colesoft™

ZXDC®

c/XDC PRIMER

for Release z2.2 for z/OS

Bob Shimizu

ZXDC®

Contents

Revision Date: December 4th, 2024

We have changed the names of our sample programs and this manual reflect the change:

- CSAMPLE has been renamed to XDCCSAMP.
- MSAMPLE has been renamed to XDCMSAMP.
- Screen colors have changedin z/XDC since the last edition, so I replaced them all.

Preface

PROPRIETARY LEGEND

z/XDC® and its documentation (collectively, "Product"), including copies thereof, are the property of Mainchord, LLC DBA Cole soft ("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 The Product may not be published, reproduced, distributed or made available to third parties for any purpose without the expressed written permission of Owner or Licensor.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.

Chapter 1 - How Does c/XDC Work?

The Architecture of the z/XDC and c/XDC Debuggers.

Broadly speaking, our debuggers are Recovery Routines. Recovery Routines get control whenever a program suffers an abnormal ending (an “abend”). z/XDC and c/XDC trap all classes of abends.

In this booklet, I’ll refer both to the c/XDC Feature and the z/XDC base product. c/XDC sits on top of z/XDC to provide C source statement support, the STEP command, Variable viewing, etc. z/XDC provides a LOT of the insfrastructure that c/XDC relies upon, so it’s appropriate to mention z/XDC at times in this Primer. In fact, when you’re debugging C code with c/XDC all the capabilities of the z/XDC base product remain very much at hand.

As your understanding grows with c/XDC you may find it useful to learn some of the basics of z/XDC. That’s why I wrote the z/XDC Primer. You can refer to that any time and begin to explore our much larger base product.

Our debuggers work by creating S0C1 abends in your code that our debuggers capture. They then present the results to you in z/XDC’s displays which show you the breakpointed code as if it will be the next statement or instruction to execute. This is the same technique used by generations of programmers - introduce a S0C1 abend, and create a dump. However, instead of showing you one huge monolithic dump, z/XDC and c/XDC create “mini-dumps”, one for each breakpoint or tracing operation. Each stopping place allows you to look around storage, view your source code, investigate variables, step through your source code, etc. This technique allows you to learn a great deal more about the actions of your program than creating a single dump and *hoping* that you’ll find the problem.

Recently, Dave Cole and his team have added the ability to use TRAP2 instructions for breakpoints and traces. The advantage is that the breakpoints are handled at the hardware level, which buys some efficiency. However, a discussion of TRAP2 support is beyond the scope of this Primer. You can start a z/XDC session and enter “HELP WHATSNEW Z22 TRAP2” to learn more if you wish.

From 10,000 feet, z/XDC includes:

- An object code viewer;
- A disassembler;
- The ability to view source statements in Assembler or C;
- The ability to follow code execution or stop it, or stop it under certain conditions that you specify;
- The ability to create debugging sessions in a TSO address space, in a batch address space or across many address spaces;
- The ability to execute in Problem or in Supervisor state;
- The ability to automate z/XDC through a “scripting” function;
- The ability to LIST many things about the z/OS environment (including the PSW, registers, the Task Structure and more that we won’t go into);
- The ability to configure the product’s screens/PFkey settings, colors, etc. in the way most meaningful to any single user and the ability to have many such configurations available (This is called the Profile Facility);

- Complete documentation available INSIDE the product under our online Help subsystem.

The z/XDC family of debugging products is designed for software engineers and system programmers tasked with building and maintaining very simple or very complex programs under z/OS. Such programs may be multitasking, run as started task or batch jobs, may be APF-authorized or run in multiple address spaces at once.

z/XDC's capabilities may be described in a single sentence: *"Using z/XDC, a knowledgeable programmer may more easily find and fix bugs in arbitrary object code within any csect within any module running under any Request Block under any Task Control Block (or SRB) within any address space on the system, including much of the internals of the z/OS environment itself."*

c/XDC brings the same level of functionality to IBM XL C/C++ and Metal C.

c/XDC allows you to follow execution from C language to Assembler and back again - seamlessly. c/XDC is tightly integrated with the z/XDC parent product, so all the traditional capabilities in z/XDC will be available to you under c/XDC.

Guys, this is important:

- z/XDC and c/XDC work by becoming (and remaining) the most recently created Recovery Routine for your program.
- All breakpoints are really X'00's or TRAP2 instructions that c/XDC inserts into the execution stream, causing S0C1 abends.
- c/XDC captures the abend and presents the program to you for further work.
- If c/XDC does not remain the most recently created Recovery Routine, then it will lose control of the debugging session and some other program will capture the S0C1 abend.

Recovery routines are created and destroyed as large programs execute, and any single program may invoke its own Recovery Routine. For example, if your program has multiple enclaves it is almost certain that each enclave will have a new Recovery Routine.

Language Environment has its own huge Recovery Routine, and part of what you'll do to use c/XDC under LE is to turn off the ESPIE that LE sets. That's because ESPIEs get called ahead of ESTAEs, and that prevents z/XDC from keeping control of the debugging session. This applies to debugging both in your TSO region and in the batch. This means you'll specify the TRAP parameter to get LE out of c/XDC's way, and I'll discuss that further down in the booklet.

Introducing HOOK

If c/XDC is happily debugging for you and another Recovery Routine is invoked, then c/XDC will lose control of the debugging session. When this happens, the next STEP or breakpoint will be captured by that newest Recovery Routine, and it will handle the S0C1 that c/XDC introduced. In other words, z/XDC can lose control of the debugging session if a new Recovery Environment is invoked. This is a Bad Thing.

To avoid this situation, Dave Cole wrote HOOK. If you're reasonably sure that a newer Recovery

Routine will be invoked during the debugging session, then you'll need to set a HOOK instead of a breakpoint. HOOK automatically disables ESPIEs that get in c/XDC's way.

HOOK creates a NEW debugging session right-here/right-now regardless of any other Recovery Routine. There are two kinds of HOOKs, Static Hooks and Dynamic Hooks.

1. A ***Dynamic*** HOOK can be set with:
   - "HOOK <address-expression>";
   - The use of the "K" shortcut command;
   - Entering "HD <program object name>" (which sets up a deferred hook).
2. A ***Static*** HOOK is one that you compile into your program or transaction.
   - For C, you may call the cxdchook() function. If you compile the cxdchook function into your code you are guaranteed to get control in that C program at the HOOK's location – until another Recovery Routine is invoked.
   - For Assembler language we offer the #XDCHOOK macro that you can assemble into your source code.

Static Hooks are excellent for working in very large systems such as CICS, DB2, IMS, etc. In that situation you really don't want to debug the entire system, but you DO want to get control when your program/transaction loads.

You may think of HOOK as the *ultimate breakpoint*. If the code containing the HOOK is executed, then you will get control of the code and will be able to work on it.

If your environment makes heavy use of its own Recovery Routines, you may have to set more than one HOOK to keep control during the session. We'll talk more about HOOK later.

The definitive reference on ALL of z/XDC's C capabilities may be found in Help Debugging C. Just start any z/XDC session and enter "Help DEbugging C" from the command line. It's ALL in there!

c/XDC and Source Code Discovery

c/XDC uses DWARF data to synchronize object code with your C source files. DWARF, now there's a strange acronym! It actually means, Debugging With Attributed Record Formats. Now you know.

Automapping

If c/XDC detects that it is in a C program object, it will automatically show you your source code. That's because so long as you haven't moved or renamed your source files - AND - you're working in C or C++ (not Metal C), then c/XDC can retrieve your source files by analyzing the DWARF data. This is a Very Good Thing.

Similarly, if z/XDC detects that it is in an Assembler program it will attempt to automatically map your Assembler source code.

With these capabilities, you can move between C and Assembler, and our debuggers will automati-

cally map your source code in either language.

In IBM XL C/C++ and Metal C, you'll have to compile your programs in such a way as to create DWARF data for them.

In IBM Assembler language, you'll assemble using the High Level Assembler (HLASM) to create ADATA data for them. Later in z/XDC you'll need to issue a SET MAPLIBS command to tell z/XDC where the source code is.

Note: If you're compiling under IBM's Metal C, and if you are running at z/XDC Maintenance level MDL-2014C (or greater) and if you are running z/OS 2.4 (or 2.3 with APAR PI99202 applied) then c/XDC will be able to find your DWARF data and will automatically map your C source.

Otherwise, Metal C users will have to learn and use the LIBRARYLISTS command.

Introducing the LIBRARYLISTS Command

The LIBRARYLISTS command tells c/XDC how to match up source and DWARF files so that it can map them and show you your source code. You can establish a default LIBRARYLIST or create lists of lists, each of which in the end, points to either DWARF or C source locations. Because C itself is pretty complex, so is LIBRARYLISTS. So, if you're working in Metal C you will find LIBRARYLISTS useful.

There are many forms of the LIBRARYLISTS command, but you'll probably end up issuing a LIBRARYLISTS REDIRECT command in order to point c/XDC to the DWARF data and your source files.

You can find the authoritative (and exhaustive) treatment on LIBRARYLISTS at Help DEBUGGING C LIBRARYLISTS. You can also issue Help COMMANDS LIBRARYLISTS to see command-level information. I'll get into deeper detail on LIBRARYLISTS later on in this booklet.

Automating LIBRARYLISTS - Introducing z/XDC Scripts

The bad news is that LIBRARYLISTS is a pretty complex facility. The good news is that LIBRARYLISTS can be automated by putting the LIBRARYLISTS commands into a z/XDC "Script". So, now let's spend a bit of time on scripts.

A z/XDC "Script" is a sequential file full of z/XDC commands that are separated by semi-colons or New Line (the Return or Enter key). You can include any z/XDC command in a script (in this case LIBRARYLISTS commands).

Here are the rules for scripts:

1. Scripts may contain ONLY simple declarative commands. There is no conditional logic in a script.

2. No z/XDC script survives an embedded GO command. if you want things to happen after a GO, then create another script to run after the GO command executes.

You execute a script with z/XDC's READ command, which opens the script for input and processes the commands in it. Information on how to use the READ command may be found at HELP COMMANDS READ. The script will run to completion unless z/XDC detects an error. It will then suspend operation and put up a message. You have the opportunity to fix whatever is wrong and then issue a READ RESUME. Full documentation is available at HELP SCRIPTS. You can choose the sub-categories RYOSCRIPTS (for write your own), or OURSCRIPTS to review a list of scripts that we've built for you.

Scripts are a great way to get repetitive "grunt work" done easily and quickly. Heck, a READ command could even be equated to a PFKey setting. Then you could wiggle one finger and many hundreds of commands might be issued! For more information on our READ command, please see Help COmmands READ.

Chapter 2 - Preparing Your C Program for Debugging

Note: c/XDC does not support debugging optimized code.

There's quite a journey from coding your original C source statements to having a running program. As you know, you first compile the program and then call the Binder to create an executable "Program Object". These days a Program Object can be a complex mix of programs written in different High-Level languages with both prologue and epilogue code. It's our job to make these complexities transparent to you. Normally you won't see your prologue or epilogue code, because normally you won't be concerned with these entities.

Here's what you need in order to compile your program for c/XDC:

1. Add the NOOPT parameter, which turns off optimization.
2. Include the DEBUG parameter of some sort (see Figure 2-2).
3. GONUMBER needs to be specified, but the DEBUG parameter turns that on by default.
4. Add a SYSCDBG or (in the case of Metal C) a SYSDWARF DD name that tells the compiler where to put your DWARF information.
5. Finally, if there is a TEST parameter in your compile JCL, please remove it.

The Compile-time DEBUG Option

You specify the DEBUG compile option DIFFERENTLY in z/OS V2R1 and earlier than in z/OS V2R2 and later.

In z/OS V1R13 and V2R1, all you had to specify in your compile-time JCL was “DEBUG.” Here’s an example of the “older” style:

```
. 000001 //ASMCSAMP JOB (COLE,UPS,,999,,),'Phone: 540-456-8536',NOTIFY=R9999,
. 000002 //             MSGLEVEL=(1,1),PRTY=12,TIME=60
. 000003 //*
. 000004 //COMPILE  EXEC PGM=CCNDRVR,REGION=0M,
. 000005 //             PARM=('/DEBUG OPTF(DD:CCCOPTS)')
. 000006 //CCCOPTS   DD *
. 000007      GOFF
. 000008      ILP32
. 000009      LANGLVL(EXTENDED,LONGLONG)
. 000010      LIST
. 000011      LSEARCH(//'DBCOLE.XDCZ22.XDCSAMP')
. 000012      MARGINS(1,72)
. 000013      RENT
. 000014      SEARCH(/usr/lpp/cbclib/include)
. 000015      SEQUENCE(73,80)
. 000016      SOURCE
. 000017      DEBUG
. 000018      SSCOMM
. 000019      XPLINK
. 000020      XREF
. 000021 //STEPLIB   DD DISP=SHR,DSN=CEE.SCEERUN
. 000022 //            DD DISP=SHR,DSN=CBC.SCCNCMP
. 000023 //SYSCDBG   DD DISP=SHR,DSN=DBCOLE.XDCZ22.XDCDWARF(DBCCSAMP)
. 000024 //SYSUT1    DD UNIT=SYSDA,SPACE=(CYL,(20,20)),
. 000025 //             DCB=BLKSIZE=32760
. 000026 //SYSTERM   DD SYSOUT=*
. 000027 //SYSCPRT   DD DDNAME=SYSPRINT
. 000028 //SYSPRINT  DD SYSOUT=*
. 000029 //SYSLIN    DD DISP=OLD,DSN=DBCOLE.XDCZ22.XDCOBJ(CSAMPLE)
. 000030 //SYSIN     DD DISP=SHR,DSN=DBCOLE.XDCZ22.XDCSAMP(SRCCSAMP)
```

Figure 2-1 - The DEBUG option under z/OS V2R1 and older

However, in z/OS V2R2 and later, you need to specify “DEBUG(LEVEL(9)).” This is because IBM has gotten around to improving DWARF support. Level 9 gives c/XDC everything it needs to do its job for you.

For z/OS V2R2 and later:

- Specify “DEBUG(LEVEL(9))”
- or you may use the complete statement: “DEBUG(FORMAT(DWARF),LEVEL(9))”.

Here’s an example of the newer style:

```
    Menu  Utilities  Compilers  Help
  ------------------------------------------------------------------------------
  BROWSE     CSW.XDCZ22.XDCJCL(CMPCSAMP)              Line 0000000000 Col 001 080
  Command ===> COmpile JCL for z/OS V2R2 and later...   Scroll ===> CSR
 ********************************* Top of Data ********************************
 //CMPCSAMP JOB (COLE,UPS,,999,,),'Caps off',NOTIFY=R9999,     FHC-2411A 00000010
 //             MSGLEVEL=(1,1),PRTY=12,TIME=60                           00000020
 //*                                                                     00000030
 //COMPILE  EXEC PGM=CCNDRVR,REGION=0M,                                  00000040
 //             PARM=('/GOFF OPTF(DD:CCCOPTS)')                          00000050
 //CCCOPTS   DD *                                                        00000060
      DFP                                                               00000070
      GOFF                                                              00000080
      ILP32                                                             00000090
      LANGLVL(EXTENDED,LONGLONG)                                        00000100
      LIST                                                              00000110
      LSEARCH(//'CSW.XDCZ22.XDCSAMP')                                   00000120
      MARGINS(1,72)                                                     00000130
      RENT                                                              00000140
      SEARCH(/usr/lpp/cbclib/include)                                   00000150
      SEQUENCE(73,80)                                                   00000160
      SOURCE                                                            00000170
      SSCOMM                                                            00000180
      XPLINK                                                            00000190
      XREF                                                              00000200
      DEBUG(FORMAT(DWARF),LEVEL(9))                                     00000210
 //* DEBUG(FORMAT(DWARF),LEVEL(0)) or just DEBUG for zOS 2.1 and older  00000220
 //STEPLIB   DD DISP=SHR,DSN=CEE.SCEERUN                                 00000230
 //             DISP=SHR,DSN=CBC.SCCNCMP                                 00000240
 //SYSCDBG   DD DISP=SHR,DSN=CSW.XDCZ22.XDCDWARF(XDCCSAMP)               00000250
 //SYSUT1    DD UNIT=SYSDA,SPACE=(CYL,(20,20)),                          00000260
 //             DCB=BLKSIZE=32760                                        00000270
 //SYSTERM   DD SYSOUT=*                                                00000280
 //SYSCPRT   DD DDNAME=SYSPRINT                                         00000290
 //SYSPRINT  DD SYSOUT=*                                                00000300
 //SYSLIN    DD DISP=OLD,DSN=CSW.XDCZ22.XDCOBJ(XDCCSAMP)                 00000310
 //SYSIN     DD DISP=SHR,DSN=CSW.XDCZ22.XDCSAMP(SRCCSAMP)                00000320
 ******************************** Bottom of Data ******************************
```

Figure 2 -2 - The DEBUG option under z/OS V2R2 and later

Above, we’ve commented out the proper DEBUG statement for z/OS 2.1 and earlier. Now you have both “flavors” of DEBUG to look at.

Notice the presence of the SYSCDBG DD statement in both cases above. It can be a Fixed Block, LRECL 80 sequential file, a PDS, a PDSE or an HFS/zFS directory or file. SYSCBDG receives the DWARF data c/XDC needs to do its job.

Note: There are two other compiler options that are unsupported. One is TEST. The other is DEBUG(FORMAT)ISD)). Please remove these compiler options from your compile step JCL.

In the compile step: If you wish to embed a static Hook into your program, invoke the LSEARCH compiler option that points to the library containing the cxdchook() header file. You can see our example above in Figure 2-2.

In the Binder step: If you plan to embed a static Hook into your program using the cxdchook() function then you also need to concatenate a library to the //SYSLIB DD name for the file containing the Hook module. That library comes from Cole Software named “DBCOLE.XDCZ22.XDCLINK”, so you’ll need to ask your installing systems programmer what name was assigned at your site. It’s a good bet that the low-level qualifier will be “.XDCLINK”.

c/XDC Compiler Support

c/XDC supports four languages:

1. IBM XL C/C++
2. Metal C
3. IBM SPC
4. Dignus™ Systems/C

Compiling Under IBM XL C/C++

JCL Snippet:

//CEE EXEC PGM=CCNDRVR, REGION=0M, PARM= '<...> DEBUG(FORMAT(DWARF), LEVEL(9)) <...>'

//SYSCDBG DD DISP=SHR, DSN= <a sequential fixed-block lrecl-80 file OR pds member>

//SYSIN DD DISP=SHR, DSN=<libraryname(member)>

Explanation:

- First, add the DEBUG parameter to your PARM field or Compiler options. Insure that FORMAT DWARF is implied or specified.
- Next add a SYSCDBG card for receiving the DWARF output area. As above, it can be a FB LRECL 80 sequential file, a PDS, a PDSE or a HFS/zFS directory or file.

You can find an example of the relevant JCL in the sample files we discuss under "Introducing XDCCSAMP, etc."

Note:

- c/XDC does not support “DEBUG(FORMAT)ISD))” or “TEST”.
- Anything other than OPT(0) is incompatible.
- NOGONUMBER is incompatible.

Compiling Under IBM Metal C

Metal C is a two-step process. First, you perform a compile step which creates Assembler statements. Then, you perform an Assembly step using CDAHLASM.

