900 Series HP 3000 Computer Systems

System Debug Reference Manual

HEWLETT PACKARD

HP Part No. 32650-90013
Printed in U.S.A. 1992

Third Edition
E0692

FINAL TRIM SIZE : 7.5 in x 9.0 in

FINAL TRIM SIZE : 7.5 in x 9.0 in

Print History

The following table lists the printings of this document, together with the respective release dates for each edition. The software version indicates the version of the software product at the time this document was issued. Many product releases do not require changes to the document. Therefore, do not expect a one-to-one correspondence between product releases and document editions.

Edition	Date	Software Version
First Edition	November 1987	A.01.00
Update 1	July 1988	A.10.00
Second Edition	October 1989	A.30.00
Third Edition	June 1992	B.40.00

FINAL TRIM SIZE : 7.5 in x 9.0 in

Preface

The *System Debug Reference Manual* (32650-90013) is written for the experienced programmer. It is a reference manual that provides information about System Debug. System Debug provides a family of low-level assembly language debugging tools for MPE XL (for both Native and Compatibility Mode code):

- Debug
- Dump Analysis Tool (DAT)
- Standalone Analysis Tool (SAT)

A certain level of knowledge is required to utilize System Debug. Specifically, familiarity with assembly code, procedure calling conventions, parameter passing conventions, and HP 3000 and HP Precision Architecture is assumed.

This manual is organized into the following chapters and appendices:

Chapter 1 **Introduction** contains an introductory overview of System Debug features and describes how to get started with the debugger.

Chapter 2 **User Interface** describes the common user interface supported by System Debug. This chapter describes expressions, types, operators, operands, functions, variable macros, error handling, regular expressions, the history stack, and Control-Y handling.

Chapter 3 **System Debug Interfaces Commands & Intrinsics** describes the commands and intrinsics (both CM and NM) that enable you to invoke System Debug either interactively or programmatically.

Chapter 4 **System Debug Command Specifications** lists the System Debug commands in alphabetic order, complete with full syntax, parameter descriptions, and examples of use.

Chapter 5 **Symbolic Formatting Symbolic Access** presents an overview of symbolic formatting and symbolic access functions.

Chapter 6 **System Debug Windows** describes the System Debug screen windows. Basic window operations are introduced, and a typical screen display is presented. Each type of window is described, along with an explanation of each field within the window.

Chapter 7 **System Debug Window Commands** lists the System Debug window commands, broken into logical groups. The window commands are then listed in alphabetical order, along with full syntax, parameter descriptions, and examples of use.

Chapter 8 **System Debug Standard Functions** lists the predefined System Debug functions in alphabetical order, complete with full syntax, parameter descriptions, and examples of use.

Chapter 9 **Dump Analysis Tool (DAT)** contains information on the Dump Analysis Tool (DAT).

Chapter 10 **Standalone Analysis Tool (SAT)** contains information on the standalone Analysis Tool (SAT).

Appendix A **Patterns and Regular Expressions** presents pattern matching and regular expressions.

Appendix B **Expression Diagrams** contains System Debug expression diagrams.

Appendix C **Emulated/Translated CM Code** describes CM Object Code Translation

Appendix D **Reserved Variables/Functions** contains a full summary of all reserved variables and functions.

Appendix E **System Debug Command Summary** contains a full System Debug command summary.

Conventions

`UPPERCASE` In a syntax statement, commands and keywords are shown in uppercase characters. The characters must be entered in the order shown; however, you can enter the characters in either uppercase or lowercase. For example:

```
COMMAND
```

can be entered as any of the following:

```
command        Command        COMMAND
```

It cannot, however, be entered as:

```
comm           com_mand       comamnd
```

italics In a syntax statement or an example, a word in italics represents a parameter or argument that you must replace with the actual value. In the following example, you must replace *filename* with the name of the file:

`COMMAND` *filename*

bold italics In a syntax statement, a word in bold italics represents a parameter that you must replace with the actual value. In the following example, you must replace ***filename*** with the name of the file:

`COMMAND(`***filename***`)`

punctuation In a syntax statement, punctuation characters (other than brackets, braces, vertical bars, and ellipses) must be entered exactly as shown. In the following example, the parentheses and colon must be entered:

`(`*filename*`):(`*filename*`)`

underlining Within an example that contains interactive dialog, user input and user responses to prompts are indicated by underlining. In the following example, yes is the user's response to the prompt:

```
Do you want to continue? >>  yes
```

`{ }` In a syntax statement, braces enclose required elements. When several elements are stacked within braces, you must select one. In the following example, you must select either `ON` or `OFF`:

$$\texttt{COMMAND} \left\{ \begin{array}{l} \texttt{ON} \\ \texttt{OFF} \end{array} \right\}$$

`[ ]` In a syntax statement, brackets enclose optional elements. In the following example, `OPTION` can be omitted:

`COMMAND` *filename* `[OPTION]`

When several elements are stacked within brackets, you can select one or none of the elements. In the following example, you can select `OPTION` or *parameter* or neither. The elements cannot be repeated.

$$\texttt{COMMAND}\ \textit{filename} \left[\begin{array}{l} \texttt{OPTION} \\ \textit{parameter} \end{array} \right]$$

FINAL TRIM SIZE : 7.5 in x 9.0 in

[. . .]	In a syntax statement, horizontal ellipses enclosed in brackets indicate that you can repeatedly select the element(s) that appear within the immediately preceding pair of brackets or braces. In the example below, you can select *parameter* zero or more times. Each instance of *parameter* must be preceded by a comma: [,*parameter*][...] In the example below, you only use the comma as a delimiter if *parameter* is repeated; no comma is used before the first occurrence of *parameter*: [*parameter*][,...]
| . . . |	In a syntax statement, horizontal ellipses enclosed in vertical bars indicate that you can select more than one element within the immediately preceding pair of brackets or braces. However, each particular element can only be selected once. In the following example, you must select `A, AB, BA`, or `B`. The elements cannot be repeated. $\left\{ \begin{array}{c} \texttt{A} \\ \texttt{B} \end{array} \right\} \mid \ldots \mid$
. . .	In an example, horizontal or vertical ellipses indicate where portions of an example have been omitted.
Δ	In a syntax statement, the space symbol Δ shows a required blank. In the following example, *parameter* and *parameter* must be separated with a blank: (*parameter*)Δ(*parameter*)
( )	The symbol ( ) indicates a key on the keyboard. For example, (RETURN) represents the carriage return key or (Shift) represents the shift key.
(CTRL)*character*	(CTRL)*character* indicates a control character. For example, (CTRL)Y means that you press the control key and the Y key simultaneously.

FINAL TRIM SIZE : 7.5 in x 9.0 in

Conventions (continued)

base prefixes — The prefixes %, #, and $ specify the numerical base of the value that follows:

%*num* specifies an octal number.
#*num* specifies a decimal number.
$*num* specifies a hexadecimal number.

If no base is specified, decimal is assumed.

bits (*bit*:*length*) — When a parameter contains more than one piece of data within its bit field, the different data fields are described in the format bits (*bit*:*length*), where *bit* is the first bit in the field and *length* is the number of consecutive bits in the field. For example, bits (13:3) indicates bits 13, 14, and 15:

```
most significant              least significant
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
| 0|  |  |  |  |  |  |  |  |  |  |  |  |13|14|15|
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
bits (0:1)                           bits (13:3)
```

FINAL TRIM SIZE : 7.5 in x 9.0 in

Contents

4. **System Debug Command Specifications**

7. System Debug Window Commands

8. System Debug Standard Functions

Tables

1

Introduction

System Debug provides a family of low-level assembly language debugging tools for MPE XL:

- Debug
- Dump Analysis Tool (DAT)
- Standalone Analysis Tool (SAT)

A certain level of knowledge is required to utilize System Debug. Specifically, familiarity with assembly code, procedure calling conventions, parameter passing conventions, and HP 3000 and HP Precision Architecture is assumed. If you do not require the features offered by an assembly language debugger, please be aware that two excellent source-level symbolic debuggers are available from Hewlett-Packard: Symbolic Debug/XL and Toolset/XL.

What Is Debug?

Debug provides non-privileged and privileged users with both interactive and programmatic debugging facilities for examining their operating environments.

Debug enables you to do the following:

- Set, delete, and list breakpoints in a program. The program executes until a breakpoint is reached, then stops and passes control to the user. When you set breakpoints, you can specify a list of commands that automatically are executed when the breakpoint is hit.
- Single step (multiple steps) through a program.
- Display and/or modify the contents of memory locations. A full set of addressing modes is offered, including absolute CM memory, code segment relative, data segment relative, S relative, Q relative, DB relative, HP

Precision Architecture virtual addresses, and HP Precision Architecture real memory addresses.

- Display a symbolic procedure stack trace, optionally displaying interleaved NM and CM calls. You can also set the current debug environment back temporarily to the environment which existed at any marker on the stack.
- Calculate the value of expressions in order to determine the correct values of variables at a given point in a program. Values can be custom formatted in several bases.
- Use new full screen displays (windows) which allow inspection of registers, program code, the current stack frame, and the top of stack. Groups of custom user windows can be aimed at important data blocks to monitor changing values dynamically.
- Display online help for all commands, predefined functions, and environment variables.
- Create and reference user-defined variables.
- Define powerful parameterized macros. Macros can be invoked as new commands to perform useful sequences of commands, or as functions within expressions that return single values.
- Define aliases for command and macro names.
- Execute commands from a file, record all user input to a log file, and record all Debug output to a list file.

What Is the Dump Analysis Tool (DAT)?

The Dump Analysis Tool (DAT) aids support and lab personnel in analyzing MPE XL system events such as process hangs, operating system failures, or hardware failures. This tool is used primarily by Hewlett-Packard support personnel.

Refer to chapter 9 for detailed information regarding DAT.

What Is the Standalone Analysis Tool (SAT)?

