

Introduction to Debug/iX

Stan Sieler
Allegro Consultants, Inc.
2101 Woodside Road
Redwood City, CA 94062
(415) 369-2303 voice
(415) 369-2304 fax
sieler@allegro.com
<http://www.allegro.com>

Chapter 1: What is DEBUG/iX?

DEBUG/iX is the machine-level debugger that is bundled with MPE/iX. It provides a multiple window view of machine code and data, along with hundreds of commands and functions that can be used with a powerful interpreted programming language.

This paper is a short introduction to Debug/iX.

DEBUG/iX is the debugger of choice (and necessity!) when you want to debug the operating system, libraries, optimized code, or Compatibility Mode (CM) code. (Note: If you have a DEBUG/iX manual prior to the October 1989 edition, you should order a new version.)

Unless specified otherwise, most of the text and examples will assume that you are in "Native Mode" (NM) when debugging, as opposed to "Compatibility Mode" (CM).

The text will sometimes refer to DEBUG/iX simply as "Debug" or as the "debugger".

1.01 How Do We Enter DEBUG/iX?

There are a variety of ways to enter DEBUG/iX:

- 1) :DEBUG, a Command Interpreter (CI) command. Note: this requires PM capability.
- 2) Passing the string "DEBUG" into the NM COMMAND or HPCICOMMAND intrinsics or the CM COMMAND intrinsic.
- 3) RUN program;DEBUG (a CI command). Note: this requires write access to the program file.
- 4) Specifying the "debug" option when creating a process with the CREATE or CREATEPROCESS intrinsics (NM or CM). Note: this requires write access to the program file.
- 5) :SETDUMP (CI command) coupled with a subsequent :RUN of a program which aborts. Note: this does not allow continuation of the program.
- 6) Calling the NM DEBUG or HPDEBUG intrinsics, or the CM DEBUG intrinsic. Some programs (e.g.: QEDIT, EDITOR) allow the user to dynamically call simple procedures, including DEBUG.

7) Executing code (NM or CM) that encounters a breakpoint.

The first time that DEBUG/iX is invoked for a particular process, it spends some time looking around and setting up information about its environment. You may notice that it takes much longer to enter DEBUG/iX for a large NM program than for a small NM program. This is because DEBUG/iX has more information to derive for NM programs than for CM programs..

1.02 Reading The Prompt

When DEBUG/iX is ready for input, it prompts with a message like:

```
DEBUG/iX B.79.06

HPDEBUG Intrinsic at: a.009f16a8 hxdebug+$e4
$1 ($30) nmdebug > c
```

The first line tells us the version of DEBUG/iX (which seems to match the version of the kernel of MPE/iX).

The next line ("HPDEBUG Intrinsic...") tells us why we entered DEBUG/iX. (Several other possibilities exist.)

The third line is telling us:

- This will be our first input into Debug (the \$1)
- Our PIN (Process Identification Number) is \$30 (hex 30 is decimal 48).
- We entered Debug from Native Mode (the "nm").
- We are in debug, not in DAT, SAT, or another debug-like tool (the "debug").
- Numbers entered without a radix prefix (\$, #, or %) will be interpreted as hexadecimal (the "\$" on the two numbers).

1.03 Exiting DEBUG/iX

Once we are in DEBUG/iX, we have several ways of getting out of it:

- 1) The "C" command will exit DEBUG/iX and continue normally. "C" stands for "Continue".
- 2) The "ABORT" command will abort our process immediately. (Unless our process is critical or holds a SIR (System Internal Resource), in which case Debug will reject the ABORT with an error message.)
- 3) The "S" (or "SS") command, which will execute one (or more) instructions and then come back to Debug (assuming we haven't aborted, or called TERMINATE or QUIT or QUITPROG). ("SS" stands for SingleStep; S is a synonym for SS.)

The ABORT command will turn off DEBUG/iX's windows prior to aborting your process, unlike the C and S commands.