JCL Snippet for the Metal C compile step:

```
//CEE       EXEC PGM=CCNDRVR, REGION=0M, PARM='<...> METAL, DEBUG <...>'
//  <additional DD cards as needed>
//SYSLIN   DD DISP=OLD, DSN=DBCOLE.XDCZ22.XDCOBJ(XDCCSAMP)>
```

You do NOT need a //SYSCDBG card.

Explanation: Add DEBUG to the PARM field or Compiler Options. This will cause the Metal C compiler to build and pass a High-level Assembler (HLASM) source file that contains the DWARF data. CDAHLASM uses that along with ADATA generated by HLASM data to create the final DWARF. You do not need to specify a SYSCDBG DD card, because the C compiler is creating Assembler source files, not a compiled C program.

JCL snippet for the Assemble step:

```
//HLASM    EXEC PGM=CDAHLASM, REGION=0M, PARM='<..> ADATA <...>'
//SYSADATA    DD DUMMY  DCB=(RECFM=VB, LRECL=27994)
//SYSDWARF  DD DISP=SHR, DSN=<sequential  FB-80 file or PDS member>
//SYSIN          DD DISP=SHR, DSN=DBCOLE.XDCZ22.XDCOBJ(XDCMSAMP)
//SYSUT8        DD UNIT=SYSALLDA, SPACE- (TRK, (100,100))
// <more dd cards as needed>
```

Explanation: For Metal C’s assembly step you invoke CDAHLASM instead of ASMA90. This is a front end to the High-Level Assembler. It takes the ADATA output along with the pseudo-DWARF information created at compile time to create the DWARF file.

Summary:

1. Add ADATA to the PARM field or Assembly Options file. This causes CDAHLASM to produce both the ADATA and DWARF data.
2. Then, add a SYSADATA DD card to receive the ADATA. However, this can be a DUMMY file because c/XDC doesn’t use this DATA.
3. Then add a SYSDWARF DD card to receive the DWARF data. It can be a Fixed Block, LRECL 80

sequential file, a PDS, a PDSE or an HFS/zFS directory or file.

4. Finally, add a temporary SYSUT8 work file. CDA-HLASM won't build DWARF data without it.

You can find an example of the relevant JCL in the sample files we discuss under "Introducing XDCC-SAMP, etc."

Compiling under Dignus™ Systems/C and C++

c/XDC now supports the Dignus Systems/C and C++ compilers for z/OS, Windows and Linux. Like IBM's Metal C, the Dignus compilers generate Assembler statements that are assembled into object code.

Things diverge here with a PLINK step that performs AUTOCALL processing. This processing creates either an object deck for final processing by the IBM Binder or a z/OS compatible executable program.

To generate the DWARF that is required by c/XDC, the following changes are needed in the compile and PLINK steps. On all platforms, the "-gdwarf" option is specified in the compiler step to instruct the compiler to generate DWARF data.

Here is a snippet of JCL that Frank provided me for the z/OS-based compile step. The -gdwarf option is italicized for clarity.

```
//DCC      EXEC PGM=DCC,REGION=0M,PARM='-@PARMS'
//STEPLIB  DD  DISP=SHR,DSN=CSWPP.DIGNUS.LOADLIB
//PARMS    DD  *
-oASMSRC,-flisting=SYSPRINT,-gdwarf,
-fwarn_disable=2181,-frent,
-I//DSN:FRANK.DIGNUS.INCLUDE
//ASMSRC   DD  DISP=SHR,DSN=FRANK.DIGNUS.ASM(FINDVNTE)
//SYSPRINT DD  DISP=SHR,DSN=FRANK.DIGNUS.LISTINGS(FINDVNTE)
//SYSTERM  DD  SYSOUT=*
//STDOUT   DD  SYSOUT=*,LRECL=133,RECFM=FB
//STDERR   DD  SYSOUT=*,LRECL=133,RECFM=FB
//STDIN    DD  DISP=SHR,DSN=FRANK.DIGNUS.SOURCE(FINDVNTE)
```

There are two options for the PLINK step. The one below specifies the -dbg parameter pointing to a USS file. If you take this path, then c/XDC will be able to find your DWARF and will automatically show you your source code under c/XDC. This is a Very Good Thing. HOWEVER, due to actions in the Dignus compile and PLINK processes, source file names can be altered and in SOME cases you may have to resort to the LIBRARYLISTS REDIRECT command, usually for just one csect. Sorry, it wasn't us...

To take this option, specify both the "-px" and "-dbg" parameters. Again, the relevant text is italicized.

```
//PLINK    EXEC  PGM=PLINK,REGION=0M,PARM='-@PARMS'
//PARMS    DD  *
-px,-dbg=/u/frank/dignus/DBOX2.dbg,
//STEPLIB  DD  DISP=SHR,DSN=CSWPP.DIGNUS.LOADLIB
//STDERR   DD  SYSOUT=*,LRECL=133,RECFM=FB
//STDOUT   DD  SYSOUT=*,LRECL=133,RECFM=FB
//SYSLIB   DD  DISP=SHR,DSN=CSWPP.DIGNUS.LIBCXX
//         DD  DISP=SHR,DSN=CSWPP.DIGNUS.LIBSTDCX
//         DD  DISP=SHR,DSN=CSWPP.DIGNUS.LIBCR
//INDD     DD  DISP=SHR,DSN=FRANK.DIGNUS.OBJECT
//SODEF    DD  SYSOUT=*
//SODEFNM  DD  SYSOUT=*
//SOREF    DD  SYSOUT=*
//SYSIN    DD  *
  INCLUDE INDD(DBOX2)
//SYSMOD   DD  DISP=SHR,DSN=FRANK.DIGNUS.OBJECT.PLINK(DBOX2P)
```

An alternative way to perform the PLINK step appears below. This option includes a DDNAME for a SYSDBG file. However (and ironically) due to the actions in Systems/C, your DWARF data will NOT be automatically discovered by c/XDC and you WILL have to use the LIBRARYLISTS command (which is not our idea of fun). The relevant lines of JCL are italicized below.

```
//PLINK    EXEC PGM=PLINK,REGION=0M
//STEPLIB  DD  DISP=SHR,DSN=CSWPP.DIGNUS.LOADLIB
//STDERR   DD  SYSOUT=*,LRECL=133,RECFM=FB
//STDOUT   DD  DISP=SHR,DSN=FRANK.DIGNUS.LISTINGS(PLNKBRKN)
//SYSLIB   DD  DISP=SHR,DSN=CSWPP.DIGNUS.LIBCRZ
//INDD     DD  DISP=SHR,DSN=FRANK.DIGNUS.OBJECT
//SYSIN    DD  *
  INCLUDE INDD(BIRKEN2)
//SYSMOD   DD  DISP=SHR,DSN=FRANK.DIGNUS.OBJECT(BIRKENP)
//SYSDBG   DD  PATH='/u/frank/dignus/birkenp.dbg',
//         PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//         PATHMODE=SIRWXU
```

After the PLINK step, you FTP your code up to the mainframe, where you will perform the Binder step. You can find an example in the z/XDC libraries. We ship it as "**DBCOLE.XDCZ22.XDCJCL(CMPCSAMP)"**, where the low-level qualifier is "XDCJCL".

If you're curious about using the LIBRARYLISTS command, I've included some instructions from Frank in the section on using LIBRARYLISTS further down.

Compiling Under Unix Systems Services

Unix command line:

xlc [...] -g [...] /path/filename[.c]

The "-g" flag is necessary for the compiler to generate DWARF data. Then, DWARF data is written to /path/filename/ with a .dbg file extension. In this case, c/XDC will find the DWARF and source datasets without further effort on your part, just as it does in XL C/C++.

Caveat: At this time, c/XDC has problems executing under USS, but it has no problem with USS-based compilations that are moved over to z/OS and run under z/OS. We hope, over time, to address this.

Introducing XDCCSAMP - The "Toy" Program That We Distribute

The code I'll show in this booklet is taken from "XDCCSAMP", which is distributed as part of the c/XDC product. The program is distributed ONLY as source code, so you will need to compile and bind locally it in order to follow along with the screenshots we use in this book.

We distribute the z/XDC product using our "factory naming convention", and your installing systems programmer may have altered the dataset names to your own local naming standard, but:

- When it left the factory, the source code for XDCCSAMP was contained in CSW.XDCZ22.XDCSAMP(SRCCSAMP)
- When it left the factory, the JCL used to compile XDCCSAM for both LE (XL) C and Metal C was contained in "CSW.XDCZ22.XDCJCL".

Consult your local systems programmer to find out how these names were changed at your site.

The JCL examples show you how you might both compile and bind your programs. Look for these members:

CSW.XDCZ22.XDCSAMP(SRCCSAMP) <-- source code for XDCCSAMP

CSW.XDCZ22.XDCJCL(CMPCSAMP) <-- compile JCL for LE C/C++

CSW.XDCZ22.XDCJCL(CMPMSAMP) <-- compile and Assemble JCL for Metal C

CSW.XDCZ22.XDCJCL(LCSAMP) <-- bind JCL for LE C/C++

CSW.XDCZ22.XDCJCL(LMSAMP) <-- bind JCL for Metal C

CSW.XDCZ22.XDCJCL(RUNCSAMP) <-- JCL to execute XDCCSAMP for C/C++*

CSW.XDCZ22.XDCJCL(RUNMSAMP) <-- JCL to execute XDCMSAMPfor Metal C*

*Note: In the last two JCL decks above, you won't see the "normal" z/XDC EXEC statement in the RUNCSAMP and RUNMSAMP JCL (which executes XDCCALL, and specifies RUNCSAMP and RUNMSAMP as PARMS to the EXEX statement). That is because XDCCSAMP includes an embedded cxdchook() function, which will cause c/XDC to get control via the Hook. That is the method we recommend, so we're following our own example.

We also recommend that you add a //XDCJSTEP DD DUMMY card to your batch JCL. This helps c/XDC and z/XDC deal with some issues having to do with multiple Task Control Blocks.

When using c/XDC in batch mode, it's better to start a debugging session by embedding a static Hook. Then, when your batch program executes, it will hit the HOOK and create a c/XDC session very gracefully. This is what happens in RUNCSAMP and RUNMSAMP.

The alternative is to set up your JCL in the "classic" z/XDC way, which is to specify:

// EXEC PGM=XDCCALL, PARM='<your program and its parms>'. <== for non-authorized programs

or

// EXEC PGM=XDCCALL**A**, PARM='<your program and its parms>'. <== for authorized programs

After this job executes, z/XDC will come up in a batch debugging session. You'll use Option 3 from the z/XDC Startup Panel to start the session, and off you go! We'll talk about all this further down.

How To Co-exist With The Language Environment

Language Environment has its own great, big Recovery Routine, and part of what you'll do to use c/XDC under LE is to turn off the ESPIE that LE sets. That's because ESPIEs get called ahead of ESTAEs, and ESPIES prevent z/XDC from keeping control of the debugging session. This applies to debugging both in your TSO region and in batch regions.

Dave has written a very good treatment of the subject in the z/XDC Help Subsystem. It's entitled, "Help DEbugging LE". We recommend that you read this topic.

This is why we ask you to specify the TRAP(ON,NOSPIE) LE run-time option. This will disable most of LE's recovery routines, which allows c/XDC and LE to co-exist. However, you can get away with specifying only TRAP(OFF) because in that case NOSPIE is implied.

Note: While using TRAP(OFF) is useful, it disables LE's recovery for things that LE needs to handle, such as an LE stack overflow event. Thus using TRAP(OFF) is a last resort

Here's a hyperlink to IBM's discussion on TRAP for z/OS V2R3:

https://www.ibm.com/docs/en/zos/2.3.0?topic=options-trap

There are two ways to specify run-time options to LE in your JCL. Here, I've shamelessly copied Dave's own words:

You may put them into a sequential file, and use DDname CEEOPTS to point to it.

Example:

```
// EXEC PGM=XDCCALL
//        PARM='<mypgm myparms>'
//CEEOPTS  DD *
<le runtime options>
/*
```

or

Put them into the PARM field. Use a slash ("/") to separate them from your programs' own parameters.

Example:

```
// EXEC PGM=XDCCALL
//        PARM='<mypgm le runtime options/myparms>'
```

or

```
//        PARM='mypgm myparms/le runtime options>'
```

Apparently, some LE environments expect the <le runtime options> to come ahead of your program's parms, while other LE environments expect them to come after your parms. Go figure!

Because of c/XDC's issues with regard to LE, we suggest the use of the cxdchook() function to get control in your C code. That's a static hook that you compile into the source code for testing purposes. The cxdchook() function is like an "uber-breakpoint" that causes z/XDC to create a debugging session right-here/right-now no matter WHAT other recovery environments exist.

The cxdchook() function (a static hook) is excellent for getting control when you want to in very complex environments (IMS, DB2, CICS, etc.)

Alternatively, you may also use EXEC PGM=XDCCALL or EXEC PGM=XDCCALLA to get control in a batch job, and then set a dynamic hook from the command line in your debugging session, *but* you should be careful not to put that hook into LE initialization, termination, prolog or epilog code *or* anything that is named or labeled, "CEE-anythingor

When you use any variant of Hook to get control in your debugging session, then z/XDC creates an entry in the STAE Control Block chain. This SCB entry can interfere with LE's cleanup. So, when you're done with your debugging session, you should issue "DELETE HOOKS" to get rid of any dy-

namic hooks that exist, and then issue “GO REMOVEXDC”. The REMOVEXDC parameter will remove the z/XDC entry from the SCB chain and LE will terminate cleanly.

Starting a Debugging Session With XDCCSAMP In Your TSO Region

After you've compiled and bound XDCCSAMP, you may try it out in z/XDC and c/XDC will immediately get control.

Here's a screenshot of the z/XDC Startup Panel filled in correctly. Because your Systems Programmer may have altered our dataset names to your local naming convention, your TASKLIB DSN will probably be different from mine:

```
 ------------------------- 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 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) ===> NO    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  ===> XDCCSAMP                    Upcase the PARM? ===> YES
     CMD/PGM PARM  ===> TRAP(ON,NOSPIE)/

     Tasklib DSNs  ===> 'CSW.XDCZ22.XDCLINKE'
                   ===>
                   ===>
                   ===>
                   ===>
                   ===>
     or DDNAME     ===>
 PARAMETER FOR OPTION 3 (BACKGROUND DEBUGGING VIA cdf/XDC)
   Connect to cdf/XDC using your current TSO Userid? (YES/NO) ===> YES
```

Figure 2-03 - Filling in the z/XDC Startup Panel

Above, I've entered a "2" in the command line. This will execute the XDCCALL program in my TSO Region. It will load itself, then attach XDCCSAMP as a subtask. Don't press Enter just yet. Let me explain the parameters on this screen first.

All the parameters in the middle section of the display starting with PARAMETERS FOR OPTION 1 OR 2, etc. are relevant only to working in my TSO region. You can see that I've invoked XDCCSAMP, the toy demo program, the parm TRAP(ON, NOSPIE)/. The trailing / is needed and indicates that anything prior to the / are the LE runtime options. Anything after the / are parameters passed to the target program. Finally I include a TASKLIB DSN where I'll find my local copy of XDCCSAMP. That library may be named differently at your installation.

Regarding the TRAP statement: If you turn LE's recovery mechanism completely off (TRAP(OFF)/) then LE won't handle stack overflows. It's a good idea to let LE handle this, so we suggest that you specify TRAP(ON,NOSPIE).

After I press Enter, c/XDC does its thing, and I see the initial debugging display.

We’ll return to this panel in Chapter 4, but first I want to introduce you to the Profile Facility...

Chapter 3 - Introducing the z/XDC Profile Facility

Our base product z/XDC provides a Profiling Facility that allows you to configure z/XDC's behavior in your debugging sessions. Among other things you can control PFKey settings and screen layouts, display colors, z/XDC's AUTOMAPPING feature and High Level Language settings.

Several new profiles have been created for your work with High Level Languages such as C. You can use the profile named C (or CEE) to get all the settings. You may use CWIDE for wide terminal displays and "C80" for narrow displays.

The command to access them is "PROFILE RESET <C/CEE/CWIDE/C80>". By far, my personal favorite is "PROFILE RESET CWIDE".

If you're a busy person in "task-mode", you can skip ahead to Chapter 4 at this point. Otherwise, please hang in there for a bit longer.

When I invoke any of the C-oriented profiles, all the choices for array depth and structures have been set to the maximum values that are possible for a terminal emulator. These settings control the number of levels of variables that are displayed. So while the user may have a 1000 level structure, c/XDC can only display the top 250 (or something) due to the geometry limits of the terminal emulator.

Note: In z/XDC's profiles both the AUTOMAP and AUTOSTEP commands are defaults. When both are invoked much goodness results for you in c/XDC. The two commands help you with automatically mapping your code and skipping the prologue area in a C program (which you usually don't care about). You can research these commands by entering "Help COmmands AUTOMAP" and "Help COmmands AUTOSTEP".

Frank wants me to mention that AUTOMAP and AUTOSTEP can also be turned off via DD statements, and the presence of these DD statements will override anything specified in your z/XDC Profile (which I'll mention later). The two DD statements are:

- **// XDCNOMAP DD DUMMY, and**
- **// XDCNSTEP DD DUMMY**

I can't understand why anyone would want to turn these helpful facilities off.

Let's get back to Profile.

From your z/XDC debugging session you can enter “PROFILE” to enter the Profile Facility. Below is the output of that command:

Figure 3-01- the PROFILE main menu

As a c/XDC user you may be most interested in “AUTOMAP PRINT and Other Settings” and “c/XDC Settings”, but let me remind you that you don’t have to touch a thing. If you’ve issued PROFILE RESET CWIDE, then we’ve already configured your profile to take the greatest advantage of all of c/XDC’s juicy wholesome goodness. I’m merely being complete in showing you where to find these settings.

If you want to investigate or alter these settings, just tab down or place your cursor next to a sub-topic, enter an “S” and press Enter. You’ll see the settings and you can override them by typing over the existing values.

A few pages back I mentioned AUTOSTEP. That appears in the final selection in the Profile main menu under “C/XDC Settings”. This panel also contains the settings for STEP and mandates how c/XDC will treat array and structure depths. You may change these settings, but they are already set to useful values.