The Standalone Analysis Tool (SAT) aids support and lab personnel in analyzing MPE XL system events such as process hangs, operating system failures, and hardware failures.

Refer to chapter 10 for detailed information regarding SAT.

How to Debug

This chapter gives a very brief introduction to debugging. For additional information, refer to the *Programmer's Guide* corresponding to the language compiler you are using. There you will find details and examples specific to your language.

How to Debug a CM Program

Compile and, using the Segmenter, prepare your program file and optional library files.

In order to take full advantage of Debug's symbolic capabilities, you must ensure that your program (and library) contain the necessary `FPMAP` symbolic records. This is easily accomplished with the Segmenter as follows:

For program files, use the `FPMAP` option when you prepare your program:

```
:PREP USLFILE, PROGFILE;FPMAP
```

For libraries, use the `FPMAP` option each time you add a segment to the library:

```
ADDSL SEG ; FPMAP
```

To debug your program, specify the `Debug` parameter in the `RUN` command:

```
:RUN CMPROG.GRP.ACCT;LIB=G;DEBUG
```

The program file is loaded, and you break at the first instruction in your program, at the main entry point.

Debug announces your arrival into the debugger. You are now ready to debug your program (set breakpoints, define macros, turn on the windows, and so on). For example,

```
:RUN CMPROG.GRP.ACCT;LIB=G;DEBUG
CM DEBUG Intrinsic: PROG  %0.22

%cmdebug > won
```

How to Debug an NM Program

Compile and link your program file and any necessary libraries.

To Debug your program, specify the `DEBUG` parameter in the `RUN` command:

```
:RUN NMPROG;DEBUG
```

The NM program file is loaded, and a temporary breakpoint is set at the external stub that is linked to your program's main entry point.

When the program is launched into execution, the temporary breakpoint is hit, and you immediately enter Debug (in NM mode). Debug announces your arrival and deletes the temporary breakpoint.

To best observe the actual entrance (through the stub procedure) into your main program, type `WON` to turn the windows on. Note that you are at a stub procedure, which is marked with a question mark:

```
> ?PROGRAM
  ?PROGRAM+0004   etc.
```

Single step a few times to advance the program through the stub and into the main body of the program. In summary,

```
:RUN NMPROG;DEBUG
Break at:   [0] PROG 31.00022e7c  ?PROGRAM
$nmdebug > won
$nmdebug > s
$nmdebug > s
```

You are now ready to debug your program (set breakpoints, define macros, turn on the windows, and so forth).

2

User Interface

The System Debug user interface is command oriented. That is, all requests for System Debug to perform some operation must be expressed as commands. Normally, commands are read either from the standard input device (`$STDIN`) in the case of DAT, or from the session LDEV using low-level I/O routines in the case of Debug. But commands may also be read from command files, sometimes known as *use files*, stored on disk.

System Debug output is displayed in one of two ways. List output is typically written to the user's terminal as a sequence of lines, but may also be automatically echoed to disk files, interleaved with the interactive command input that generated it. System Debug also offers a tiled window facility, which provides an interpretation of the machine state as well as code and data memory areas. The windows are updated to reflect changes in the displayed areas that occurred between commands.

This chapter discusses the various data types supported by System Debug and how values of these types are created or accessed, manipulated, and stored. Other topics, such as error handling, Control-Y startup processing, error handling, Control-Y management, and debugging at the console, are also discussed.

For detailed information of the syntax, operation, and output of individual commands, please refer to chapter 4. Windows, and the commands that control them, are explained in chapters 6 and 7.

Command Line Overview

System Debug displays a prompt when it is ready to accept a command interactively. The standard prompt looks like this:

```
$10 ($42) nmdebug >
```

The first number is the current command number. This is the number that is assigned to the command entered at the prompt. Blank lines do not cause the command number to increase. The number in parentheses is the process identification number (PIN) of the current process. If Debug is entered from the CI, then this is the CI's PIN.

The dollar signs in front of the numbers indicate that the current output radix is hexadecimal. Except for a few obvious exceptions, most numbers are displayed in the current output base. The abbreviations for numeric radices are

```
% - octal,    # - decimal,    $ - hexadecimal.
```

The `nmdebug >` part of the prompt is composed of two parts. The first, `nm`, indicates that the current mode of System Debug is native mode. The other possibility is `cm` for compatibility mode. The second part, `debug`, identifies the name of the tool being run. Another possibility for this is `dat`.

The prompt can be changed with the `ENV` command as follows:

```
$10 ($42) nmdebug > env prompt "mode ' > '"
   nm >
```

Command names can be entered in either upper- or lowercase and may be followed by their parameters, separated from one another by either blanks or commas. The specifications of individual commands may also describe special parameters that are also accepted.

Comments can be entered on any command line, and are introduced by the sequence `/*`. Everything on a command line after the `/*` is ignored:

```
CMD1  parm1    /* this is a comment...
```

Long commands may be spread across several lines by using the command continuation character "`&`". Command lines ending with this character are continued on the following line. The special prompt `cont >` is used to indicate that command continuation is in progress:

```
$nmdebug > wl 'This is a long &
cont > line broken into&
cont >   three parts.'
This is a long line broken into three parts.
$nmdebug >
```

The semicolon separates multiple commands entered on the same line:

```
CMD1; CMD2; CMD3; ...
```

A command list can be formed by enclosing multiple commands within curly braces. Command lists are syntactically single commands, and are frequently used as command parameters:

```
b myproc, 1,, {CMD1; CMD2; CMD3}
```

Unterminated command lists, which are introduced with a left curly brace, can be continued on successive input lines without the use of the command continuation character. The command prompt changes to indicate that a multiline command list is being read, and it displays the current nesting level of the braces. When the final closing right brace is encountered, the prompt changes back to the normal command line prompt:

```
$nmdebug > if p1 > 0 then {
{$1} multi > wl "parm is:" p1;
{$1} multi > var curbias = p1+bias}
$nmdebug >
```

Data Types

Several data types are supported by System Debug. This section introduces each of the types by giving the mnemonics by which they are known, along with a description of the data which they represent.

Integer Types

Three sizes of signed and unsigned integers are supported:

`S16`	Signed 16-bit integer.
`U16`	Unsigned 16-bit integer.
`S32`	Signed 32-bit integer.
`U32`	Unsigned 32-bit integer.
`S64`	Signed 64-bit integer.

All of the signed types obey the properties of twos complement binary arithmetic. The type `S64` has not been fully implemented, and it supports only those values in the range $-2^{52}..2^{52} - 1$. Other than this restriction, `S64` values behave as if they consume 64 bits.

Boolean Type

Data of type BOOL may assume the values TRUE and FALSE. Integer values also are generally accepted where BOOLs are called for, and when this occurs, zero (0) is taken to be FALSE; all other values are TRUE.

String Types

The type STR is used to represent variable-length character (text) data. Strings quoted with single and double quotes (' and ") represent literal text. But strings quoted with the back-quote character (`) are sometimes interpreted as regular expressions, which are used to match other text. Refer to appendix A for a discussion of how patterns and regular expressions can be constructed for use in pattern matching.

Pointer Types

System Debug supports many different kinds of pointer types, but most are actually variations of the same theme. Pointers come in two sizes, long and short, and both may be interpreted quite differently depending on the current mode of System Debug.

The most frequently used pointer types are *long pointer* (LPTR) and *short pointer* (SPTR). An LPTR is simply a pair of 32-bit numbers separated by a dot, sometimes called a *dotted pair*. What the two numbers actually mean is unspecified by the type. Instead, the context in which the LPTR is used determines the meaning. An SPTR is just one 32-bit number, and it is often thought of as being the low-order (rightmost) part of an LPTR. When used in CM, both long and short pointer values are often range-checked to verify that they fit within 16 bits.

The remaining pointer types are variations of long pointers (that is, they are all dotted pairs). However, unlike LPTRs, they project an additional meaning on the dotted pair. Since the interpretation of pointers is heavily dependent on the mode of System Debug, the rest of this discussion deals with each mode individually.

Compatibility Mode Pointers

An LPTR in CM is usually a *segment.offset*. If a CM LPTR refers to data, then the segment number is the DST number of the addressed data segment, and the offset is the CM word offset from the beginning of the segment. If a CM LPTR refers to code, there are many possible interpretations of the segment number, and without additional information the LPTR is ambiguous. It is for this reason that the additional long pointer types exist. Their purpose is to differentiate LPTRs. Most users who work with CM code are probably familiar with the logical code segment numbers assigned by the Segmenter. The Segmenter's -PREP command assigns logical code segment numbers to program file segments, while the -ADDSL command assigns logical code segment numbers to SL file segments. These segment numbers always begin with zero (0) in each program or SL file. System Debug allows users to refer to loaded CM code using these logical code segment numbers through use of the following logical code pointer types:

PROG	Program file long pointer.
GRP	Group library file long pointer.
PUB	Public library file long pointer.
LGRP	Logon group library file long pointer.
LPUB	Logon public library file long pointer.
SYS	System library file long pointer.

Logon group and public libraries are loaded only by the CM LOADPROC intrinsic.

The above long pointer subtypes are by far the preferred choice for specifying code addresses. Since System Debug also displays CM code addresses logically, it usually is not necessary to refer to CM code segments by the CST/CSTX segment numbers assigned to them by the CM loader. However, low-level system debugging sometimes requires this method of addressing, and it is supported by the following absolute code pointer types:

CST	Absolute CST long pointer.
CSTX	Absolute CSTX long pointer.

