page down you'll see a list of them along with a brief description of what they do. Some of them are fairly sophisticated. Put an S next to any scripts that intrigue you, press Enter and you can learn more about them.

Running Scripts under z/XDC

- To invoke a Script, you use z/XDC's READ command. The reference for that command is, of course, Help COmmands REAd.
- If a script abends in execution, it will stop at the command that it couldn't process. You can investigate the error, fix it and issue READ RESUME.
- If you take the time to master scripts and the READ command, you'll be able to automate a great deal of your work and save a commensurate amount of time.
- READ commands can be set to PFKey definitions. Wiggle one finger and make lots of cool stuff happen. Nice.

It seems that each of our customers has one user on site who ibecomes a script expert, and that person's pioneering work ends up being shared by the team. Perhaps you'll be that person.

The REXX interface

“But I really-really need to have conditional logic in my scripts!”, you say. OK, in that case you may want to investigate our REXX interface. The reference for that is HELP REXX.

z/XDC’s REXX interface allows you to create user-written “commands” that you can call from within your z/XDC session. That is beyond the scope of this Primer, so we’ll finish our digression, and go back to mapping.

Mapping DSECTS - the DMAP and USING commands

What about dsects? z/XDC can provide maps for dsects as well. For csects, you use the MAP command. For dsects, you use z/XDC’s DMAP command. Let’s play with it.

I’ll issue the LIST TASKS command. Here’s the display. I”m about to try to format TCB#9 (the current task) with my F line command:

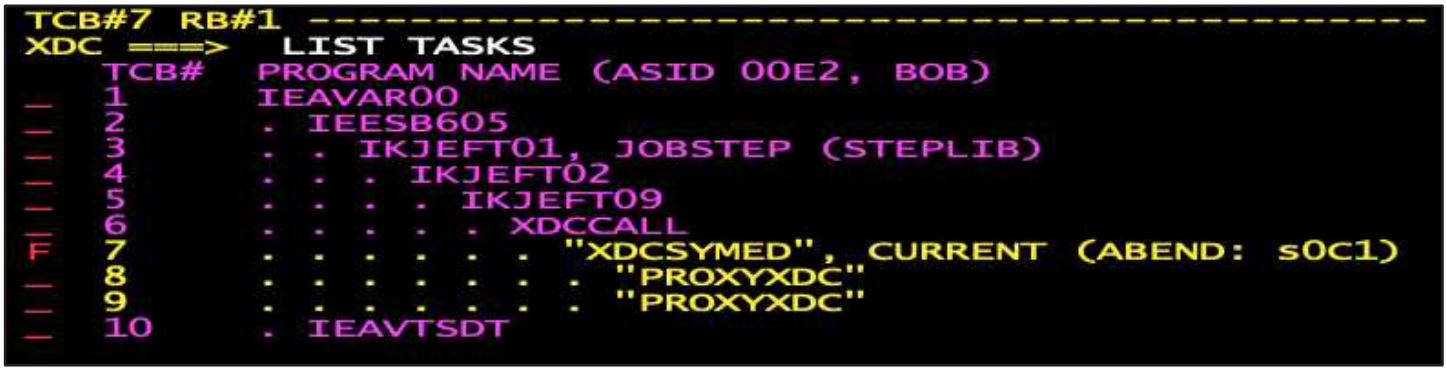


Figure 4-5.

Next, I press Enter and get this:

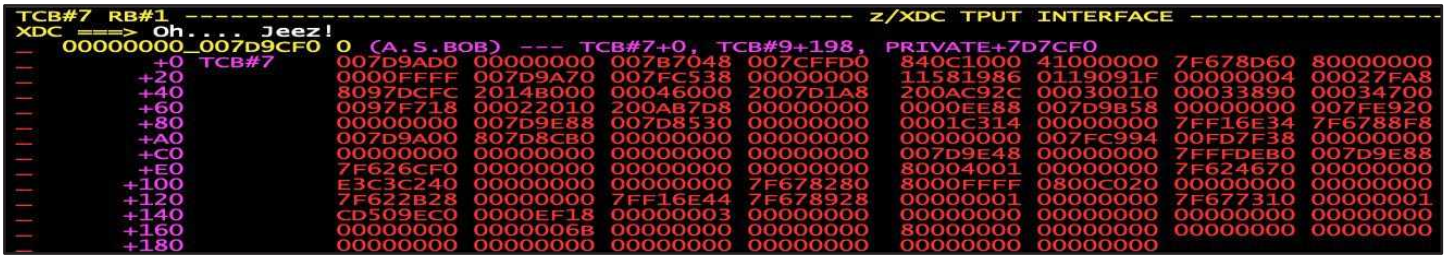


Figure 4-6.

E-yuck! z/XDC had no way of disassembling the TCB, so it reverted to hex-EBCDIC. Perhaps you can puzzle it out with your years of experience by just looking at the offsets, but I am much too lazy.

Let’s improve this display. We’re about to learn the DMAP and USING commands.

Issue DMAP XDCMAPS.TCBFIX. z/XDC goes out looking in a load module called XDCMAPS for a map called TCBFIX. It grabs TCBFIX and reads it into z/XDC’s storage.

Figure 4-7.

IBM Control Block Maps That Dave Cole Made for You

The XDCMAPS module contains over 1300 IBM control blocks already mapped out for you in ADATA record format. In the XDCMAPS module, each csect is a series of DS statements that make up a map of a single IBM control block.

XDCMAPS serves as a handy resource for programmers. It is also an example of how you might go about making up a load module of all your application's most treasured control blocks. Then, everyone on your team can share them. Of course, z/XDC handles the inline dsects in your assembly as well.

Note: There is a Helper for DMAP, too. Enter DMAP ? to see the panel. Then, fill in the values and press Enter. I didn't show it here to save a bit of space (and a tree or two). You get the idea.

The LIST MAPS command

Now, I'll issue a new list command, LIST MAP". This gives you a global view of all the maps defined within your debugging session:

Figure 4-8.

There are our Link-edit and CSECT maps for XDCSYMED an entry for TCBFIX. But TCBFIX's memory location is marked "INACTIVE". What does that mean?

The USING Command

z/XDC has no inherent notion of where you might choose to park a DSECT. Assembly language has a USING directive. z/XDC has a USING command.

Issue USING TCBFIX.TCBRBP TCB#n (where the n is your particular current task). Now TCBFIX's origin (TCBRBP) is assigned to the start of TCB#n (TCB#n is an EQUATE. EQUATE names resolve to address expressions. How convenient).

Now you can issue FORMAT TCB#n and there is your TCB dsect all formatted out, with its names and their values. You could also have issued LIST TASKS and used the "F" line command next to TCB#n.

I'll stack these commands using the semi-colon (our command delimiter) to save some time:

Figure 4-9.

This display is SO MUCH easier to understand! Remember, all of z/XDC's point-and-shoot capabilities are available to you here as well as the Shortcut commands. A question mark in column one will show you what Shortcut commands you may use here.

Floating Equates

The USING command has a number of parameters, one of which is the F parameter. If you issue a USING command, like this:

USING <map.origin> <address> F

Then, the origin point of the dsect map will float in memory with the resolution of <address>. That's perfect in situations where your input record is referenced by the contents of a register. In that case, you might issue this:

USING inrec.firstname R15? F

The above is a nonsensical example, but I think you grasp the implications. In fact, the F operand can be used with ANY USING command (not just those that refer to registers). This means that whole structures of floating maps can be created that automatically react when a program changes the organization of its control blocks. Now you can SEE field names and their contents very easily. The whole structure will react as the program does "in nature". If your map structures get complicated you might prefer to build a Script to do this sort of "spade-work".

Here's a quiz: In the XDCMAPS module, there's a dsect map for the SDWA. There is also a pre-built equate called, @SDWA. Can you create a dsect map for your System Diagnostic Work Area?

The USING Helper Dialogue

The USING command is a very complex command, because it has to be. There are a LOT of parameters to learn and explore, so Frank Chu has written a Helper Dialogue for it.

Figure 4-10.

Read the explanatory text and fill in the relevant parameters . Press Enter to run the command or PF4 to abandon it.

The AUTOCLONE Parameter

z/XDC's USING and EQUATE commands both accept an "AUTOCLONE" parameter.

Do you remember the LIST EQUATE command's output? You will have seen "RB#1" through "RB#3" and "TCB#n" equates for each of the Task Control Blocks in your address space. LIST ESTAE will also create equates like "SCB#1", "SCB#2", etc. AUTOCLONE gives you the same ability to create lists of equates that map elements in a queue or pointers in a chain. Such things might be named "PTR#1", "PTR#2, "PTR#3", etc.

You can use AUTOCLONE to create "lists" of equates that are named alike, and resolve to the elements in a queue, or pointers in a chain. It's a pretty esoteric topic, so I'll skip any display of it here.

You can reference this topic in the Help system by entering HELP COMMANDS USING AUTOCLONING, or HELP COMMANDS EQUATE AUTOCLONING.

The LIST LKEDMAP command

A while ago, I issued the LIST MAPS command, and you saw the Link-edit map, and the csect map names. That reminds me to show you the LIST LKEDMAP <address> command. Now that we've mapped our XDCSYMED module, that display will have more meaning for us. As usual, the definitive reference can be found at Help COmmands LISt LKedmap.

Here goes:

Figure 4-11.

End of digression. Back to the DMAP/Using commands.

Well, we've successfully mapped a single, static DSECT. That's nice, but there can be many of the same DSECTS placed in memory. In that case you can "clone" an existing DSECT definition by issuing DMAP newname oldname.

Issue DMAP NEWTCB TCBFIX. Now issue LIST MAPS. There is NEWTCB. It is inactive. Issue USING NEWTCB.TCBRBP TCB#8. Now you have mapped the eighth TCB in the address space. Do you see the possibilities?

Time to review. We have used a bunch of commands:

DISPLAY <a-e>
FORMAT <a-e>
LIST PSW
LIST REGS/FREGS/AREGS/CREGS
LIST Rn/Frn/ARn/CRn
LIST FIXED/FLOAT <a-e>
LIST TASKS
LIST RBS (TCB#n)
LIST SUBPOOLS (subpool-id)
LIST PGMS
LIST MAPS
LIST EQUATES
LIST RSA
LIST LSE (or LIST LSTACK if you prefer)
LIST TIOT
LIST ESTAE
LIST LKEDMAP

WHERE

END NOASK (you can also enter END COMPLETELY)

PROFILE

KEYS

TRACE

MAP

DMAP

USING

SET QUALIFIER

READ

We've introduced Shortcut commands and the point-and-shoot variants of z/XDC's commands.

You now know that there are Helper dialogues for FIND, DMAP, MAP and USING (just state the command's name with a blank, then a question mark).

I've probably skipped a few commands in the list above, so bear with me. Remember, to reference a z/XDC command, issue HElp COmmands <command-name>.

Independent Study: Issue Help COmmands LISt and investigate all the choices available to you. I bet you'll find something interesting that I didn't mention in this book.

Take a break. You've earned it. Next, we'll discuss breakpoints (AKA "traps") and tracing within z/XDC.

Chapter 5: Breakpoints and Traces

Our next subject is breakpoints (also known as “traps”) and traces. Let’s start our debugging session completely over. This will give you a chance to get familiar with ending a session and starting a session.

1. If you are in your session now, just issue END NOASK. You’ll be taken back to the z/XDC Startup Panel.
2. Enter Option 2 on the command line, tab forward to CMD/PGM Name ===> and fill the field with “XDCSYMED”. If it’s already there, you can skip this step.
3. Press Enter. You will see the first screen with informational messages on it.
4. Issue SET MAPLIB DBCOLE.XDCZ21.XDCADATA (your local equivalent).
5. Issue MAP PSW?.
6. Put a Q next to the csect on the resulting display. Now you have issued SET QUALIFIER for this module/csect.
7. Issue W, or WHERE. That will FORMAT where your program is stopped right now at XDCSYMED+0.
8. Issue T. You will now see the STM instruction that begins the entry linkage.

That was a little review exercise. You’re now ready to go.

Remember, the z/XDC command delimiter is a semi-colon (“;”). You can stack commands on the command line using this character.

Using X’00’ or TRAP2 Instructions for Breakpointing/Tracing

For decades, z/XDC has used X’00’ to create it’s breakpoints and perform its tracing functions. However, at z/XDC Release z2.2 and above you have a choice. You may use the classic X’00’ OR Trap2 instructions.

There are two consequences:

1. System overhead for processing abends is avoided. There’s no program check. The System’s Recovery Termination Manager does not get involved. No search occurs for the right Recovery Routine. Instead, with TRAP2 instructions, the hardware simply passes control to the Trap Handler routine, which in this case is z/XDC.
2. Also, the distinction between the Error Level and Retry Level environments (a z/XDC construct that the product has to manage) just goes away.

You can issue SET TRACE TRAP2 to initiate this action or SET TRACE ZERO to set it back. This can also be changed in your Profile under Display/Change FORMAT TRACE AND FIND Settings.

The Breakpoint Commands

There are six kinds of breakpoints (traps) in z/XDC, and eight kinds of traces. Each of these can take a conditional statement. Also, any of the traps and traces can have another z/XDC command (or commands) associated with the trap or trace. Thus, you can set a trap which only gives control to you under certain circumstances, and prior to giving you control, z/XDC issues another command(s) on your behalf. It's a far-reaching facility, limited only by your imagination. Here we go:

AT <address>

Sets a permanent breakpoint (retained until the end of the session, or deleted manually via the OFF command). Theshortcut command equivalent is A. The reference is found at, HELP COMMANDS AT.

ATX <address>

A hard to remove breakpoint. The Off command must specifically refer to the unique name of this command in order for it to be removed. The reference is found at HELP COMMANDS ATX.

T <address>

Sets a transient breakpoint (when reached by execution, these breakpoints are automatically deleted). The line command equivalent is T. The reference is found at HELP COMMANDS TRAP.

AD module.csect+offset

Sets a permanent deferred breakpoint (the breakpoint is scheduled for whenever the stated module is loaded into memory). The reference is found at HELP COMMANDS ADEFFERED.

TD module.csect+offset

A "transient" version of the above. The reference is found at HELP COMMANDS TDEFERRED.

HOOK <address>

This is a special kind of "breakpoint: that not only intercepts program execution, but also insures that an z/XDC environment is created in order to properly receive the intercept (more about this later.). The reference is found at HELP COMMANDS HOOK.

Creating a Permanent Breakpoint

I'll create a breakpoint on the command line aith AT +4 , and issue F_+0 so you can see the result:

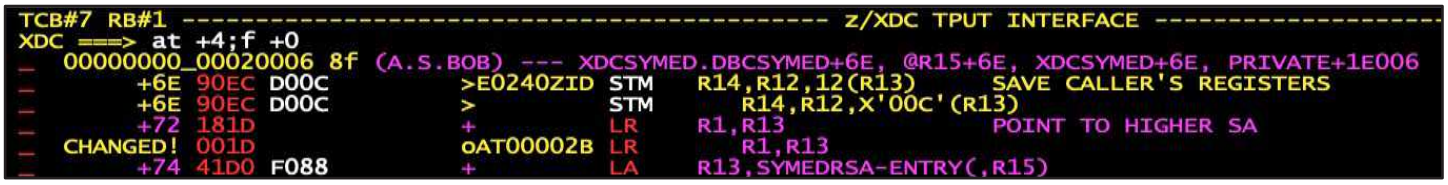


Figure 5-1.