Here's the panel itself:

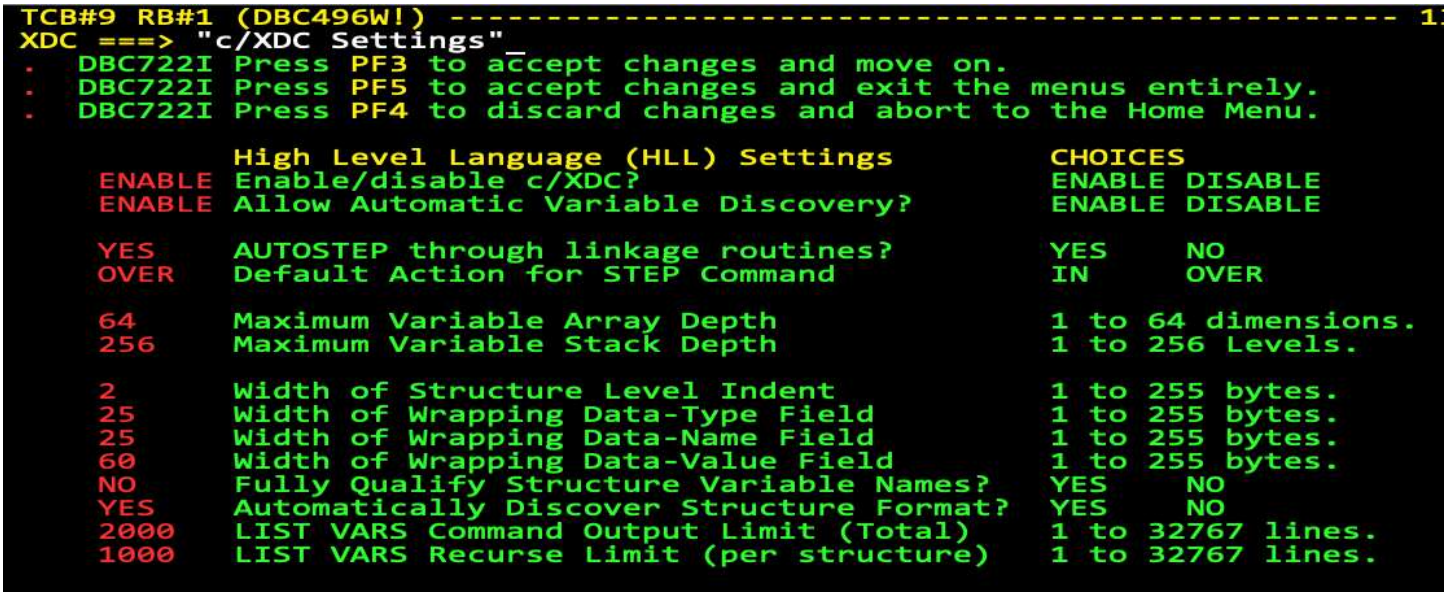


Figure 3-02 - Contents of the Profile Facilitiy's Last Panel

When you're done changing things, you can press PF3 to get out one level or PF3 multiple times to get all the way out. By the way, you can't issue any other z/XDC or c/XDC commands while you're in the Profile Facility. You have to "END" your way out of Profile entirely before you can use other commands.

To save your changes issue "PROFILE SAVE XDC" (or "PROFILE SAVE <blank>") which will make these settings your "default" profile settings, or you can issue "PROFILE SAVE <up-to-5-character-name>", which will create a named profile in your ISPF Profile library.

Note: Why only five characters and not eight? Because z/XDC prefaces all your profiles with "XDC" (or XDC's current "clone name" which I won't get into) so that profile "BOB21" will have a real name of "XDCBOB21".

To retrieve a saved profile, you can issue PROFILE READ <up-to-5-character-name> and retrieve all the cool settings you have created in your named profile.

Note: If you intend to use c/XDC for batch programming you should add a DDCARD named "XDCPROF" to your JCL so z/XDC can find your profiles when debugging in batch address spaces.

If you ever get confused or totally messed up, issue PROFILE RESET CWIDE again. That will restore the factory default settings that we distribute with the z/XDC product (or the settings your installing system programmer has established for you).

You can have as many named profiles as you like, and each can be set up for a particular programming problem. You could call up a profile that is specifically tailored for a certain application or group of applications with PF Key settings (perhaps one that executes a script for the LIBRARYLISTS command) and screen layouts that make your debugging a great deal easier.

Example: Suppose that I'm going to spend some days/weeks/months working on an application named XYZZY. Why not start my debugging session, issue PROFILE READ XYZZY, and have EVERYTHING set up just the way I like it for the XYZZY environment?

NOTE: For C-centric programmers, there are two profiles that will help you a great deal. They are column width-dependent. For 80 column displays ("narrow displays), issue "PROFILE RESET C80". For 132 column displays, issue "PROFILE RESET CWIDE".

Both of these profiles have AUTOMAP set to "YES", and both of these profiles set c/XDC for the maximum depth of arrays and structures.

One last nice thing about profiles: Whenever you arrive at a screen, colors, PFKey settings and layout that you really like, you may issue PROFILE SAVE without any parms, and that will become your personal default for as long as you like.

The definitive Help topic for Profiles is contained in Help Profiles. Start a debugging session, enter Help Profiles from the command line and read to your heart's content.

How to Configure Your Work-space

You can arrange your 3270 interface to suit your needs by creating windows. z/XDC uses two kinds of windows: The Working Window and zero or one or more Watch Windows, which we also refer to as Display Windows. Have a look at the screenshot below:

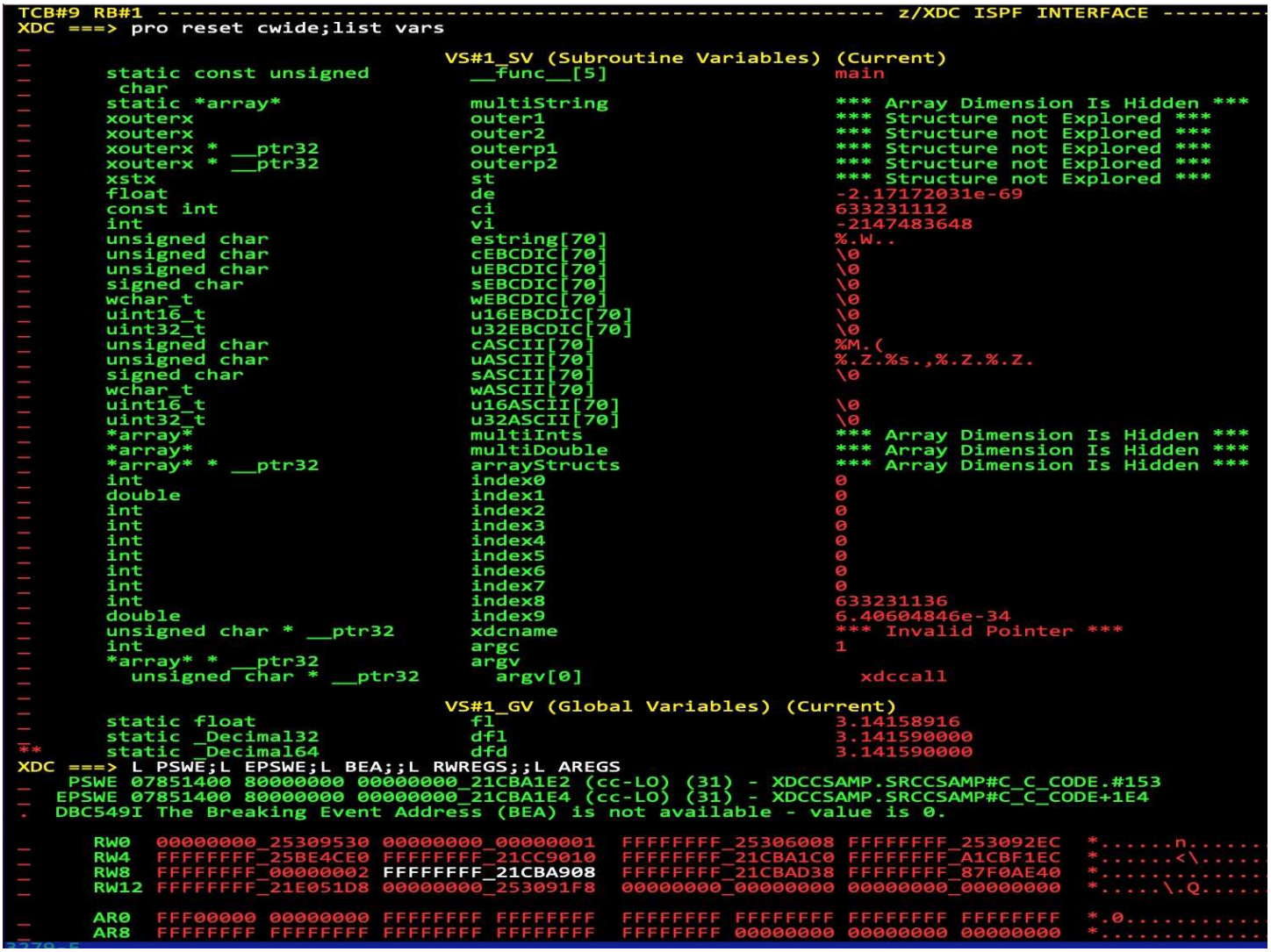


Figure 3-03 - Working Window vs. Watch Window

The Working Window is the top window in the display which contains the first instance of “XDC ===>”. That’s where your code will be displayed, where you’ll STEP through it, where you’ll analyze the source statements, perhaps make changes to the underlying object code, zap new values into variables or alter the flow of your code.

In Figure 3-02 The Watch Window is the second instance of “XDC ===>”. In c/XDC’s “CWIDE” profile, Dave has created a watch window containing stacked commands to show the Program Status Word, the Error-level Program Status Word (you don’t need to know about this) and both the General Registers and the Access Registers. I hate it.

If you’re a C-centric programmer you really don’t care about these things. But you have the opportunity to alter the commands in any Watch Window’s command line to suit yourself. This is, again, the factory default, which I hate.

Instead, insert the “LIST VSTACK” command. These values won’t change for the “toy” program

XDCCSAMP, but in your own programs these underlying structures WILL change. New entries are created in LIST VSTACK's output whenever execution goes into a function. GLOBAL entries are available for all functions. Functions and local subroutines will pop into and out of existence depending upon function calls.

Here's what your screen looks like with the LIST VSTACK command in the Watch Window's command line:

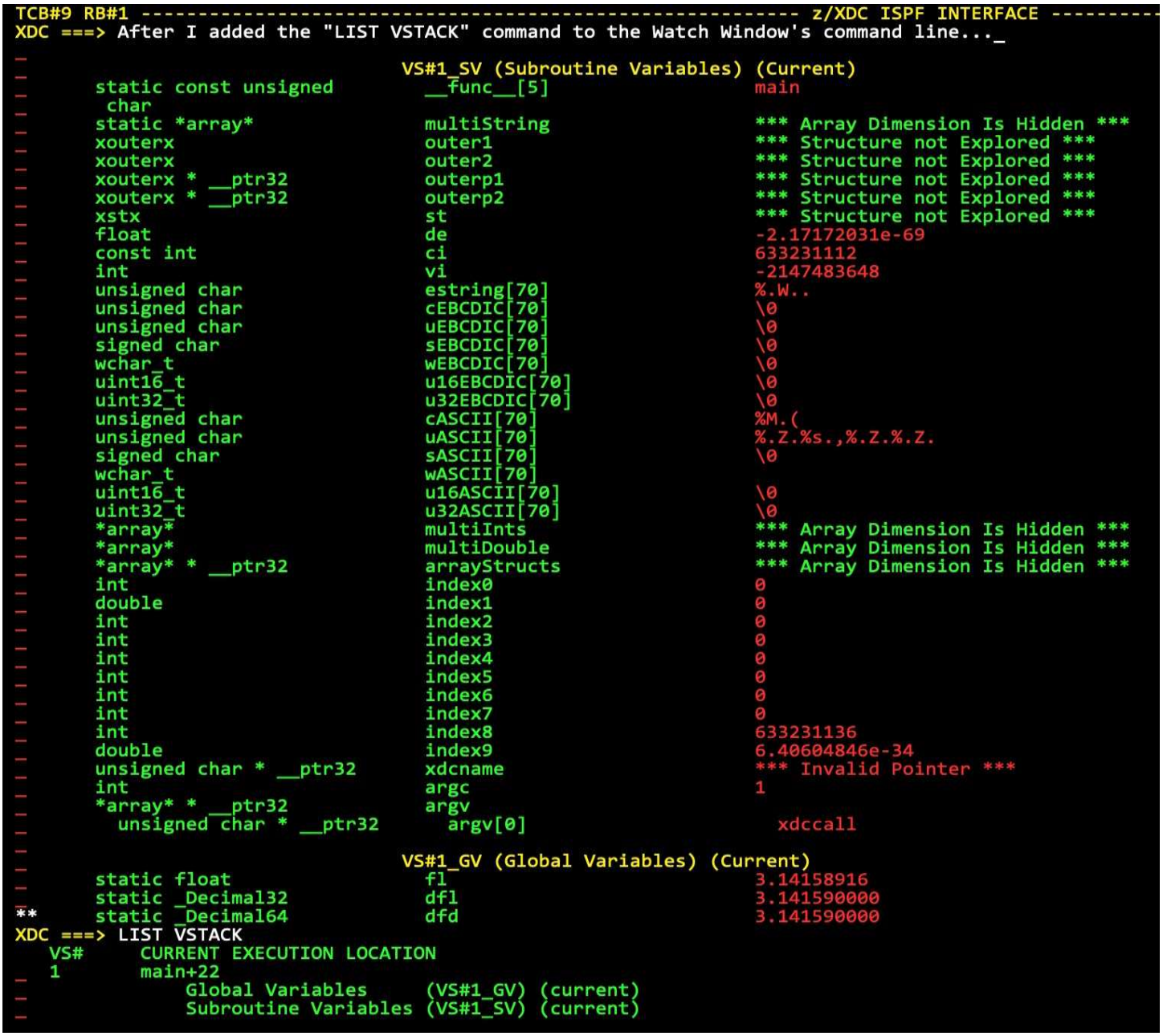


Figure 3-04 - After adding the LIST VSTACK command

What are Watch Windows Good For?

You may create up to fourteen Watch Windows. Typically, they are used to "keep an eye" on things you're interested in as your program executes. You can put LIST or DISPLAY commands into the Command Line of Watch Windows that resolve to variables, arrays, command buffers, registers or anything at all that you wish to monitor. As your code executes these displays (if altered by the code) will shift and move so that you can see the effect of your code on them in real-time.

One idea would be to create a Watch Window to see how a variable is manipulated by the code. When it's no longer of interest to you, delete that Watch Window.

If you want to keep the screen layout you've created, you can issue PROFILE SAVE <blank> (or PROFILE SAVE XDC) to make this your default screen layout. You may also issue PROFILE SAVE <up-to-5-character-name> to save it as a named profile. The resulting Profile will be stored in your ISPF Profile Library under the name "XDC<up-to-five-character-name>". For example, your FRED Profile will appear as "XDCFRED".

How do I create and destroy Watch Windows?

So, how are Watch Windows created or deleted?

1. In the "Factory Default" profile, PFKey13 will issue a SET WINDOW CREATE command.
2. Conversely, PFKey1 will issue a SET WINDOW DELETE command. This can be used to delete any Watch Window. You can't delete the Working Window.

To create a Watch Window, position your cursor anywhere on the display and press PFkey13. z/XDC will create a Watch Window on that row, with a command prompt (XDC===>"). The new Watch Window will inherit the command from any previous Watch Window. Overtype the old command with anything you like, press Enter and the changes will be saved.

Again, if you arrive at a layout you really like, you can issue the PROFILE SAVE or PROFILE SAVE <5-character-name> to save this layout for another time.

No, z/XDC does not provide a GUI. z/XDC's windows aren't movable/sizable with full-screen or minimizing capabilties. z/XDC is (and remains) a 3270 interface, which imposes a *lot* of limitations on how we can present information to you. On the other hand, we're doing just about everything that a 3270 interface *can* do.

A future version of z/XDC and c/XDC may utilize a more modern GUI interface. For now, this is what we can do, and if you take time to learn Profile and the use of SET WINDOW CREATE/DELETE you may get pretty comfortable in this environment.

Chapter 4 - Starting A c/XDC session

I'm going to start out by exploring XDCCSAMP in my own TSO address space. Then, I'll show you around the major features in c/XDC.

Here, I'm setting up c/XDC to run against XDCCSAMP in my own TSO address space. Normally, you'll be working on programs running in Batch Regions (and I'll discuss how to do that later on).

Note: If you're interested in learning how to debug Batch Jobs NOW, then issue HELP DEBUGGING BATCHJOBS. Hint: The essential change is to alter the EXEC PGM= statement from EXEC PGM=myprogram to EXEC PGM=XDCCALL PARM='my program and its parms'. If your code executes APF Authorized, then you may say EXEC PGM=XDCCALLA, etc.

Along the way, I'll include the command as well as the result in each of the figures in this book. That's not how c/XDC really works (because the command line is cleared after each command is executed) but I want you to see the command as well as the result. With that in mind, let's proceed.

Here is how Frank suggested that I fill in my z/XDC Startup Panel:

```
------------------------- 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 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) ===> NO    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  ===> XDCCSAMP                    Upcase the PARM? ===> YES
    CMD/PGM PARM  ===> TRAP(ON,NOSPIE)/

    Tasklib DSNs  ===> 'CSW.XDCZ22.XDCLINKE'
                  ===>
                  ===>
                  ===>
                  ===>
                  ===>
    or DDNAME     ===>
PARAMETER FOR OPTION 3 (BACKGROUND DEBUGGING VIA cdf/XDC)
  Connect to cdf/XDC using your current TSO Userid? (YES/NO) ===> YES
```

Figure 4-01 - The z/XDC Startup Panel filled in for XDCCSAMP

Once again,the really important parameters are "TRAP (ON,NOSPIE)/". That turns off the Language Environment's "interference" with c/XDC and allows c/XDC to do its job.

Next, I press Enter to start the debugging session in my TSO Private Area. I get this:

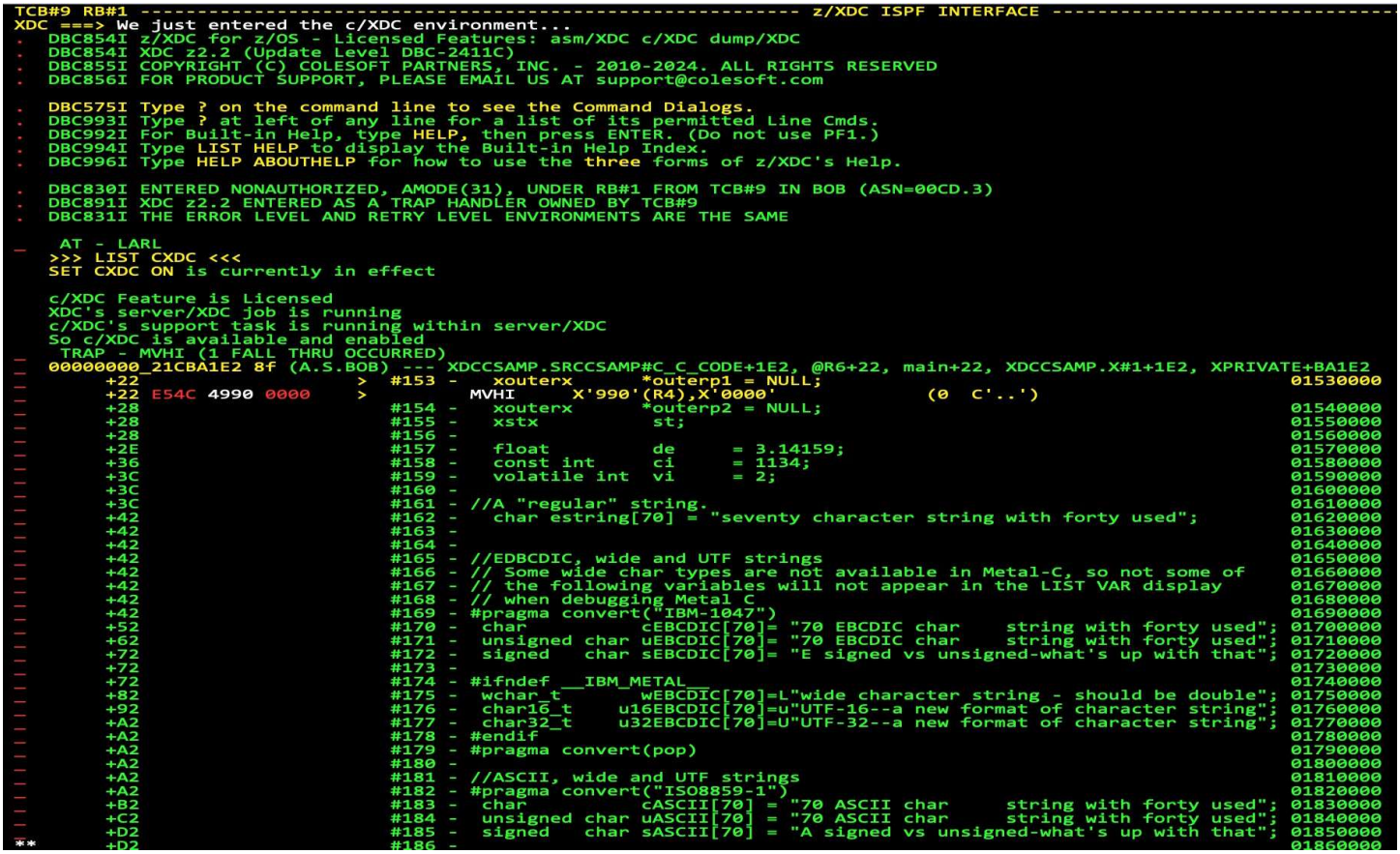


Figure 4-02 - c/XDC's Initial Display

A number of things have just happened. Let's go over them.