CM program segments are assigned numbers in the CSTX, while CM SL segments are assigned numbers in the CST. CST and CSTX segment numbers start with 1. The following illustration depicts the relationships between CM logical code segment numbers and absolute ones.

```
Logical              Absolute       Absolute             Logical
System Library       CST            CSTX                 Program
Segments             Segments       Segments             Segments

+-------+           +--------+     +--------+           +--------+
| SYS 0 |  ----->   | CST 1  |     | CSTX 1 |  <-----   | PROG 0 |
|       |           | sys 0  |     | prog 0 |           |        |
+-------+           +--------+     +--------+           +--------+
| SYS 1 |  --+      | CST 2  |     | CSTX 2 |  <-----   | PROG 1 |
|       |    |      | sys 5  |     | prog 1 |           |        |
+-------+    |      +--------+     +--------+           +--------+
| SYS 2 |    |      | CST 3  |     | CSTX 3 |  <-----   | PROG 2 |
|       |    |      | sys 11 |     | prog 2 |           |        |
+-------+    |      +--------+     +--------+           +--------+
|       |    +-->   | CST 4  |     | CSTX 4 |  <-----   | PROG 3 |
|       |           | sys 1  |     | prog 3 |           |        |
+-------+           +--------+     +--------+           +--------+
| SYS n |           | CST 5  |
|       |           |        |
+-------+           +--------+
                    | CST 6  |
                    |        |
                    +--------+
Logical             |        |
Group Library       |        |
Segments            +--------+
             +-->   | CST 74 |
             |      | grp 0  |
+-------+    |      +--------+
| GRP 0 |  --+      | CST 75 |  <-+
|       |           | grp 3  |    |
+-------+           +--------+    |
| GRP 1 |  ----->   | CST 76 |    |
|       |           | grp 1  |    |
+-------+           +--------+    |
| GRP 2 |                         |
|       |                         |
+-------+                         |
```

```
| GRP 3 |  ----------------------+
|       |
+-------+
```

Note that the following pairs specify the same segment:

```
(logical)  PROG 1   <-->   CSTX 2   (absolute)
(logical)  SYS  1   <-->   CST  4   (absolute)
(logical)  GRP  3   <-->   CST  75  (absolute)
```

Native Mode Pointers

An `LPTR` in NM is usually a *sid.offset* virtual address. As such, NM `LPTR`s are unambiguous, even without some context of use. However, it is still useful to tag NM long pointers to code by using a type that expresses the code's logical origin. Thus, the following logical code pointer types are available for NM code addresses:

`PROG`	Program file long pointer.
`GRP`	Group library file long pointer.
`PUB`	Public library file long pointer.
`SYS`	System library file long pointer.
`USER`	User library file long pointer.
`TRANS`	Translated CM code long pointer.

Individual space IDs (SIDs) are assigned to each loaded NM program or library file by the NM loader. These numbers should be expected to be different each time the files are loaded. The `LOADINFO` command displays the relationships between loaded NM code files and their assigned SIDs.

Note the following differences between CM and NM logical code pointers. First, the CM types `LGRP` and `LPUB` do not exist for NM code, since addresses of this type are generated only by the CM `LOADPROC` intrinsic. Next, the types `USER` and `TRANS` are specific to NM. `USER` is a long pointer to a location in a user library file which was loaded by the `XL=` option of the `RUN` command. Since more than one such user library may be loaded, the type `USER` also includes the name of the user library file with which the long pointer is associated. Finally, the type `TRANS` is used to refer to a location in NM code which was translated from CM. Although the original CM code came from either a CM program file or one of the group, `PUB` or `SYS` SL files, the type `TRANS` gives no information about which one. A conversion function, NMTOCMNODE, can be used to convert NM TRANS addresses to CM logical code pointers, which reveal the originating CM code locations. Refer to appendix C for a discussion of CM object code translation node points and breakpoints in translated CM code. Finally, the types CST and CSTX do not apply to NM code. The analogous NM type is simply an NM LPTR.

Extended Address Types

The *extended addres* (EADDR) type is available for cases where the 32-bit offset part of a long pointer isn't large enough. An EADDR is a dotted pair, where the offset part to the right of the dot is 64 bits wide. An EADDR is effectively equivalent to an LPTR when it's offset part is representable in 32 bits. The *secondary address* (SADDR) type is a special form of EADDR, where the dotted pair is interpreted as a disk LDEV and disk byte offset. This is currently the only instance where an extended address is necessary.

Type Classes

All of the elementary data types introduced above are organized into type classes. These classes are particularly useful when defining parameters to functions and macros. By declaring a parameter to be of a particular type class, all actual values passed are automatically checked to be a member of the class.

The type tables below give the names of the type classes and show which elementary types belong to them.

Type Table

Class		Type	
INT		S16	Signed 16-bit integer.
INT		U16	Unsigned 16-bit integer.
INT		S32	Signed 32-bit integer.
INT		U32	Unsigned 32-bit integer.
INT		S64	Signed 64-bit integer.
BOOL		BOOL	Boolean.
STR		STR	Variable-length character string.
PTR		SPTR	Short pointer (offset).
PTR	LONG		Long pointer subclass. See table below.
EADDR		EADDR	Extended address.
EADDR		SADDR	Secondary address.

Long Pointers

Class			Type	
LONG			LPTR	Long pointer
LONG	CPTR			Code pointers
LONG	CPTR	LCPTR		Logical code pointers
LONG	CPTR	LCPTR	PROG	Program file
LONG	CPTR	LCPTR	GRP	Program group library
LONG	CPTR	LCPTR	PUB	Program account library
LONG	CPTR	LCPTR	LGRP	Logon group library
LONG	CPTR	LCPTR	LPUB	Logon account library
LONG	CPTR	LCPTR	SYS	System library: SL (CM), NL (NM)
LONG	CPTR	LCPTR	USER	User library (NM)
LONG	CPTR	LCPTR	TRANS	Translated object code (NM)
LONG	CPTR	ACPTR		Absolute Code Pointers
LONG	CPTR	ACPTR	CST	Absolute CST (CM only)
LONG	CPTR	ACPTR	CSTX	Absolute CSTX (CM only)

Literals

Literals represent specific values of one of the data types supported by System Debug. This section explains how to construct and interpret literals.

Numeric Literals

Numeric literals are a sequence of digits that are valid in the indicated radix. If the digits are not preceded by one of the base prefix characters, **%**, **#**, or **$**, the current input base is assumed.

Examples of valid numeric literals are the following:

```
#2048
$fff
%1762
26
```

The type of a numeric literal is determined by the smallest amount of storage required to store the value and by whether or not the literal is treated as being signed. The presence of a preceding minus sign, which must always precede the base prefix character, does not affect the sign of the literal. Such minus signs are treated as unary operators and are not considered to be parts of literals.

Octal and hex literals are considered to be signed if the representation of the unsigned digits fits into the natural word size of the current mode of System Debug (16 bits for CM, 32 bits for NM), and the high-order bit of the word is 1. Decimal literals are always unsigned.

Examples:

```
$nmdebug > env outbase '#'  /* set output base to decimal
#nmdebug > wl $ffffffff     /* S32 - sign bit 1, NM word size
#-1
#nmdebug > wl $ffff         /* U16 - sign bit 1, but not NM word size
#65535
#nmdebug > cm               /* switch to CM
#cmdebug > wl $ffff         /* S16 - sign bit 1, CM word size
#-1
#cmdebug > wl $ffffffff     /* U32 - sign bit 1, but not CM word size
```

```
#4294967295
#cmdebug >
```

Pointer Literals

Short pointer literals are represented by numeric literals. Essentially, this means that wherever a short pointer is required, a numeric literal that fits in 32 bits is accepted and is silently converted to the type `SPTR`.

Long pointer literals of type `LPTR` are entered as a pair of (32-bit) numbers separated by a dot, forming the so-called dotted pair. Long pointer literals are entered in the form *sid.offset*. When the *offset* part exceeds 32 bits, the type of the literal becomes `EADDR`.

Examples are:

`$c0002040`	short pointer literal
`3f.204c`	long pointer literal (SID=`3f`, offset=`204c`)

String Literals

String literals are formed by enclosing an arbitrary sequence of ASCII characters within either single quotes (') or double quotes (").

The same type of quote used to start the string (single or double) must be used to terminate it. For example, `'abc'` and `"abc"` are valid string literals, but `'abc"` is not.

A string which is defined with single quotes can contain one or more double quotes within the string body, and vice versa. For example, `"don't fret"` and `"SEG'ONE"` are valid strings.

In order to include the same quote character that is used as the string delimiter within the string itself, that quote character should appear in duplicate within the string. For example, the apostrophe in`'don&'&'t'`comes out as `don't`.

Examples of string literals are:

```
'Rufus T. Firefly'

"OB'"

'xltypes:pib_type.parent'

'The sun isn''t shining and I''m feeling so sad.'
```

Regular Expression String Literals

A special class of string literals called regular expressions is formed by enclosing an arbitrary sequence of characters with the backquote character (`). Refer to appendix A for a discussion of how patterns and regular express can be constructed for use in pattern matching.

Operators

An operator denotes an operation to be performed on existing values to create a new value of a particular type.

Operators are classified as arithmetic, Boolean, relational, address, and concatenation. A particular operator symbol may occur in more than one class of operators. For example, the symbol '+' is an arithmetic operator representing numeric addition, as well as string concatenation.

The table below summarizes the System Debug supported operators by operator class, and lists the possible operand and operator result types. The following subsections discuss the operators in detail.

Operators

Class	Operator	Operand Types	Result Types
Arithmetic	`+` (addition) `-` (subtraction) `*` (multiplication) `/` (division, quotient) `MOD` (division, modulus)	`INT, PTR`	`INT, PTR`
Boolean	`AND` (logical and) `OR` (logical or) `NOT` (logical not)	`BOOL, INT`	`BOOL`
Bit	`BAND` (bitwise and) `BOR` (bitwise or) `BNOT` (bitwise not) `<<` (left shift bits) `>>` (right shift bits)	`INT, PTR`	`INT, PTR`
Relational	`<` (less than) `<=` (less than or equal to) `=` (equal) `<>` (not equal) `>=` (greater than or equal to) `>` (greater than)	`BOOL, INT, PTR, STR`	`BOOL`
Address	`[ ]` (indirection)	`PTR`	`U16, U32`
String	`+` (concatenation)	`STR`	`STR`

Arithmetic Operators

Arithmetic operators perform integer arithmetic. The operators include the familiar `+`, `-`, `*`, `/`, and `MOD`. The operator `/` computes the integer quotient of two numbers, while `MOD` computes the remainder. The result of `MOD` is always nonnegative, regardless of the sign of the left operand. This implementation of `MOD` is the same as that in HP Pascal, which defines the result of `i MOD j, j > 0`, to be

```
i - k * j
```

for some integer `k`, such that

```
0 <= i MOD j < j.
```

The operation `i MOD j`, where `j <= 0`, is illegal.