1.04 DBUGINIT

When DEBUG/iX is first invoked for a process, it reads and executes the commands in a file called DBUGINIT (the formal name of the file is DBUGINIT.logongroup.logonaccount). (This file may be file-equated, if desired.) DBUGINIT must be an simple ASCII file without sequence numbers (i.e.: un-numbered).

The following is a typical DBUGINIT file:

```
/* Turn on CRON...
set cron /* may be undesirable if typeahead is on
wl "CRON on"
/* open SYMOS file, prevent errors...
ignore quiet; {
    symopen symos.osb08.telesup;
    wl "Opened SYMOS"}
/* enable compiler debugging traps, if any...
trap trace_all arm
```

1.05 HELP!

DEBUG/iX has on-line help for all commands, functions, and variables. If you know the name of a command that you want help on, simply enter: HELP commandant

For example, if we want help on the ABORT command, we'd say:

```
HELP ABORT
```

1.06 Case Sensitivity

The only time that DEBUG/iX cares about the case (upper/lower) of your input is when you are entering a Native Mode symbolic address.

Due to an unfortunate decision in the early days of MPE XL, names for Native Mode procedures (and other code-oriented symbols) are case sensitive. The thinking was apparently that only by case sensitivity could the operating system differentiate between things like the C "fopen" routine and the FOPEN intrinsic. So, rather than being robust and searching for your symbolic names in both uppercase and lowercase, DEBUG/iX takes NM symbolic names literally.

DEBUG/iX commands, functions, variables, macros, symbolic types, and CM symbolic names may be uppercase, lowercase, or mixed case.

One tip to remember: Most intrinsics have uppercase names. Notable exceptions: The TurboIMAGE intrinsics and the V/Plus intrinsics have lowercase names.

1.07 The "LIST" Commands

DEBUG/iX has a number of commands that will list things for you. This includes listing: commands, functions, and variables. These commands end with the word "LIST". With two exceptions (LIST and REGLIST), every command that ends in the letters "LIST" can have the "IT" omitted.

The DEBUG/iX "LIST" commands are:

<u>Command</u>		<u>What does it list</u>
ALIASList	*	aliases (synonyms) for commands
CMDList	*	DEBUG/iX commands
ENVList	*	environmental (DEBUG/iX pre-defined) variables
ERRList		most recent errors
FUNCList	*	DEBUG/iX functions (procedures that return values)
LOCList	*	local variables (variables local to macros)
MACList	*	macros (user defined procedures)
MAPList	+	files that have been opened with MAP command
PROCList	*	symbolic names in current program or library
SYMList	*	Pascal/iX type and constant names
VARList	*	Variables ("global" variables)

The commands marked with an asterisk (*) expect a wildcarded pattern as their first parameter (e.g.: CMDL @Z@). (MAPList is documented as expecting a pattern, but does not accept one!) If no pattern is given, "@" is the default. Only the uppercase portion of the above commands is required by debugger.

Two other commands end in "LIST": LIST and REGLIST. These commands differ from the above and will be covered later.

One interesting aspect of the CMDLIST, FUNCLIST, ENVLIST, and MACLIST commands is their ability to optionally display all of the available help information for each item. As an example, if you entered:

```
CMDL , , all
```

you would get about 300 pages of output, the equivalent of chapter 4 of the DEBUG/iX manual.

Output from any DEBUG/iX command should be interruptible with control-Y. Unfortunately, control-Y is sometimes "lost". If control-Y is not working you will have to decide whether to hit Break and :ABORT or to sit back and wait.

Note: Control-Y can often be rejuvenated by entering :listf foobar at the next DEBUG/iX prompt.

The REGLIST command lists the values of over 60 different CPU registers as shown below:

```
mr R1 $1
mr R2 $96f9c4
...
mr R30 $40332f40
mr R31 $2
mr SR0 $a
...
```

```

mr SR7 $a
mr TR0 $75c200
...
mr TR7 $84941018
mr ISR $331
mr IOR $40332a78
mr IIR $489f0000
mr EIEM $ffffffff
mr RCTR $ffffffff
mr SAR $11
mr PID1 $492
mr PID2 $25c
mr PID3 $0
mr PID4 $0
mr IVA $ec000
mr ITMR $60142780
mr CCR $80
mr EIRR $0
mr IPSW $6000f
mr PCSF $a
mr PCOF $96f9c4
mr PCSB $a
mr PCOB $96f9c8

```

If the output of the REGLIST command could be captured to a disc file, then a USE of that file would rebuild the entire set of registers for the user. (The MR command changes the value of a register.)