All breakpoints receive a unique name, with a naming structure. Thus, the breakpoint name AT00002B would denote a permanent breakpoint (the AT prefix), a family-id of"00002 and a member name of B.

Here is where z/XDC's penchant for showing two representations of the current instruction come into play.

1. z/XDC puts the word CHANGED! into the SOURCE CODE representation of the display so that you KNOW something is now different from the source code as it was Assembled.

2. z/XDC has altered the OBJECT CODE opcode from a LR instruction to X'00', which will cause an S0C1 abend when execution reaches it.

Executing to a Breakpoint - the GO Command

The GO command will cause your program to resume. It will execute until either it hits another breakpoint, encounters a HOOK or abends on its own ("That never happens in MY code!")

If I issue the GO and then the Format_+0 commands at this point, here is what I'll get:

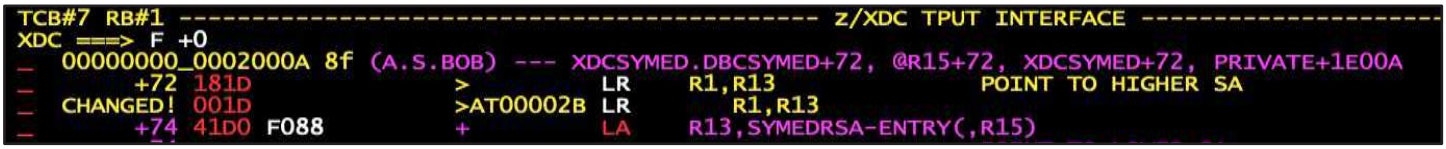


Figure 5-2.

Now, the Current Display Pointer equate (@CDP) points to the breakpointed instruction.

If you create a breakpoint from the command line and if the breakpoint's address is somewhere within the current display, z/XDC won't automatically display it for you. That is because z/XDC can't tell if the breakpoint is in this csect, this module under this RB or in another Task or *even in this address space.* So you may become used to re-issuing FORMAT or WHERE to accomplish that. You *could* set your breakpoint and ignore this, because sooner or later z/XDC will have to repaint the screen, or you can enter something like this: **AT <address>;F_+0**.

On the other hand, if you use the line command equivalents for breakpoints (A and T), then z/XDC DOES know that the command was entered on this screen and will automatically repaint the display for you,

Creating a Transient Breakpoint

You can create a Transient breakpoint either by entering T <address> on the command line or by using the T shortcut command in column one of any FORMATed display. Since I created a permanent breakpoint with the command line last time, I'll use the Shortcut command this time:

Figure 5-3.

Now you see a new breakpoint prefix of TR, which indicates that this breakpoint is a transient breakpoint. What's nice is that *when execution reaches a transient breakpoint, then that breakpoint is automatically removed*. You could enter GO at this point. z/XDC would release XDCSYMED for execution by the system and would capture control at the transient breakpoint. Then, that breakpoint would be deleted. Transient breakpoints are intended to be used only once.

The LIST BREAK Command

Remember our engineering principle: If you can create a structure in z/XDC then you can both LIST it and DELETE it. Let's go see our breakpoints:

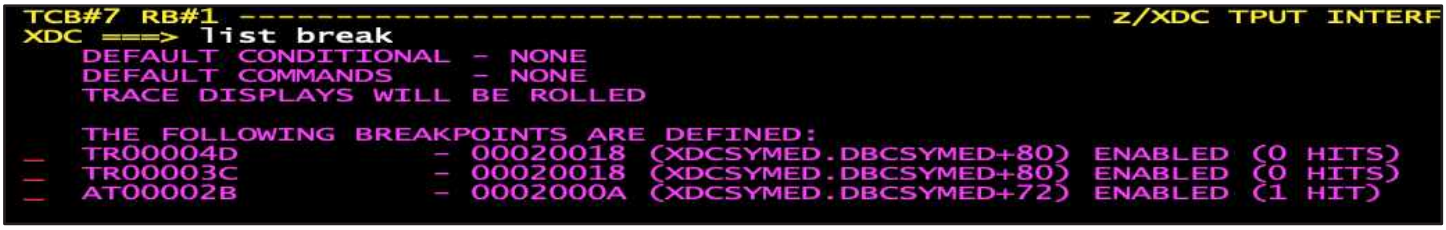


Figure 5-4.

You could have breakpoints all over the address space, and this display will serve as a central repository. Could you use the F Shortcut command to format at any of these places? Absolutely!

Deleting breakpoints

You can use the OF" command to delete breakpoints. From the command line you can issue OFF <breakpoint-name". An example would be OFF AT00002B, which would remove the permanent breakpoint in the display above.

It is SO MUCH easier to use the O Shortcut command to delete your breakpoint instead of using OFF from the command line. The O Shortcut command will work either in your FORMATed display or in the LIST BPTS display. Try it if you like.

You can also delete every breakpoint with OFF ALL, or simply the OFF command. Be careful with that. You may not care to delete ALL your breakpoints if you're deep in a complex debugging session!

Disabling/Enabling breakpoints

If you've been paying attention to the LIST BPTS display, you've deduced that there is information displayed for each breakpoint. You've probably grasped that the virtual address is shown, along with the relative offset of the breakpoint. You've seen that z/XDC keeps track of how many times a breakpoint has been "hit".

Now you're wondering, "What does ENABLED mean?"

z/XDC allows you to toggle a breakpoint so that its placement and name are retained, but the original opcode is restored. This allows you to turn off a breakpoint without deleting it by using the X Shortcut command. X will work in both the FORMATed display and the LIST BPTS display. Then, the descriptor in LIST BPTS will change from ENABLED to DISABLE". If you use X on the FORMATed display, then you can watch the opcode being restored to it's proper value from X'00". It's momentarily interesting, like watching the laundry go around in the dryer.

Try X if you care to. I'm going to issue OFF ALL in preparation for the next topic. I'll also issue F_ +0"to FORCE z/XDC to put the current instruction at the top of my display.

Creating Groups of Breakpoints

You can set multiple breakpoints on your currently formated screen by using multiple A or T shortcuts. This action will create a group of breakpoints all in the same family. In the panel belw, I've created three transient breakpoints:

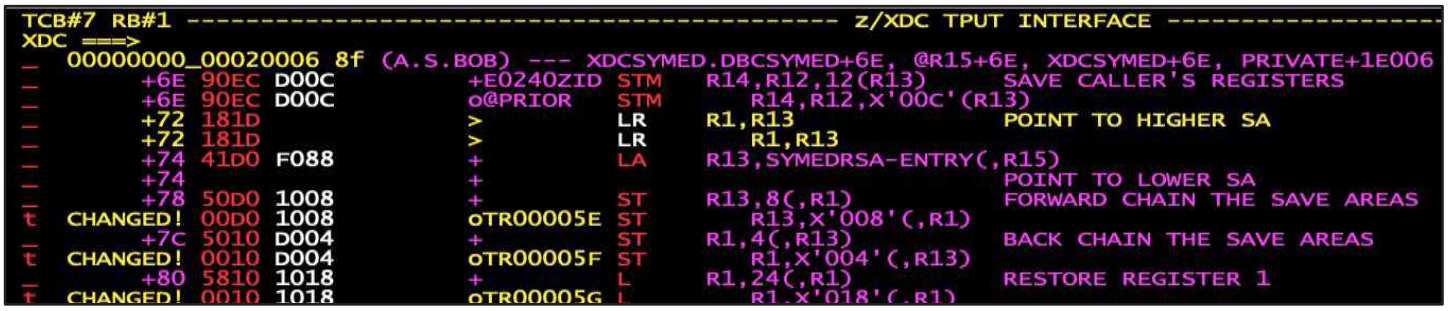


Figure 5-5.

If you were to type GO at this point, z/XDC would proceed to the first breakpoint in the family and stop execution there. *It would also automatically delete every other member of the breakpoint family.* This is really convenient in certain situations.

In fact, Dave Cole has used that scheme in the default setting for PFKey 11. That sets up a transient breakpoint using the NXSEQ Address Function so that six transient breakpoints are set at the next six instructions. Then, the GOT command is issued and execution is captured somewhere in the family.

Since you asked, "GOT" means "GO for Tracing". It causes the screen to be repainted only once. Now you know.

You might use PFKey 11 if you encountered a BALR type instruction out to a sub-routine, one that you do not care to trace through (at least this time). In structured program environments execution returns to the next instruction after the calling instruction. Dave's setting for PF11 handles this as well as the next six instructions (in case execution returns within this range of instructions). You wiggle one finger and the subroutine executes without your having to trace through it. Nice.

Deferred Breakpoints

z/XDC's deferred breakpoints are useful if you are working with a program that dynamically loads its own modules. Do you really want to start up a debugging session in the main program and then trace a gazillion instructions until the program you are really interested gets loaded? I didn't think so!

By entering AD module+offset or TD module+offse" we can schedule a breakpoint for whenever a module loads. As you might have supposed, AD makes a permanent deferred breakpoint, and TD makes a transient deferred breakpoint.

Since our toy program XDCSYMED doesn't load any other modules, you can't try it here. Perhaps your own work would be a better test-bed for this command.

That makes up the breakpoints. Here's a short review:

- You can set them on the command line explicitly or implicitly by way of Shortcut commands (A or T).
- You can see your breakpoints with the LIST BREAK command.
- You can delete them by issuing an OFF command on the command line or O as a shortcut command.
- You can toggle breakpoints with the X shortcut command to ENABLE/DISABLE them.

We will talk about HOOK further along in this booklet. It will be worth your time to become familiar with this important facility. But first, let's talk about the Trace commands.

The Tracing Commands

There are eight ways to trace program execution:

T	Trace a single instruction.
T B	Trace to next branch instruction.
T BY	Trace to next branch instruction that will branch.
T BN	Trace to next branch instruction *that will not branch.*
T *	Loop trace (stop whenever execution returns "here").
T SA	Trace to after the next storage alteration instruction.
T SB	Trace to before the next storage alteration instruction.
T W	Trace Watch (A special case. We'll explain this further down).

Note from Dave Cole on T BY: "By branching instructions I mean any instruction that will change the flow of execution. This includes B-*whatevers*, J-*whatevers*, SVCs, PCs - even such wierd things as LASPs, LPSWs and everything else that has the potential of changing the PSWs Instruction Address Field."

I think it would be a waste of space to demonstrate these commands in this book. Why don't you try a few and let us know if they DON'T work? THAT would get our attention, because this part of z/XDC *is reliable as a hammer*.

Caveat: The toy program XDCSYMED actually does useful work. It removes redundant SYM entries from a load module (which the Binder introduces in order to waste your disk space). Sooner or later, XDCSYMED is going to issue SVC 19 to open a file. There is no input file catalogued, so XDCSYMED will abend and you'll get some nasty-gram messages. If that happens to you, just restart the debugging session, go through your opening sequence (MAP, SET QUALIFIER, whatever) and keep following along.

Digression: Now that we're into trapping and tracing, there are two commands you'll find very useful. They are LIST BEA and LIST BRANCHES.

The LIST BEA Command

In z/XDC Release z2.1, Dave Cole introduced the LIST BEA command. The Breaking Event Address 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 (in this case) prior to the abend or breakpoint by which z/XDC received control. It is a great help for debugging wild branches to unreasonable locations. In effect, LIST BEA answers the question, How did we get HERE? In the case of XDCSYMED the BEA is pretty simple, as below:

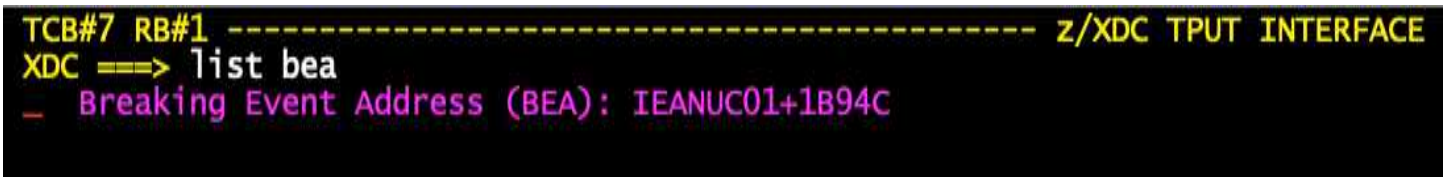


Figure 5-6.

The LIST BRANCHES Command

z/XDC keeps track of the trace commands done in a debugging session. Just now I did a few traces and issued the command. Here's the output:

Figure 5-7.

As your program executes, LIST BRANCHES populates the output of this command in LIFO order, with the most recent branch address at the top of the stack. How to learn more? Yes, you guessed it: The reference is Help COmmands LISt BRAnches.

Dave Cole on the Branch History Table

This is a table that z/XDC creates that records branching information every time z/XDC receives control. This includes times when z/XDC receives control but decides to let execution silently resume. Examples:

- During conditional traces and at conditional traces when the condition resolves false.
- During T BY traces at branches that do not branch.

To take advantage of this, do the following:

- Set multiple TRAPS, all with FALSE as the conditional expression. Example: TRAP here there somewhere else (FALSE)
- This means that whenever they are hit, they will always allow execution to resume.
- However, under the covers, z/XDC will record each hit in its Branch History Table.
- To actually recapture execution, use a normal TRAP wherever would be appropriate.
- Once z/XDC is back in control, issue LIST BRANCHES to see where execution has been.

The List Operands Command

In Release z2.1, Dave Cole introduced the LIST Operands command. This handy tool allows you to see the operands for both the current instruction and the prior instruction. These commands make the most sense when they are placed in a Watch Window, as I have done below. Beware: You can issue LIST OPERANDS for any <address>, but the farther you get away from the current instruction, the less likely LIST OPERANDS will be able to give you true results. So, it's best to stick with the defaults, LIST OPERANDS, and LIST OPERANDS PRIOR:

Figure 5-8.

For clarity above, I used two semi-colons, which has the effect of putting a blank line between the output of the two LIST OPERAND commands. This is purely for readability - another trick I've learned from Dave.

Conditional Statements

All the z/XDC breakpoints and traces can take an attached conditional statement. Conditional statements are of two varieties: counting conditions and value-testing conditions.

Counting Conditional Statements

Counting conditions give you control after a certain number of iterations. You can say something like "T * (50)" and an internal counter is initialized to zero. Each time execution passes through the address you specify, the counter is incremented and some time in the future (50 times in this example) you will get control again. z/XDC will accept up to 9999 loops for a counting condition.

Value-Testing Conditional Statements

Value-testing conditions allow you to compare a location's contents against a supplied value and return control when the desired relation proves to be true. You could enter something like: AT mymod.mycsect.mylabel (R15?,NE,0000FCAD). When the memory pointed to by the contents of R15 (in 31-bit mode) no longer equals X'0000FCAD' you'd get control.

You could also set a Value-testing Conditional statement to keep an eye on a storage location and give you control if it gets changed/corrupted. This helps you solve memory alteration bugs, which can be

really hard to find.