Unary minus is also allowed, but note that the `-` operator must precede any base prefix character for numeric literals. This means that

```
-#32767
```

is allowed, but

```
#-32767
```

is not.

Arithmetic operands are restricted to the classes `INT` and `PTR`. In general, the types of the operands determine the result type of an arithmetic operation. In certain cases, one of the operands may be converted to another type before the operation is performed (see the following discussion).

Arithmetic on the INT Class

When both operands are of the `INT` class, the result of the arithmetic operation is also an `INT`. The type of the result is the largest type of the two operands, unless this type is not large enough to represent the result. In this case, the next larger type that can hold the result is used. The order of the two operands does not affect the result type.

The `INT` types are shown below in order of size:

smallest: `S16, U16, S32, U32, S64` :*largest*

The following examples illustrate the result types of some simple arithmetic operations.

```
  2      +     5      =    7              1     +    65535    =    65536
(U16)        (U16)       (U16)          (U16)      (U16)          (U32)

  2      -     5      =   -3              1     -    65535    =   -65534
(U16)        (U16)       (S16)          (U16)      (U16)          (S32)
```

Pointer Arithmetic

Arithmetic between a pointer and an integer is just like arithmetic between two integers, except only the offset part of a pointer contributes to the operation. With short pointers, only the (unsigned) low-order 30 bits are used. With long pointers, the entire 32-bit offset is used, treated as a `U32`. With extended address pointers, the 64-bit offset is used. The type of the result is that of the pointer, with the same bits that contributed to the computation being replaced by the result. Negative results, and results that cannot be represented with the available bits, cause an overflow condition.

The most common arithmetic operation between two pointers is subtraction, and the result is of type `S32` or `S64`. Other arithmetic operations may be performed between two pointers, but both pointers, whether long, short or extended, must reference the same space IDs. As with pointer/integer arithmetic, only the low-order 30 bits of a short pointer's offset contribute to the operation. The result is placed back in the same bits of the larger of the two operands, when they differ in size, which determines the result type. Note that if the two pointers are logical, their types must be identical due to the space ID check mentioned above.

Boolean Operators

The Boolean operators are `AND`, `OR`, and `NOT`. They perform logical functions on Boolean and integer operands and produce Boolean results. Integer operands are considered to be `FALSE` if they are 0, otherwise they represent `TRUE`.

The operation of the Boolean operators is defined below.

`AND` Logical and. The evaluation of the two Boolean operands produces a Boolean result according to the following table:

a	b	a AND b
T	T	T
T	F	F
F	T	F
F	F	F

`OR` Logical or. The evaluation of the two Boolean operands produces a Boolean result according to the following table:

a	b	a OR b
T	T	T
T	F	T
F	T	T
F	F	F

`NOT` Logical negation. The Boolean result is the logical negation of the single Boolean operand as defined in the following table:

a	NOT a
T	F
F	T

Examples of the use of Boolean operators are listed below:

```
NOT 0              result = TRUE
NOT 6              result = FALSE
1 AND 0            result = FALSE
1 AND 6            result = TRUE
(1<2) OR (4<2)     result = TRUE
```

Bit Operators

The bit operators are `BNOT`, `BAND`, `BOR`, `<<` (shift left), and `>>` (shift right). They perform bitwise logical operations on their operands and return the result as the type of the largest operand type.

BAND, BOR, and BNOT

These operators perform the indicated logical operation bit-by-bit on their operand(s), which are treated as unsigned integers of the appropriate size. When the sizes of the operands differ, they are aligned at the rightmost bits, with the smaller operand extended on the left with zeros. When a long pointer and an extended address are `BAND`ed or `BOR`ed together, the operation is performed separately on the SID and offset parts, with the offsets aligned at the right.

For example, when a `U16` is `BAND`ed with a `U32`, the `U16` is treated as a `U32` whose high-order 16 bits are all zero.

The definitions of the logical operations `BAND`, `BOR`, and `BNOT`, are the same as those for the Boolean operators `AND`, `OR`, and `NOT`, respectively, where the Boolean operands `TRUE` and `FALSE` are represented by the integer values 1 and 0, respectively.

<< and >>

These operators shift the first operand (the *shift operand*) left or right by the number of bits specified by the second operand (the *shift count*). The type of the result is the same as that of the first operand. For right shifting, if the shift operand is signed (`S16` or `S32`), sign extension is used when shifting. Otherwise, zeros move in from the left. For left shifts, zeros always move in from the right. Negative shift counts reverse the direction of the shift.

Relational Operators

The relational operators `<`, `<=`, `=`, `<>`, `>=`, and `>` compare two operands and return a Boolean result. Unless the comparison is for strict equality (`=` or `<>`), the operands must be members of the same primary type class (`INT/BOOL`, `STR`, or `PTR`).

Comparisons of integers and/or Booleans are based on the normal mathematical order of the integers, substituting 0 for `FALSE` and 1 for `TRUE`.

Comparisons between two long pointers are performed by first comparing their SIDs and, if equal, comparing their offsets, with each comparison being made as if the pointer parts were of type `U32`. Two short pointers are compared as if they were of type `U32`. When a short pointer is compared to a long pointer, the short pointer is first converted to a long pointer, and the comparison is then made between the two long pointers. Extended addresses behave similarly to long pointers in comparisons.

A comparison between two pointers with different SIDs is considered to be invalid unless the comparison is for strict equality (`=` or `<>`). System Debug recognizes the two special nil pointers `0` and `0.0`. These may only be involved in comparisons for strict equality, and `0` is considered to be equal to `0.0`.

Examples of pointer comparisons are listed below:

```
wl 1.200 < 1.204             TRUE
c0000200 >= c0000100         TRUE
```

```
1.200 < 2.30                   invalid
0.0 = sptr(0)                  TRUE
a.0 = sptr(0)                  FALSE
```

String comparisons are performed character by character, using the order defined by the ASCII collating sequence. If the two strings are not the same length, but are equal up to the length of the shorter one, the shorter string is considered to be less than the other.

Examples of string comparisons are listed below:

```
"abc" < "abcde"                TRUE
"Big" <= "Small"               TRUE
"Hi Mom" = "Hi " + "Mom"        TRUE
```

Indirection Operator

Square brackets ([ ]) are used as the indirection operator to return the value at the address they enclose.

The syntax of the indirection operator is shown below.

Note Please note that the non-bold square brackets in the following table are used to denote optional syntax, and are not meant to represent the literal square brackets (presented here in bold) of the indirection operator.

Indirection	Default Alignment	Return Type
[[*prefix*] [`VIRT`] *virtaddr*]	4 byte	(`S32`) 4 bytes
[[*prefix*] `REAL` *realaddr*]	4 byte	(`S32`) 4 bytes
[[*prefix*] `SEC` *ldev.offset*]	4 byte	(`S32`) 4 bytes
where [*prefix*] can be any one of the following:		
`BYTE`	byte-aligned	(`U16`) 1 byte
`U16`	2-byte-aligned	(`U16`) 2 bytes
`S16`	2-byte-aligned	(`S16`) 2 bytes
`LPTR`	4-byte-aligned	(`LPTR`) 8 bytes

These additional address specifications are supported (*without* the prefix):

[`ABS` [*offset*]]	(`S16`) 2 bytes
[`DL` [*offset*]]	(`S16`) 2 bytes
[`DB` [*offset*]]	(`S16`) 2 bytes
[`Q` [*offset*]]	(`S16`) 2 bytes
[`S` [*offset*]]	(`S16`) 2 bytes
[`P` [*offset*]]	(`S16`) 2 bytes
[`DST` *dst.offset*]	(`S16`) 2 bytes
[`CST` *cst.offset*]	(`S16`) 2 bytes
[`CSTX` *cstx.offset*]	(`S16`) 2 bytes
[`CMLOG` *lcptr*]	(`S16`) 2 bytes

Address specifications for the indirection operator contain an *address mode keyword*. All address modes can be used in both NM and CM.

The default address mode is `VIRT` (NM virtual address). Virtual addresses can be specified as short pointers, long pointers, or full NM logical code addresses.

`REAL` mode addresses physical memory in the HP Precision Architecture machine.

`SEC` mode addresses secondary storage. The address is always specified in the form of a long pointer or extended address to indicate the LDEV and byte offset.

`VIRT`, `REAL`, and `SEC` mode addresses are always automatically 4-byte-aligned (backwards to the nearest NM word boundary) before any data is retrieved. The indirect contents result value is returned as a signed 32-bit (`S32`) value.

Additional address modes provide access to compatibility mode data structures. In these modes, addresses are interpreted as CM word (16-bit-alignment) addresses, and the indirect contents result value is returned as a signed 16-bit (`S16`) value. The following CM modes are supported:

- `ABS` mode accesses emulated compatibility mode bank 0 addresses. This terminology is derived from absolute memory addressing in the HP 3000 architecture.
- `DL` mode addresses are DL-relative.
- `DB` mode addresses are DB-relative.
- `Q` mode addresses are Q-relative.
- `S` mode addresses are S-relative.
- `P` mode addresses are P-relative.
- `DST` mode accesses a word at the specified data segment and offset.
- `CST` mode accesses a word at the specified CST code segment and offset.
- `CSTX` mode accesses a word at the specified CSTX code segment and offset.

Since the default addressing mode is `VIRT`, a special CM mode `CMLOG` is provided to indicate that the address is a full CM logical code address.