DEBUG/iX has the ability to direct its output to a disc file, with the LIST command. The syntax for the LIST command is:

```

LIST filename
LIST <ON | OFF | CLOSE>

```

The "LIST filename" form of the command opens a new disc file of the specified name. (The file is opened as ?.) DEBUG/iX will now send a copy of all input and all output to the file, as well as to the terminal.

By default, DEBUG/iX assumes that you want all output to a "LIST" file paginated, so it opens the file with carriage control and writes a page header every 60 lines. This can be defeated, as shown in the following example which prevents carriage control from being associated with the file and disables the page header logic:

```

:file foo; nocctl
env list_paging false
list *foo

```

Be careful when using the LIST command: DEBUG/iX seems to unconditionally purge any old copy of the specified file before trying to open a new copy.

DEBUG/iX also has problems when you use the LIST command to open a spooled printer as shown:

```

:file lp;dev=lp
list *foo

```

Prior to MPE XL 3.0, the above LIST command would hang waiting for a console =REPLY.
(Replying NO would allow DEBUG/iX to continue, and would eventually produce a printer listing.)
On MPE XL 3.0, the LIST command no longer hangs in this manner, but it does open two printer files.
(The first file is empty, and probably results from a misguided attempt to purge the "old" file.)

1.08 Numbers & Expressions

Whenever DEBUG/iX wants a value, you can use an expression. This is a very powerful statement, when you think of the ramifications. If Debug is looking for a value, it evaluates your input according to the following (rough) rules:

- 1) Does it "look" like a number? This determination is based upon the current input base (hex, decimal, or octal).

If yes, then it is treated as the smallest form of number that will not lose information. For example "77" (without the quotes) would be treated as an unsigned 16 bit integer whose value is decimal 77, or decimal 63, or decimal 127, depending on the current input base (decimal, octal, or hex). "-#20000" would be treated as a signed 16 bit integer. "-#40000" would be treated as a signed 32 bit integer.

- 2) Does it "look" like a string? Strings start with a double quote (") or a single quote ('), and are terminated by whichever quote character you started the string with. (DEBUG/iX does not care which pair of quotes you use.) If you want to embed the same quote character in a string that you are using as the delimiter, simply double it. For example, the following two strings are functionally identical:

```
"Cat 's"  
'Cat ' 's'
```

- 3) Is it a local variable? Note: It is usually not necessary to put an exclamation mark (!) in front of any kind of variable name.
- 4) Is it a global variable or an environmental variable? Environmental variables are simply predefined variables controlled by DEBUG/iX.
- 5) Is it a macro call? DEBUG/iX does not differentiate between typed and untyped functions. From DEBUG/iX's viewpoint, every macro returns a value. If a macro does not have a "return" statement in it, then the default return is the value 0.
- 6) Is it a built-in function?

Chapter 2: Breakpoints

A "breakpoint" is a code address that you have told DEBUG/iX to "break" execution at. When your process attempts to execute the instruction at such an address, we say it "hits" the breakpoint.

The B command (for Breakpoint) sets a breakpoint at a specified code address. When your process tries to execute a breakpoint, it will pop up into DEBUG/iX. Breakpoints default to being "local" to your process, but if you have PM capability you can set a breakpoint for another process if you know its PIN (Process Identification Number), and you can setup system-wide breakpoints, that will affect

every process. (Note for PM users: you cannot safely set a breakpoint for code that will be running on the Interrupt Control Stack (ICS).)

The syntax for the Breakpoint command is:

```
B address [count] [ LOUD | QUIET ] [cmd]
```

"Address" is optionally qualified with a ":pin#" to set a breakpoint for a specified process, or with a ":@" to set a system-wide breakpoint.