1. Before showing you this display c/XDC did something pretty cool. It noticed that I was debugging a C/C++ program, and breakpointed its way through the Language-Environment prolog so that the first instruction to be displayed was the first executable C/C++ statement in the user's code. c/XDC bypassed displaying the LE-prolog, which you generally don't care about.
2. z/XDC then displayed the normal messages it displays at the beginning of any debugging session which, among other things, shows you the version of the product being executed. You can safely ignore those paragraphs. HOWEVER, notice the last line of the first paragraph. That's our support email address. PLEASE reach out with your questions and we'll help you.
3. c/XDC then issued a LIST CXDC command to prove to you that the c/XDC feature is available
4. Then, c/XDC automatically MAP'd your code, stopping you at the first statement in your C program, filling the Working Window with source code.

Note: A C program statement can take up many more than just one line of the display. If you see two white asterisks at the bottom of the working window, you have the opportunity to PageDown to see the rest of the program or statement. In fact, you can ALWAYS use PageDown to scroll through the program if you like.

Notice the yellow text at offset +22 near the middle of the Working Window. That's the C statement that will execute next.

Issue PROFILE RESET or PROFILE RESET CWIDE (for wider terminal displays) and we'll get everything set up for debugging in c/XDC.

Here's an example of the result:

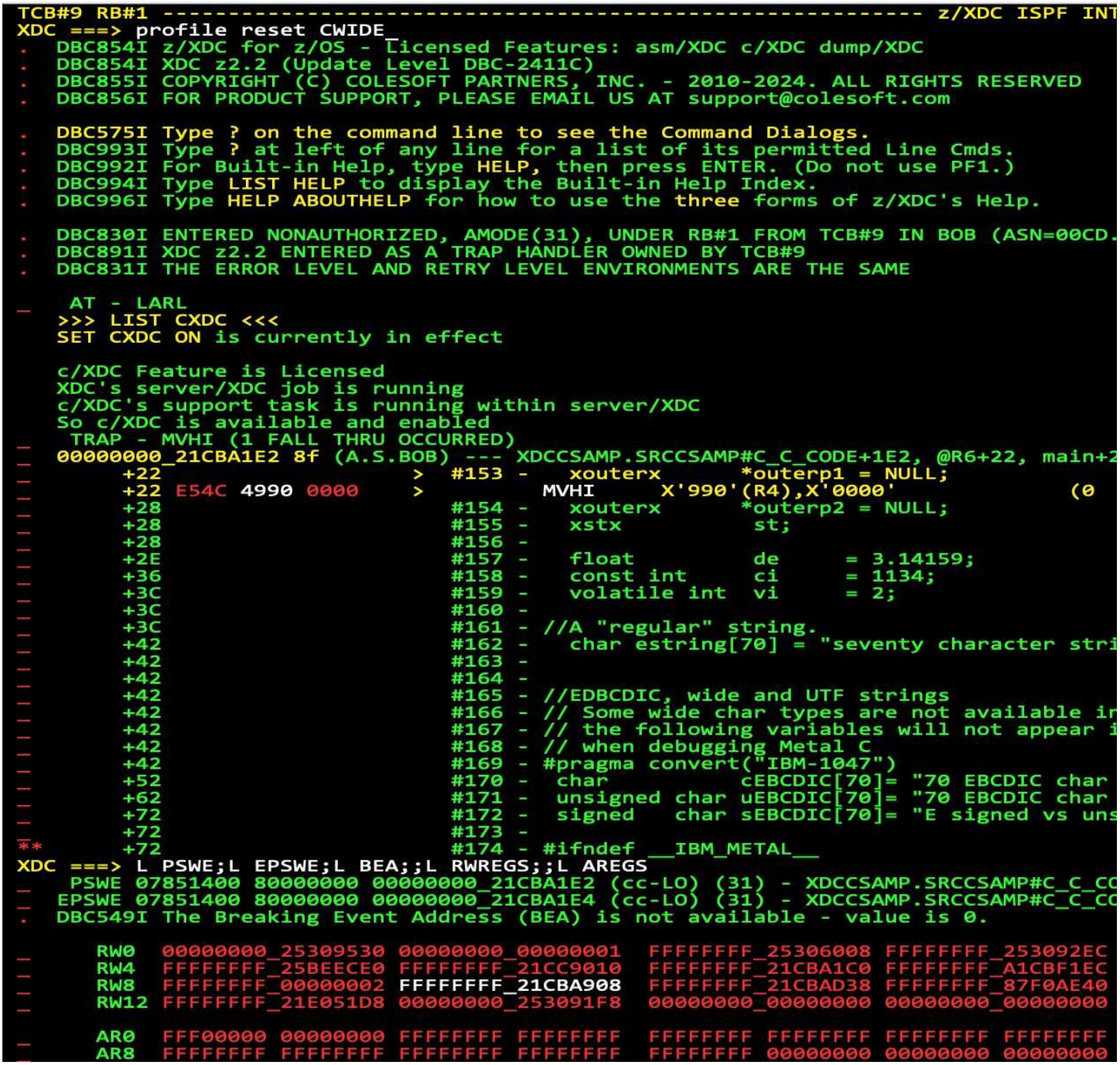


Figure 4-03 - Result of Profile Reset CWIDE

Again, here is Dave's preferred view of the world. As I discussed above, you may delete the Watch Window entirely by putting the cursor anywhere in it and pressing "PF1" - OR you may put whatever command YOU WANT in there instead.

The SET QUALIFIER command

z/XDC can establish a default module or csect for you. In the C world we refer to a csect as a Compilation Unit. Anyway, if you establish these defaults, then you can refer to individual executable statements in your program by statement number.

To do this, you may issue SET QUALIFIER <address-expression>. The address-expression is any virtual address that resolves inside your program. PSW? would be a good one to try.

However, it is much easier to just put your cursor in column one of your display next to any line of code and enter a “Q” - one of z/XDC’s many shortcut commands. Q accomplishes a SET QUALIFIER command without having to specify an address-expression, because z/XDC resolves it for you.

After you’ve established your defaults with the Q shortcut command, you can view any executable statement in your code this way: **F .#nnn** where:

1. **F** stands for “FORMAT” (show me source code);
2. **.#**” is our way of telling c/XDC to expect a statement number and;
3. **nnn** is a statement number.

Frank tells me that the “#” above isn’t static. When execution is in the csect that contains the main(), then it will be a “#”. Other csects will range from “A” to “Z”, and from “AA” to “ZZ” depending upon the sequence in which c/XDC process them.

The definitive resource is contained in Help COmmands SEt Qualifier.

Address Expressions - Telling c/XDC Where to Apply a Command

If I were to hand you a hammer and tell you merely, “Please hammer” that would be meaningless to you. But if I say, “Please hammer that nail in”, then you’d know where to apply the hammer. The “where” in both z/XDC and c/XDC is referred to as an address-expression.

Address expressions are composed of three elements: **Bases**, **Operators** and **Offsets**.

1. A **base** can be a statement number in your program. It can be a register, or the Instruction Location Counter of the Program Status Word. It can be a pure virtual address, or any EQUATE that you define in storage.
2. An **Operator** can one of four arithmetic types: (“+” for addition), (“-” for subtraction), (“/” for division) and (“*” for multiplication) and three indirect operators: (“%” for 24-bit addressing), (“?” for 31-bit addressing) and (“!” for 64-bit addressing). Indirect operators may be “stacked”.
3. An **offset** can be some numeric hexadecimal value, such as 0C or 24A, etc. If you want to express a decimal number then follow the number with a “N. So, 100N is a decimal 100. 100 without the N suffix means hexadecimal 256.

Let’s see some examples.

- If I tell c/XDC Format PSW? then it will show you source code (if properly mapped) at the NEXT statement that will execute, and in this case it will assume that you’re using 31-bit addressing (Whenever you tell z/XDC PSW it means use the ILC portion of the PSW, not the many other parts of the PSW.).
- I could also enter Display R12???+A and c/XDC would give you a raw storage “dump” using the contents of register 12 as a starting point. Using 31-bit addressing, it would give you the third “pointer” hung off that register (three question marks stacked) and add 10 decimal bytes. THAT’s what this particular Display command would actually show you.
- If I issue a Q shortcut command (for SET QUALIFIER - discussed later on), and then issue F .#156 then c/XDC will show me my source code at statement number 156.

My personal favorite method is to *let z/XDC handle the address-expression for me*. Shortcut commands are one way of accomplishing this, and they’re discussed later on. But my all-time favorite address expression is <command>-space-plus-zero or +0.

If I tell z/XDC to F +0 or M +0 or D +0 I’m telling z/XDC and c/XDC to do a Format, or a “Map or a “Display command *where I am right now*. I get away with this because z/XDC maintains an equate called @CDP” which stands for the “Current Display Pointer. So I’m telling z/XDC to use the equate @CDP as my base, and since z/XC constantly updates that for me, I don’t have to express the virtual address or statement number.

Address expressions are a huge topic in z/XDC, and you can find the definitive reference in our Help Subsystem under HELP ADDRESSING.

Following Program Execution - The STEP command

After you've gotten control in your C program, you'll probably want to watch it execute. The simplest way to do that is with the STEP command. Issuing STEP without any parameters will cause the system to execute a single C statement, and will give you control on the next statement that will be executed. Presuming the previous screen wherein we established our session, I'll show you the result of a single STEP. To execute a statement, you can use issue the STEP command (abbreviation: ST). If you're using c/XDC's CWIDE or C80 profiles you can also use PF9 for a STEP. Afterwards, you see this screen:

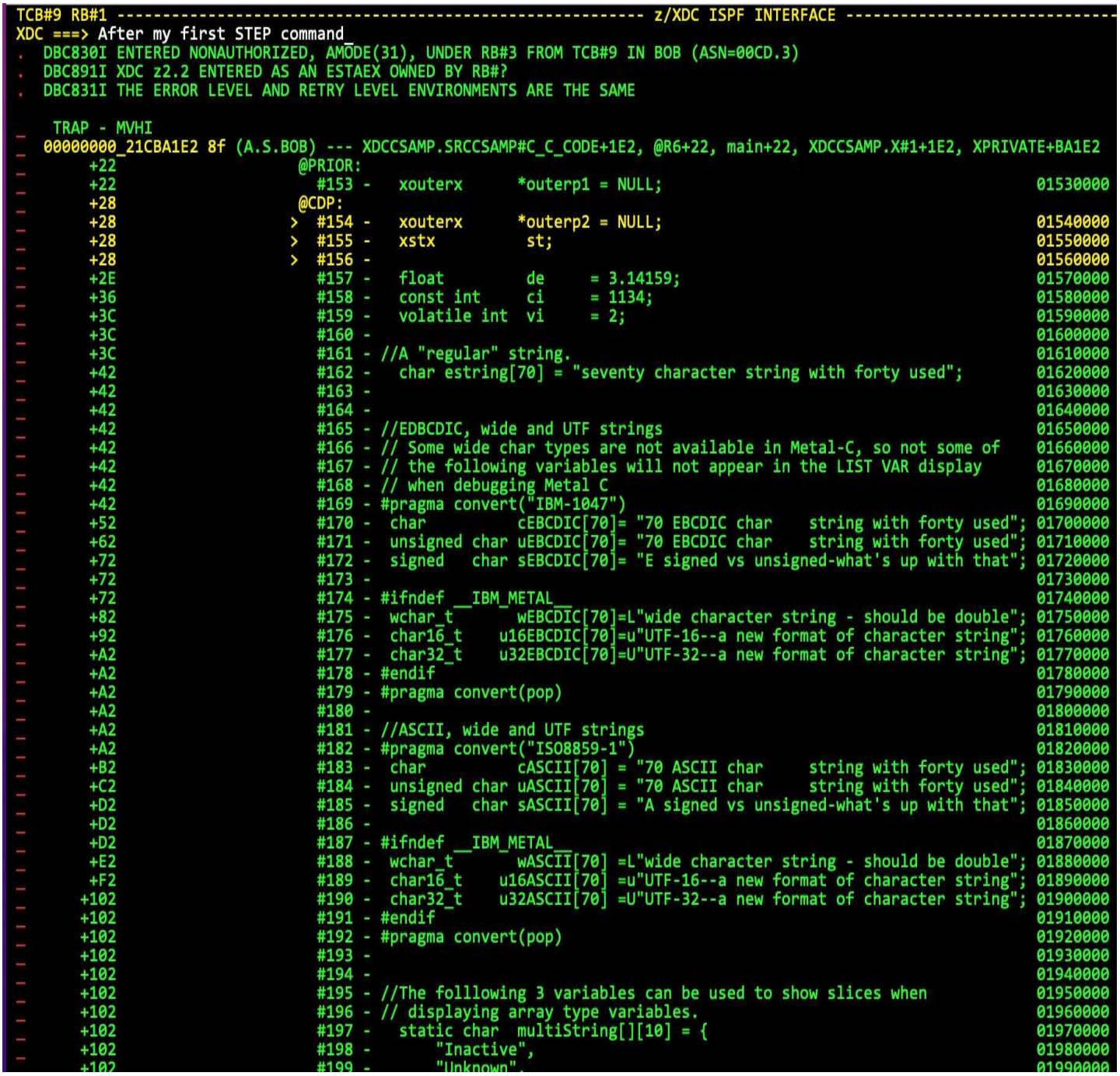


Figure 5-1 - After the first STEP command

As you can see, c/XDC has now advanced to the next statement at offset +28 in the program.

There's much more you can do with STEP. If your program executes a subroutine or function call, then you can choose to STEP IN to examine the subroutine or function, you can STEP OUT to resume execution after the subroutine or function and you can STEP OVER if you don't care to look at the subroutine or function.

Note: There are some rules that c/XDC follows when you attempt to STEP INTO a function. It will only work when the current C statement:

- Calls a function;
- The function is mappable;
- The Target function is not part of the LE runtime

Otherwise, c/XDC just does a STEP OVER.

What's pretty cool is that z/XDC pays attention to your local environment. If the subroutine happens to be written in Assembler, then z/XDC will AUTOMAP it and attempt to load source statements for the Assembler subroutine (if it can find them).

In this fashion you can debug from C to Assembler and back - seamlessly. To you, that's very convenient. To a guy like me, that's a strong marketing point! I did say I was in Sales....

The definitive resource can be accessed at Help COmmands STep.

Variable Discovery - The LIST VARS Command

c/XDC offers a LIST VARS command that allows you to query all variables or any single variable, discover the content of arrays and structures. Here's the output of the LIST VARS command for the XDCCSAMP program. However, because XDCCSAMP hasn't yet populated these variables, please be patient for just a bit.

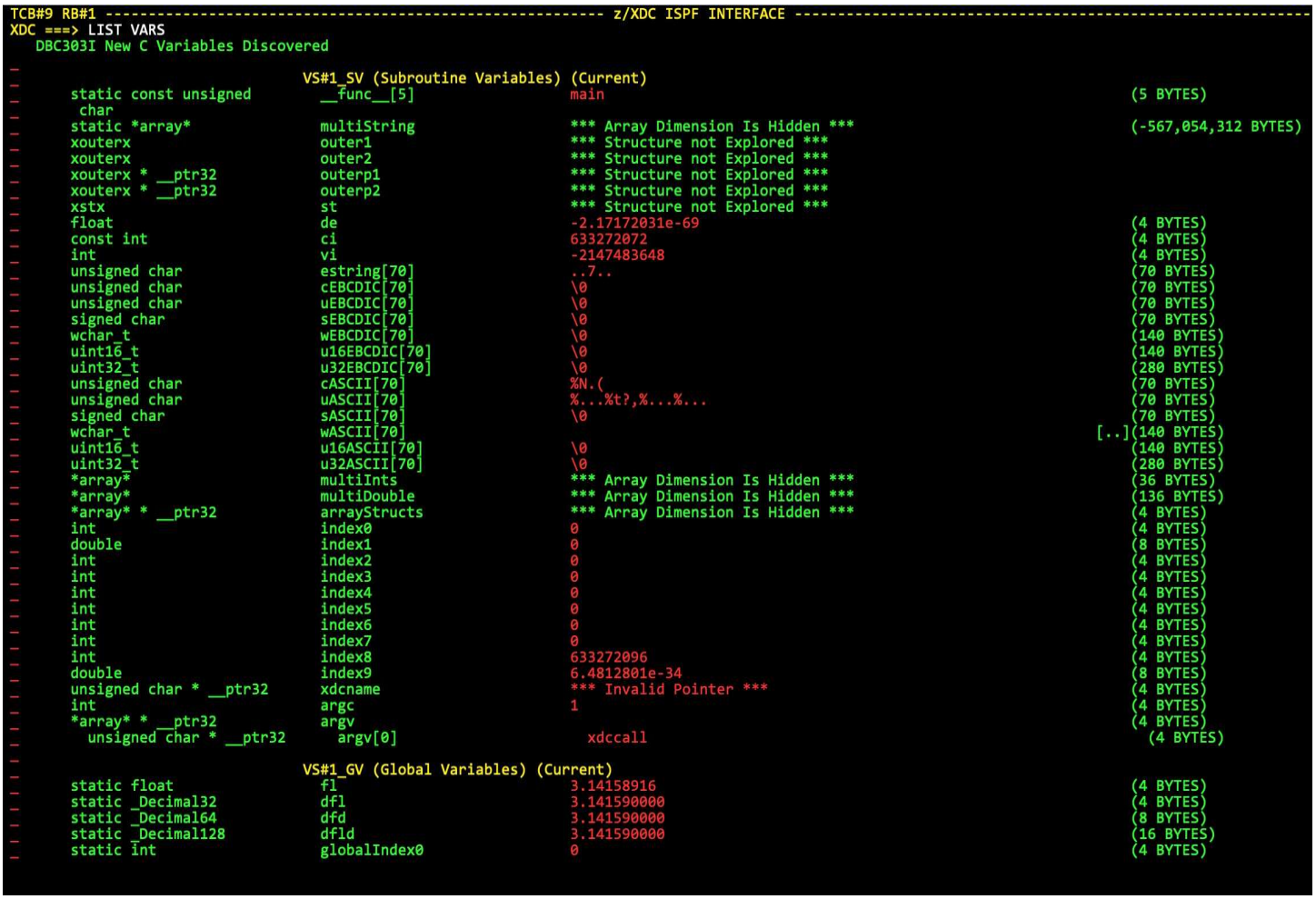


Figure 5-2 - Output of the LIST VARS Command

Now, the XDCCSAMP program isn't a huge compilation unit, so I got away with the default LIST VARS command, which shows ALL variables. c/XDC shows you the kind of variable that each is, its name, the value if it's not a structure and the size of each variable. You may also issue LIST SIZEOF <variable-name> if you care to.