Note

Nesting of indirection operators uses a significant amount of stack space. A stack overflow could occur if the user's stack is small and a large number of nested indirection operators are used.

Indirection Operator Examples:

`$nmdebug > w1 [r25]` `$400c6bd0`	Contents of virtual address, contained in register R25.
`$nmdebug > w1 [400c6bd0]` `$3f`	Contents of virtual address, specified as a short pointer.
`$nmdebug > w1 [r25]` `$3f`	Indirect operator can be nested.
`$nmdebug > w1 [3dc.204c]` `$f4000`	Contents of virtual address, specified as a long pointer.
`$nmdebug > w1 [HPFOPEN+2c]` `$6bcd3671`	Contents of virtual address, specified as a NM logical address.
`$nmdebug > w1 [REAL tr1]` `$2cb20`	Contents of real memory address, which is contained in register TR1.
`$nmdebug > w1 [SEC 1.0]` `$804c2080`	Contents of secondary storage at address: LDEV 1 offset 0.
`$nmdebug > w1 [c0004bc1]` `$804c2080,`	Contents of virtual address which is automatically 4-byte-aligned back to address c0004bc0.
`$nmdebug > w1 [byte c0004bc1]` `$4c`	Contents of the byte at byte virtual address c0004bc1.
`$nmdebug > w1 [u16  c0004bc1]` `$804c`	Contents of two bytes (as unsigned) at 2-byte-aligned address c0004bc0.
`$nmdebug > w1 [LPTR 402d5c63]` `$a.472280`	Contents of eight bytes found starting at 4-byte-aligned address 402d5c60, returned as a long pointer.
`$nmdebug > w1 [S16 real 3d3]` `$3fff`	Contents of two bytes (as signed) found in real memory at 2-byte-aligned memory address 3d2.
`$nmdebug > w1 [BYTE REAL 3d3]` `$ff`	Contents of the byte found in real memory at address 3d3.
`$nmdebug > w1 [LPTR REAL 4c]` `$31c.2200`	Contents of eight bytes found starting at 4-byte-aligned address 3d0, returned as a long pointer.
`$nmdebug > w1 [REAL 4c].[REAL 50]` `$31.2200`	Same as above.
`$cmdebug > w1 [DST 22.203]` `%20377`	Contents of data segment 22 offset 203.

Indirection Operator Examples:

`$cmdebug > w1 [S-2]` `%0`	Contents of S-2.
`$cmdebug > w1 [cmlog fopen+3]` `%213442`	Contents of the instruction found at CM logical code address `FOPEN`+3.
`$nmdebug > w1 [cst 12.432]` `$6`	Contents of code segment 12 offset 432.
`$nmdebug > w1 [cst %12.%432]` `$6`	Same as above but from NM instead of CM.
`$nmdebug > w1 [virt CSTVA(%12.%432)]` `$6`	Same as above. The `CSTVA` function is used to translate `CST` %12.%432 to its virtual address.
`$cmdebug > w1 [Q-3]` `%17`	Contents of Q-3.
`$nmdebug > w1 [virt dstva(sdst.q-3)]` `$f`	Same as above. Contents of Q-3.

Concatenation Operator

The concatenation operator (`&+`) concatenates two string operands. Examples of the use of this operator are listed below:

```
$nmdebug > var s1 = "abc"
$nmdebug > var s2 = "def"
$nmdebug > var s3 = s1 + s2
$nmdebug > wl s3
abcdef
$nmdebug > var s4 = s3 + '123'
$nmdebug > wl s4
abcdef123
$nmdebug >
```

Expressions

Expressions are formulas for computing new values from a collection of operators and their operands. Operator precedence, in combination with the use of parentheses, determines the order of expression evaluation. When two or more operators of the same precedence occur at the same level of evaluation, they are evaluated from left to right.

Expression operands may be literals, variables, functions, macros, and symbolic procedure names, each of which denotes a value of some type. Examples of valid expressions are:

`$12`	Simple numeric literal
`pc + 4`	Predefined variable
`FOPEN + 12`	Symbolic procedure name
`[dst 2.104]`	Indirection - contents of DST 2.104
`(count < 5) and (q>200)`	Boolean expression with relational operators
`strup('hello') + "MOM"`	Standard function result

Operator Precedence

The precedence ranking of an operator determines the order in which it is evaluated in an expression. The levels of ranking are:

Precedence	**Operators**
highest	`[ ]`
.	`NOT, BNOT`
.	`<<, >>, BAND, BOR`
.	`*, /, MOD, AND`
.	`+, -, OR`
lowest	`<, <=, =, >, >=, <>`

Operators of highest precedence are evaluated first. For example, since * ranks above +, the following expressions are evaluated identically:

```
(x + y * z)    and    (x + (y * z))
```

When operators in a sequence have equal precedence, evaluation proceeds from left to right. For example, each of the following expressions are evaluated identically:

```
(x + y + z)    and    ((x + y) + z)
```

Variables

System Debug provides variables in which values may be stored for use as operands in expressions. Variable names must begin with an alphabetic character, which may be followed by any combination of alphanumeric, apostrophe ('), underscore (_), or dollar sign ($) characters. Variable names are case insensitive and may not exceed 32 characters.

System Debug supports two levels of variable scoping: global and local. Global variables are defined by the `VAR` command and exist for the lifetime of the System Debug session (unless removed by the `VARD` command):

```
$nmdebug > var v1 $2f
$nmdebug > var s2 = "hello mom"
$nmdebug > var p3:lptr = 2f.102c
```

The type of a variable is determined by the type of the expression which computes its value. The optional :*type* syntax which follows the variable name imposes a check on the expression type for that particular assignment only. It does not establish the variable's type over its entire lifetime. A value of a different type may be assigned to the same variable by a subsequent `VAR` command.

Local variables are defined by the `LOC` command only from within macro bodies and exist only for the lifetime of the macro in which they are defined. Local variable definitions nest with macro execution level, and they supercede global variables of the same name. Note that local variables normally are not visible from outside the macro in which they are created (that is, from macros called by the one in which they are created). To make local variable visible to called macros, the environment variable NONLOCALVARS must be TRUE.

```
loc v1 200
loc s2 = "new string"
```

Note that, although a macro cannot reference the value of a global variable once a local variable of the same name has been defined, it may change the global value by using the VAR command instead of LOC.

!variable

The use of the letters a through f to denote hex digits implies the possibility of ambiguity between hex constants and variable names composed of just these characters. System Debug warns the user of this occurrence when such variables are defined by the VAR and LOC commands, but uses the value of the constant when the name occurs in an expression. This may be overridden by preceding the variable name with the exclamation point as follows:

```
$nmdebug > var a 123
Variable name collides with hex numeric literal.  (warning #55)
  Name: "a"
$nmdebug > wl a+1     /* a is a hex constant here
$b
$nmdebug > wl !a+1    /* !a references the variable a
$124
$nmdebug >
```

Environment Variables

System Debug provides a large collection of predefined environment variables, the names of which are reserved and may *not* be replaced by user-defined variables with the VAR and LOC commands.

Several environment variables provide access to the current System Debug execution environment. Examples of these variables include the current input radix and the prompt string. Other environment variables are used to access key components of the state of the machine being examined. For example, all of the machine registers defined in the HP 3000 and HP Precision Architectures are available as environment variables. Subject to the context of use, some of these variables may be set by the user with the ENV command. The environment variables that correspond to the CM and NM machine registers are also accessible through the MR (modify register) and DR (display register) commands. All environment variables may be read (accessed) as expression operands. Some environment variables also require privileged mode for modification access.

The ENV command in chapter 4 gives a detailed description of each of the predefined environment variables and specifies which ones may be modified and which ones are read-only.

Predefined Functions

A large collection of predefined functions exist that provide access to the machine being debugged, as well as those which perform various operations on values of the data types supported by System Debug.

Syntactically, a function reference appears as an operand in an expression and is denoted by its name, followed optionally by a list of parameters surrounded by parentheses. Multiple parameters are separated from one another by either spaces or commas. Functions evaluate to a single value of some type.

Detailed descriptions of all the System Debug predefined functions may be found in chapter 8.

Macros

System Debug supports an extensive macro facility that allows users to define a sequence of commands that may be invoked either as a command or as a function in an expression. The MAC command is used to define a macro, as the following examples illustrate:

```
$nmdebug > mac double (n=2) { return n * 2 }
$nmdebug > mac formattable (entry=1) { ... }
```

Reference to macros as functions in expressions look exactly like references to predefined functions:

```
$nmdebug > wl double (1)
$2
$nmdebug > wl double (double (1))
$4
$nmdebug >
```

Macro parameters may be defined as being either required or optional (as indicated by the presence of default parameter values in the macro definition). When all of a macro's parameters are optional and it is referenced as a function without any parameters, the enclosing parentheses are optional:

```
$nmdebug > wl double ()
$2
$nmdebug > wl double
$2
$nmdebug >
```

When macros are used as commands, the parentheses surrounding the parameters may be omitted:

```
$nmdebug > formattable 3
...
$nmdebug > formattable (3)
...
```

However, since macro command parameters may still be surrounded by parentheses as an option, care must be used when the first parameter is an expression that begins with a parenthesis of its own. In this case, the parenthesis is seen as the beginning of a parenthesized list of command parameters, and not as belonging to the expression for the first parameter. Thus, parameters surrounding the entire command list are required when the first parameter starts with a parenthesis:

```
$nmdebug > formattable (current_entry + 1) * 2      /* wrong
$nmdebug > formattable ((current_entry + 1) * 2)    /* right
```

Procedure Name: Symbols

Symbolic procedure names, which represent logical code addresses of the type class `LCPTR`, may be used as operands in expressions. Thus, to determine the virtual address of the procedure `FOPEN`, the `WL` command may be used as follows:

```
$nmdebug > wl FOPEN
SYS $a.345498
$nmdebug >
```

In the above example, since no System Debug variable named `FOPEN` was found, the expression evaluator searched for the symbol in the currently loaded program file and libraries, finding it in `NL.PUB.SYS`.

