

An Introduction to XDB, the MPE/iX Symbolic Debugger

Bruce Toback
OPT, Inc.
(602) 996-8601

I. Introduction

While in an ideal world, all programs would work properly immediately upon entry and compilation, this is a rare occurrence for most programmers. Programmers have devised various strategies for diagnosing and fixing program logic errors, including hardware logic analyzers, software debuggers, program progress messages, integrity checks and program assertions, running the program "by hand" and explaining the program logic to a sympathetic co-worker, spouse or pet in the hope that exposition might somehow lead to enlightenment. This paper discusses the use of HP's Symbolic Debugger, also called XDB, for finding program logic errors.

XDB is an interactive program which allows the programmer to exercise very close control over the execution of a program to be diagnosed. The programmer is permitted to execute a target program one procedure, statement or machine instruction at a time, and in the interim to examine and change the values of variables or to change the program flow.

Unlike the System Debugger, XDB requires its object code to be specially prepared. This makes it more suitable as a development tool than as a diagnostic tool. In addition, XDB is an extra-cost option, so it may not be available on a production system if development and production tasks are split between systems.

The purpose of this paper is not to recapitulate the XDB manual. Rather, it is to provide a beginner's guide to use in lieu of the manual, and to give some flavor of XDB's capabilities. The manual should be consulted for detailed information about XDB commands. The last section of the paper gives several techniques that are not obvious from the manual, but which can provide enormous convenience when debugging complex systems.

II. Preparing A Program For Symbolic Debugging

XDB needs considerable information about the program being debugged in order to operate. All MPE/iX compilers except RPG are capable of generating this information and storing it in the object file together with normal object code. Since the debugging information takes up extra space in the program file, and in particular since XDB cannot be used on optimized programs, this step is an optional part of compilation. Compiling a program with the symbolic debugging option does not require that the program be run under the debugger.

COBOL. The COBOL compiler uses a compiler option statement to turn on symbolic debugging. The statement

```
$SYMDEBUG `XDB`
```

instructs the compiler to begin emitting symbolic debugging information in the format required by XDB. (The compiler can also emit other kinds of symbolic debugging information.)

Pascal. Pascal also uses a compiler control command of the form:

```
$SYMDEBUG `XDB`$
```

to activate symbolic debugging. As with COBOL, the option must be at the front of the program.

FORTRAN. The FORTRAN 77 compiler uses a compiler control statement of the form

```
$SYMDEBUG XDB ON
```

to control symbolic debugging. The FORTRAN compiler allows the directive to appear between program units in order to control symbolic debugging information on a unit-by-unit basis. This can be useful when debugging programs that benefit greatly from optimization: by turning off symbolic debugging for those units that require optimization, program execution times can be kept to a minimum. However, certain kinds of optimization will still be inhibited.

C. The C/iX compiler's debugging options are controlled by a command line option. When compiling, include the -g flag either in the compiler's INFO string or in the CCOPTS CI variable:

```
:CCXL STEST, UTEST, $NULL;INFO="-g"
```

or

```
:SETVAR CCOPTS "-g"  
CCXL STEST, UTEST, $NULL
```

In addition, programs should be linked with the a small debugging library, xdbend.lib.sys:

```
LINK FROM=...,xdbend.lib.sys;to=...
```

This library isn't actually a library, but a workspace for XDB. If the target program isn't linked with this library, XDB will be unable to execute certain commands but will still provide basic debugging capabilities.

III. Basic Commands

The basic requirements for any debugger are to allow control over execution of the program and to allow the programmer to monitor the program's progress. XDB provides a variety of ways to accomplish both these tasks.

To begin debugging, invoke XDB using the name of the target program as an argument:

```
:xdb testprog
```

XDB will read in the program file, display the lines around the target program's primary entry point, and await a command:

XDB displays program code above the inverse-video header line, and allows command input below. The greater-than symbol at line 95 in the code section indicates the next line to be executed. If the debugger is unable to access the program source code, or if the module containing the entry point lacks symbolic debugging information, XDB displays assembly language instead of source code. XDB can still be used to set breakpoints in object code, or in modules for which debugging information is available.