You can also use value-testing conditions in conjunction with traces, though for performance reasons you'd probably want to be sparing with them. You see, if I enter the command T (.mylabel,GT,'F0000000') z/XDC will make the comparison at each and every step of the program. You might want to press Enter on this puppy and go to lunch! This could bring the system to its knees. Anyway, it's going to take a while…

Here is a handy diagram on how to employ Value-testing Conditional statements with breakpoints:

Value-testing Conditions

Value-testing Conditions allow comparison of the contents of an object (located at some address expression) to a constant that is supplied in the Conditional Expression. All relational operators (Equal, greater-than, less-than, greater-than-or-equal-to, etc.) are supported. You can even enter a bit mask and do the comparison at the bit level by way of "AND-ing" or "Or-ing" bit strings against the supplied bit mask. Here is the syntax of a Value-testing Conditional Expression:

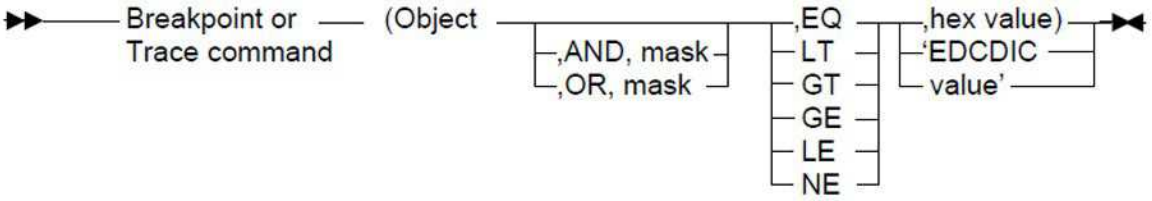


Figure 5-9.

Trace Watches

Now that we've shown you Conditional Statements we can FINALLY discuss the last Trace command: Trace Watch.

A Trace Watch is a conditional expression that is not associated with any breakpoint or trace, but is evaluated at every breakpoint or trace.

Trace Watches can't stop execution by themselves. Thus, in order for a Trace Watch to be effective, z/XDC has to receive control from time to time by other means.

Dave Cole on Trace Watches

From Dave himself:

I often find it useful to create conditional expresses that are not associated with specific traces or traps. This is what TRACE WATCH (conditional expression) does.

- You can create any number of WATCHs.
- WATCHs are evaluated when:
  - When z/XDC receives control due to a TRACE or TRAP.
  - And the TRACE or TRAP has a conditional expression,
  - And that expression resolves FALSE.
  - Then the WATCHs are evaluated one by one.
- If any WATCH evaluates TRUE, then z/XDC stops your program's execution and passes control to you at the terminal.
- On the other hand, if all WATCHs evaluate FALSE, then z/XDC allows your program's execution to resume.
- In other words, WATCHs are logically OR'd both amongst themselves and with respect to the trap or trace by which z/XDC received control:
- If any of them say "stop", then the program is interrupted, and the user receives control.
- All of them must say "don't stop" before z/XDC will allow program execution to silently resume.

So I use Watches as follows:

- I will set any number of TRAPs or ATs wherever appropriate, all with a conditional expression that is simply (FALSE). Example: TRAP <here there somewhere else> (FALSE)
- I will then create one or more TRACE WATCHs each having a suitable conditional expression.
- I will also set an unconditional trap somewhere appropriate as a guaranteed stopping point.
- I will then issue GO to kick it all off.

Note, a LIST BRANCHES command might have useful information in this scenario as well.

The TRACE WATCH command accepts all of the same operands as all other TRACE, TRAP, and AT commands.

Unfortunately, I am reluctant to recommend trying T BY (FALSE) Conditional Trace.

- Yes, it will follow your program's execution
- And yes, it will populate the Branch History Table
- And yes, you can set up suitable WATCHs and unconditional TRAPs to stop execution under appropriate circumstances.
- **BUT** T BY will follow execution nearly everywhere... including into subroutines both as deep and as far as they go! That's a rat hole that frequently the trace will never climb out of... Especially if (for example) ESTAE[X] macros are encountered along the way.

However, I am currently working on a new kind of trace called T BNC (Branch No Call). It will work like a T BY except that it will not follow linkage instructions (JAS, BASR and friends) into subroutines. Once that is published, a T BNC (FALSE) will be a much more viable command to use. I will then re-publish the AUTOTRCE script, which will use the T BNC tracing command - *Dave*

Causing a Trace Watch to Stop Execution

"But wait", you say. "You said that a Trace Watch can't stop program execution by itself. I might find that a VERY useful thing." We hear you.

You may be content to set a WATCH and let it be checked along with your other breakpoints/traces, OR you could become a bit more pro-active and set any other breakpoint or trace and specify the conditional expression (FALSE), like this:

T BY(FALSE)

Then, your WATCH will be checked not only at any of the other defined breakpoints or traces but ALSO at each successful branch-type instruction, without stopping execution of the program UNLESS the WATCH tests TRUE at one of these points.

So, let's recap this useful technique:

1. You set up any number of Trace Watches that you need (and there's your compound condition).
2. You set up a T BY (FALSE).
3. You enter GO.
4. z/XDC gets control at each of the successful branches in your program. It stacks and processes all the Trace Watches. If all of them prove FALSE, then control is returned to the system and your program continues to run. If any of them prove TRUE, then execution stops at the last T BY command and gives you control.
5. You could issue LIST BRANCHES to see the most recent Branch (in the BEA) OR all branches that z/XDC has "witnessed" during the current session.

Really, you can use ANY trap or trace command with the (FALSE) or (TRUE) parameters. T BY is just darned convenient! Read Help Breakpoints Conditional Watch and Help COmmands Trace Watch for more information.

The global reference for this entire chapter can be found at HELP BREAKPOINTS. Breakpoints are a

HUGE topic within z/XDC's Help system, and well worth your time spent studying them.

z/XDC and Storage Alteration Bugs

Sooner or later you will encounter a storage alteration bug, where address X is being over-written and you have no idea who the culprit is. I get this question often, and our normal response is "For some situations SLIP is a better alternative." That's hardly enough information for you.

Recently Dave Cole had to answer this question, and as usual his response is comprehensive. Here's what Dave had to say about this sort of problem:

Unfortunately, while there are many great things that can be done with z/XDC, monitoring for storage alterations is not one of them, at least not in the broad, general sense that one usually thinks of when confronting this problem.

For that kind of monitoring, you'd need to use IBM's SLIP command. It's documented in the z/OS MVS System Commands manual (ieag100_v2r3.pdf), and also here in Knowledge Center.

Note that the SET SLIP,SA,[...] command allows you to define what storage you want to monitor, and what you want done when the storage alteration finally occurs.

One of the actions you can request is ACTION=RECOVERY which is IBM-speak for "abend the guy" (abend code s06F). Well, if z/XDC happens to have already been established as a recovery routine for the program being abended, then he will get control when the alteration occurs, and you can then use your session to investigate.

z/XDC does have a much more limited monitoring that sometimes will give you a better, more targeted result than you can achieve with SLIP, but there are limitations:

1. While SLIP can monitor storage for alteration by anyone anywhere, with z/XDC you must choose specific locations in your program's execution at which z/XDC will check to see whether or not an alteration of interest has occurred.
2. In other words, z/XDC cannot be used to broadly monitor a location in storage and trap anyone who is altering that storage no matter what code is doing it. For that kind of monitoring, where you have no idea where to start, you'll have to use SLIP.
3. But if you already have some idea of where to start looking, such that you can choose specific points in your code where checking will help you narrow the culprit down, then z/XDC can be helpful.

In other words, you have to be able to pick one or more specific spots within your code where it would be helpful for z/XDC to check to see whether or not an alteration has occurred. If you can do that, then you can use conditional traps to monitor one or more locations of interest.

A "conditional trap" is a breakpoint which, when execution reaches it, z/XDC receives control and performs one or more tests that you define. If any test evaluates as TRUE, then your program's execution is stopped, and you will receive control at your terminal within z/XDC.

On the other hand, if all tests evaluate as FALSE, then your program's execution is silently resumed as

if nothing happened.

When you can use conditional traps, there are some advantages they have over SLIP:

1. You can be selective about what alterations matter vs. what don't. For example, if the problem you're having is not that a field is being changed, but rather that someone is making it negative (or whatever), z/XDC can trap for that. In other words, if some changes are ok while others are not, conditional traps can be selective about which alterations to trap for.
2. The target of a conditional trap need not be a specific location in storage. The location checked can be different each time it is checked. For example, if you have multiple instances of a cblock (such as cblocks on a chain or in a table), and the corruption involves an instance of a cblock that differs from one time to the next, you can craft a conditional expression that is based on an anchoring field or register, or that is based on a table index, or whatever. Thus, you could automatically test a different instance of the cblock each time the conditional trap is reached.
3. Using what we call trace watches, you can create multiple tests to be performed at every conditional trap. Thus, you can monitor for multiple issues that might occur at multiple locations.

These kinds of fine tunings are not possible with SLIP.

For further information, we have a comprehensive series of topics in the Built-in Help starting at HELP BREAKPOINTS CONDITIONAL. Give that a read, then get back to us as more specific questions arise.

I hope this helps,

Dave Cole

Associating Commands with Breakpoints and Traces

Whether you use breakpoints or traces, and whether you use conditional statements or not, you can associate other z/XDC commands with any breakpoint or trace command. This means that whenever the breakpoint or trace is done, another z/XDC command or group of commands will be executed on your behalf. The associated command delimiter is a single quote (“‘”). You type a single quote, then a string of z/XDC commands that are separated from one another by semi-colons.

You could enter something like AT .MYLABEL ‘LIST R12;D MYEQUATE’, and then issue GO. When the breakpoint at .mylabel was reached you'd see the contents of R12 as well as the memory at the location pointed to by the equate MYEQUATE.

Let me try another command string:

T BY ‘LIST RWREGS;LIST PSW F;F +0 10;LIST BEA;RETRIEVE’

With this one command string I accomplish:

- A Trace to the next successful Branch, then en-masse;
- A list of the entirety of the 64-bit registers;
- A complete display of the PSW;

- A Format for ten source code lines;
- a display of the Breaking Event Address, and;
- A Retrieve of the entire command string. Now I can just keep pressing Enter and keep gathering all this information into my session log!

Sanity check: The single quotation mark is used to set the outer boundaries of commands strings that you associate with breakpoints, traces and Trace Watches. The semi-colon (";") is used to delimit z/XDC commands within that command string or on a command line.

Associated commands allow you to stack additional z/XDC commands for execution at a breakpoint or in conjunction with a trace, but it doesn't end there. You could also invoke an z/XDC Script via the READ command and accomplish a great many things. You could also call up a special PROFILE and configure the screen or other session characteristics to the needs of the moment. z/XDC supports a TSO command exit, so you could go into the TSO environment at a breakpoint for just a moment and issue a TSO command or even a CLIST.

You are limited only by your imagination. There is a LOT of engineering here, all designed to allow you to combine z/XDC's "primitive" functions into surprisingly sophisticated sequences. You can stack commands, run scripts when a conditional statement proves true, change your PROFILE (and your PFkeys or screen colors) when a breakpoint is reached.

Back-Tracing your program

Dave Cole recently taught me a new trick. Perhaps it's an old trick, but it was new to me. You can "back-trace" your program's execution. Here's how that works:

Dave avoids using the single-step "T" command, because he feels that's tedious. Instead, he uses "T BY" which traces from successful branch to successful branch. In the default Profile that we distrib-tute with z/XDC, PFKey 10 is set up to perform a "T BY".

Here's the cool trick: Once you've been tracing your program for a while, you may issue "T" then press PageUp. This will show you the previous Branch address. You can also issue "UP T" from the command line if you like. This is a great way to see where your program has been.

If you prefer to follow execution in a "forward" direction, you can issue "UP MAX" to take yourself to the beginning of the session log, and issue "T" and PageDown. Alternatively you can use "DOWN T" to achieve the same result.

Learning this, I decided to try it. However, I created a new Watch Window and populated that window with a LIST BRANCHES command. Now, when I issued “T-PageUp” or “UP T” I could keep track of the branching addresses as I went along.

Here I’ve set up such a screen layout. I’ve issued PFKey 10 several times and then a single “UP T”:

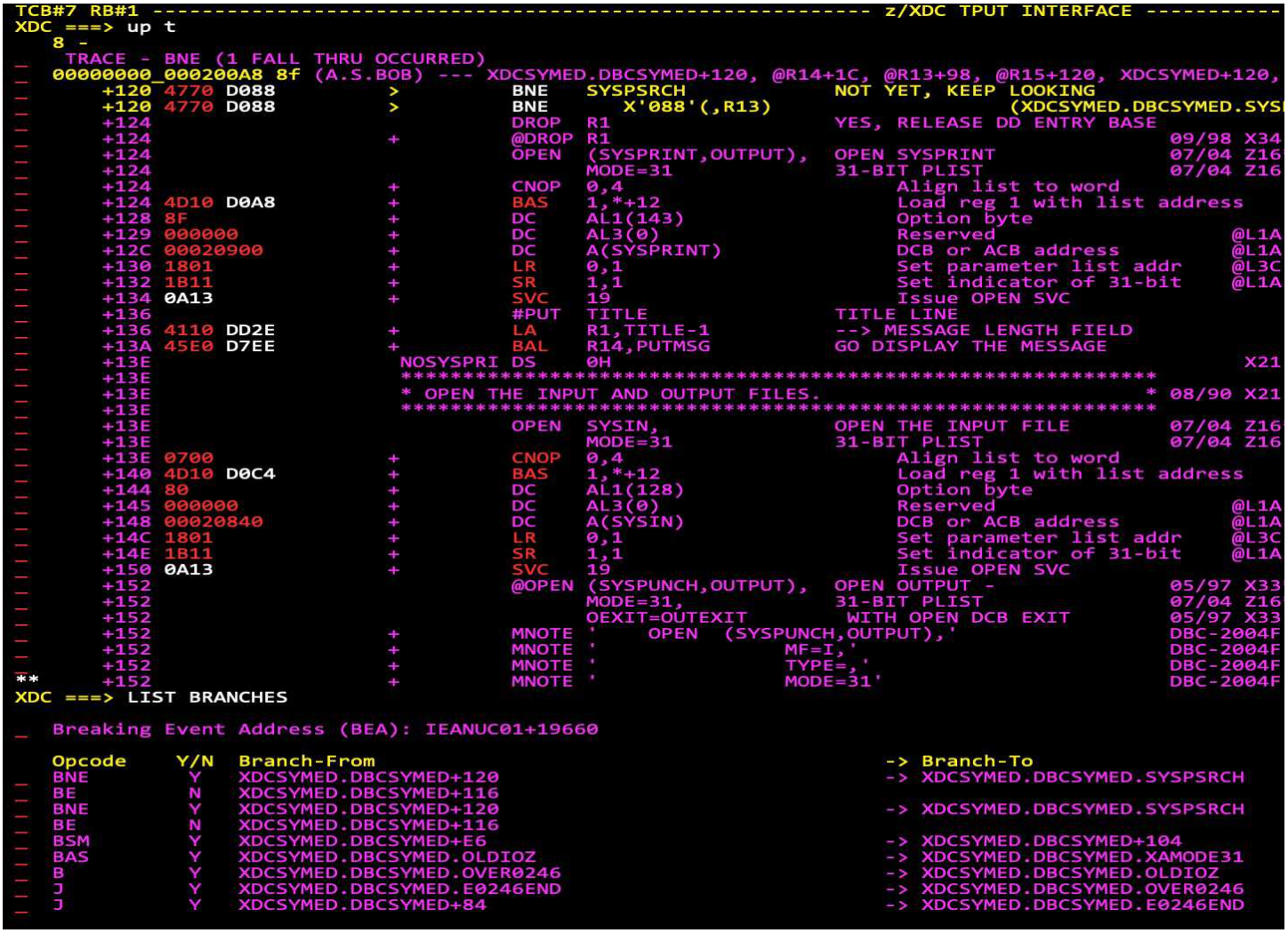


Figure 5-010

If I were a real programmer, this might help me to keep track of where I am whenever I back-trace my code.