Note: Frank reminds me that variable names in C are case-sensitive. So while case generally doesn't matter in z/XDC it DOES matter in c/XDC.

You can also query individual variables with LIST VARS <variable-name>. Below, I'll demonstrate that by issuing list vars index0:

Figure 5-3 - Output of LIST VARS INDEX0

There are three columns of information being displayed: The type of variable c/XDC has detected, the name of the variable and the value seen there. Some values are visible now (in the early stages of this program) and some are not. If you see, "Structure not Explored" that is self-explanatory.

I couldn't fit the entirety of the display here, so we don't see that the variable consumes 4 bytes of space.

The definitive resource for the List Vars command can be found at Help COmmands LISt VAriables.

The LIST SIZEOF Command

c/XDC also allows you to query the amount of storage taken up by a variable (or all variables if you wish). This capability has also been added to the output of the LIST VARIABLES command, putting the size information way out to the right of the display. Hence, it may not be available to you on your 3270 emulator depending upon how many columns of data you have set it to. You can always issue RIGHT nn (where "nn" is the number of columns you want to shift) to see the size information in LIST VARIABLES (after which you can issue LEFT MAX to view from column one again). The appropriate online reference for this command is Help COmmands LISt SIZeof.

You can issue LIST SIZEOF ALL, if you wish, but I'll drop a tiny screen below.

Here I'll issue LIST SIZEOF estring:

Figure 5-4 - Output of LIST SIZEOF estring

Introducing Shortcut Commands

You may have noticed that there is a red underscore in column one of many of c/XDC's displays. That's a z/XDC construct; a place to put shortcut commands. These are one-character abbreviations of some of z/XDC's coolest commands.

There are a bunch of these shortcut commands, and in this book I'll use the ones I think you'll need most often. If you're curious (and I hope that you are) then issue Help SHortcutcommands and you'll see the entire list. Tab next to one that interests you, put an S next to it and press Enter. You've just delved into the Help subsystem.

One of my all-time favorite shortcut commands is ?. That's right. Put a question mark in column one of any z/XDC display and z/XDC will display all the shortcut commands that could apply in that situation. In other words, the display is context-sensitive. You'll see a zXDC message and all the shortcut commands that are allowed in the current context. If you want to read up on them, home your cursor, enter an "H" and press Enter.

Any time that z/XDC or c/XDC displays a message you can put H next to that message and press Enter. It's a hyperlink into our Help subsystem.

Back to c/XDC and variable displays.

Frank wants me to show you some of the neater aspects of c/XDC's variable displays. Before I do that, I want XDCCSAMP to populate some of those variables, and the way to do that is to let it run a bit. Embedded in XDCCSAMP further down is a cxdchook() function. This is a static Hook that was compiled into XDCCSAMP. Here, I issued SET QUALIFIER, and then issued FORMAT to look at statement 250 in the XDCSAMP program.

Figure 5-5 - The cxdchook function

The GO command

Later on, we'll talk about setting breakpoints in your program. Breakpoints may be set on any current display or in other parts of the program well off the current screen.

To execute to that breakpoint you issue the GO command. This allows the code to execute to the breakpoint where z/XDC gets control, does a BUNCH of stuff under the covers and presents it to you as your C program, stopped right there.

I would like you to issue GO now, so that XDCCSAMP will execute to the cxdchook() function, by which time all the variables will have been populated.

When c/XDC returns control to me, I see the image below. c/XDC executed the Hook, and created an entirely new debugging session at the Hook.

Here's a screenshot of what I see after I enter GO, and the Hook executes:

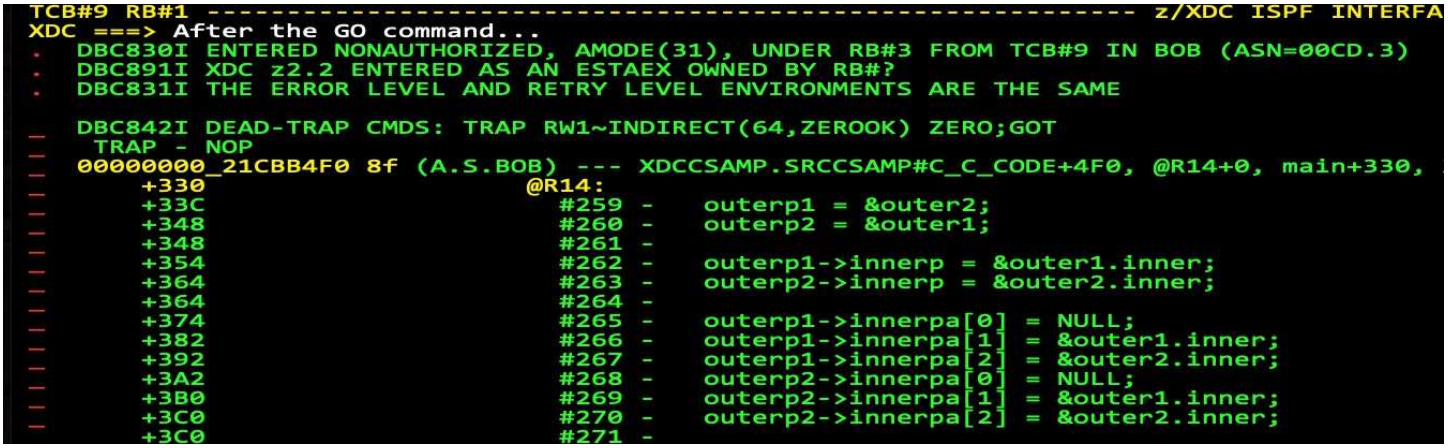


Figure 5-6 - After Executing the Hook

Now we can show you some variables.

Show a floating point number:

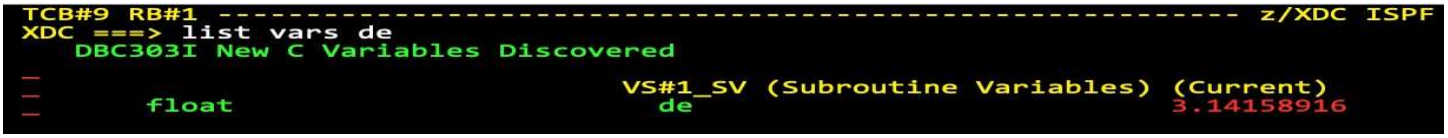


Figure 5-7 - List Vars DE

Let's have a look at a pure ASCII string. we'll issue "list vars cASCII":

Figure 5-8 - List Vars cASCII

This gives us the opportunity to show you some new shortcut commands that are specific to high-level languages.

First up is the L shortcut command. It's context sensitive. In some situations it will perform a LIST command of some sort. However, in this contect it will allow to look at variables.

In this context, I will use the L Shortcut command to drill into or (in some cases) drill out of a variable. Let's try that.

Here's an output of the LIST VARS command, and next to one of the variables I've placed an L:

Figure 5-9 - Placing an L in the output of LIST VARS

After I press enter, the L command is executed and I drill into uint16_t:

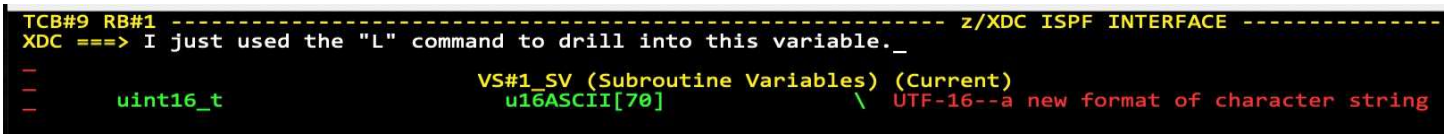


Figure 5-10 - After the L command processes

If I use a | next to the variable cASCII, then c/XDC will convert the display of cASCII (not actually convert the cASCII variable) from EBCDIC to ASCII representation:

Figure 5-11 - Showing cASCII as EBCDIC

To flip it back to EBCDIC, I use the asterisk ("*"):

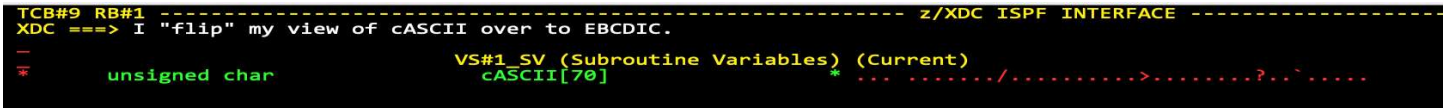


Figure 5-12 - Showing cASCIII as EBCDIC

One variable, estring[70] is a null-terminated string. I can show it as a length-terminated string with the / shortcut command:

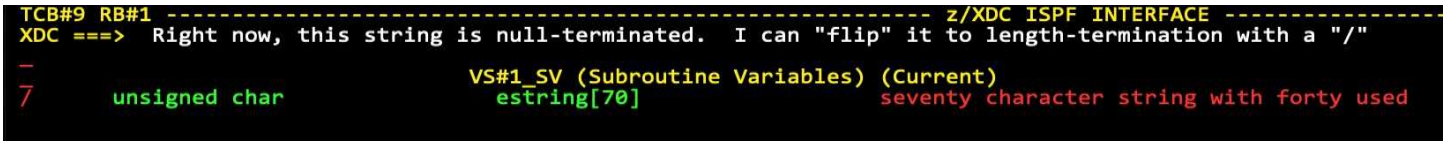


Figure 5-13 - Viewing estring as a Length-Terminated String

To flip it back, I use \ to change it back to a null-terminated string:

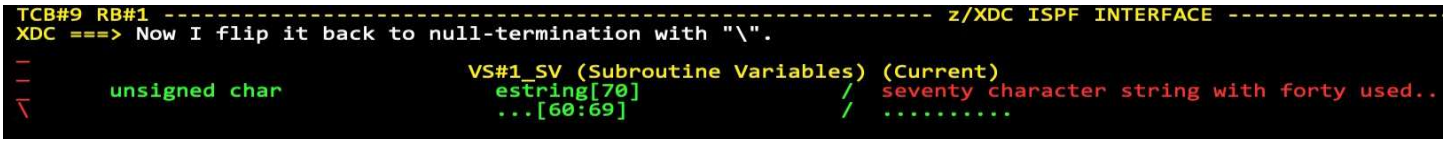


Figure 5-14 - Viewing estring as a Null-Terminated String

I can also use the L shortcut to drill down into an array. First, I'll issue LIST VAR, then I'll put an L"

next to the multiString array:

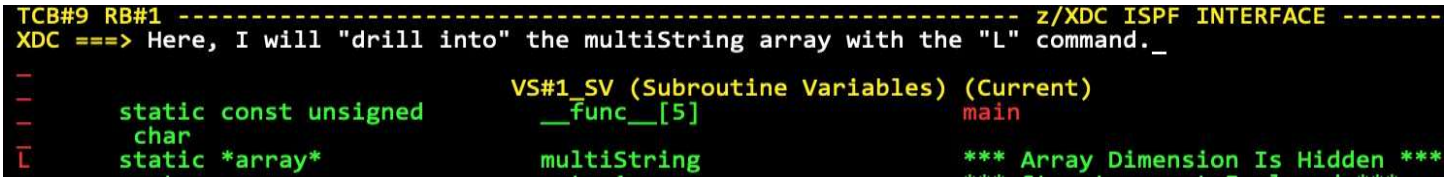
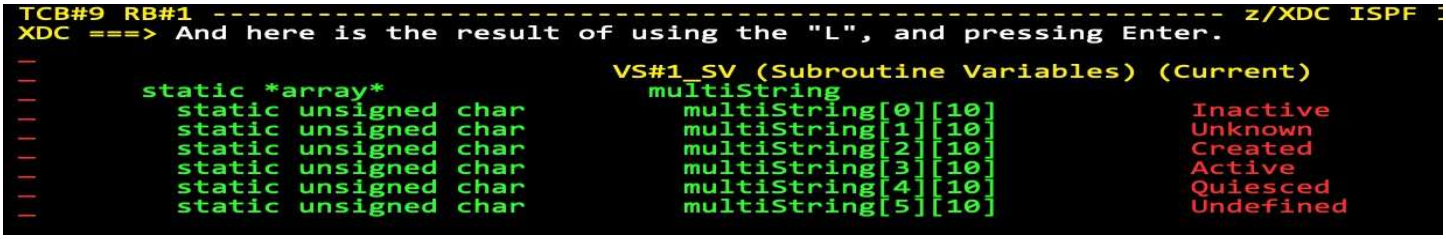


Figure 5-15 - Putting an L into the multiString Array Display

After pressing Enter, I see the effect of drilling into multiString:



5-16 - After Using the L Shortcut Command

Parsing the Internals of Arrays and Structures

c/XDC allows you to selectively view “slices” of arrays and structures. The basic syntax is:

LIST VARS <variable-name>[n][n] Shows discrete elements

or

LIST VARS <variable-name>[n:n][n:n] Shows a range of elements

or

LIST VARS <variable-name>[n,n,n][n,n,n] Shows specific elements within a range

The bracketed integers tell z/XDC what you want to see. However, you must count from zero, so to see the tenth of anything use 9. Frank had to correct me on this, so I’ll save you the trouble! Frank also told me that anyone who codes in C would immediately understand this, but I believe in being as clear as I can.

Let's try drilling into the multiString variable:

```
TCB#9 RB#1 ------------------------------------------------------------
XDC ===> These are examples of "slices" of variables.
   4 - list vars multiString[1][0]
   DBC303I New C Variables Discovered

_                                  multiString[1]
_       static unsigned char         multiString[1][0]          U
   5 - li vars multiString[1][0,1,2]

_                                  multiString[1]
_       static unsigned char         multiString[1][0]          U
_       static unsigned char         multiString[1][1]          n
_       static unsigned char         multiString[1][2]          k
   6 - li vars multiString[3][0:4]

_                                  multiString[3]
_       static unsigned char         multiString[3][0]          A
_       static unsigned char         multiString[3][1]          c
_       static unsigned char         multiString[3][2]          t
_       static unsigned char         multiString[3][3]          i
_       static unsigned char         multiString[3][4]          v
   7 - li vars multiString[3][4:0]
```

5-17 - Using LIST VARs agains multiString

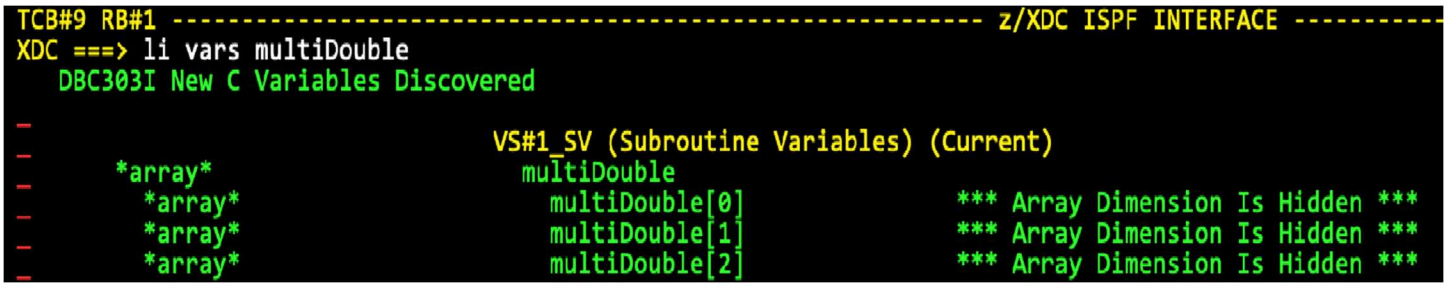
There are four List VAriables commands above. I got lazy and backed up in my session log to show them.

There's another interesting variable in multiDouble. This is the code from XDCCSAMP's source that builds this three-dimensional table. multiDouble is basically a three-dimensional "box" of numbers.

```
//a multi-dimension array of doubles
  double multiDouble[3][2][4] = {
        {
          {0.1, 1.2, 2.3, 3.4},
          {4.5, 5.6, 6.7, 7.8},
        } ,
        {
          {8.9, 9.10, 10.11, 11.12},
          {12.13, 13.14, 14.15, 15.16},
        } ,
        {
          {16.17, 17.18, 18.19, 19.20},
          {20.21, 21.22, 22.23, 23.24},
        }
  };
```

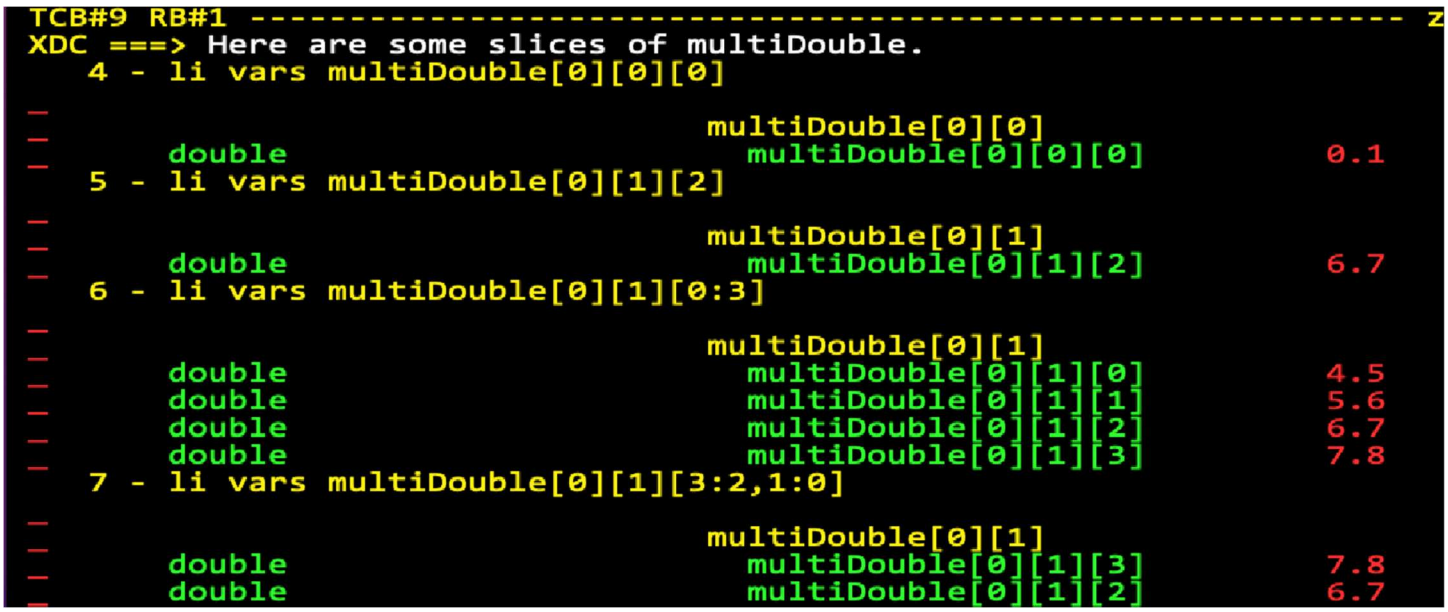
5-18 - Definition of multiDouble from the XDCCSAMP source code

Now we can use List Variables against multiDouble. The first command doesn't reveal very much.