Procedure name symbols stand for slightly different locations depending on the mode of System Debug. In CM, they stand for the starting address of the code bodies that they name. In NM, they stand for the entry address. Since compilers may emit constants before executable instructions in System Object Modules, breakpoints should always be set at entry addresses. To find the entry address of a CM procedure, the procedure symbol name should be prefixed by the question mark (`?`), as explained below.

When searching program files and libraries for procedure symbols, System Debug behaves differently depending on its mode. In NM, procedure names are case sensitive, and the program file and libraries are searched in the following order:

NM search order: *first* ... `PROG, GRP, PUB, USERs, SYS` ... *last*

In CM, procedure names are case insensitive, and the following search order is used:

CM search order: *first* ... `PROG, GRP, PUB, LGRP, LPUB, SYS` ... *last*

Each of the above search orders, which visit all currently loaded files, is known as a full search path. Note that this order is the same as that used by the CM and NM loaders in satisfying external references in program files and libraries, as specified in the `LIB=` and `LIBLIST=` parameters of the `RUN` command.

Variations of certain commands, such as `BREAK, DISPLAY, MODIFY, TRANSLATE, FREEZE,` and `UNFREEZE`, restrict the search path for procedure name symbols in their parameters to a single loaded code file. In addition, certain coercion functions (`PROG, GRP, PUB, LGRP, LPUB, SYS`) also restrict the search path for procedure name symbols in their parameters to a single loaded code file. This allows references to procedure symbols in a particular library, that would otherwise be inaccessible if they were redefined in preceding libraries on the full search path.

Two symbol tables are present in NM executable libraries and program files. The first symbol table is called the Loader Symbol Table (LST) and is utilized by the native mode loader. It contains only exported level 1 procedure names, which are hashed to support fast symbol name lookups.

The second symbol table is called the System Object Module (SOM) symbol table. This symbol table contains all compiler-generated symbols (procedure, data, internal labels, try/recover, and so on), which are maintained in no particular order. Any lookup attempt must be made sequentially through the symbols.

If the SOM symbols are being searched and an ambiguous name is entered, the first symbol that matches the name found during the sequential search of the symbol table is used.

The symbol table used by the expression evaluator for symbol lookups is based on the environment variable `LOOKUP_ID`. The variable may take on any of the following values. (The default setting is `LSTPROC`.)

`UNIVERSAL`	Search exported procedures in the SOM symbols.
`LOCAL`	Search nonexported procedures in the SOM symbols.
`NESTED`	Search nested procedures in the SOM symbols.
`PROCEDURES`	Search local or exported procedures in the SOM symbols.
`ALLPROC`	Search local/exported/nested procedures in the SOM symbols.
`EXPORTSTUB`	Search export stubs in the SOM symbols.
`DATAANY`	Search exported or local data SOM symbols.
`DATAUNIV`	Search exported data SOM symbols.
`DATALOCAL`	Search local data SOM symbols.
`LSTPROC`	Search exported level 1 procedures in the LST.
`LSTEXPORTSTUB`	Search export stubs in the LST.
`ANY`	Search for any type of symbol in the SOM symbols.

Note Using the SOM symbol table is noticeably slower than using the LST.

!procedure_name

Just as System Debug variable names composed of only the letters "A" through "F" may conflict with hex constants, so may procedure name symbols. Preceding such name symbols with an exclamation point makes the expression scanner see the name as a symbol instead of a hex constant. However, System Debug variable names take precedence over procedure name symbols, so the variable name `ADD` makes a procedure of that name invisible in expressions. In this case, the functions `CMADDR` and `NMADDR` can be used to locate the procedure names.

?procedure_name

Sometimes the address that a procedure name symbol represents is not appropriate for a particular use. By preceding a procedure name symbol with a question mark, a different address is returned, depending on the mode of System Debug.

In CM, `?procedure_name` returns the entry point address for the named procedure instead of its start address. This is the address of interest when setting CM breakpoints. In NM, the question mark prefix returns the export stub address of the procedure. This is the entry location used by callers from external modules. Please refer to the *Procedure Calling Conventions Reference*

Manual (09740-90015) for a detailed discussion of export stubs and native mode procedure organization.

Operand Lookup Precedence

When expressions are scanned and parsed, they are ultimately broken down into a series of tokens, which represent either operators or operands. The preceding sections of this chapter introduced all the possibilities for operand tokens in expressions, thereby answering the question, "What sorts of things can be used as operands?" This section deals with the converse: "Given an operand, what sort of thing is it?"

The process of evaluating an operand token can be modeled by a list of possible interpretations of a token. The unknown token is tested against each of the possibilities in the list, in the specified order, with the first match determining the token's meaning.

The following list determines the interpretation of an operand token:

1. Test for a string literal or a numeric literal in the current input base.
2. Test for a predefined variable.
3. Test for a user-defined variable.
4. Test for a predefined function.
5. Test for a macro.
6. Test for a procedure name symbol in the current mode, subject to the search path in effect.
7. If still unresolved, fail.

There are two operand modifiers that, when prefixed to an operand, alter the above search order for that operand. The exclamation point (!) signals that the operand to which it is prefixed is not to be treated as a numeric literal. This prevents the token from being mistaken as a hex constant and initiates the operand search at step 2.

A question mark prefix (?) indicates that the operand is to be treated as a procedure name symbol and that the entry point or export stub address of the named procedure is being referenced instead of its starting address. The search for such symbols begins with step 6.

Command Line Substitutions

Command line scanning proceeds from left to right and is done in two phases. The first preprocessing phase scans a command line for the vertical bar character (`|`), which introduces the following syntax:

`|`*expression*`[:`*fmtspec*`][~]`

When the command preprocessor recognizes the above syntax, it removes all the characters associated with it from the command line and replaces them with text representing the value of the expression. The *expression* part of the substitution syntax may be any valid expression as previously described in this manual. In particular, there are no special restrictions placed on command line substitution expressions.

The optional :*fmtspec* represents special formatting directives that may be used to control the formatting of the value of the expression when it is converted to characters and inserted back into the command line. *Fmtspec* is always specified as a string literal and is fully defined by the `W` (`WRITE`) command in chapter 4.

The optional closing tilde (`~`) character is used to terminate the command line substitution string when it appears adjacent to text that is not to participate in the substitution. The tilde is always removed as part of the substitution.

During the preprocessing phase, a command line is scanned repeatedly until no command line substitutions are performed. Note that, after an individual substitution is performed, scanning continues after the point of substitution. If the substituted text causes another substitution (by containing a new vertical bar character), it is processed during the *next* scan of the command line.

The special meanings of both the vertical bar and the tilde are cancelled when they are immediately preceded by the backslash (\) escape character. After the preprocessing phase of command line scanning is finished, the escape characters are removed, leaving the following vertical bar or tilde by itself. The practice of using the escape character to remove the special meaning of some other character is known as *escaping*, and is often used in string literals, particularly in regular expressions. Refer to appendix A for a discussion of how patterns and regular expression can be constructed for use in pattern matching.

Command line substitutions are performed on every command line, including those which define macros. If a macro definition is to contain a command line substitution to be performed when the macro is executed, it should be escaped to prevent it from being performed when the macro is defined.

Command line substitution is subject to the current state of the `CMDLINESUBS` environment variable. If set to `FALSE`, command line substitutions are not performed.

Examples of command line substitutions are listed below:

Assuming the following declarations have been entered,

```
var grp   = 'PUB'
var acnt  = 'SYS'
var cmd   = 'SYMOPEN'
var const = $20
var n     = $1
```

the following examples demonstrate command line substitutions:

```
symopen myfile.|grp~.|acnt
```

becomes

```
symopen myfile.PUB.SYS
```

while

```
while n < |const:"#" do {cmd1;cmd2;cmd3}
```

becomes

```
while n < #32 do {cmd1;cmd2;cmd3}
```

which saves many searches for the constant. And

```
while |n < |const do {cmd1;cmd2;cmd3}
```

becomes

```
while $1 < $20 do {cmd1;cmd2;cmd3}
```

which will loop infinitely. Next consider the following:

```
$nmdebug > var n "mom"
$nmdebug > wl "|n"
mom
$nmdebug > wl "\|n"
|n
```

Note how the presence of the backslash cancels the command line substitution.

Aliases

Aliases may be established for command names, macros, and even other aliases. By defining an alias for one of these objects, one is merely specifying an alternative name by which the aliased object may be referred. Note that this defines an *alternative*, rather than a change, and affects no other aspect of the thing being aliased. For instance, the alias has no effect on the parameters of an aliased command. Once established, the alias name may be used wherever the original name is valid.

Command Lookup Precedence

The second phase of command line scanning is performed after the preprocessing phase, in which command line substitution is performed. In the second phase, the command name is extracted from the command line and is interpreted according to the following sequence:

1. Search for the command in the alias table. If found, repeat this process recursively with the aliased name until the search fails. Infinitely recursive aliases result in an error. Proceed with the aliased command name, if found.
2. Search for the command in the command table.
3. Search for the command in the window command table.
4. Search for the command in the macro definition table. If found, execute the macro as a command, discarding any macro return value.
5. If still unresolved, then fail.

Error Handling

System Debug employs an error stack for error messages and maintains the environment variable ERROR for detection of errors by control commands. When an internal error is detected, appropriate error messages are pushed onto the error stack and the variable ERROR is set to the error number of the last error generated.