That’s the breakpoint/trace facility. Fool around with it. Ask questions. You can always Enter HElp COmmands {AT | AD | TD | T}, etc.

Chapter 6: Changing Things

We have come a long way in this book. We've learned to invoke z/XDC. We've learned to look around"with DISPLAY, FORMAT, LIST and FIND. We've MAPped and SET QUALIFIER with our csects, and used DMAP/USING to work with DSECTS. We've learned how to set breakpoints and perform traces. Now perhaps we have found "a problem" in our code. Now we can change things in order to test the functionality of our fix.

You might use the J Shortcut command to"jump the PSW around in order to do unit-testing of a section of code. You might change a register or memory value to trigger an event you'd like to trace. If your program is APF-authorized you can literally go-anywhere/do-anything. You have a powerful facility for use in z/XDCs environment alteration commands.

Here's a brief overview of what environment alterations are possible:

ZAP - used to alter:	• Opcodes and operands • Assembler instruction mnemonics • Memory locations • Register contents • The Instruction Location Counter in the PSW
J Shortcut command	• Alter the ILC portion of the PSW to point to any instruction.
SET - used to alter the PSW's:	• Address mode • Address Space Control Mode • Condition Code • External Interrupt Mask • IO Interrupt Mask • Execution Key • Execution State (problem state/Supervisor state)
GETMAIN/FREEMAIN	Create/delete storage in any subpool.
COPY	Move data areas around in storage.
LOAD/DELETE	Obtain/delete copies of modules.
POST	Post an Event Control Block.
HOOK	Create an debugging session for z/XDC wherever you like.

Be Careful when Changing Things

Now is a very good time for me to remind you to BE CAREFUL when you use z/XDC's environment alteration commands, ESPECIALLY if you're executing APF-authorized.

You can't hurt anyone but yourself when you work on XDCSYMED unauthorized in your TSO region's Private Area. But out in the wide-wide world - if you're authorized - you can subvert or alter a running production system.

- **YOU MUST BE CAREFUL WITH z/XDC!**
- **YOU CAN CRASH AN OPERATING SYSTEM IN THE TIME IT TAKES TO RELIEVE PRESSURE ON THE ENTER KEY!**

Here's a scary example. Normally z/XDC will not follow a subroutine's execution - it won't leave the Private Area of an address space. So, if you come to SVC 19 and you do a T command z/XDC will NOT trace the SVC and you'll receive control at the instruction following the call to SVC 19. Kind and gentle.

Whether you're running z/XDC APF-authorized or not, Tracing a Supervisor Call (SVC) or any system routine will have no "bad" effect on you or anyone else. That is because z/XDC has a default behavior of SET TRACE LOCAL. It won't allow the scope of a trace to leave the Private Area of an address space.

However, IF you are APF-authorized AND you issue SET TRACE GLOBAL, then when you encounter an SVC or a subroutine and you trace it, you will now be looking at the entry point to the SVC or subroutine. *Think about it*: You've introduced an S0C1 abend into a facility that may be called hundreds of times while you're sitting there thinking. NOT KIND. NOT GENTLE.

There are perfectly acceptable reasons that you may need to do just this sort of work legitimately for the greater good of your enterprise. z/XDC is a sophisticated tool for extremely knowledgeable and capable software engineers. That's what it was built to do. But, like a chainsaw, which you can use to cut up a pile of wood, z/XDC will do your bidding. In the blink of an eye, and before you can say "OH NO!" z/OS has crashed. That chainsaw did exactly what you made it do - cut off your leg!

That's why we have very strong language in our Lease Agreements on the subject of liability. If you use a perfectly functioning copy of z/XDC to do something careless or evil, we won't accept the blame. BE CAREFUL.

If you've just been terrified, GOOD. I want you to be. Now for some good news:

z/XDC and System Security

z/XDC makes SAF calls to your security system whenever a user tries to do something "interesting". You can control z/XDC completely with security rules.

- Some folks may not be able to access z/XDC at all.
- Others may use it only for non-authorized code.
- Others may be allowed to go-anywhere/do-anything within a certain group of programs.
- Others may be allowed to do anything they like - wherever/whenever they like - because their jobs require it.
- Some pure development shops run without any security. Crashes and "learning experiences" are expected, yet unwelcome events.

You can control z/XDC's capability with security rules. This makes the issue political, not technical. The definitive resource on this topic can be found at HELP SECURITY. Please review it.

In many cases it is better to work with z/XDC on isolated, test-only systems, especially for working with authorized code, the development of SVC routines, system exits and the like.

Someone like a Dave Cole could use z/XDC to fix a broken production system, but he's very, very good and far more careful than the normal programmer. Perhaps you are too. Vaya con Dios.

The ZAP command

ZAP, or the Z Shortcut command equivalent, can be used to change memory, opcodes/mnemonics and register contents. Let's play with this capability.

Zapping from the Command Line

Here, I issue LIST RWREGS, then I ZAP RW0 with a new value. I then enter two semi-colons (to create a blank line) and enter another LIST RWREGS. z/XDC changes the display of the register to a high-visibility color to remind me that I've changed it.

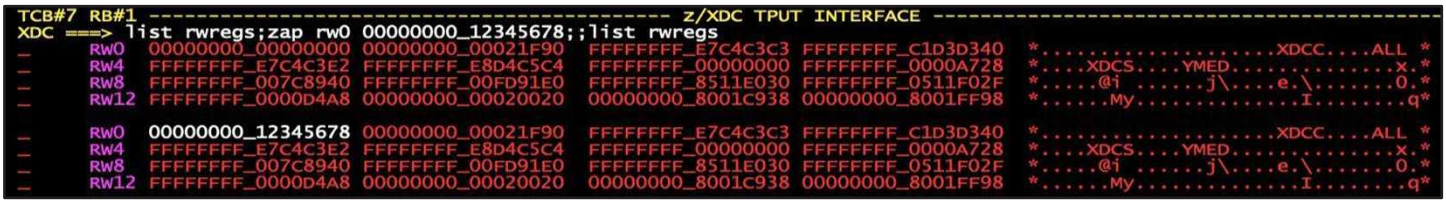


Figure 6-1.

I could just as easily issue ZAP PSW <address> and z/XDC would alter the ILC portion of the PSW, changing the current instruction for you. Alternatively, the J line command forces the PSW to point to any line of code desired.

The reference is, of course, HELP COMMANDS ZAP. Above I'm using the command line form of the ZAP command which is (generally) ZAP <address> =<string>. If the string is EBCDIC, I use single quotes around the string value.

Other examples:

ZAP R15,01234567	This forces the calue 01234567 into R15.
ZAP .myflag,.stuff	The address that the label ".stuff" resolves to is stored at the location named ".myflag".

I have found that I can use a comma (or not) and the equal sign (or not) in the ZAP command. z/XDC doesn't seem to care either way.

The VERIFY Command

z/XDC also offers a VERIFY command that has the same syntax as the ZAP command. This allows you to write scripts that change things, like a SUPERZAP. Sometimes it's useful to check the contents of memory before we over-write them. The reference is, of course, Help COmmands Verify.

Zapping using the Z Shortcut command

I prefer to use the point-and-shoot method of zapping. To do this, tab down to the one-byte area in column one next to the desired line of code or memory and put a Z into it. Now, tab wherever you like and change whatever you want with the new value. If the cursor doesn't land on the place you want to change after a tab, then you can't change that thing by overtyping (like the PSW). Using the point-and-shoot scheme, you can overtype object code, or opcodes. This is a handy way to change program values on the fly.

You cannot normally overtype a two-byte opcode with any opcode that's wider. z/XDC keeps track of instruction length and will not normally allow you to do so. HOWEVER, if you really want to overlay an instruction opcode with an opcode of a different length, you CAN do it if you issue SET NOILC. This turns z/XDC's Instruction Length Checking off. Vaya con Dios.

If you want to change a register, put a Z next to its line in the display window and overtype it. It's just that easy.

You can also change hex or character data. Essentially anywhere the cursor stops as you tab across the panel can be changed by over-typing what's already there! Here's one example, where I wrote into a patch area in XDCSYMED:

Figure 6-2.

Here's another, where I altered an instruction at XDCSYMED+74from LA to STM (naughty of me; I know):

Figure 6-3.

z/XDC has put the word CHANGED! into the display to remind me that I've done the zap command. The old STORE instruction is gone.

How Do You Un-do a Zap?

Basically, you have to remember what the value of the thing you changed was BEFORE you zapped it. There's no Edit/Undo command in z/XDC. My advice is DISPLAY what you're going to change, and use PageUp into the session log to see what the value was - or merely trust that a PageUp will contain what you need in order to un-do the change. Then you ZAP again to remove the change you made.

The SET PSW Command

If you want to change the resume address in the PSW, that's a simple ZAP PSW <address> command, or the J shortcut command. This changes ONLY the Instruction Location Counter in the PSW (the next instruction that will be executed). But there are times in which you may want to change other portions of the PSW. The SET PSW command does that.

You can issue SET PSW <field> <value> to alter the contents of the Program Status Word, as below. SOME of the changes here require that you be APF-authorized.

SET PSW (amode)	You can enter 24, 31 or 64. Use LIST AMODE to verify what you've done.
SET PSW (Address Space Control Mode)	You can enter PRIMARY, SECONDARY, HOME and AR.
SET PSW (Condition Code)	Too many choices to list. See HELP COMMANDS SET PSW CC
SET PSW (External Interrupt Mask)	EXT=ON EX=OFF There are synonyms for these functions, but these two are straightforward.
SET PSW (I/O Interrupt Mask	IO=ON (and various other ways to set it on) IO=OFF There are synonyms for these functions, but these two are straightforward.
SET PSW (Key)	Changes the storage access key in the retry level PSW. SET PSW KEY=<range from 0 to F>
SET PSW PROG	Enables or disables the four Program Mask bits in the retry level PSW SET PSW PROG=<h>, where <h> is a single hex digit.
SET PSW	Change the execution state of the retry level PSW. You can enter PROBLEM or SUPERVISOR. You must be running z/XDC APF-authorized in order to issue this command.

The GETMAIN/FREEMAIN commands

z/XDC allows you to issue the GETMAIN command to obtain or free additional storage.

Here's the syntax: GETMAIN <length> <subpool-number> <HIGH/LOW>

The HIGH/LOW parameter controls whether your new storage will be above or below the line (in 24-bit or 31-bit storage).

When you issue GETMAIN, two equates are created for you automatically. The first is AREA, which always references the storage most recently GETMAIN'd. That's so you can write scripts that refer explicitly to AREA. The second equate is AREA#1-n where n is the number of times you've issued GETMAIN. So the second GETMAIN'd area would receive an equate of"AREA#2, but AREA would be updated to the value you just GETMAIN'd and remain current.

[If you understood my use of the word "GOTMAINED" then you're just as twisted as I am. Poor you.]

Let's try GETMAIN. I'll ask for 100 (in hexadecimal) bytes in Subpool 78:

Figure 6-4.

Now, let's go look at it. I'll issue D AREA and z/XDC will show it to me.

Figure 6-5.

If I prefer to see this storage as ASCII, I can issue SET FORMAT ASCII, and redisplay the stoage. That, however, would bias ALL of z/XDC's subsequent storage displays. I might find it better to issue D AREA ASCII and see ASCII only for the duration of that DISPLAY command.

What can I do with my GOTMAINed storage? Anything I like. Remember, you can zap code into storage, JUMP the PSW somewhere and execute the code. Voila! You have created a patch area.

Your storage could also be a repository of some data that you wish to keep, for later reference.

Our intention is this: "Anything your program can do programmatically, you can do interactively." You can simulate conditions, move the PSW around and do unit tracing, test "what if" scenarios, even create lethal inputs to test the robustness of your solution.

The COPY Command

One day when thinking about this I had a great new idea for a z/XDC command: A Move Characters Long (MVCL) command!

Breathlessly, I went to Dave Cole. "Dave, I have a great idea! Why not make an MVCL command? It would do a Move Characters Long operation and we could populate GETMAIN'd areas of storage with data from other addresses!" I was SO proud of myself.

Dave fixed me with a stare, "Bob, have you bothered to read about the COPY command?" Sure enough, I read HELP COMMANDS COPY, and I never asked that particular question again.

Briefly, the syntax is: COPY <target-addr> <target-length> <source-addr> <source-length>. The reference may be found at "Help COmmands COPy".

The LOAD/DELETE Commands

The command syntax is “LOAD <module-name> <loadlibrary-name/omitted>.

You can choose to load or delete a module. In that case, z/XDC will issue a LOAD SVC for the desired module. If there is a reusable or reentrant copy already in storage, z/XDC will increment the use count by 1.

Below, I’ve issued a LOAD for IEFBR14:

Figure 6-6.

Notice all the good information you get about the module you’ve loaded. It’s just part of the service...

To delete a module, just issue DELETE <module-name>. z/XDC will issue a message <module-name> DELETED. It’s just that simple. You don’t really need a picture of this, do you?

The POST Command

I’ve never POSTED an Event Control Block in my life, but perhaps you will.

The syntax is POST <address> <hi-order-code/low-order-code> You can omit the values for <hi-order-code> and <low-order-code> and z/XDC will make assumptions for you - probably the values that you want anyway.

How did I know that? The reference is HELP COMMANDS POST.

Below I’ve issued POST AREA;D AREA. I figured that it wouldn’t hurt to do a POST operation on my GETMAIN’d storage at AREA. Sure enough, z/XDC put a X’40’ into the first byte:

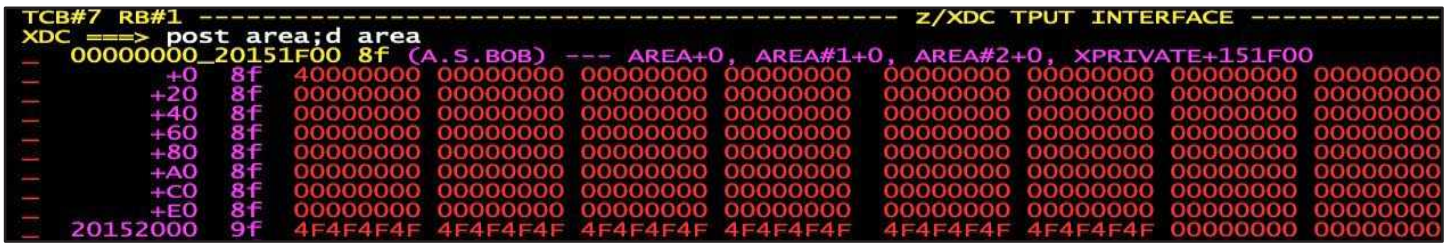


Figure 6-7.