5-19 - List Vars multiDouble

I'll show you some slices of multiDouble that may be more edifying.



5-20 - Slices of multiDouble

In order to give you the definitive resource, I've chosen to drop Dave Cole's own words in below, from Help Debugging C Variables.

Rules for Arrays - Displaying Selections of Individual Members

To show just a single member, specify the indices for that member. Example: For a 3-dimensional array named MYCUBE, use (for instance):

L V MYCUBE[12][7][3] (Bob's comment: Think in terms of 3D space using the X, Y and Z axes.)

You can use numeric variables instead of constants. Example:

L V MYCUBE[H][W][3]

Notes regarding variables:

- Non-integer values are floor'd down to the next lower integer.
- Character strings consisting of a numeric value are permitted.
- Pointer variables can be used as well.

To display a selection of members, use a comma delimited list of numbers (or variables) for one or more of the indexes. Example:

L V MYCUBE[1,7,4][3,2][3] This displays six members from MYCUBE.

To display a range of members, use a [number:number] syntax. Example:
L V MYCUBE[3:6][2:4][5]

This displays a 4x3 rectangular slice of MYCUBE. Note, specifying backwards ranges (such as [6:3]) is permitted.

All of the above can be combined. Example:
L V MYCUBE[3:N,9][2,W][5,10:8]

Managing the Display of Structures

To display a member within a structure, specify its name qualified by the structure's name. Examples:

L V MYSTRUCTURE.THISMEMBER

L V MYSTRUCTURE.INNERARRAY[3][4].INNERSTRUCTURE.MEMBER

To display an entire structure, start by referencing the structure's name. Examples:

L V MYSTRUCTURE

L V MYSTRUCTURE.INNERARRAY[3][4].INNERSTRUCTURE
A single line will be displayed with one of the following captions:

Structure not Explored
or
Structure is Hidden

Then use the L shortcut command to drill down one level into the structure. (You also use the L shortcut to climb back out of a drill-down).

Changing String Variables

At the time of this writing, we have disabled the ability to zap strings from the LIST VARS command output by using the Z shortcut command. You CAN use Z to change number variables, but not strings. Not yet, anyway. We intend to support zapping of the wide strings such as UTF16, wchar_t, char32_t, etc., but that's not implemented yet. For the time being, Displaying or Formatting storage containing strings and then zapping them is the way to go.

Let's issue LIST VARS against the variable "estring":

Figure 5-21 - Displaying a variable in storage

If I put a Z next to estring and attempt to overtype the value there, it won't work. But I CAN use Z if I use the Display or Format commands. So, I'll issue Display estring instead:

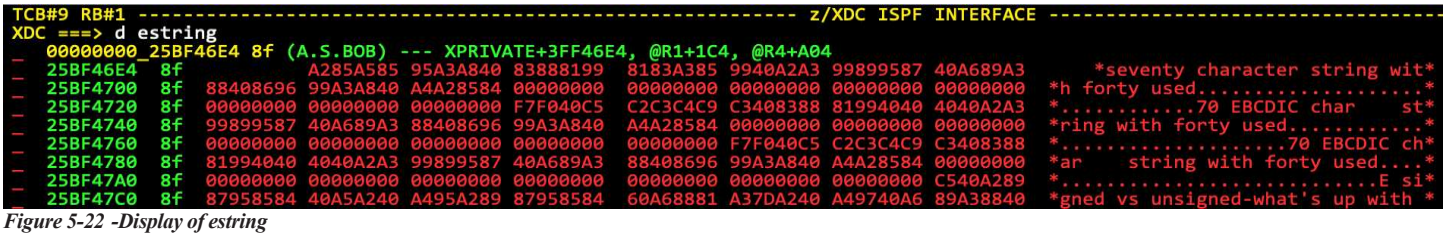


Figure 5-22 -Display of estring

The Display command above shows you the raw storage without any attempt at formating.

I can also issue a Format command against estring, thusly:

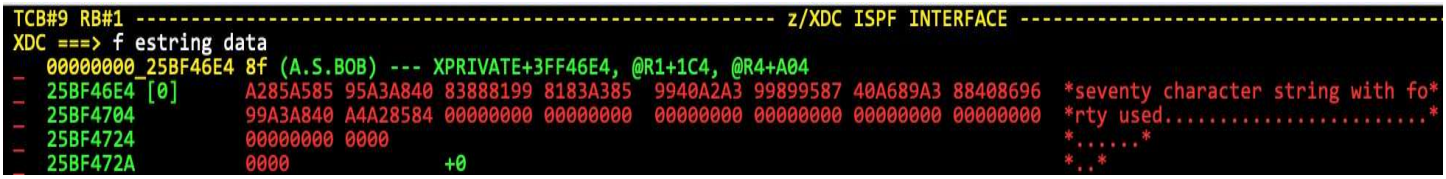


Figure 5-23- Format of estring

Above I've used the DATA parameter of the Format command. This overrides z/XDC's bias towards interpreting storage contents as machine instructions.

Either way that I view estring, I can now use the “Z” shortcut command to alter my string:

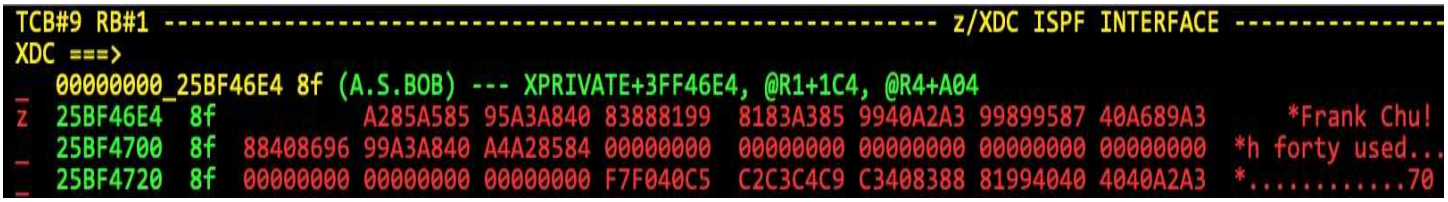


Figure 5-24 - Zapping estring

Now the estring variable contains “Frank Chu!” instead. Note that I don’t have to change the EBDCIC area at the right of the display. I could just have easily altered the hex storage there as well. Changing the EBCDIC area is just easier for me to understand because I don’t have convert to hex in my head. Nor could I.

The definitive resource can be found at Help COmmands Zap.

The LIST VSTACK Command

c/XDC gives you the ability to query and display all available stack frames and their associated variables (global, subroutine and local).

You can use the LIST VSTACK command to display these pools. c/XDC will display stack frames from oldest to newest showing resume addresses and information about the pools owned by the frame. You have the opportunity to put an L next to the entries so you may drill down into them and display the variables for each pool.

Here is the output of the LIST VSTACK command:

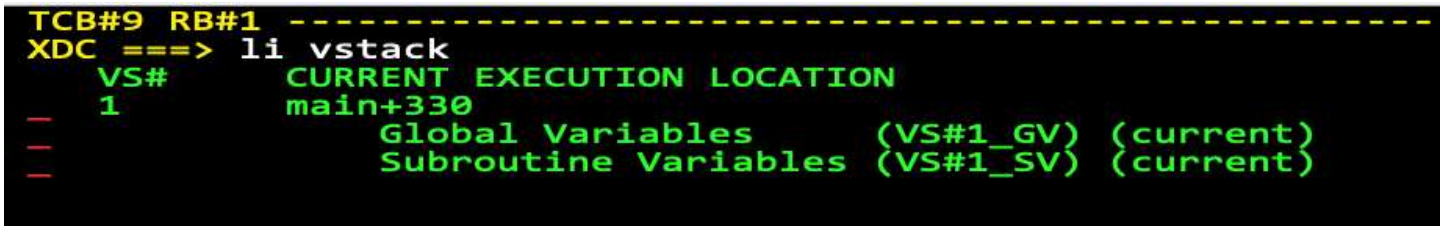


Figure 5-25 - Output of the List VSTACK command

c/XDC creates equates that name the various stacks and pools of variables. in our parlance, VS#1_GV would refer to Stack Number 1, Global Variables. Similarly, VS#1_SV would refer to the Stack Number 1, Subroutine Variables. Finally, VS#1_LV would refer to Stack Number 1, Local Variables. There can be any number of stacks with variable pools associated with them.

Note: If you have a programming environment where functions call functions call functions, then LIST VSTACK will give you a handy way of keeping track of where you are at any time. It's similar to z/XDC's LIST TASK command.

As you may expect, the definitive resource can be found at Help COmmands LISt VSTack.

Variable discovery is all very nice, but how do we get back to our program? That's simple. Issue the WHERE command. c/XDC will refresh the Working Window with your program, stopped at the next statement to be executed. If you prefer, you may type WHERE on the command line, but it's easier to use the abbreviation W. AND, if you're too lazy to hit "Home" to get your cursor onto the command line, you can use W as a shortcut command in any column 1 of the display.

z/XDC offers many one-character shortcut commands, and so far these are the ones we've used:

W	Performs "WHERE" command, showing you where your program is stopped.
?	Displays all the shortcut commands available in "this situation"
Q	Establishes a "default" module and csect (or Compilation Unit)
L	When used in the Working Window, will drill up or down. When used in a Watch Window, will expand or collapse a variable.
Z	A "zap" command. You may overtype raw storage with any value you like.
/	Displays the entirety of a character array (length termination).
\	Displays only that portion of a character array that contains information (null termination).
|	Display a character string in ASCII format
*	Display a character string in EBCDIC format

Those are only the few shortcut commands that we've used so far. There will be more in this book, and there are plenty more in z/XDC. Whenever you get curious about them, you can start up a z/XDC session and issue HELP SHORTCUTCOMMANDS for the complete reference.pro re

The LIST LKEDMAP Command

Many Program Objects are made up of many pieces. They can include Assembler code, C code, COBOL, FORTRAN - anything. You can have loads of compilation units inside your Program Object, all glued together by the Binder. c/XDC supports a LIST BINDER command to show you the anatomy of your Program Objects. Unfortunately, because we're dinosaurs, our LIST BINDER command is really named LIST LKEDMAP, because the ancestor of the modern Binder was the Linkage Editor.

The problem with the LIST LKEDMAP command is that it produces too much information for a C-centric programmer. It shows the prolog/epilog information that just isn't relevant, so Cole Software has introduced a CFRIENDLY parameter to the command

Below I will issue LIST LKEDMAP PSW? CFRIENDLY. This will cause z/XDC to show me the Binder map of XDCCSAMP.

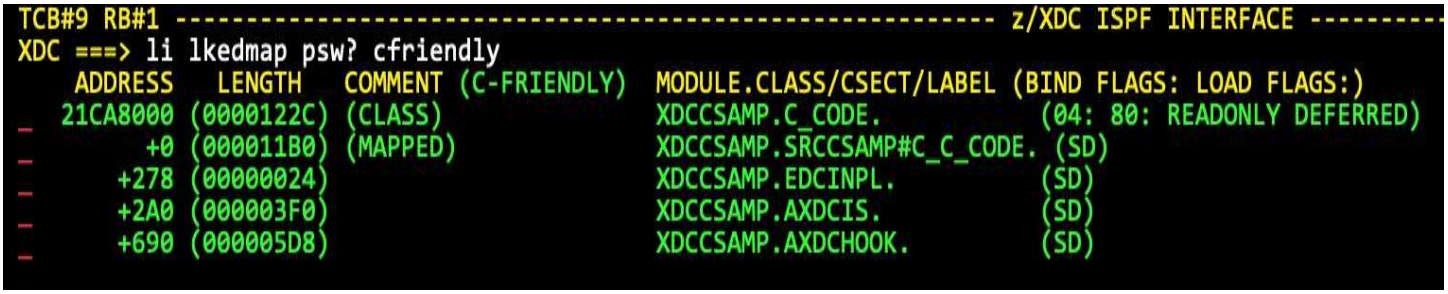


Figure 5-26 - Output of the LIST LKEDMAP command

If I was interested in debugging somewhere else in this Program Object, I could use F to format it, M to Map it, or even K to set a Hook. See Help COmmands LISt LKedmap.

The SCANLOG Command

In your work environment, you may end up with LOTS of functions and CSECTS in your Program Objects, and finding them by paging up and down can be tiresome. z/XDC allows you to search for strings of characters in its displays. To do this:

- Issue the FORMAT or DISPLAY command out to a certain distance. This puts the output of the command into your session log. Example: F <starting-address> <distance> or “F +0 1000”. You can use DISPLAY in this way as well.
- SCANLOG <string>

SCANLOG accepts SCAN as an abbreviation. You can specify the PREVIOUS parameter and z/XDC will search up in the session log. You can specify numeric parameters, and z/XDC will search within columns. An example would be SCAN cxdchook 12 35 and z/XDC would look for the string cxdchook between columns 12 and 35 in the session log.

Bet you can guess the name of the reference: See Help COmmands SCanlog for more information.

Stopping Execution of A Program - Breakpoints in c/XDC

Using STEP to follow execution is all very nice, but you may wish to skip ahead and stop your program further on. To do that you will use z/XDC’s breakpoint facilities. The basic procedure is to create a breakpoint(s) and then issue GO” to let the code execute to the breakpoint.

All z/XDC breakpoints insert a X’00’ (or a TRAP2 instruction) into the execution stream. When executed, the X’00’ will cause a S0C1 abend that z/XDC will capture.

The two simplest forms of z/XDC breakpoints are AT, which creates a permanent breakpoint, and T which creates a transient breakpoint.

- A Permanent Breakpoint (“AT”) has a shortcut command variant: A
- A Transient breakpoint has a shortcut command variant: T

What’s the difference? An AT breakpoint will persist until you explicitly turn it off with the OFF command (which has a shortcut of O). A Transient breakpoint is executed only once, and then is automatically removed. Each has it’s advantages, depending upon the situation.

If you set a breakpoint in a “wrong” place, such as the declaration of a variable, then c/XDC will actually place that breakpoint at the next executable statement.

Breakpoints and how to use them is a HUGE topic in z/XDC’s Online Help Subsystem. You can find all the sordid details at Help Breakpoints.

Permanent Breakpoints

Let's select a line of code in XDCCSAMP (Statement #265) and put an A next to it. This what it looks like:

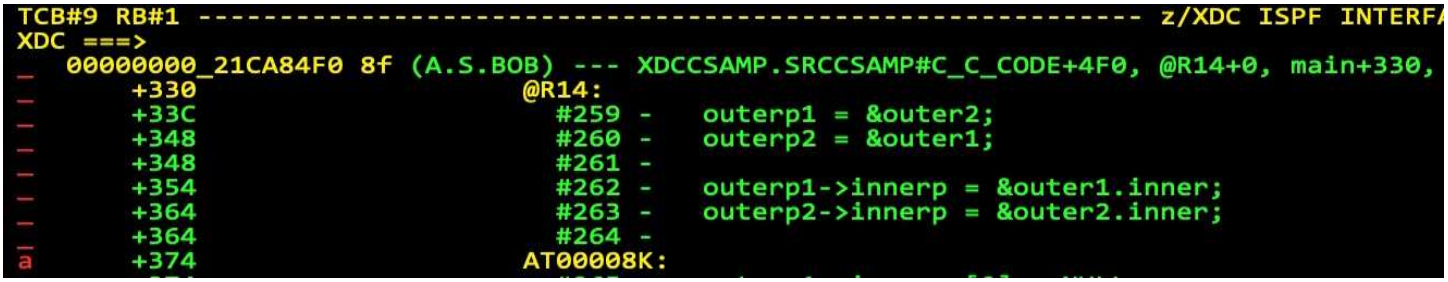


Figure 5-27 - Put an AT into the code stream

Now you can see the breakpoint. It has been assigned a unique name of AT00008K. The name consists of the breakpoint's type (AT), a 5-digit family ID (00008), and an alpha character that designates the member of the family (K). This naming convention allows you to manipulate breakpoints by type, family and individuals within a family.