While the highest-level error messages are typically displayed on the user's terminal, lower-level (intermediate) errors are usually pushed silently onto the error stack. All errors can be inspected with the ERRLIST command:

```
$nmdebug > dv 1234.98127345
$ VIRT 1234.98127344 $
Display error.  Check ERRLIST for details.  (error #3800)
$nmdebug > errl
$1:  Display error.  Check ERRLIST for details.  (error # 3800)
$1:  data read access error  (error #805)
$1:    READ_CMWORD   bad address: $ VIRT 1234.98127344
$1:  Virtual read failed  (error #6000)
$1:       VADDR= 1234.98127344
$1:  A pointer was referenced which contained a virtual address outside
of the bounds of an object.
$nmdebug >
```

The error stack can be reset (cleared) with the ERRDEL command:

```
$nmdebug > errd
```

The System Debug command interpreter (CI) checks the variable ERROR after each command is executed. When an error condition is detected (ERROR < 0), all pending commands (in loops, command lists, macros, and so on) are aborted. The command stack is flushed, and the outermost prompt is issued. Note that only negative ERROR values constitute an error. Positive values represent *warnings*, and do *not* cause command stack execution to cease.

The IGNORE command protects the next single command, command list, macro, or use file from being aborted if an error is detected. IGNORE has the same effect as the CONTINUE command of the MPE XL CI.

Although the IGNORE command prevents abnormal command termination, it does *not* automatically prevent generated errors from being displayed. The QUIET option of the IGNORE command suppresses the error messages as well.

While the IGNORE command affects just the following command or command list, the environment variable AUTOIGNORE may be set to TRUE to cause errors for all commands to be ignored and is equivalent to entering an IGNORE LOUD command before each one.

User-defined macros can take advantage of the error handling mechanism. A user error message can be pushed onto the error stack with the ERR command, and the ERROR variable can be explicitly set to a negative value. For example,

```
$nmdebug > ERR "a very nasty error happened"
$nmdebug > ENV error -125
```

Control-Y

System Debug allows the user to prematurely terminate command execution by entering a Control-Y (press and hold the CONTROL key and press Y). Command loops, display loops and modification loops can be interrupted with this mechanism.

When Control-Y is entered during window updates, interrupted output lines may disturb portions of the windows. When this occurs, redraw the windows with the RED (redraw) command.

Note

There is only one Control-Y handler per session. When Debug is entered, it takes ownership of the Control-Y handler. When Debug is exited, it returns the Control-Y handler to the process that owned it when Debug was entered.

If other processes are active in a session while Debug is being used, it is possible for one of the other process to steal Control-Y ownership from Debug. In this situation, when Debug exits it will, in effect, *steal* Control-Y back from the current owner and give it to the process that owned it when Debug was entered. If Control-Y is stolen from Debug, it is also possible to create infinite loops in Debug from which there is no way out (for example, "`while TRUE do {}`").

Both DAT and Debug rearm the Control-Y trap after every CI command (for example, the "`:`" command).

Command History, REDO

System Debug maintains a very short history of command lines in the form of a stack. Commands in the stack can be displayed with the `HIST` (or `LISTREDO`) command, and may be reexecuted with the `DO` command or edited prior to reexecution with the `REDO` command.

Commands read from outer level or interactive input are pushed onto the history stack. Currently, commands read from `USE` files are also pushed onto the stack. Commands executed as part of macro commands are *not* pushed.

Debug Input/Output: The System Console

Under normal circumstances, Debug Input/Output is typically directed to the user's terminal. However, during the following occasions, Debug I/O is redirected to the MPE XL system console:

- During the bootstrap process (until the system is up), all Debug I/O is directed to the system console.
- All system process debugging uses the system console.
- All job debugging uses the system console. The environment variable `JOB_DEBUG` allows jobs to enter Debug.
- The environment variable `CONSOLE_DEBUG` can be used to cause all processes that are entering Debug for the first time to use the system console.
- The environment variable `CONSOLE_IO` can be used to cause all debugging for the current process to be directed to the system console.
- The environment variable `TERM_LDEV` allows the use of any terminal for debugging. A privileged procedure, `DEBUG_AT_LDEV (ldev : ldev_type)`, is also available to enter the debugger and direct I/O to the specified terminal LDEV.

When Debug is using the system console, the following technique is recommended to prevent confusion while sharing the console with the CI:

```
$cmdebug > :restore
```

Running `RESTORE` prevents unwanted terminal reads from the console's CI.

See the `ENV` command for detailed descriptions of all of the environment variables mentioned above.

Automatic DBUGINIT Files

Debug supports the automatic execution of commands within special initialization files named `DBUGINIT`. These files must be in the form of a USE file as described by the `USE` command.

Debug first tests for an initialization file (`DBUGINIT`) in the same group Uand account as the program that is being debugged. Next, Debug looks for an initialization file in the user's logon group and account (if different).

Based on the existence of these special files, it is possible to execute initialization command files from both the program's group and account and the user's logon group and account.

The following initialization sequence is possible for Debug:

```
1) DBUGINIT.ProgGrp.ProgAcnt  (program group/account)
2) DBUGINIT.UserGrp.UserAcnt  (user's group/account)
```

Refer to chapter 9 for a discussion of initialization files used for DAT.

FINAL TRIM SIZE : 7.5 in x 9.0 in

3

System Debug Interface Commands and Intrinsics

Debug may be invoked directly through an integrated set of commands and intrinsics. All MPE V intrinsics are supported. In addition, several new intrinsics have been added to enhance the functionality of MPE XL and take advantage of the new debugger. The commands and intrinsics allow you to enter the debugger from three different paths:

- Directly from a command interpreter (CI) command in a session.
- From a program through an intrinsic call.
- From the system during an abnormal process termination (a process abort).

Many of the commands and intrinsics that make up the system debugger interface also allow you to specify an optional character string containing Debug commands . If supplied, this string is passed to Debug for execution as part of debugger initialization.

The MPE XL commands and intrinsics allow you to do the following:

- Enter Debug from a program or in a session directly from the CI.
- Generate stack trace upon demand from within a program.
- Execute a defined series of Debug commands from a session, job, or program.
- Arm a call to Debug to take place during the process abort sequence.
- Disarm the call to Debug during the process abort sequence.

The Debug commands and intrinsics are described in the following sections. For additional information, refer to the *MPE XL Commands Reference Manual* (32650-90003) and the
MPE XL Intrinsics Reference Manual (32650-90028).

Debug Interfaces

Debug may be invoked directly or indirectly: directly from the CI of a session, or from an intrinsic call within a program; indirectly through arming a call to Debug in the case of a process abort.

The MPE XL CI commands are identical to the MPE V commands, with the exception that the user may specify an optional command string to be passed to Debug when it is invoked. The following is a list of the available MPE XL CI commands and their syntax:

```
DEBUG [commands]

SETDUMP [DB [,ST [,QS]][;ASCII]][;DEBUG="commands"]

RESETDUMP
```

All intrinsics can be called from NM with the exception of `STACKDUMP'`. This intrinsic is not supported in native mode and is found only in the CM intrinsic file. Only those intrinsics available in MPE V are callable by the CM user. The following table summarizes which intrinsics are callable from compatibility mode (CM) and native mode (NM):

Callable From	**Intrinsic Name**
CM/NM	`DEBUG`
CM/NM	`RESETDUMP`
CM/NM	`SETDUMP`
CM/NM	`STACKDUMP`
CM	`STACKDUMP'`
NM	`HPDEBUG`
NM	`HPRESETDUMP`
NM	`HPSETDUMP`

Note that no HPSTACKDUMP intrinsic is present. It is intended that the user call HPDEBUG to produce a custom stackdump when desired.

Direct Calls

If you want to invoke Debug from the CI of the current session, use the `DEBUG` command. This command is implemented through intrinsics. The CI simply calls the `DEBUG` or `HPDEBUG` intrinsic. Note that this command requires privileged mode (PM) capability.

```
DEBUG
DEBUG/XL A.00.00

DEBUG Intrinsic at:  a.00702d74 hxdebug+$24
$1 ($25) nmdebug >
```

The following example shows a call to Debug with a command to display the registers and then return to the CI.

```
:DEBUG DR;C
DEBUG/XL A.00.00

HPDEBUG Intrinsic at: a.006b4104 hxdebug+$130
R0 =00000000 006b0000 006b4100 00000002 R4 =40221a80 40221638 402213d8 00000400
R8 =00000001 40200268 40221558 402215c4 R12=402213d4 00000000 00000000 00000000
R16=00000000 00000000 00000000 0000000c R20=00000000 0000000b 0000007f 40221a80
R24=40221add 00000001 00000001 c0200008 R28=0000000b 00000000 40221c58 00000000

IPSW=0006000f=jthlnxbCVmrQPDI  PRIV=0   SAR=0011 PCQF=a.6b4104   a.6b410

SR0=0000000a 00000188 0000000a 00000000 SR4=0000000a 00000188 0000000b 0000000a
TR0=00616200 00646200 00005600 00545274 TR4=40222168 00000001 00000001 00000018
PID1=0184=00c2(W) PID2=0000=0000(W)     PID3=0000=0000(W) PID4=0000=0000(W)

RCTR=ffffffff ISR=0000000a IOR=00000000 IIR=87e0211a IVA=000aa800 ITMR=35b49924
EIEM=ffffffff EIRR=00000000 CCR=0080
:
```

Debug may also be invoked with the `HPDEBUG/DEBUG` intrinsic calls from within any program. Native mode programs enter Debug assuming that the user will be viewing the native mode environment (program, stack, registers); this is referred to as NM Debug. Compatibility mode programs enter Debug assuming that the user will be viewing the compatibility mode environment; this is called CM Debug.

Process Abort Calls

You may arm a call to Debug which occurs in the event of a process abort. The call may be armed by:

- The SETDUMP command.
- The SETDUMP intrinsic.
- The HPSETDUMP intrinsic.

Once a SETDUMP command or intrinsic has been issued, all new processes created are affected. Both the setdump attribute and the DEBUG command string are inherited by new child processes. This feature may be disarmed by the following:

- The RESETDUMP command.
- The RESETDUMP intrinsic.
- The HPRESETDUMP intrinsic.

If the Debug process abort call has not been armed through one of the SETDUMP interfaces, and a process abort occurs, an abbreviated stack trace is produced. This abbreviated trace shows only the most recently called procedure in the program file and in each library being used. This is done for both the CM and NM stacks.