XDB commands, unlike those of most MPE subsystems, are case-sensitive. The commands `S` and `s`, for example, are distinct: the lowercase `s` causes XDB to step by one statement, or into a procedure or paragraph if the statement is a call, while the uppercase `S` causes XDB to step by one statement, but treat paragraph or procedure calls as atomic units.

XDB responds to a null command (a carriage return) by re-executing the last command. This is convenient for repeated single-stepping.

Begin program execution either by using one of the single-step commands, or by using the `R` (run) command. The `R` command also allows program parameters similar to a normal MPE command line. To run the program with an `INFO` string, for example, use the XDB command:

```
R;info="someinfo"
```

A lowercase `r` as a run command tells XDB to run the program ignoring any previously-specified parameters.

In general, simply running the program under the debugger provides no special advantages. It is necessary to set breakpoints so that the program can be stopped at points of interest. XDB provides several ways to set breakpoints using the `b` command:

```
b 457 Break at the given line  
b ProcessOneRecord Break at the given paragraph or procedure name
```

A breakpoint set with the `b` command is permanent: the program will stop at the breakpoint every time execution flows through that point, until the programmer deletes the breakpoint. XDB also provides temporary breakpoints, which are automatically deleted after being activated once. Use the `c` command to set temporary breakpoints, specifying the location in exactly the same way as for the `b` command. (The `c` command with no arguments simply causes the program to continue from the current breakpoint.)

While the program is stopped at a breakpoint, XDB allows the user to examine the values of variables. Global variables, local variables and procedure arguments can all be examined, and XDB will evaluate arbitrary expressions using the syntax of the language of the current module (e.g., if the currently-displayed module is written in C, XDB accepts C expressions).

Use the `p` command to print variables or expressions:

```
p currentValue           Prints the value of the variable currentValue
p myArray[5]            Prints the value of the 5th element of myArray
p myArray[i + 1]       Prints the value of the (i+1)th element
```

XDB will also perform format conversions:

```
p txCode\x             Prints the value of txCode in hexadecimal
```

Since XDB allows any arbitrary expression to be used in the print command, using an assignment expression will cause the value of the variable to change:

```
p txCode = 5           Changes the value of txCode to 5
```

XDB will also print the new value in this case. The debugger understands the values of enumerated constants in C or Pascal; these can be used in place of integer values.

Breakpoints can contain lists of commands to be executed when the breakpoint is activated. Almost any command can be included in such a list, but the most useful are the print and if commands:

```
b MyProc {p arg1}
```

XDB will break on entry to `MyProc` and print the value of `arg1` before pausing for command input. Similarly, using the `if` command can create a conditional breakpoint:

```
b MyProc {p arg1; if arg1 < 5 {c}}
```

XDB will break on entry to `MyProc` and print the value of `arg1` as before, but in this case, if the value of `arg1` is less than 5, XDB will automatically continue program execution.

IV. XDB Tips and Tricks

When writing software “platforms” — code bases that are expected to have a long life — most software engineers design in debugging aids from the start. If XDB is to be available, these debugging aids should take advantage of the debugger’s unique capabilities.

Debugging Procedures. XDB's ability to execute arbitrary expressions and return their values extends to calling user-specified procedures. That is, if `MyProc` is a procedure that takes one integer and one string argument, XDB allows:

```
p MyProc(5, "a string")
```

and will call `MyProc` using the specified arguments. This can be used in many different ways:

- To test a procedure by calling it “by hand” using various parameter values.
- To print the values of complex data structures.
- To execute procedures written specifically to check data structure integrity.
- To create a “test script” for a procedure by using XDB's recording facility.

The ability to print complex data structures or execute integrity checking procedures is particularly valuable. Combining conditional breakpoints with integrity checking procedures creates a powerful mechanism for detecting program errors:

```
b MyProc {if CheckWorkList(workListPtr) > 0 {c}}
```

Assuming that `CheckWorkList` is a function that returns a nonzero value if the structure represented by its argument is valid, this command will cause XDB to execute the `CheckWorkList` procedure every time `MyProc` is entered, and stop program execution if a problem is detected.