That’s it for the environment alteration commands. Again, anything your program can do programatically, YOU can do interactively. By now perhaps you’ve grasped the implications for automating some of your more repetitive tasks. Scripts can be equated to PFKeys. Alterations could be invoked from within a Script (remember to use the VERIFY command!). The more time you put into designing fancy

scripts, the less time you'll spend typing.

The HOOK Command

In any serious Assembler application it's probable that there are a host of recovery routines built to protect the base code in case of improper input and other oddities. The idea is to shield the code by handling errors in a graceful manner. If the application ABENDs and comes down with a dump that's NOT graceful.

z/XDC is itself an recovery routine, which means that it's the abend handler/debugger for your code *right up to the moment that your own code invokes another recovery routine*. After that z/XDC will lose control of the debugging session - and this new recovery routine handles the S0C1 abend instead of z/XDC.

The answer to this problem is HOOK. The appropriate reference is Help COmmands HOok. You may consider Hook as the *Uber-breakpoint*. No matter what, you WILL get control where you want it.

HOOK allows you to create a breakpoint that builds a z/XDC session on the fly. No matter what recovery routines exist at the moment you invoke it, HOOK will ensure that z/XDC is the most recently invoked recovery routine for your debugging session. So, for example, if you KNOW that an ESTAE is going to be invoked in your own code before you hit a breakpoint, use a HOOK instead. Remember, you also have the LIST ESTAES command to show you where your current ESTAES are.

In z/XDC Release z21 Dave and his team completely re-wrote Hook. Hook is now able to operate in non-SVC-friendly environments. So, a Hook may be set in all the traditional enviroments but also in PC Routines, SRB Routines, FRR-protected code, Locked code, Disabled code - just about anywhere! The Hook may be placed in common storage if you care to do so. You may specify the ASID or the ASNAME of the address space in which you want your Hook. The address space doesn't have to exist at the time you set a Hook.

Hook is available in two varieties, dynamic and static:

A dynamic hook can be invoked:

- As an interactive command: HOOK <address>"
- As a line command: K.
- As a deferred command: HD <module-name+offset>.

A static hook is a macro: Assemble #XDCHOOK into your base code.

When you invoke a dynamic HOOK, z/XDC take 6 bytes of storage where you have set your hook, and puts it into storage in a control block called "HOOKCBE". When the Hook fires, the Hook is deleted and the 6-bytes is moved back into your code stream.

Be careful. If a HOOK is placed into a module in common storage, *it can be executed by any program which calls a routine containing a Hook.*

If a HOOK is set in any address space, it persists for the life of that address space or until it is executed. A LIST HOOKS command will only be able to see the Hook while it's active. If a debugging session is

terminated, *the Hook still exists*. Again, be careful.

Here I am going to set a HOOK four bytes past the STM in the XDCSYMED program with the K Shortcut command:

```
TCB#7 RB#1 ------------------------------------------ z/XDC TPUT INTERFACE -------------------
XDC ===>  Using the "K" shortcut command to set a hook...
_   00000000_00020006 8f (A.S.BOB) --- XDCSYMED.DBCSYMED+6E, @R15+6E, XDCSYMED+6E, PRIVATE+1E006
_        +6E 90EC D00C          >E0240ZID STM    R14,R12,12(R13)      SAVE CALLER'S REGISTERS
_        +6E 90EC D00C          >         STM      R14,R12,X'00C'(R13)
_        +72 181D               +         LR     R1,R13               POINT TO HIGHER SA
k        +74 41D0 F088          +         LA     R13,SYMEDRSA-ENTRY(,R15)
_        +74                    +                                     POINT TO LOWER SA
```

Figure 6-8.

I press Enter and this happens:

```
TCB#7 RB#1 ------------------------------------------ z/XDC TPUT INTERFACE -------------------
XDC ===> Voila!
_   00000000_00020006 8f (A.S.BOB) --- XDCSYMED.DBCSYMED+6E, @R15+6E, XDCSYMED+6E, PRIVATE+1E006
_        +6E 90EC D00C          >E0240ZID STM    R14,R12,12(R13)      SAVE CALLER'S REGISTERS
_        +6E 90EC D00C          >         STM      R14,R12,X'00C'(R13)
_        +72 181D               +         LR     R1,R13               POINT TO HIGHER SA
_        +74 41D0 F088          +         LA     R13,SYMEDRSA-ENTRY(,R15)
_        +74                    +                                     POINT TO LOWER SA
_   CHANGED! C0F40001           HOOK0001           -1057751039        *{4..*
_        +78 50D0 1008          +         ST     R13,8(,R1)           FORWARD CHAIN THE SAVE AREAS
_   CHANGED! 7512                                   +29970            *..*
```

Figure 6-9.

You can see the Hook next to the CHANGED! reminder (HOOK0001). Next, I enter the GO command and see:

```
TCB#7 RB#1 ------------------------------------------ z/XDC TPUT INTERFACE -------------------------------------------------------
XDC ===> Actually z/XDC suppressed all the information about my new session. I had to back up in the log to show it to you.
   19 - go
.  DBC830I XDC z2.2 (Level FHC-1903E)
.  DBC830I ENTERED NONAUTHORIZED, AMODE(31), UNDER RB#3 FROM TCB#7 IN BOB
.  DBC830I (ASN=00E2.3)
.  DBC891I XDC z2.2 ENTERED AS AN ESTAEX OWNED BY RB#7
.  DBC831I THE ERROR LEVEL AND RETRY LEVEL ENVIRONMENTS ARE THE SAME

_  DBC842I DEAD-TRAP CMDS: EQUATE HOOKCBE RW13~INDIRECT(64) DATA;SET ZAP SPROT;TRAP
                           RW1~INDIRECT(64,ZEROOK) 'DELETE HOOKS HOOKCBE;WHERE;DOWN M-1' ZERO
                           SAVE=NO REMOVAL=PURGE;GOT
_   TRACE - LA
   >>> DELETE HOOKS HOOKCBE <<<
   >>> WHERE <<<
_  00000000_0001FF98 8f (A.S.BOB) --- XDCSYMED.DBCSYMED+0, @R15+0, XDCSYMED+0, PRIVATE+1DF98
_        +0                    DBCSYMED CSECT ,                                                    07200000
_        +0                             @DBCCB ,                                                   07500000
_        +0                    +DBCSYMED LOCTR ,                                  09/02 z12 01-00000
_        +0                             TITLE 'OBJECT DECK SYMBOL TABLE EDITOR'    09/93 x22 07600000
_        +0                    +        @TITLE '[ALVL=2] OBJECT DECK SYMBOL TABLE EDITOR'  09/98 x34 01-00000
_        +0                     ********************************************************************  07700000
_        +0                     *        STANDARD NON-REENTRANT ENTRY LINKAGE                     *  07800000
_        +0                     ********************************************************************  07900000
_        +0                     ENTRY    #ENTER '&XNAME, &COPYRIT:    ',                10/90 x21*08100000
_        +0                              SAVTYPE=(LOCAL,SYMEDRSA), RSA TYPE AND NAME     05/97 x33*08200000
_        +0                              ESDTYPE=ENTRY                                   10/90 x21 08400000
_        +0                    +        ENTRY ENTRY              MAKE NAME EXTERNALLY AVAILABLE    01-#ENTER
_        +0 47F0 F06E          +ENTRY   B     E0240ZID-ENTRY(,R15)                              X01-#ENTER
_        +0                    +                               SKIP AROUND THE MODULE ID
_        +0                    oENTRY   EQU     * DBCSYMED @R15
_        +0 47F0 F06E          oXDCSYMED B      X'06E'(,R15)
_        +4 69                 +        DC    AL1(E0240ZID-E0240MID)                            X01-#ENTER
_        +4                    +                               LENGTH OF TEXT
_        +5 C5D5E3D9 E840      +E0240MID DC   C'ENTRY '        ENTRY NAME                         01-#ENTER
_        +B F1F261F0 F761F1F8  +        DC    C'11/22/16 '     ASSEMBLY DATE                      01-#ENTER
_       +13 40                 o                64            * *
_       +14 F1F04BF4 F5406040  +        DC    C'04.28 - '      ASSEMBLY TIME                      01-#ENTER
_       +1C A961E7C4 C36B40C3  +        DC    C'z/XDC, COPYRIGHT (C) COLESOFT PARTNERS, INC. - 2010-20X01-#ENTER
_       +1C                    +              16. ALL RIGHTS RESERVED:  '
_       +24o          D6D7E8D9 C9C7C8E3 404DC35D 40C3D6D3  C5E2D6C6 E340D7C1 D9E3D5C5 D9E26B40  *OPYRIGHT (C) COLESOFT PARTNERS, *
_       +44o          C9D5C34B 406040F2 F0F1F060 F2F0F1F8  4B40C1D3 D340D9C9 C7C8E3E2 40D9C5E2  *INC. - 2010-2018. ALL RIGHTS RES*
_       +64o          C5D9E5C5 C47A4040 40                                                       *ERVED:    *
_       +6Do                   00                 0                                           *.*
_        +6E 90EC D00C         +E0240ZID STM    R14,R12,12(R13)      SAVE CALLER'S REGISTERS    01-#ENTER
_        +72 181D              +         LR     R1,R13               POINT TO HIGHER SA         01-#ENTER
_        +74 41D0 F088         >         LA     R13,SYMEDRSA-ENTRY(,R15)                        X01-#ENTER
_        +74                   >                                     POINT TO LOWER SA
_        +74 41D0 F088         >@CDR     LA       R13,X'088'(,R15)
```

Figure 6-10.

I now have a SECOND debugging session. You can see that z/XDC has deleted the Hook and done a bit of other housekeeping.

The LIST HOOKS command

As you might expect, you now have a use for the LIST HOOKS command. However, because my previous Hook already fired and was deleted, I had to create another for this screenshot:

Figure 6-11.

I also (quite predictably) have a DELETE HOOKS command, OR I can put an O or X next to any Hook in my LIST HOOKS display.

Chapter 7: Working with Other Address Spaces

Up until now we've been working exclusively on XDCSYMED in the Private Area of our address spaces. It's time to expand our scope a bit.

There are two ways in which to access other address spaces with z/XDC. One is Cross Domain Facility, which allows you to run z/XDC on programs in a Batch Region. This is probably how you'll use z/XDC most of the time, and we'll be talking about that in the next unit. The other way to use z/XDC in another address space is to invoke Foreign Address Space Mode, or "FASM" for short. FASM is pretty cool. I'll talk about it first.

Foreign Address Space Mode

Dave Cole conceived of FASM originally as a handy way of checking out another address space. It was never designed as a way of changing things - only as a "look-don't-touch" interface. However over the years we've experienced some mission creep. Nowadays (if your security system lets you) you can ZAP the address space, and you can HOOK it.

To invoke FASM you have to execute z/XDC as an APF-authorized program. So, I'll issue END NOASK and close out the program. BUT WAIT: If you followed along with me and created one or more HOOKs in your address space, z/XDC has merely taken you one step back to the next most-recent debugging session. You want to get ALL the way out, so you may care to issue END COMPLETELY.

Here I am back at the z/XDC Startup Panel. I'm going to run XDCSYMED as an authorized program. You can see where I've put Option 2 into the command line. I've also answered "YES" to "Should CMD/PGM start APF auth?"

```
 ---------------------------- z/XDC STARTUP PANEL ----------------------------(v6)
 OPTION  ===> 2                                          XDC's name ===> XDC
 1  XDCCMD   - Debug TSO CMD processor - will be passed a  CPPL
 2  XDCCALL  - Debug other PGMs in TSO - will be passed an OS PARM field
 3  XDCCDF   - Debug background jobs or tasks via  cs-cdf/XDC

 PARAMETERS FOR OPTION 1 OR 2 (FOREGROUND DEBUGGING IN TSO):
   Does CMD/PGM use ISPF services? (YES/NO) ===> NO    Dialog APPLID ===>
   Should CMD/PGM start APF auth?  (YES/NO) ===> YES   Quick Start?  ===> NO
   Should AUTOSTEP be allowed?     (YES/NO) ===> YES
   Should RENT and REFR pgms be loaded into key-8 Storage?  (YES/NO) ===> YES
   Script DSN  ===>
     CMD/PGM Name  ===> XDCSYMED                    Upcase the PARM? ===> YES
     CMD/PGM PARM  ===>

     Tasklib DSNs  ===>
                   ===>
                   ===>
                   ===>
                   ===>
                   ===>
     or DDNAME     ===>
 PARAMETER FOR OPTION 3 (BACKGROUND DEBUGGING VIA cs-cdf/XDC)
   Connect to cs-cdf/XDC using your current TSO Userid? (YES/NO) ===> YES
```

Figure 7-1.

I press Enter, and z/XDC comes up. Note that I am now running APF-AUTHORIZED:

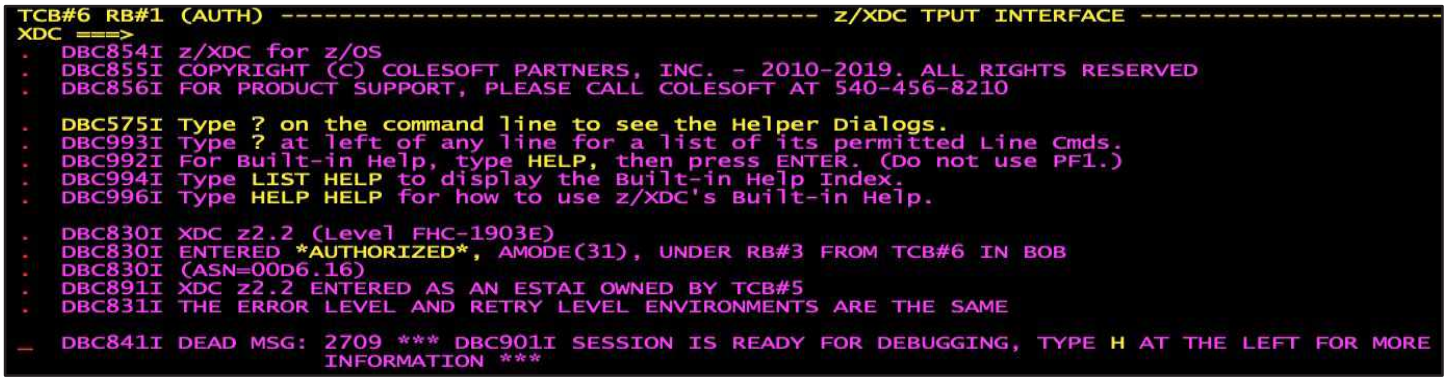


Figure 7-2.

At this point you may want to sit back and watch me work. *Please DON'T follow along.* I don't know whether you CAN execute z/XDC APF-authorized. I also KNOW that you don't have a test-bed piece of code to play with as I do. For now, follow along with me, and when you're confident (and have a test address space to monkey with) you can try this on your own.

Invoking Foreign Address Space Mode

To invoke FASM, you issue SET ASID <job-name> or SET ASID <nnn> where nnn is the number assigned to the address space in the Address Space Vector Control Table. Being the simple sort I am, I will issue SET ASID WTOR. WTOR is a toy program that I can play with without messing up a REAL job in our development system.

Figure 7-3.

At this moment I'm not merely looking at the WTOR address space. According to Dave Cole, I AM the WTOR address space. He considers it an important distinction. He's the boss, so it IS important!