A simple way to put a breakpoint 2 instructions from where you are now is:

```
b pc + 8      /* native mode
b p  + 2      /* compatibility mode
```

The "count" parameter specifies how many times the breakpoint must be "hit" before you actually pop into DEBUG/iX. The default is 1, which means you will enter DEBUG/iX every time the breakpoint is hit. If a count of "2" is used, then every other time you hit the breakpoint, you will enter Debug/iX. And, "other every" time you will silently continue without entering DEBUG/iX.

By default, breakpoints are "permanent", unlike MPE V, where the breakpoints defaulted to temporary (or, "one shot"). (Of course, permanent for non-system-wide breakpoints really means "for the life of the process".)

If "count" is negative, then the breakpoint is deleted when it finally enters DEBUG/iX. One nice usage of this is to set a "temporary" breakpoint at 2 instructions from where you are now:

```
b pc + 8, -1   /* NM
b p  + 2, -1   /* CM
```

The "LOUD/QUIET" option (default is LOUD) lets you decide whether or not DEBUG/iX should announce the fact that you have hit a breakpoint. The QUIET option can really speed things up when you are hitting a breakpoint thousands of times.

The "cmd" option is a DEBUG/iX command to be executed when you enter DEBUG/iX for the breakpoint. Compound commands (a list of commands separated by semicolons and surrounded by braces ("{}")) are allowed. If a "cmd" is specified, then it will be executed. If the command does not have a "c" command within it, then you will be left in DEBUG/iX after the command finishes.

The following example shows how the "cmd" option can be very helpful in finding a problem. Let's say that your NM program is opening several hundred files with the FOPEN intrinsic, but is having a problem when a file fails to open. We can setup a breakpoint at the exit from FOPEN that will quietly continue each time until the failing FOPEN is encountered. We can couple this with a breakpoint at the start of FOPEN that will remember the name of the file that is being opened:

```
b FOPEN, , quiet, {var save_file_address = r26; c}

b ?FOPEN+8, , quiet, {if r28 <> 0 then c
                      else {
                        wl "Failed to open file: ";
                        dv save_file_address, 10, s} }
```

With the first of the above breakpoints, every time we enter FOPEN the debugger will save the first parameter (which is the address of the name of the file being opened) in a global debugger variable

called "save_file_address". The second breakpoint will check that FOPEN has returned a valid (non-zero) file number. If it has not, then we know that the open failed, and display the name of the file with a DV command. We could then, if we have PM capability, do a dynamic call of PRINTFILEINFO for more information about the failure:

```
= nmcall ( "PRINTFILINFO", 0 )
```

Note: The "nmcall" function requires MPE XL 2.2 or later.

2.01 Breakpoint At Procedure Return

It is often desirable to set a breakpoint at the return point from a procedure call. The following shows an NM and a CM method of setting a breakpoint at the return address from a procedure.

```
lev 1; b pc, -1; lev 0    /* NM example  
lev 1; b p, -1; lev 0     /* CM example
```

(The "lev 0" is optional for both.)

Another method, only valid for NM code and only fully correct when you are at the entry to a procedure, is:

```
b sr4.r2, -1
```

With any of the above techniques, the breakpoint can be made permanent by omitting the "-1".

Chapter 3: Tracing Your Stack

When you enter DEBUG/iX, you often want to look at the history of how you got there. DEBUG/iX is entered because your code called DEBUG or HPDEBUG, because you hit a breakpoint, or because you triggered a trap that the debugger was watching.

DEBUG/iX would like to tell you what procedure you are in, and (perhaps) what procedure called it. The identity of the current procedure is determined from the value of the Program Counter register (pc). The "caller" procedure is a little more difficult to determine.