Now, I'll enter GO. The GO command is how you resume execution of the program. And here we are, stopped at the breakpoint:

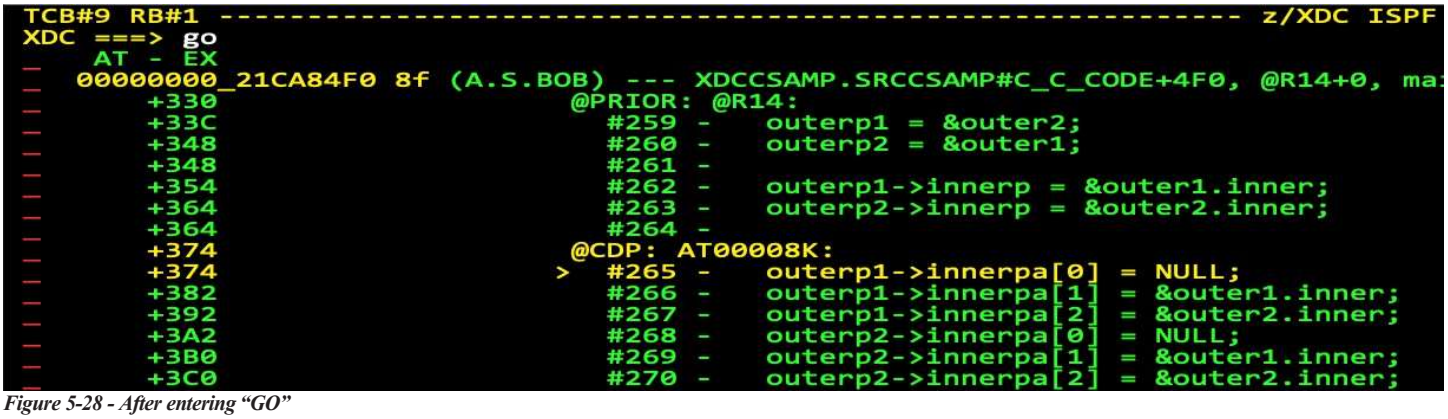
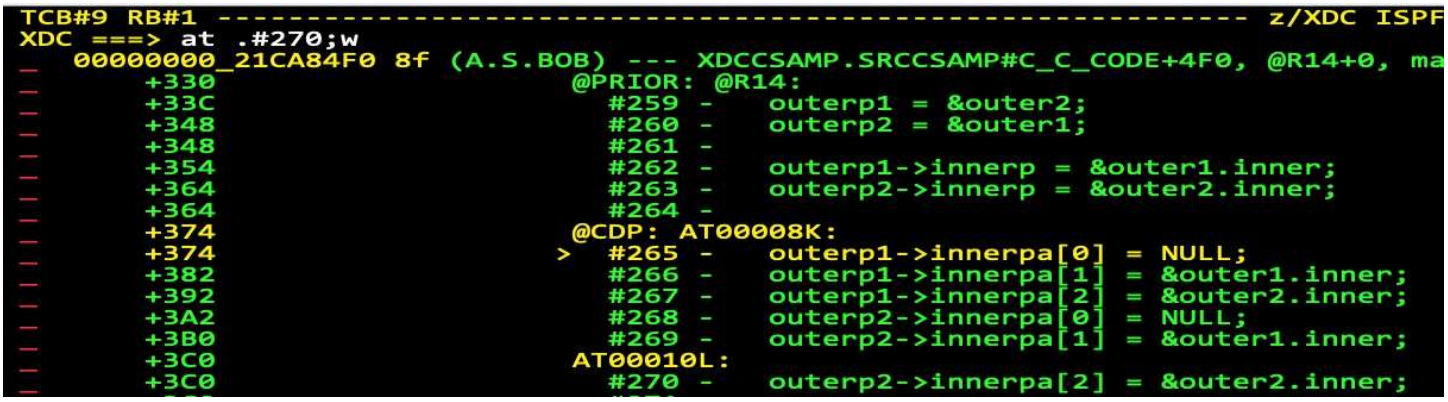


Figure 5-28 - After entering "GO"

You don't have to use an A shortcut command to create a breakpoint. You can also set them from the command line.

Since I previously used the Q command to set a default compilation unit for my session, I can issue AT .#statement number. Here I'm issuing AT .#270;W I'm stacking the AT command and a WHERE command to force z/XDC to repaint the screen (The semi-colon is z/XDC's command delimeter). z/XDC can't tell that the breakpoint is placed on this screen, so in normal operation it won't refresh the screen. The WHERE command forces it do do so.

Here's the display:



5-29 - Issuing AT From The Command Line, With A Stacked WHERE Command

Now we see the new breakpoint at .#270. It has its own unique name, "AT00010L".

The LIST BREAK command

It would be a handy thing to be able to see all the breakpoints that I create. The LIST BREAK command accomplishes that:

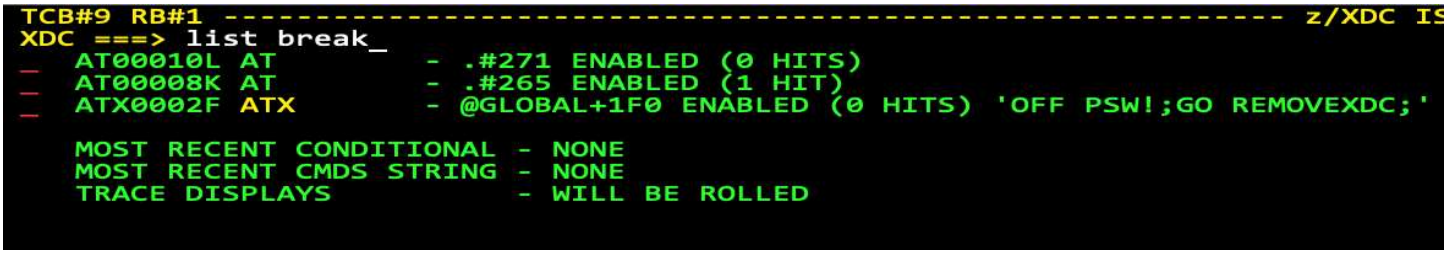


Figure 5-30 - Output Of The LIST BREAK Command

In the screen-shot above, you see both of the breakpoints that I've set manually.

The THIRD breakpoint with the ATX prefix is a persistent breakpoint AUTOSTEP creates upon initial entry to a C debugging session. You needn't be concerned about that. c/XDC will handle it for you transparently.

For the authoritative reference, see Help COmmands LISt BREak.

Transient Breakpoints

Transient breakpoints are one-time-only breakpoints. While a Permanent breakpoint (set with A as a shortcut command or with AT <statement-number> on the command line) sticks around until it's deleted, a Transient breakpoint is automatically deleted after being used once.

You can set a Transient breakpoint with the T shortcut command, or by entering T <.#statement number> from the command line.

A Transient breakpoint is also assigned a unique name, but with a TR prefix. c/XDC creates a family ID and a member name within that family.

So as to save us all some tedium, I'll issue a GO command to execute to AT00010L, and then I'll set a Transient breakpoint at statement .#273. Then I pressed Enter, and here's the result:

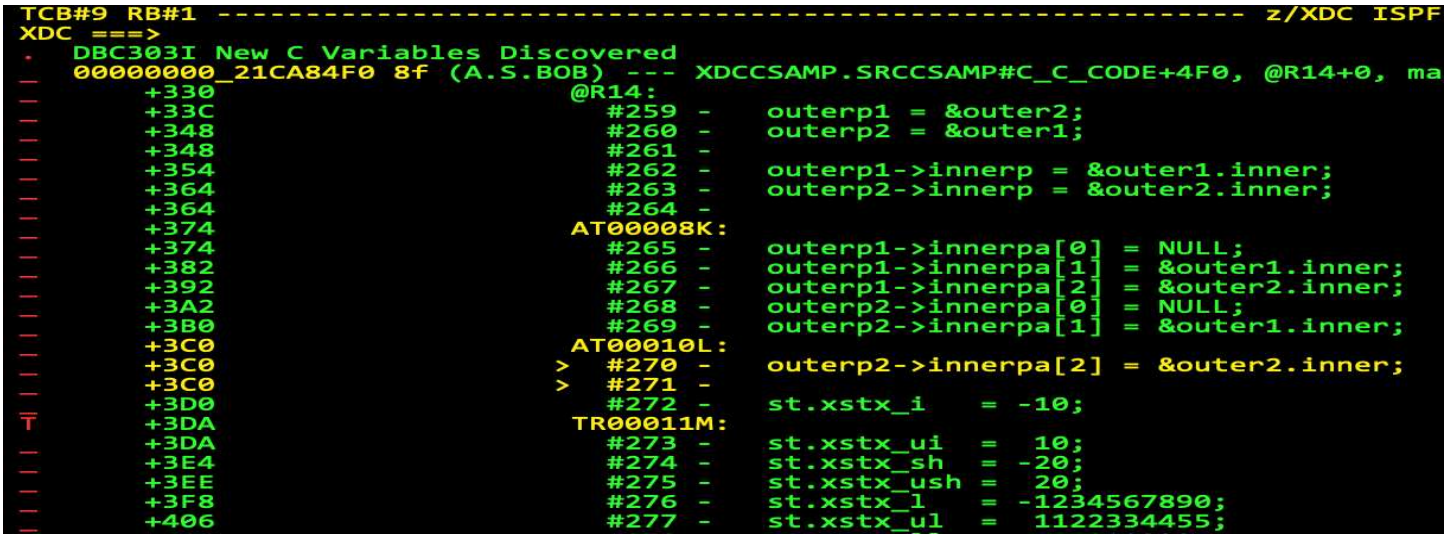


Figure 5-31 - After Setting a Transient Breakpoint

After you set a Transient breakpoint and enter GO, you'll get control at that statement and the breakpoint will be deleted.

You don't have to set only one breakpoint at a time. You can use any mixture of A and T anywhere on the panel. All breakpoints set in this way will be assigned to their types (AT or TR prefixes) and a family number relevant to each prefix, with individual member characters.

What's really cool about a family of Transient breakpoints is that if execution reaches ANY in that family then all the breakpoints in that family are also deleted! It's a way of capturing control at any likely outcome of a particular pice of logic. Rather than have me kill any more trees, or annoy more electrons with a tedious screenshot, just try it. It works.

Deferred Breakpoints

z/XDC also supports Deferred breakpoints of both the permanent and transient variety. They are useful in situations where the module you want to set a breakpoint in hasn't been loaded into storage yet. These breakpoints are beyond the scope of this booklet, but you can research them at Help COmmands ADeferred.

A couple of simple examples would be these:

AD <compilation-unit>.X#1 <== The keyword .X#1 is Cole-speak for physical start of module which may (or may not) be its entry point. This construct sets up a deferred breakpoint for the compilation-unit, and each and every time the program loads, you'll get a breakpoint there.

You could also issue AD <compilation-unit>+offset and c/XDC will set the breakpoint at some offset into the program.

TD <compilation-unit>.X#1 would set a transient deferred breakpoint. This will get control ONLY ONCE, the next time the program loads.

Note: For REENTRANT and REFRESHABLE programs, when loaded for the very first time into storage, an AD will trigger and set a breakpoint. Subsequent loads are processed by zOS using the already existing copy in storage. Because a new copy of the program is not loaded into storage, the AD will not trigger. However, if the program is deleted completely from storage and loaded again, the AD will trigger again.

For NON-RENT/NON-REFR programs, every single time it is loaded into storage, the AD will trigger. Every copy will have it's own breakpoint.

Deleting Breakpoints

You can use the OFF command to delete breakpoints you don't need any more.

Choices:

- From the command line you can issue OFF <breakpoint-name>
- You can use the O shortcut command next to any breakpoint in either the source statement view of your program or in the LIST BREAK command's output.

If you want to get rid of all breakpoints you can issue OFF, (or OFF ALL) from the command line.

Disabling/Enabling Breakpoints

You can also turn a breakpoint off temporarily by putting X shortcut command next to it. In this case the statement is restored, but the breakpoint definition is retained. All breakpoints that you X show up DISABLED in the LIST BREAK Command's output. You can enable them again with a subsequent X.

HOOK - the Uber-Breakpoint

Because z/XDC (and c/XDC) are Recovery Environments themselves, they can be supplanted or over-ridden by other, newer Recovery Routines. In this case, the debugger can lose control of the debugging session and your application will typically go down with a S0C1 abend.

If your program environment changes enclaves, it is almost certain that you'll change Recovery Environments. Then your breakpoints won't do you any good. c/XDC has lost control. Let's fix that.

If you're reasonably certain that another Recovery Environment will take effect, then you should set a Hook instead of a breakpoint.

Hooks come in two varieties: Dynamic Hooks and Static Hooks.

A **Dynamic Hook** may be set in three ways:

- With HOOK .#statement-number;
- With a K Shortcut command;
- With the HD <compilation-unit-name> plus offset <==Just like a deferred breakpoint, but it's a Hook instead.

A **Static Hook** is set with the cxdchook() function.

When you set a HOOK, c/XDC stores the statement you set the Hook at, and inserts a JLU instruction (a long jump).

- You can see your dynamic Hooks with a LIST HOOK command. Static Hooks won't be displayed.
- You can delete your Hooks with a DELETE HOOKS command, or DELETE HOOKS <hook-name> or by using the O"command next to any Hook on your source display.

Note: You can research the cxdhook() function in z/XDC's Help Facility by entering "Help DEbugging C CXDChook".

When execution reaches a dynamic Hook, z/XDC creates a new debugging session right here/right now. It automatically deletes the Hook and replaces the code that was originally there. Finally, it presents the code to you as if the statement you Hooked will be the next statement to execute. Static Hooks are NOT deleted. They're in place forever if they've been compiled into your Program Object.

After you've set an initial Hook, you can use breakpoints or perform tracing operations - at least until you change enclaves or Recovery Environments again. Then you'd execute another Hook (probably a dynamic one), and keep on trapping or tracing.

How To Set A Static Hook In Your C Program

Here is a snippet of code from our "toy" program XDCCSAMP, at statements .#250 and .#251:

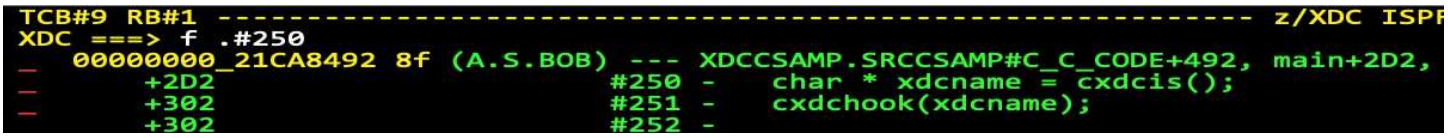


Figure 5-32- The cxdchook() function in XDCCSAMP

Statement .#250 deserves some explanation. z/XDC allows your site to run multiple versions of z/XDC simultaneously. We refer to these different versions as "clones". So, for release z2.2 of our product the clone name could be Z22. For release z2.1 of our product the clone name might be Z21. If we change the clone name, say to XXX, then the product will use XXXCALL instead of "XDCCALL".

This allows the installing systems programmer to perform tests on one version of the product while the user community is accessing another. Most of the time, however, your production version of the z/XDC product's clone name will be simply XDC. Now if Dave refers to a "clone name" in his documentation, you will know what he means

Frank says:

“cxdcis() is used along with a special ddname //XDCISxxx DD DUMMY to control at runtime what clone will be called by cxdchook(). So if you define a //XDCISZ22 DD DUMMY, cxdcis() will return “Z22”, which can then be passed along to cxdchook() to trigger. Typical usage would be cxdchook(cxdcis());.

Or you can assign it to a variable if you care to:

char xdcCloneName = cxdcis();
cxdchook(xdcCloneName);”

So, statement .#250 is our way of accomodating local changes to z/XDC’s current clone name. If you haven’t changed clone names then the example in statement .#250 will be unchanged in your code.

Statement .#251 invokes the static hook. When execution reaches this statement, you’ll get a whole new debugging session *inside* the current one.

You use Hook in different ways depending on what you’re trying to do. For large transaction processing systems, it’s probably better to make the source code change, and insert a static Hook. For other situations a dynamic Hook is the way to go. If you’re ever unsure, please ask us.

Anatomy of a Static Hook, and the Libraries You’ll Need

Let’s drill down. The following information has been lifted from Help Debugging C CXDCHOOK.

The cxdchook() function has two parts:

1. CXDCHOOK is a C language header file that you need to include into your C program. It can be found in the CSW.XDCZ22.XDCSAMP sample library. Ask your Systems Programmer for the library’s actual name at your Data Center.
2. AXDCHOOK is a load module that is located in the CSW.XDCZ22.XDCLINK load library. Again, you’ll have to ask your Systems Programmer for the libarry’s actual name at your Data Center.

The source for AXDCHOOK can be found in CSW.XDCZ22.XDCASM(SRCAHOOK).

Basically the cxdchook() function invokes the #XDCHOOK macro which creates the debugging environment and passes control to z/XDC and to you. See Help HOoks Static 3xdchook for more information.

cxdchook Compile Time Requirements

In order to call cxdchook(), you need to make some changes to your C program.

You need to add LSEARCH(//’CSW.XDCZ22.XDCCSAMP’) to your compile time options file. See CSW.XDCZ22.XDCJCL(CMPCSAMP) for sample JCL.

You will need to include the cxdchook header file into your program. The following statement will do the trick: #include cxdchook.h

Then at the point in your code where you want to start a debugging session, you need to call cxdchook(). Use this statement: cxdchook(XDC). When execution passes to the function, an XDC session will be started.

If you are using a clone of XDC (for example Z22), then the statement you need to use is cxdchook("Z22").

It is possible to run different levels of z/XDC at any site, by changing the clone name. Normally, your site will be running the production version of our product under the standard clone name of XDC So, you probably won't have to invoke the cxdchook function with anything other than cxdhook() or perhaps cxdchook("XDC").

cxdchook Bind Time Requirements

The cxdchook function's code is contained in a load module named AXDCHOOK which is located in CSW.XDCZ22.XDCLINK. In order for the Binder to find the AXDCHOOK load module you need to:

- Add CSW.XDCZ22.XDCLINK to your //SYSLIB concatenation.
- Add an INCLUDE statement to the Binder's //SYSLIN input file.
  Example: INCLUDE SYSLIB(AXDCHOOK).

In CSW.XDCZ22.XDCLINK you will find several modules for using cxdchook() in the 31-and 64-bit environments:

- AXDCHOK6 ==> 64 bit version of cxdchook()
- AXDCHOOK ==> 31 bit version of cxdchook()
- AXDCIS64 ==> 64 bit version of cxdcis()
- AXDCIS ==> 31 bit version of cxdcis()

Let me remind you of the cxdcis() statement that you saw in the screenshot above. That's how to express a "clone name" to c/XDC. So, when working in a 64-bit environment you must use AXDCIS64, and for 31-bit environments you mus use AXDCIS.

I really want you to be able to ignore clone names, because unless you're running multiple versions of our product (unlikely) then the clone name will always be XDC.

Be careful with Hook

Depending upon system security rules, z/XDC may permit you to set a Hook in Common Storage. ***The people who use our product in this way must know exactly what they are doing***, because carelessly setting a Hook where it doesn't belong can create a debugging session where it doesn't belong. That might be a Very Bad Thing.

In fact, you should be careful with z/XDC in general. If security rules permit it, the user can also zap storage outside the Private Area in Common Storage. That too can either be a Good or a Very Bad Thing.

The good news is that every time you try to do something interesting with z/XDC a SAF call is made

to your security system. If there is an appropriate rule in effect there is NO security exposure, but that depends entirely upon your Security Administrator. z/XDC is a power tool, like a chain-saw or a drill. Be careful with power tools.

Hook is a powerful instrument. Remember, Hook is the uber-breakpoint. It WILL create a debugging session wherever you set it. If I've just scared you a bit, that was on purpose.

The definitive resource on Hooks can be found at Help COmmands HOok. I strongly urge you to read this and understand it.

A Cumulative Review of Short-cut Commands

Let's do another short review of the Shortcut commands, adding in the new ones I've just talked about:

A	Performs an "AT" command, placing a permanent breakpoint at the desired line of code.
T	Performs a "TRAP" command, placing a transient (one time only) breakpoint at the desired line of code.
F	Performs a "FORMAT" showing you source statements at a specific address.
L	When used in the Working Window, will drill up or down. When used in a Watch Window, will expand or collapse a variable.
O	Performs an "OFF" command, deleting a breakpoint definition.
Q	Establishes a default module and csect (Compilation Unit).
W	Performs "WHERE" command, showing you where your program is stopped.
Z	A "ZAP" command. Tab to the field you want to change, and over-type it.
?	Display all the shortcut commands available in "this situation"
/	Displays the entirety of a character array - even the stuff you don't care about (length termination).
\	Displays a only the useful portion of a character array (null termination).
|	Display a character string in ASCII format
*	Display a character string in EBCDIC format

Again, you can get the entire story on Shortcut commands by entering Help SHORTCUTCOMMANDS.