In the FASM environment I have access to only a subset of z/XDC's commands. So, my Looking Around commands all work. I can FIND, DISPLAY, FORMAT and LIST to my heart's content. I CANNOT set a breakpoint. I can't do tracing. That's because at the moment, z/XDC is not the ESTAE for any code in this address space. Not yet, anyway.

Leaving Foreign Address Space Mode

When you're done poking around in another address space with FASM, enter "SET ASID HOME. You can also SET ASID <blank> and either way z/XDC takes you to your home adress space. All is well. You may end the authorize debugging session or go to another address space if you wish. But, in this case I want to keep going with FASM to show you something pretty darned cool. Again, please follow along with me.

Remember, if system security allows me, I can ZAP. And. if system security allows me, **I can HOOK!** You see where this is going?

How To Hook a Running Program with Foreign Address Space Mode

Let's orient ourselves. I'll issue LIST TASKS;;LIST RBS TCB#4. I can't merely issue LIST RBS because in the FASM context z/XDC doesn't have any concept of a current task.

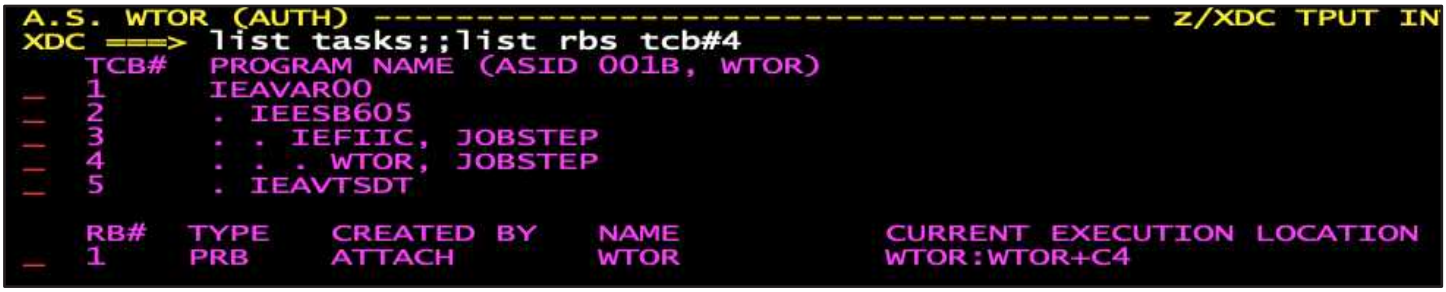


Figure 7-4.

Now I can put an F next to RB#1 to FORMAT where WTOR is executing. I'll skip showing you that step.

OK, here's the WTOR program, a very small piece of code. I'm going to set a HOOK in this program:

Figure 7-5.

I press Enter and see that the HOOK has been set:

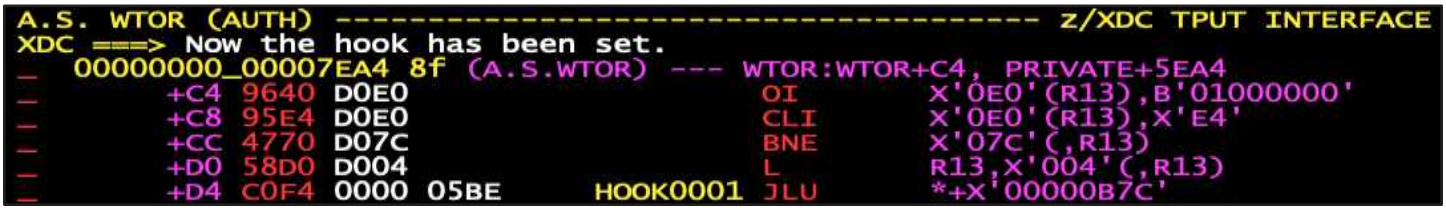


Figure 7-6.

That is all I want to do in this address space. Now, I issue SET ASID HOME and get out of the WTOR address space.

Figure 7-7.

My next step is to give the WTOR address space an interrupt, one that will cause it to execute the hook that I just set. So, first I issue “END NOASK” to close out my session and enter =i.log so I can see the Operator Console:

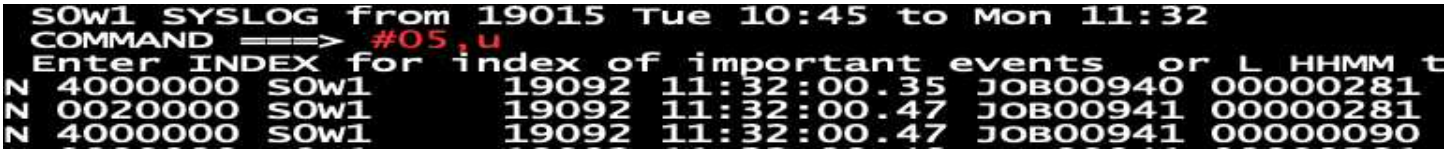


Figure 7-8.

Job #05 refers to my WTOR job, so I issue #05,U to provide it with an interrupt. This will start up WTOR and execute the hook. Next I navigate to the z/XDC Startup panel and issue Option 3 to invoke the Cross Domain Facility’s Router task:

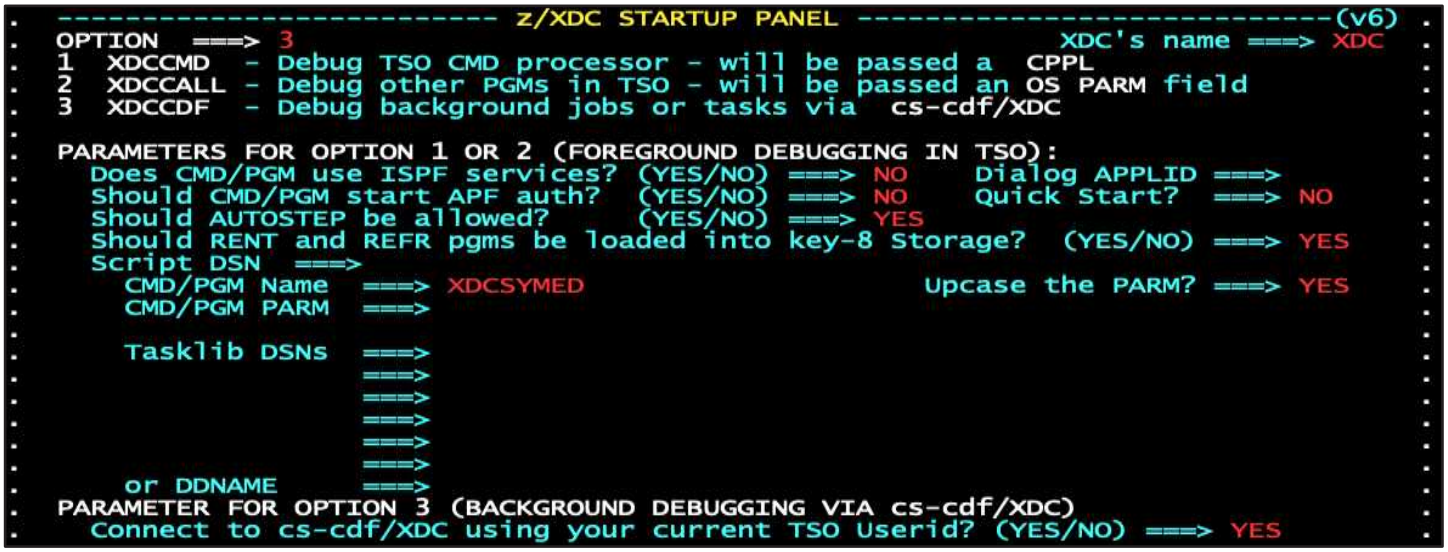


Figure 7-9.

When I press Enter, I see a list of jobs waiting to be debugged. I put an “S” next to WTOR:

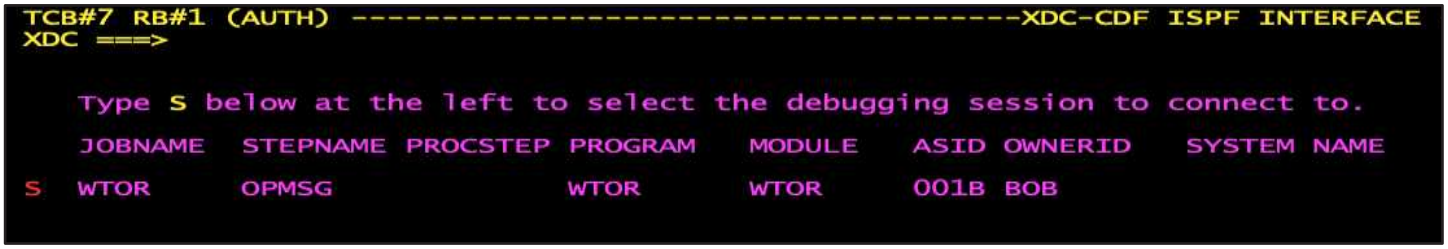


Figure 76.

Hey, look at that! I'm now debugging the WTOR job!

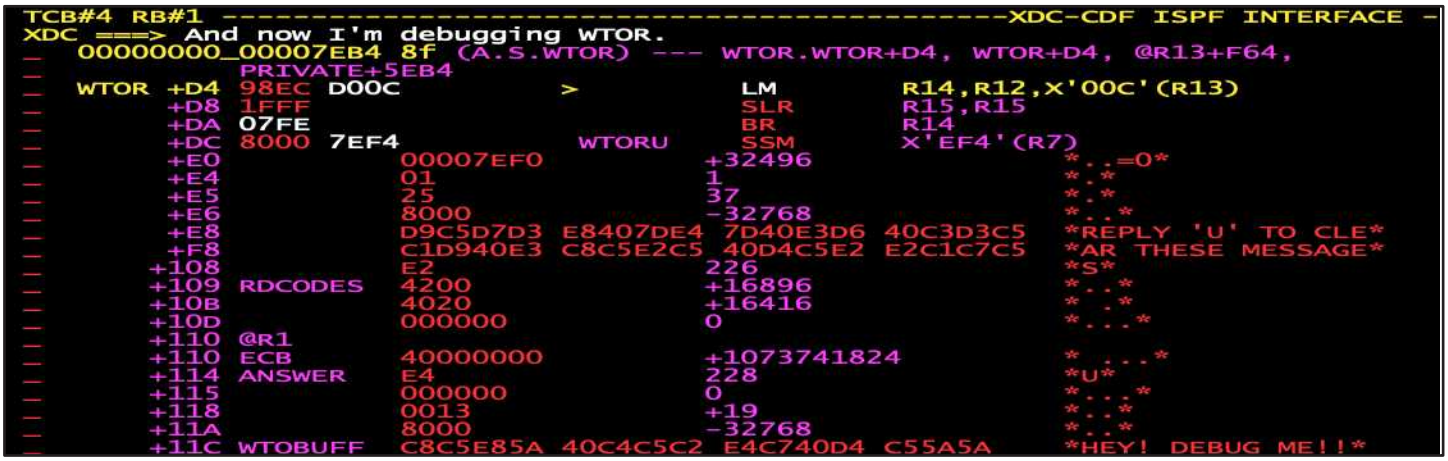


Figure 7-11.

What I just did was HUGE. I "hijacked" a currently running program with Foreign Address Space Mode using HOOK, and went right back in using Cross Domain Facility to debug it. C'mon, admit it: That's GOOD STUFF.

Here are the steps:

1. Start z/XDC APF-authorized.
2. Issue SET ASID <job-name> to navigate to the job you're interested in.
3. Issue LIST TASKS;;LIST RBS <TCB#n> (Two semi-colons yield a blank line. It's prettier).
4. Set a HOOK wherever you like.
5. Issue SET ASID HOME to return to your home address space.
6. Issue END NOASK to get out of the session.
7. Find some way to force the target code to execute the hook.
8. Go back into z/XDC's Startup Panel and issue a 3.
9. Put an S next to your high-jacked program and press Enter.

This scenario is for situations in which you DON'T expect to debug a program, but decide to on an ad-hoc basis. **It's very powerful, and very cool.**

Now that I'm in Cross Domain Facility (not FASM), ALL my z/XDC commands work. It's just as if I were debugging WTOR in my own TSO address space.

The Cross Domain Facility

z/XDC's Cross Domain Facility allows you complete functionality when working within another address space. CDF allows you to submit a batch job under the control of z/XDC and debug it, whether it's multitasking, APF-authorized, running as a started task - whatever. In fact, we expect that you'll use Cross Domain Facility almost exclusively for your debugging work, except when working with TSO command processors or things that you can run in your TSO region's Private Area.

In the previous topic, we created a Cross Domain Facility session on the fly by using the HOOK command in conjunction with Foreign Address Space Mode. That was an example of arbitrarily debugging a running batch job on an unplanned basis. This Section will discuss using z/XDC when you DO plan to debug a batch job.

Conceptually, CDF allows you to communicate with your batch job through a VTAM interface. All of z/XDC's commands and actions take place in the batch address space, and the results are piped to you at your TSO terminal or VTAM session.

To retain control of your debugging session with z/XDC, you must ensure that z/XDC is and remains the newest ESTAE for the desired bit of code. Background reading on this subject is found in HELP DEBUGGING ESTAES.

The HOOK command is a great way to ensure that you retain control in a batch job. HOOK is also available to you as a macro ("#XDCHOOK") that you can assemble into your source code. Using the macro form of HOOK is a great way to work in large systems that have their own robust ABEND handling. Example: If you were working on a single transaction in a transaction processing environment, you normally wouldn't want to debug the entirety of that system (CICS, IMS, etc.). Instead you want control when your program loads, and assembling a #XDCHOOK into your code would be a great way to assure that. You'll need your own system to do this without affecting others, but it would work.

Changing your JCL for Cross Domain Facility

First, you need to get initial control of your code under z/XDC. This can be done with a single JCL change to your EXEC statement. The change to make is this:

OLD: // EXEC PGM=<yourprogram>, PARM='itsparms'

NEW: // EXEC PGM=XDCCALL, PARM='yourprogram itsparms'

That looks entirely familiar, doesn't it? Just as we started out running XDCCALL under TSO READY you're issuing, XDCCALL <yourprogam> in batch.

And, as you may expect, if you want to run your batch program APF-authorized, you'll say // EXEC PGM=XCDCCALL**A**, PARM=<'your program and its parms'>.

Optional DD Names for use with Cross Domain Facility

There are many other DD statements you may invoke with Cross Domain Facility. Some are for very specific purposes, and you may research them at HELP DDNAMES. But the only JCL statement that you MUST change is the EXEC card.

Here are some of the DD names I like to include.

You can include a DD statement pointing to your ISPF profile library so z/XDC can find and use your favorite session profile. The syntax would be:

// DD XDCPROF <dataset-name> <== this points to your ISPF Profile library. Now you can take your favorite, custom profiles with you into the batch environment.

You may also establish a DD statement named XDCCMDS. This dataset will contain any z/XDC scripts that you wish to access via the READ command. The syntax would be:

// DD XDCCMDS <== points to a dataset containing a script of z/XDC commands that will execute automatically before your program begins.

You might want z/XDC to run its AUTOMAP process. That syntax would be:

// DD XDCMAPLB <== points to a dataset containing the ADATA for one or more of the programs you wish to debug. When execution passes to z/XDC, it will search for your ADATA and AUTOMAP will bring in your source code!