The TR command (for "TRace") will try to list the procedure your code is in, the procedure that called it, the procedure that called that procedure, and so on, all the way back to the outer block of the program. The process of looking at each procedure to determine who called it is called "walking the stack". The result of listing the names of each of these procedures in reverse chronological order is called a "stack trace"

Unlike the Classic HP 3000 instruction set, PA-RISC does not have a "procedure call" instruction. This means that the procedure calling mechanism is defined entirely by the software. The "Instruction Set and Procedure Calling Conventions" Manual (HP part # 09740-90015) documents this convention. Although it has a few shortcomings, it works reasonably well for most uses. DEBUG/iX is aware of the Procedure Calling Convention (PCC). In order for DEBUG/iX to be able to determine what procedure called the current procedure, DEBUG/iX must consider information including:

- is the current code a procedure or millicode?
- is the current procedure a "leaf" procedure? (leaf procedures are those procedures that do not call any other procedures)
- Has the procedure stored the return address into the stack?
- how big is the stack frame for the procedure?

Sometimes, DEBUG/iX cannot answer all of these questions. When this happens, DEBUG/iX is unable to completely trace the stack.

3.01 TR Command

This section discusses the TR command. TR causes DEBUG/iX to print a stack trace for the "current" process. To get a stack trace for another process, first switch to it with the PIN command, then use the TR command.

The syntax for the TR command is:

```
TR [ # ], [Interrupts] [Dual] [Full]
```

The first parameter to TR, which we usually omit, tells DEBUG/iX to stop the stack trace after printing that number of markers. This is useful when you are stack tracing a process that is dozens or hundred of procedures "deep". (Of course, such a process probably suffers from severe design problems, which is why you are probably debugging it!) When the number is omitted, DEBUG/iX will trace as far "back" into the stack as possible.

Each of the other options for the TR command may be abbreviated to just their first letter.

3.02 Sample Stack Trace

When the demo program (http://www.allegro.com/papers/debug_ex1.html) is run as follow:

```
:resetdump
:run demo.pub
```

it aborts with the following:

```
**** Bound violation or range error (TRAPS 12).

ABORT: SAMPLE.PUB.SIELER
NM PROG 604.00005b30 parse_info.find_non_blank+$44

PROGRAM TERMINATED IN AN ERROR STATE. (CIERR 976)
```

If a :RESETDUMP command was in effect, then the following occurs when the program aborts:

```
:setdump
:run sample.pub

**** Bound violation or range error (TRAPS 12).

ABORT: SAMPLE.PUB.SIELER
      PC=604.00005b30 parse_info.find_non_blank+$44
NM* 0) SP=4033a168 RP=604.00005cd0 parse_info+$94
NM  1) SP=4033a168 RP=604.000062f8 PROGRAM+$d0
NM  2) SP=4033a120 RP=604.00000000
      (end of NM stack)

R0 =00000000 00000051 00005cd3 00000001 R4 =c0000000
81c07f80 c7e80000 00000000
R8 =00000000 00000000 00000000 00000000 R12=00000000
00000000 00000000 00000000
R16=00000000 00000000 00000000 00000020 R20=00000020
00000050 00000051 20202020
R24=20202020 403310cc 403310c8 40331008 R28=00000000
4033a168 4033a168 40331067

IPSW=0026000f=jthlNxbCVmrQPDI PRIV=3 SAR=000b
PCQF=604.5b33 604.5b37

SR0=0000000a 00000000 00000000 00000000 SR4=00000604
000005c8 0000000b 0000000a
TR0=00733200 00793200 000005c8 00879100 TR4=c70a0000
4033ad6c 4033abc0 c7e81018
PID1=05d0=02e8(W) PID2=0058=002c(W)
PID3=0000=0000(W) PID4=0000=0000(W)

RCTR=00000000 ISR=00000604 IOR=00000000 IIR=b0207761
IVA=0013c000 ITMR=fe0719b0
EIEM=ffffffff EIRR=00000000 CCR=0080
```

```

**** PROCESS ABORT INTERACTIVE DEBUG FACILITY ****

$1 ($22) nmdebug > tr,i,d
      PC=604.00005b30 parse_info.find_non_blank+$44
NM* 0) SP=4033a168 RP=604.00005cd0 parse_info+$94
NM  1) SP=4033a168 RP=604.000062f8 PROGRAM+$d0
NM  2) SP=4033a120 RP=604.00000000
      (end of NM stack)
$36 ($22) nmdebug > c

PROGRAM TERMINATED IN AN ERROR STATE.  (CIERR 976)
:

```

The data shown by the TR command is worth discussion in more detail. Consider the following stack trace :

```

:editor
HP32201A.07.20 EDIT/3000 SUN, JAN 26, 1992, 12:49 PM
(C) HEWLETT-PACKARD CO. 1990
/:debug
DEBUG/iX B.79.06

HPDEBUG Intrinsic at: a.0096f9c4 hxdebug+$144
$39 ($43) nmdebug > tr
      PC=a.0096f9c4 hxdebug+$144
* 0) SP=40332b10 RP=a.009787b4 exec_cmd+$7e4
   1) SP=40332690 RP=a.0097a244 try_exec_cmd+$c8
   2) SP=40332640 RP=a.00977e3c command_interpret+$358
   3) SP=403321e8 RP=a.0097af64 xeqcommand+$198
   4) SP=40331e80 RP=a.0097e370 prog_execute_cmd+$20
   5) SP=40331e00 RP=a.0097e31c ?prog_execute_cmd+$8
      export stub: a.0049dda4 COMMAND+$7d4
   6) SP=40331dd0 RP=a.0049d5bc ?COMMAND+$8
      export stub: a.004c2514 arg_regs+$28
   7) SP=403317e8 RP=a.004a85cc nm_switch_code+$978
   8) SP=403316b8 RP=a.00497aec Compatability_Mode
      (switch marker frame)
   9) SP=403312e0 RP=a.00909c64 outer_block+$150
   a) SP=403310f0 RP=a.00000000 _traplib_version
      (end of NM stack)

```

The first line of the TR output:

```
PC=a.0096f9c4 hxdebug+$144
```

reports where the process is currently executing. PC is the Program Counter. The value is shown as a 64-bit address, with the "a" being the space ID, and the 0096f9c4 being the offset within space \$a. Code addresses with a \$a as the space ID are within NL.PUB.SYS, the kernel of the operating system.

The second line:

```
* 0) SP=40332b10 RP=a.009787b4 exec_cmd+$7e4
```

tells us a variety of things:

- the "*" means: this is the stack marker associated with the program counter (pc).
- the "0)" means: this is the "top of stack", or the most current stack marker. The next marker says "1)", meaning that it is one level "down" into the past.
- the "SP=" tells us what the value of the top-of-stack pointer (SP) is for this marker. The value for the second marker (#1) is less than for the top marker (#0). The difference is the number of bytes that the top procedure allocated as a stack frame.
- the "RP=" tells us where the current procedure will return to. Note: This part of the stack trace is the most misleading. It is tempting to look at a line, like "3)" and think that the stack trace is telling us that when we return to xeqcommand+\$198, the stack pointer will be \$403321e8. Unfortunately, this is an artifact of a somewhat poor choice in the layout of the TR command's output. The line is actually telling us: when we return to command_interpret+\$358 (we got this address from the "2)" line), the SP value will be restored to 403321e8 and the RP value will be restored to a.0097af64, which (in case we wanted to know) happens to be within xeqcommand.

In the above stack trace, the ninth marker:

```
8) SP=403316b8 RP=a.00497aec Compatability_Mode  
(switch marker frame)
```

told us that DEBUG/iX found a stack marker that indicated that the process had switched from Compatability Mode (CM) into Native Mode. When such a switch takes place, a marker is left on the CM stack and on the NM stack. Marker "8)" is an example of such an NM stack marker.