The LIBRARYLISTS Command

As of April 2021 the LIBRARYLISTS command may NOT be necessary. In earlier editions of the Metal C compiler, IBM did NOT imbed the location of the corresponding DWARF file, and Cole Software had to build a work-around. That work-around is the LIBRARYLISTS command, which is rather complex to understand and use.

That all went away!

Now, Metal C programs when compiled with DWARF on z/OS 2.3 (with APAR PI99202 installed) and z/OS 2.4 and later, now embeds the location of the DWARF file. This allows c/XDC to automatically load in the source image for CSECTs that have DWARF without the need of setting up LIBRARYLIST REDIRECTS. In order to exploit this new feature, z/XDC must be at maintenance level MDL-2104C or higher.

LIBRARYLISTS is a complex command, and I REALLY hope you can avoid ever using it. Frank says,"'If I am forced to use the LIBRARYLISTS command, I often create a z/XDC Script to do it." Even Frank avoids LIBRARYLISTS. Hint: See HELP SCRIPTS to learn more about scripting z/XDC's command language.

If you ever find yourself in a situation where you MUST use the LIBRARYLISTS command, here is an orientation.

The LIBRARYLISTS command establishes a linkage between your compilation unit and its source (or DWARF) files. You will need to use this command in four situations:

- If you have moved or renamed your source code since the compilation step
- **If you are compiling under IBM's Metal C (and do not meet the requirements in the beginning of this section)**
- If you moved or renamed the DWARF files after compilation
- The code was compiled using inline source which was contained in the JCL.

As mentioned above, c/XDC uses DWARF data in order to show your C source code. In the case of XL C/C++ c/XDC can automatically find the source files using the DWARF data resident in the program object - IF you haven't moved or renamed them. That's the magic of AUTOMAP in the default Profiles.

In the case of Metal C the route to a finished Program Object is a two-step process:

1. First, the compilation produces Assembler source, not compiled C code.
2. Then, in a subsequent step, the Assembler source code is Assembled via CDAHLASM (a modified form of the Assembler).

During the Assembly step, SYSDWARF DD card is specified, so the DWARF data is created. However, the location of the DWARF data is not stored in the resulting program object. Finally, the Binding process is performed and then you have a Program Object to execute.

In the z/XDC Assembler debugging tool - our base product - there is a SET MAPLIBS command that points z/XDC to your Assembler source code. LIBRARYLISTS is the C language equivalent.

Unfortunately, because we have to provide for some insanely complex programming environments, LIBRARYLISTS is itself insanely complex.

The basic concept is easy to grasp: You're telling c/XDC that this DWARF file now resides in that DWARF library, and this source file now resides in that source library.

You can use LIBRARYLISTS to perform 8 separate functions.

1. ADD <== Add new lists of libraries to an existing list, or reorder the list;
2. CHECKPOINT (or CKPT) <== Saves all libraries into z/XDC's storage
3. DELETE <== Delete lists or patterns of names from LIBRARYLISTS
4. DEFAULT <== identify the default list, if you want to establish one;
5. REDIRECT <== Cause z/XDC to look "here" for source or DWARF library datasets;
6. RESET <== Clear one or all LIBRARYLISTS definitions;
7. SHOW <== Display all LIBRARYLISTS definitions;
8. TEST <== Test pattern and redirection settings;
9. PIZZA <== Forget about LIBRARYLISTS and go out for lunch (just kidding!).

The good news is that you can build a z/XDC script containing your LIBRARYLISTS command, and issue z/XDC's READ command against it. That's what Frank has done, and he has graciously offered to share one of his scripts with you, below:

Figure 5-33 - Frank Chu's sample LIBRARYLISTS script

Frank is a careful guy. First, he deletes all his maps. Then, he issues a LIBRARYLISTS RESET ALL to clear pre-existing LIBRARYLISTS definitions.

The real substance of the script is the LIBRARYLISTS REDIRECT command, which tells c/XDC that the DWARF data for a Metal C csect called SRCCAMP ("PAT='-METALC-CSECT(SRCCSAMP)') can be found in USE=DBCOLE.XDCZ22.XDCDWARD(XDCMSAMP).

Finally, Frank issues a MAP PSW? to cause to process the DWARF file and map the csect using the names of the source files contained in the DWARF. Simple enough, right? Well, in this case it is. Let's drill down a tiny bit.

The PAT= parameter allows you to specify datasets using wild-card characters ("*", "**" and "?") to point to your source files, allowing you to slice-and-dice every which way. Then, LIBRARYLISTS can create lists of datasets based upon these patterns. For very complex programming environments you can create lists of lists based upon patterns that you set. There's no way I can clearly explain all this to you, the new user.

Frank says:

"It depends on if you are redirecting source or dwarf. What PATTERN contains is the original path/ dataset name of the dwarf file that was created or the source file used at compile time.

For DWARF redirects under Metal C, the pattern is a special construct: -METALC-CSECT(. ...), where you specify the CSECT name between the parentheses.

There's a special construct for source files: -INLINE-CSECT(...), where you specify the csect name in between the paranthesis and the use points to a copy of the original source."

Here is more wisdom from Frank. please bear in mind that the actual dataset names Frank uses WILL NOT be found in our libraries. It's an example.

"The LIBRARYLIST command can be a bit hard to understand at first. Basically, you are defining a series of text patterns and an associated set of datasets containing DWARF or source files. These patterns are checked to resolve the location of dwarf and source files during C mapping."

Let's presume that you've moved or renamed your DWARF and SOURCE files since the compile operation. The simplest LIBRARYLIST REDIRECT is to create a one-to-one relationship between the old location information and the new:

```
LIBR DWARF RED PAT=CSW.Z22.DWARF(XDCCSAMP) USE=SYS2.COLE.XDC22B.DWARF(XDCCSAMP)
```

With this REDIRECT set, when c/XDC tries to open CSW.Z22.DWARF(XDCCSAMP) for DWARF data, LIBRARYLIST will instead prompt c/XDC to open SYS2.COLE.XDC22B.DWARF(XDCCSAMP) instead.

You can also use wildcards in the PATTERN and symbolic substitution in the USE list. For example:

```
LIBR DWARF RED PAT=CSW.Z22.DWARF(*) USE=SYS2.COLE.XDC22B.DWARF(&1)
```

This REDIRECT causes any attempt to open a member in CSW.Z22.DWARF to look for the member in the SYS2.COLE.XDCZ22B.DWARF library instead.

Expanding on this even more, you can establish a set of "candidate lists" that can be set as the USE target. This is useful if you have a set of set of datasets that contains dwarf and source files for specific programs/projects.

LISTA:

#1: SYS2.COLE.XDC22A.DWARF(&1)

#2: SYS2.COLE.XDC22B.DWARF(&1)

#3: SYS2.COLE.XDC22C.DWARF(&1)

#4: SYS2.COLE.XDC22D.DWARF(&1)

LIBR DWARF RED PAT="CSW.Z22.DWARF(*)" USE="LISTA"

In this example, LISTA names four libraries that contain DWARF data. The REDIRECT instructs c/XDC to search through LISTA looking for the first matching dwarf member.

All of the examples are for DWARF but they apply for CSOURCE as well.

Finally, because there is no DWARF location recorded for Metal C programs, you have to specify a

keyword (-METALC-CSECT”) as part of the pattern so that c/XDC understands that it is looking at a Metal C program.

The following Metal C example references our distributed XDCMSAMP program, which you can compile and bind using the following JCL files:

CSW.XDCZ22.XDCJCL(ASMMSAMP) <-- compile, assemble step

CSW.XDCZ22.XDCJCL(LMSAMP) <-- bind step.

For the XDCMSAMP program, you will need to set up the following redirect:

```
LIBR REDIRECT DWARF PAT=-METALC-CSECT(SRCCSAMP) USE=SYS2.COLE.XDC22B.
XDCDWARF(XDCMSAMP)
```

Again, “-METALC-CSECT” is the special construct required for Metal C mapping. Specify the name of the CSECT you wish to map as part of the construct.

You can also use wildcards here as well:

```
LIBR REDIRECT DWARF PAT=-METALC-CSECT(*) USE=SYS2.COLE.XDC22B.XDCDWARF(&1)
```

IBM does not provide the ability to verify that the DWARF data c/XDC reads is really for the CSECT it is trying to map. So, with LIBRARYLIST you can easily “lie” to c/XDC and it will happily display incorrect variable data and source images.

If you do use LIBRARYLIST and c/XDC displays something unexpected, chances are the that wrong DWARF or source file was read in the mapping attempt. Look through the mapping messages generated by c/XDC to see what DWARF/source file it actually read and adjust the REDIRECT to point to the correct members.

If you are compiling LE C/C++ or Metal C using source statements inlined as part of the JCL, you will need to establish CSOURCE redirects using -INLINE-CSECT(..) as the pattern. You will need to save copy the inlined source into a dataset on disk and point the redirect to it.”

Using LIBRARYLISTS with Systems/C to create your Maps

This is all from Frank:

To map your Dignus compiled program, follow the Metal C steps for c/XDC. Specifically, you will need to define LIBRARYLIST REDIRECTS to associate a CSECT to a source file so that c/XDC can read in the source images. If you used the //SYSDBG ddcard with PLINK or if you cross compile on Windows/Linux, you will also need to provide a LIBRARYLIST REDIRECT to associate the CSECT to a dwarf file.

For example:

LIBR DWARF REDIRECT PAT=-INLINE-CSECT(csect-name) USE=some.DWARF.library(dwarf-member-name)

LIBR CSOURCE REDIRECT PAT=-INLINE-CSECT(csect-name) USE=some.SOURCE.library(source-member-name)

MAP module.csect-name

Cross compiling on Windows and Linux can generate some absurdly long path names for source and DWARF files.

For example, this an actual LIBR REDIRECT that I use for testing:

LIBR CSOURCE REDIRECT PAT="C:\MyFiles\Projects\Dignus\jsmn\example/C:\MyFiles\Projects\Dignus\jsmn\jsmn.h" USE=FRANK.DIGNUS.INCLUDE(JSMN)

Using wildcards in LIBRARY REDIRECTS is highly recommended.

The previous REDIRECT can be shorted to just:

LIBR CSOURCE REDIRECT PAT="\jsmn.h" USE=FRANK.DIGNUS.INCLUDE(JSMN@H)**

Note: Here is how to access the "formal" treatment on the LIBRARYLISTS command in our on-line Help Subsystem: Start a debugging session and enter HELP DEBUGGING C LIBRARYLISTS. Be prepared to spend some time on it. Then, if you still want assistance, please call us. That's what we're here for.

You can read about the Librarylists command itself at Help Commands LIBrarylists.

The MAP Command - A Handy Alternative to the Librarylists Command

Now that you've spent some time learning Librarylists, I have good news: You have an alternative to using it.

Our base product z/XDC includes a MAP command. Basically, MAP searches for ADATA or C Source and loads it into storage so you can SEE your source statements. As a C-centric programmer you may not need to use the MAP command because of z/XDC's AUTOMAP feature, which is very handy for everyone.

Recently the MAP command has been updated, and you Metal C users may like this. Instead of using the LIBRARYLIST command you can issue:

```
MAP <address-expression> DWARFLIB=<your-dwarf-dataset> SOURCELIB=<your-source-dataset>
```

In the case of XDCMSAMP, which is our Metal C variant of the SRCCSAMP program, I can start a session debugging XDCMSAMP and issue this:

```
map psw? dwarflib=cswmap psw? .xdcz22.xdcdwarf(xdcmsamp) sourcelib=dbcole.xdcz22.xdcsamp(srccsamp)
```

Above, I’m using the address-expression “psw?” which points to the current program. It works!

Here's an example of this command in the context of XDCMSAMP, our Metal C program:

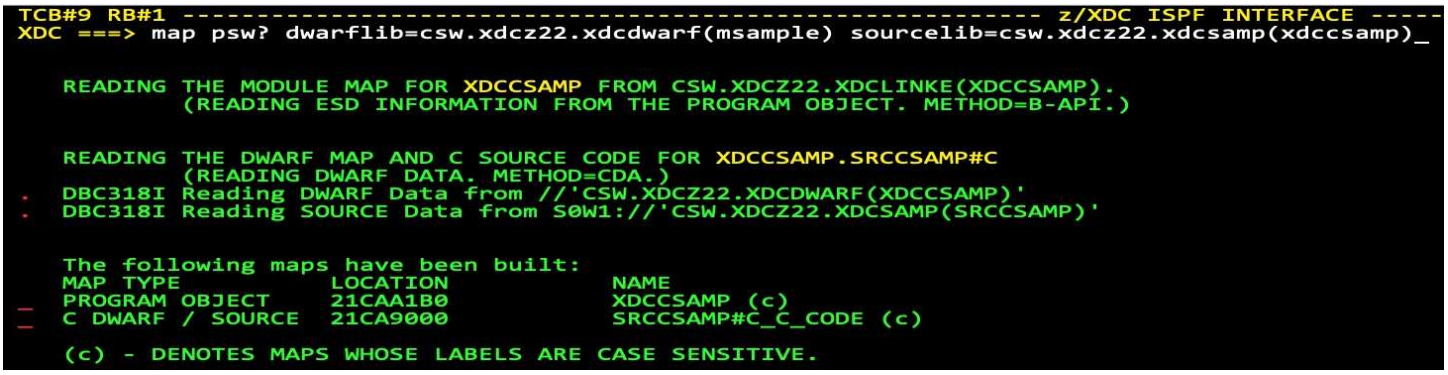


Figure 5-34 - An example of the MAP command's recent new parameters.

MAP with the DWARFLIB and SOURCELIB parameters is another way of telling c/XDC where to find your DWARF and SOURCE datasets, nicely by-passing the need for LIBRARYLISTS. Check it out.

How to end a c/XDC debugging session

Ending a c/XDC or z/XDC session is easy. You just use the END command. If you're debugging in your TSO Private Area (by way of the startup panel), then issue END COMPLETELY. This shuts your c/XDC session down. However, LE doesn't always "play nice" with c/XDC, and you may see an abend on your screen. You may ignore this.

Here's Frank's explanation:

"When an enclave comes up, LE runs through its initialization processes and sets itself up as the newest recovery handler on the SCB chain. In order for c/XDC to work, it must be the newest handler on the chain, so underneath the covers it hits a HOOK on the first executable C statement in main(). It's the magic of AUTOSTEP.

On enclave termination, LE does its cleanup. It FREEMAINs the recovery in storage and removes the new entry on the SCB chain. Without XDC in place, this works fine. But because we added ourselves to the SCB chain, LE removes the wrong entry and a S0C4 pops when it's XDC's time to clean up. We get around that by placing a command type breakpoint before LE's termination, and a GO REMOVEXDC is issued to remove ourselves from the picture. AUTOSTEP sets up the breakpoint as well.

However, if you start the debugging session using a HOOK of some kind, then you must manually issue GO REMOVEXDC to remove XDC's SCB from the recovery environment. Otherwise LE will abend with a S0C4 during it's cleanup."

Thanks, Frank!

When debugging in batch, I tell folks that if they want to separate their TSO session from a batch job they are debugging under z/XDC, they can issue the command pair DISC;GO.

With c/XDC, and if you entered the session using a dynamic hook it would be better to issue DISC;GO REMOVEXDC.

Frank says:

“When in batch (and TSO), using XDCCALL with AUTOSTEP, we will do our own clean up.”

For the definitive reference on End and Go, see Help COmmands ENd, and Help Commands GO, respectively.

How to learn more about z/XDC and c/XDC

Folks, that is a VERY abbreviated overview of c/XDC and its major features. So abbreviated, in fact, that I strongly recommend that you investigate the z/XDC Primer as well. That book has more to do with Assembler language, but it gets into a lot of the detail that I didn’t want to duplicate here. So, your next learning goal might be to read the z/XDC Primer.

Beyond that, z/XDC’s internal documentation is the source for all knowledge regarding this remarkably powerful product. You can start learning about z/XDC by entering Help DEbugging. For C-specific information you can enter Help DEbugging C. It’s all in there.

There are also several training videos available to you on Youtube. Dave Cole spent weeks preparing training videos that you may find valuable. Please check out: http://www.colesoft.com/training-videos/

There is yet another resource available to you: “The REAL z/XDC Quick Reference”. This is a .pdf document that contains a lot of the basic concepts in z/XDC and c/XDC as well as an alphabetic list of the most important commands complete with descriptions and diagrams. if you really want to drill down deep into z/XDC please see “The z./XDC Railroad Track Reference”.

Of course the very best resource is Cole Software. We’re always happy to help you, and if you want some assistance you can call us. I’m at (800) XDC-5150. The guys in engineering can be reached at support@colesoft.com. Frank, Dave and our president Calin all watch this email address.

With all these resources available, I felt it unnecessary to include all of the C-specific commands and their diagrams in what is supposed to be a “How to get started” document. However, it would be a good idea for me to list all the C-specific commands that we’ve introduced for z/XDC. They appear on the next page.

A Table of c/XDC-specific Commands

c/XDC delivers new commands and functionality. Also, MANY of z/XDC's existing commands have been altered to support c/XDC. Here are the new C-specific commands that are introduced for z/XDC Release z2.2:

cxdchook()	Creates a static hook (like the #XDCHOOK macro for Assembler) in your C Code. Consider this as the "uber-breakpoint", the ability to create a c/XDC debugging session anytime or anywhere. See Help Debugging C CXDCHook for more information.
STEP	Allows you to execute one C/C++ or Metal C statement. You can step INTO a subroutine, OUT of a subroutine and OVER a subroutine.
LIBRARYLISTS	Allows you to tell c/XDC where source statement libraries are. It's a rather large construct, deserving some study.
LIST LKEDMAP	Shows entry points in your compilation unit. Use the CFRIENDLY parameter.
List VSTACK	Allows you to review the contents of the Variable Stack Pools (AKA, "Language Environment Stack Frames").
List VARIABLES	Allows you to view variables individually or in groups. All array or structure architectures are supported.
List VSETTINGS	Allows you to configure c/XDC's displays of array dimensions and stack depths. Normally, you won't have to mess with this.
AUTOMAP	c/XDC automatically shows you your source listing - if c/XDC can find it!
List PARSEORDER	C variables and Assembler variables differ a great deal. This is how you tell z/XDC which order in which to parse your variables. Normally, you won't have to mess with this.
LIST SIZEOF	Displays variable names and the amount of storage they consume.
List STEP	Query how c/XDC interacts with subroutine calls or functions. Normally, you won't have to mess with this.
List CXDC	Detect whether (or not) the c/XDC Feature is available to your debugging session.
List VDISPLAY	Allows you to configure c/XDC's variable display. Normally, you won't have to mess with this.
Set STEP	Control how the STEP command interacts with subroutine calls or functions. Leave the settings as they are unless you feel compelled to change them.
Set VDISPLAY	Configure the output of the List VARIABLES command. Leave this alone, too.
Set VSETTINGS	Alter c/XDC's array dimensions and structure depths. These values are already set to maximums, so you probably won't have to mess with them.
ZAP	After displaying a variable, use the Z shortcut command to alter them.

All of these commands deserve your study, and they can easily be found in z/XDC's Help Subsystem by entering "Help COmmands <command-name>".

Conclusion

Folks, that's all that is new and wonderful with the c/XDC product. As the product grows and evolves, we'll add to this Primer. For now, you should have enough understanding to navigate and use c/XDC.