Next, you submit your batch job and it will go into execution.

XDCCALL (or XDCCALLA for APF-authorized sessions) gets loaded into memory, attaches your program as a sub-task and specifies itself as ESTAI. It then issues a message to the console, parks itself at the entry point to your program and goes into a WAIT.

Now, if your friendly neighborhood Operator doesn't CANCEL your job, you should be able to access it. To do that, you issue Option 3 from z/XDC Startup Panel. You will be shown a list of the jobs awaiting debugging services, and you may select one. XDCCDF makes the connection between you and your batch program, and now you can display, trace, set breakpoints, and zap to your heart's content.

Here's the JCL for my XDCSYMED job:

```
   Menu  Utilities  Compilers  Help
 BROWSE     BOB.CNTL(XDCSYMED) - 01.04              Line 0000000000 Col 001 080
 Command ===>                                          Scroll ===> CSR
****************************** Top of Data ******************************
//XDCSYMED JOB (CSW,UPS),'phone: 540-456-8210',CLASS=A,MSGCLASS=D,      XDCSYMED
//         MSGLEVEL=(1,1),NOTIFY=R9999,TIME=1439
//*
//DEBUG    EXEC PGM=XDCCALL,PARM=XDCSYMED
//XDCPROF  DD   DISP=SHR,DSN=BOB.ISPF.ISPPROF
//SYSPRINT DD   SYSOUT=*
//XDCTRACE DD   SYSOUT=*
***************************** Bottom of Data ****************************
```

Figure 7-12.

I submit this job, and go to z/XDC's Startup Panel (which you've seen). I enter Option 3 and get this display. I put an S next to the XDCSYMED job.

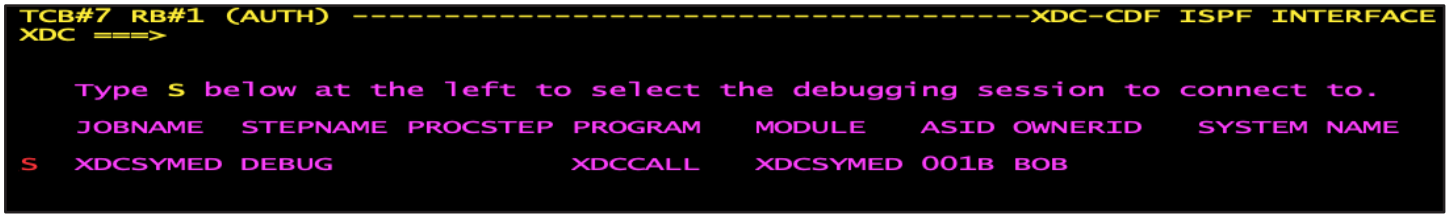


Figure 7-13.

And here we are. Now, each and every z/XDC command is available to me. I can issue MAP/SET QUALIFIER, DMAP/USING, LIST, FORMAT, TRAP, TRACE - everything.

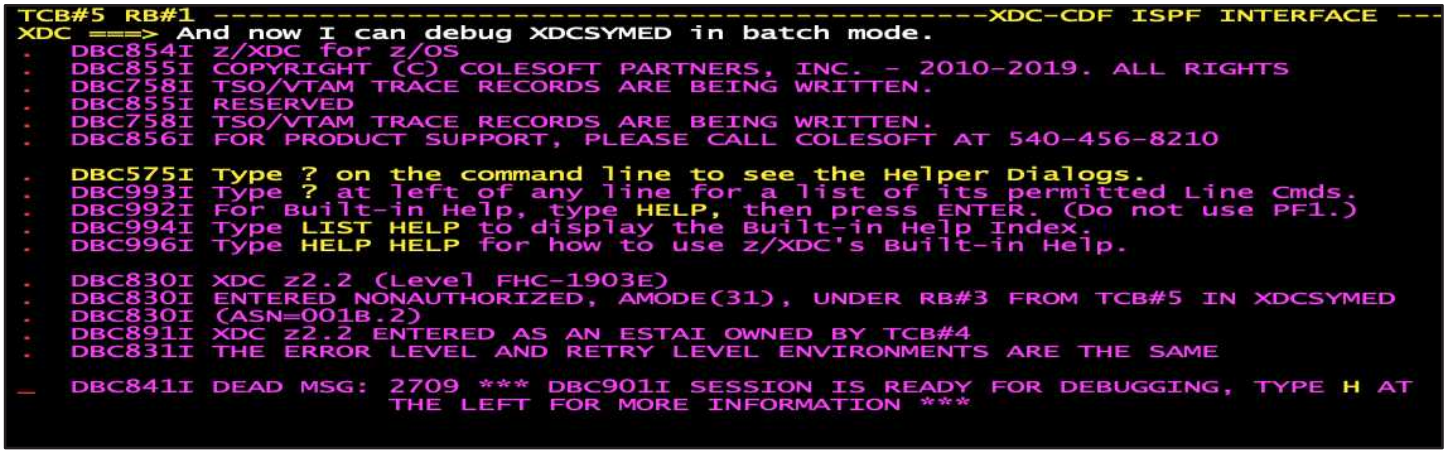


Figure 7-14.

Leaving the Cross Domain Facility Environment

Sooner or later, you'll accomplish what you set out to do, and you'll want to leave CDF and work on something else. To do this you issue the DISCONNECT command. Your batch job suspends and whenever you want to come back to it, you go to the z/XDC Startup Panel and issue Option 3 again.

Another thing to do is to DISCONNECT and then cancel the job from an Operator Console. Neat and clean. A handy abbreviation for DISCONNECT is "DISC".

However, in some situations you may want to let your program run. **DO NOT JUST ISSUE GO!!!** That might lock your terminal and you may have to cancel your TSO session.

Instead, consider these actions:

1. DELETE MAPS
2. DELETE MAPLIBS
3. DELETE CACHE
4. OFF ALL
5. DELETE HOOKS

You're being a neat and tidy engineer by cleaning up the address space. Then, issue:

DISC;GO

This gives YOU a DISCONNECT and the batch job a GO. Then the two of you part company gracefully.

However, your program is still running under z/XDC's ESTAE. At the next ABEND, or if a breakpoint or a HOOK (that you forgot to remove) is executed, the batch program will again go into a wait and give you the opportunity to debug it.

I**t is often best to just cancel the job and resubmit it** NOT under z/XDC's control.

That said, IF z/XDC got control in a batch job by means of a hook, then it is possible to enter DISCONNECT;GO REMOVEXDC.

The REMOVEXDC parameter can remove z/XDC's STAE from the STAE Control Block Chain, but there are a lot of "Ifs" and "it depends". For more information on this, enter Help COmmands GO, and examine the REMOVEXDC parameter.

Chapter 8: Help

Every word of our documentation is contained in the Built-in Help Subsystem. In fact, if you read the printed manuals, you will soon conclude that the printed manuals are just a “dump” of the Help System. You’re right. Except for minor formatting the manuals ARE a dump of the Help System. When read in its .pdf format it’s distinctly NON-linear, which can put people off. However, using Adobe’s powerful FIND function is a great way to zero in on a Help topic you may be interested in.

It is a good idea to become conversant with the Help System’s structure. Once you adapt to the structure as we present it, you’ll be able to find anything you like with relative ease.

Remember, if you get stuck, I am more than happy to receive a phone call from you at (800) XDC-5150. That’s my number, and if I can’t answer your question, I’ll GET you an answer!

You can obtain Adobe Acrobat or .html formatted versions of the z/XDC documentation at our web site, which can be found at:

http://www.colesoft.com/product-support/zxdc-support/zxdc-documentation.html.

z/XDC’s Help subsystem is a large collection of ISPF Panels - a structure that goes down and out many levels. There are perhaps 1200 or more pages in the system.

It is all hung off one main panel - Help. So, to access it you merely type HELP in a z/XDC display (not in the Profile facility or a Helper dialogue - those are separate within z/XDC).

The first thing displayed in any z/XDC Help panel is the unique name of that panel, with strangely capitalized words. They are capitalized so that you know the minimum abbreviation of the panel’s name. So, while one could enter “Help commands” and see the list of z/XDC’s commands, the minimum abbreviation is “H CO”, and the panel’s name appears as “Help COmmands”. It’s a bit strange, but you’ll get used to it.

Wherever you see an underscore in column one of a Help panel, that is an opportunity to put an “S” next to it and press enter. This will take you to a sub-topic. This is as close as we can get to a real hyperlink in a 3270 display.

Once you become familiar with the naming conventions in z/XDC’s Help subsystem you’ll find your way to what you need more quickly. You may also learn new facts about z/OS itself. That’s because z/XDC runs so low in the OS that Dave often HAS to explain what the system is doing in this or that circumstance.

If z/XDC ever generates a message, you could enter HELP MESSAGES DBCnnn, but it’s much easier to tab over the period character that precedes any z/XDC message, put an H there and press enter. That’s a hyperlink into our Messages Manual. Handy.

Accessing z/XDC's Built-in Help

To access Built-in Help, merely type Help at the command line.

Figure 8-1.

New users beware: The Help PFKeys, espeically BACK AND FORWARD don't work like the back button on a browser.

There are two categories of information. The non-highlighted topics are informational, tutorials where you can learn about concepts. The highlighted topics are references, such as COMMANDS or SCRIPTS.

Navigating in Help

During your Help session, your PFKeys change to settings specific to the Help environment. You can PageDown or PageUp, but you can also do a Previous or Next function. This is intended to make our Built-in Help read more like a book, but beware: What Dave Cole thinks is the “next” topic may not be what you expect.

I use the Help PFkeys with caution, and most of the time I SET HKEYS OFF to get rid of the clutter. The PFKeys’ functions don’t disappear and are still active. They’re just not in my face. In fact, I’m going to SET HKEYS OFF for the duration of this section so you can see more of the Built-in Help topics.

You can navigate through Help by putting an S next to any sub-topic with an underscore in column one and pressing Enter. That’s a hyperlink to the next page of information, and it’s a pretty fast way to get around inside Help.

You can also invoke the unique name of any panel and display it directly. After a while you’ll intuit the naming structure in Help all by yourself.

It’s possible to get lost in Help, if you don’t learn the syntax. One common mistake is to enter HELP MAP, when you REALLY meant that you wanted syntax for the MAP command. If you want to read the (voluminous tutorial) on mapping, enter HELP MAP. If you want the syntax for the MAP command, enter “Help COmmands Map”.

The List Help Command

In order to provide a site map for the Help system, Dave Cole wrote the LIST HELP command. Here's the base panel:

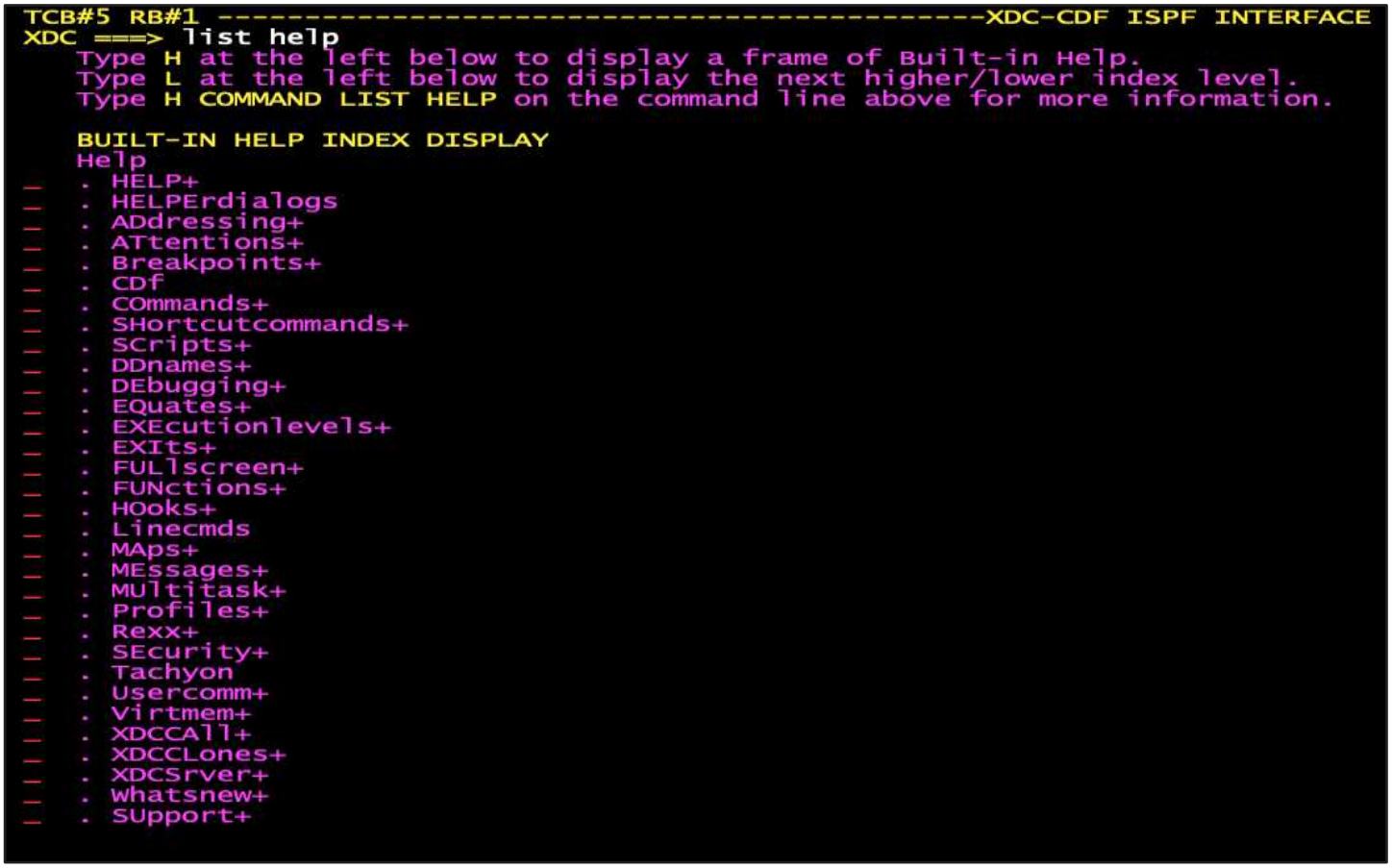


Figure 8-2.

You can enter LIST HELP <subtopic> and z/XDC will present all the subtopics available under that major topic. You can put an L shortcut command next to any area of interest to expand the topic, and see the sub-topics (if any) under it.

When you've found what you want you can put an H next to any topic and that sub-topic will be displayed. LIST HELP is a pretty cool interface.

Useful Help Topics

There are several useful Help topics that I've discovered over the years. One of the most useful is HELP DEBUGGING

Help DEbugging

Figure 8-3.

Help DEbugging is your portal to learning how to deploy z/XDC in a number of environments. Come here if you want to debug exit, or learn how to run z/XDC as a Functional Recovery Routine ("FRR") or how to work with LE programs or PC routines, etc. There's LOADS of useful information that can be accessed from this panel.

You may also try out LIST HELP DEBUGGING.