Normally, the TR command assumes that you only want to see marker that are in the "current" mode (nm for this example). If you want to see both modes, the ",Dual" option can be used on the TR command:

```
tr, d
```

Or, you can manually switch to the opposite mode (in this example, CM), and ask for a simple TRace:

```
$3b ($43) nmdebug > cm
%74 (%103) cmdebug > tr
      SYS % 205.7317 SWITCH'TO'NM'+%4 (MITroC CCG) SUSER1
* 0) SYS % 205.7317 SWITCH'TO'NM'+%4 (MITroC CCG) SUSER1
  1) SYS % 140.4430 COMMAND+%43 (MITroC CCG) CISEG1
  2) PROG % 7.1407 (mItroc
  3) PROG % 7.1535 (mItroc
  4) PROG % 7.3404 (mItroc
  5) PROG % 7.30 (mItroc
  6) SYS % 145.0 ?TERMINATE (MITroc CCG) CM

%75 (%103) cmdebug > = pin
%103

%76 (%103) cmdebug > c
/:comment: we are back to the EDITOR prompt, and we
/:comment: know that :EDITOR's PIN is 67 (octal %103).
```

When we say that a process that is "in Compatibility Mode", we mean that it is either executing CM instructions via the emulator or that it is executing CM instructions that have been translated to NM instructions via the Object Code Translator (OCT). In either case, the emulator or the translated code will often jump into small NM assembly "helper" routines. When we ask DEBUG/iX to trace the stack for a process that happens to be in some of these routines, the TR command may be unable to correlate the helper routine with the original CM (or OCT) instruction address.

Chapter 4: Data: Displaying & Modifying

This section discusses the DEBUG/iX commands for viewing and modifying data.

4.01 DV Command

Data can be viewed in a variety of manners. The basic syntax for the Display Virtual command (DV) is:

```
dv address [#words] [base] [#words/line] [#bytes/"word"]
```

Note: DEBUG/iX quietly rounds your address down to the nearest multiple of 4. This can be quite misleading if you were trying to display data starting at an address of the form $n*4+2$, which is typical of 50% of the shortint variables in a Pascal/iX program.

The #words to display defaults to 1.

The output base defaults to the current OUTBASE value (typically hex for NM, octal for CM).

Base options are: ASCII, Both, Code, Decimal (or #), Hex (or \$), Octal (or %), and String. The base may be abbreviated to a single letter. Two bases are particularly useful at times: Both and String. The Both base shows the output in hex (or whatever the current OUTBASE value is) and in ASCII, four 32-bit words per line. The String base displays the data as ASCII text (with nonrenewable displayed as dots) with no blanks every 4 (or 2) characters. If the String base is used without the #words/line

option, then DEBUG/iX will simply display the starting address, a quote ("), and then all of your data as ASCII (line after line, with no addresses), followed by a trailing quote (") at the end. If you use the #words/line option with the String base, then it will put a quote (") after that many words, and then (on a new line) display the next address, a quote ("), and continue. A particularly useful combination of String and #words/line is:

```
dv address, #words, S, $e
```

The value of \$e for #words/line results in the maximum amount of text that will cleanly fit on an 80-character line.

Examples:

```
dv dp + 18
dv dp + 100, #20, , , 2
```

DEBUG/iX has many commands to display data, all beginning with "D". These include: dr, ddb, ddq, dds, dd, and even more. For a complete list of data display commands, do:

```
CMDL ,display
```

4.02 Modifying Data

This section briefly discusses techniques for modifying data.

The basic command to modify data at a virtual address is the "MV" command. Its syntax is:

```
mv address [#words] [base] [VALUE value2 value3 ... ]
```

If new values are not provided, DEBUG/iX will prompt for them by first displaying the address, then the current value in both hex (or the specified base) and ASCII.

Note: Unless the "quiet_modify" environmental variable is set to true, the MV command will echo the old values even if the new value is provided on the command line.

You can give a register a new value with the MR command:

```
MR register [value]
```

Like the MV command, MR will prompt for a new value.

If you are "sitting" at the start of a procedure that you would like to avoid executing, the MR command can be used to skip over the code in the procedure as follows:

```
mr pc, r2
c
```

The above can be combined with a breakpoint to automatically skip calling a particular routine. For example, to prevent a program from successfully calling the NM QUIT intrinsic, try:

```
b QUIT, , , {mr pc, r2; c}
```

This trick is not applicable to CM code.

5. End

There's a lot more to Debug/iX than can be squeezed into a short paper...I encourage you to try it out for yourself, because it can dramatically reduce debugging time on MPE/iX.