Help SUpport

If you get “stuck” or need to reach us for any reason , this is the page for you:

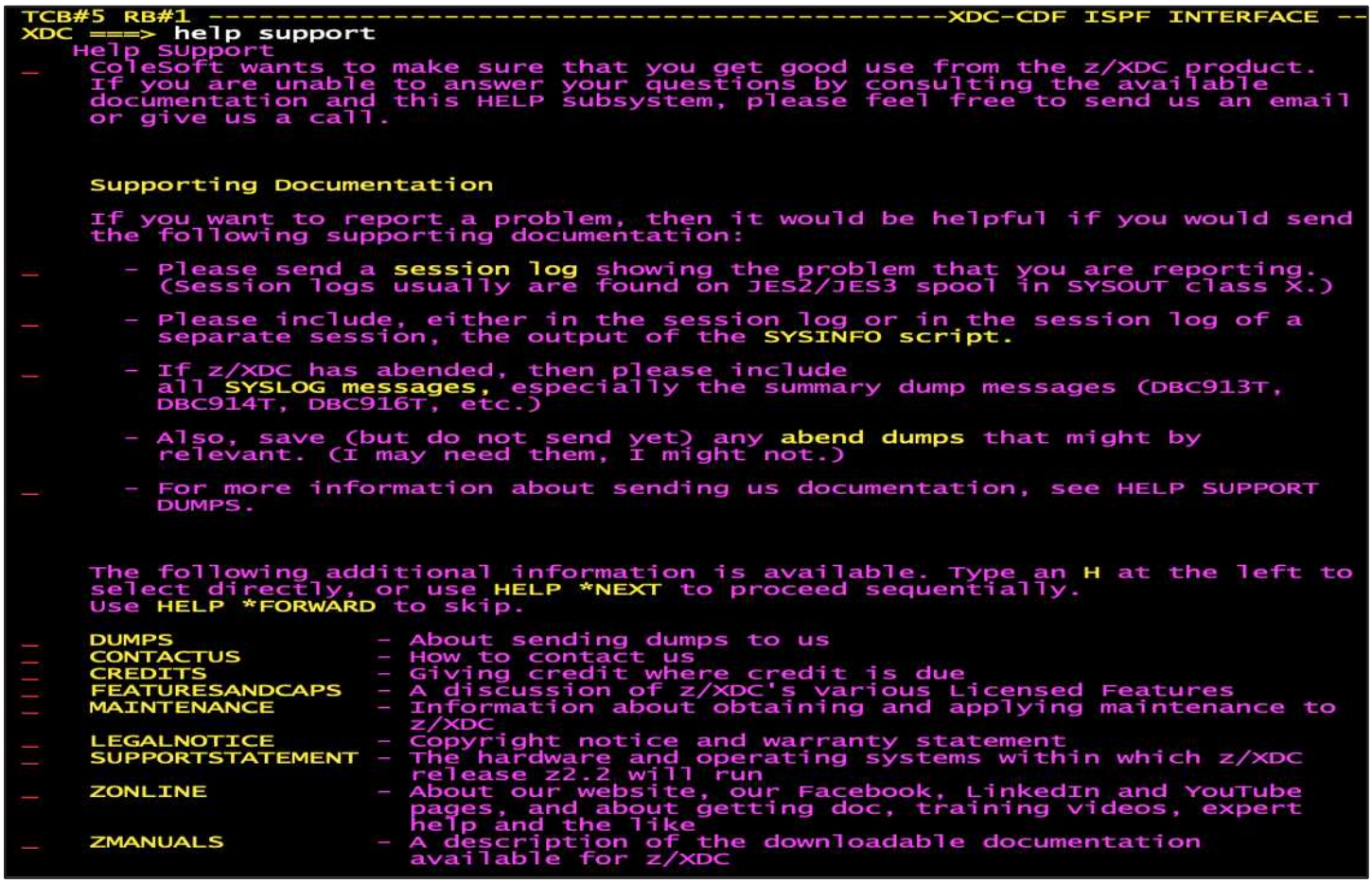


Figure 8-4.

- The DUMPS section tells you how to format and transmit a dump to us.
- The CONTACTUS link gives you access to our phone numbers and e-mail addresses.
- The CREDITS page is Dave’s way of thanking his colleagues. Awww... Thanks, Dave!
- The FEATURES link discusses the components of your z/XDC system.
- The MAINTENANCE link discusses our approach and packaging of maintenance decks.

Help COmmands

This is an alphabetic list of all of z/XDC's many commands. I'll issue a challenge at this point. Please enter Help COmmands each morning and research a single command. Do this each day for a week or two and you'll soon know more about z/XDC than I do!

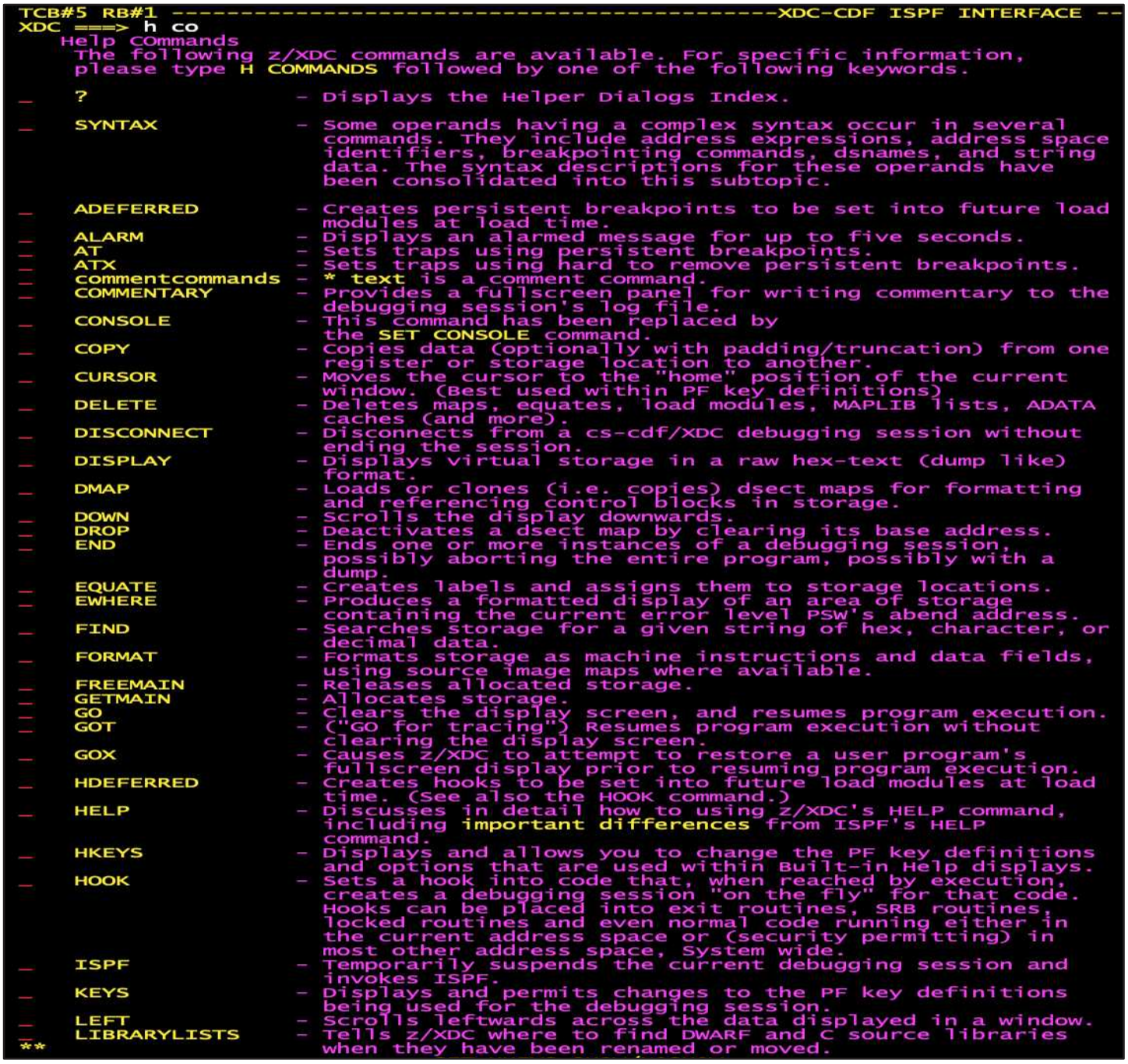


Figure 8-5.

Any time you see two asterisks at the bottom of the page, it means that more information may be seen by pressing PF8 to page down. In the case of Help COmmands, that's several pages.

Another learning exercise would be to sit with z/XDC each day for a few minutes, investigating all the List commands. Try Help COmmands LISt. Whoa... That's a LOT of List commands!

You can issue Help COmmands SEt and gaze upon the multitude of SET commands.

Help Whatsnew

This is a quick way to find out what new features have been added to any version of z/XDC.

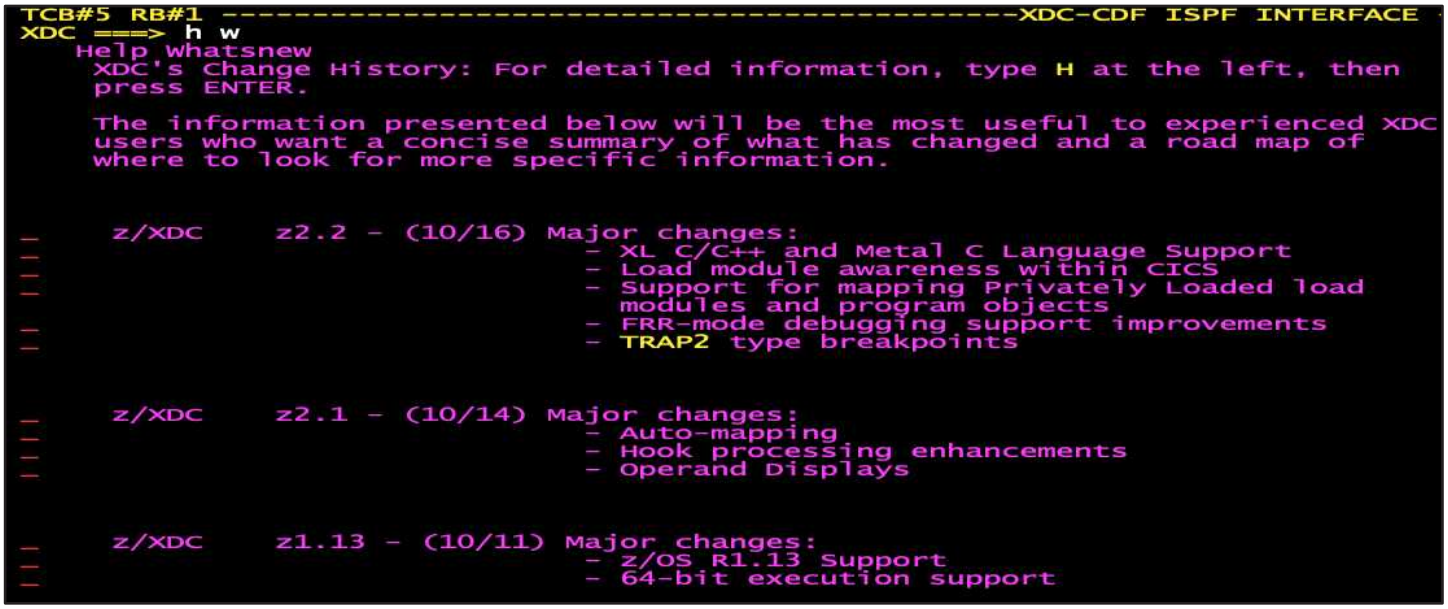


Figure 8-6.

This is like an online Release Guide that spans many iterations of the z/XDC product. Use the S”shortcut key to drill into any sub-topic. You can always enter H W Z22 and see all the changes for z/XDC Release z2.2.

Unfortunately, busy task-oriented people have no more time to read HELP WHATSNEW than they have time to read our Release Guides - but we DO try.

Accessing z/XDC's Messages

If you should create an error in z/XDC, the product will display a message, like DBCnnn. Now, you COULD enter HELP MESSAGES DBCnnn from the command line and directly access the message information in Help, but it's *so much easier to put an H next to the error message and press Enter!*

Figure 8-7.

z/XDC understands the SET command name, but since Dave hasn't implemented a FRANGLE (yet), z/XDC puts a helpful asterisk under the first thing it did NOT understand. z/XDC didn't even stoop to consider my OFF-PLANET parameter. Pity.

I put an H next to the message and press Enter. Watch the birdie:

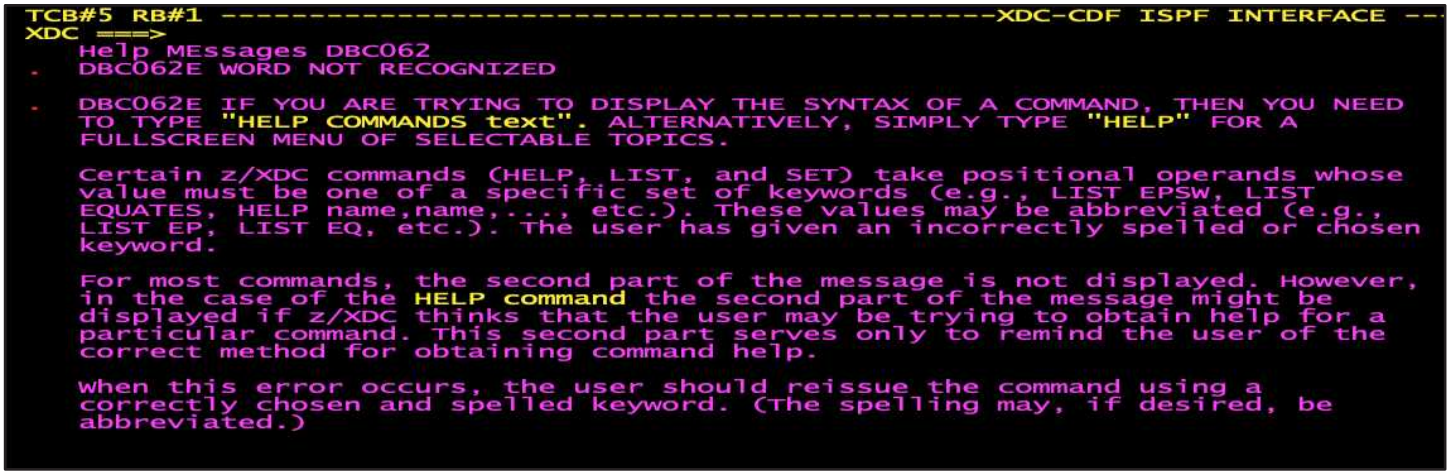


Figure 8-8.

If I have been working in z/XDC and need to remind myself of a command's syntax, I can issue HELP COMMANDS <command-name> and pop right into Help. When I have researched the command, I can enter it on the command line. z/XDC will take me out of help back to my session and will execute the command. Nice.

Reminder: You can't enter a z/XDC command from within PROFILE, or the HELPER Dialogues. Profile and Helper dialogues require you to exit their environments before you can execute commands outside their context. I've made this mistake often enough myself.

There is a quick tour of Help. People tell me it's ugly and difficult to navigate, but that there's useful information if they learn the language. That said, the most effective form of help is to CALL ME and ask a question. If I'm at my desk, I'll be back to you in near real-time. We want you to debug happily, so feel free to call me at (800) XDC-5150.