The following is an example of a CM program aborting *without* invocation of SETDUMP.

```
:run cmbomb
**** PROGRAM ERROR #4 :INTEGER DIVIDE BY ZERO
ABORT: CMBOMB.DEMO.TELESUP
**** PROCESS ABORT TRACE ****

NM SYS   a.006d7798 dbg_abort_trace+$30
   CM SYS  %  27.261    SWITCH'TO'NM'+4        SUSER1
   CM PROG %   0.1215   TEST_ARITH_TRAP+24     SEG'
PROGRAM TERMINATED IN AN ERROR STATE.  (CIERR 976)
:
```

The following example is the same as above except that the code was compiled with a native mode compiler.

```
:run nmbomb
**** Integer divide by zero (TRAPS 30)

ABORT: NMBOMB.DEMO.TELESUP
**** PROCESS ABORT TRACE ****

NM PROG  191.00006b20 test_arith_trap+$28
PROGRAM TERMINATED IN AN ERROR STATE.  (CIERR 976)
:
```

If the SETDUMP command (or intrinsic) is invoked before running this program, a full dual stack trace and a register dump is produced when the process aborts. Consider the following example:

```
:setdump
:run nmbomb
**** Integer divide by zero (TRAPS 30)

ABORT: NMBOMB.DEMO.TELESUP
**** PROCESS ABORT STACKDUMP FACILITY ****

       PC=191.00006b20 test_arith_trap+$28
NM* 0) SP=40221178 RP=191.00006e8c do_traps+$2ac
NM  1) SP=40221140 RP=191.00007c08 PROGRAM+$360
NM  2) SP=402210f8 RP=191.00000000
     (end of NM stack)

R0 =00000000 00000000 00006e8f c1c60000 R4 =81c2b6c0 00000001 c0000000 00000000
R8 =00000000 00000000 00000000 00000000 R12=00000000 00000000 00000000 00000000
R16=00000000 00000000 00000000 00000061 R20=00000020 00000191 00000005 0000003a
R24=0000001a 00000000 00000005 40200008 R28=0000018d 00000000 40221178 00006b23

IPSW=0006ff0f=jthlnxbCVmrQPDI  PRIV=3   SAR=0000 PCQF=191.6b23   191.6b27

SR0=0000000a 0000000a 0000018d 00000000 SR4=00000191 0000018d 0000000b 0000000a
TR0=00616200 00646200 0000ac00 00545274 TR4=40221de8 00000001 00000001 00000022
```

```
PID1=018a=00c5(W) PID2=0000=0000(W)     PID3=0000=0000(W) PID4=0000=0000(W)

RCTR=00000000 ISR=00000191 IOR=00000000 IIR=b3202000 IVA=000aa800 ITMR=ad40a0fd
EIEM=ffffffff EIRR=00000000 CCR=0080

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

$2 ($22) nmdebug >
```

Note that in the above example, the user is left in Debug. At this point, the user is able to enter any Debug command. The process may even be resumed (see the `CONTINUE` command in chapter 4).

It is possible to specify what action should be taken when a process aborts by providing a list of commands for Debug to execute. In the following example, a simple message is printed if the process aborts.

```
:setdump ;debug="wl 'Oh my, our process is aborting !!!'"
:run cmbomb
**** PROGRAM ERROR #4 :INTEGER DIVIDE BY ZERO
ABORT: CMBOMB.DEMO.TELESUP
**** PROCESS ABORT STACKDUMP FACILITY ****

Oh my, our process is aborting !!!

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

Notice that the user was not left in Debug after the command string was executed. In order to be left in Debug, several criteria must first be met:

- The abort did not occur while in system code, and
- The process entered the abort code through a native mode interrupt. Such aborts are typically caused by arithmetic and code-related traps (see the XARITRAP and XCODETRAP intrinsics).

Most CM programs fail these checks and are returned to the CI without entering Debug.

The SETDUMP functionality is also accessible programmatically with the SETDUMP and HPSETDUMP intrinsics. Refer to the following pages for detailed descriptions and examples.

Debug Command and Intrinsic Descriptions

The commands and intrinsics used with the Stackdump system debugger interface are described on the following pages. The programming examples are written in Pascal. Refer to the appropriate language manual set for details of calling system intrinsics from other languages.

:DEBUG Command

PRIVILEGED MODE

Enters Debug from the CI.

Syntax

```
:DEBUG [commands]
```

Parameters

commands A series of Debug commands to be executed before the Debug prompt is displayed. The string may be up to 255 characters long. All text on the command line following `:DEBUG` is passed unaltered to Debug. Note that the commands should not be quoted.

Discussion

The `:DEBUG` command enters Debug directly from the session CI. Optional Debug commands may be entered on the command line, and they will be executed before the Debug prompt is displayed.

If the optional commands contain a Debug command that returns the user to the CI, any further commands are left pending on Debug's command stack. The next time Debug is entered, any pending commands are executed before the Debug prompt is displayed. If no commands were specified, Debug displays its prompt and waits for the user to enter interactive commands. This command is ignored in a job.

Example

The example below calls Debug to produce a stack trace and return to the CI.

```
:debug  trace;c
DEBUG XL A.00.00

HPDEBUG Intrinsic at: a.006b4104 hxdebug+$130
     PC=a.006b4104 hxdebug+$130
* 0) SP=40221c58 RP=a.006b8e7c exec_cmd+$73c
  1) SP=40221ac8 RP=a.006ba41c try_exec_cmd+$ac
  2) SP=40221a78 RP=a.006b8638 command_interpret+$274
  3) SP=40221620 RP=a.006bae5c xeqcommand+$1d0
  4) SP=40221210 RP=a.006b7604 ?xeqcommand+$8
       export stub: 7d.000068dc main_ci+$94
  5) SP=40221178 RP=7d.00007420 PROGRAM+$250
  6) SP=40221130 RP=7d.00000000
     (end of NM stack)
:
```

:RESETDUMP Command

Disarms the Debug call that is made during abnormal process termination.

Syntax

```
:RESETDUMP
```

Discussion

The `:RESETDUMP` command disarms the Debug call which is made during abnormal process termination. If the setdump feature was not previously armed by one of the Setdump intrinsics or commands, this command has no effect. The command affects all processes subsequently created under the current session or job. If performed in BREAK mode, existing processes are not affected by the command.

Example

Since there are no parameters or options for this command, the example is quite simple and straightforward:

```
:resetdump
:
```

:SETDUMP Command

Arms the Debug call that is made during abnormal process termination.

Syntax

```
:SETDUMP [DB [,ST [,QS] ] [;ASCII] [;DEBUG="commands"] ]
```

Parameters

commands	A quoted string of system Debug commands, up to 255 characters long. If not specified, this parameter defaults to a command string that produces a dual mode stack trace and a register dump.
DB, ST, QS, ASCII	These parameters are provided for compatibility with MPE V. If specified, they are ignored.

Discussion

The :SETDUMP command enables automatic execution of a set of Debug commands when a process terminates abnormally (aborts). This command affects all processes subsequently created under the current job or session. That is, the setdump attribute and the *commands* parameter are inherited by any new process.

During the process abort sequence, Debug executes the commands specified in the *commands* parameter. Any output is sent to the process's standard list file ($STDLIST). Any commands that require input generate an error message.

If the process that aborts is being run from a job, the process terminates after executing the command string. If the process is being run from a session, after the specified command string has been executed, Debug stops to accept interactive commands with I/O performed at the user terminal, contingent upon the following requirements:

- The abort did not occur while in system code, and

- The process entered the abort code through a native mode interrupt. Such aborts are typically caused by arithmetic and code-related traps (see the `XARITRAP` and `XCODETRAP` intrinsics).

Note CM programs usually fail these tests.

Once Debug accepts interactive input, you can enter any Debug command. You may choose to resume the process or have it terminate (refer to the `CONTINUE` command in chapter 4).

If the cause of the abort is a stack overflow, the command list is ignored and a stack trace is sent to `$STDLIST`, after which the process is terminated with no interactive debugging allowed.

Examples

The first example arms the Setdump feature. No parameters are specified, so the default command string is assumed (the default command string produces a stack trace and register dump).

```
:setdump
:
```

The following example also arms the Setdump feature but specifies a list of commands to be executed if the process aborts.

```
:setdump ;debug="w 'Process abort at ';w pc; wl ' ' nmpath(pc)"
:
```

DEBUG Intrinsic

Enters Debug.

Callable from: NM, CM

Syntax

```
DEBUG;
```

Discussion

The `DEBUG` intrinsic calls Debug from an interactive program. The intrinsic call acts as a hard-coded breakpoint. Execution of the calling program is halted, and the Debug prompt is displayed.

If the call is made from a batch program, it is ignored.

Refer to the *MPE XL Intrinsics Reference Manual* (32650-90028) for additional discussion of this intrinsic.

Condition Codes

This intrinsic does not return meaningful condition code values.

Example

The following example is a code fragment from a Pascal program. It declares `DEBUG` as an intrinsic and then calls it.

```
PROCEDURE call_debug;

   procedure debug; intrinsic;

BEGIN
   debug;
END;
```

HPDEBUG Intrinsic

Enters Debug and optionally executes a specified set of system Debug commands.

Callable from: NM

Syntax

```
HPDEBUG (status, cmdstr [,itemnum, item] [...]);
```

Parameters

status **32-bit signed integer by reference (optional)**

The status returned by the `HPDEBUG` intrinsic call. The variable is a record containing two 16-bit fields, with the error number in the high-order 16 bits and the intrinsic subsystem number in the low-order 16 bits.

cmdstr **character array (optional)**

A packed array of characters from 255 to 1024 bytes that contains the Debug commands to be executed. The first character in the array is recognized as the command delimiter. The last character in the command string must be followed immediately by the same delimiter.

itemnum **32-bit signed integer by value (optional)**

The item number of an `HPDEBUG` option as defined in the following `HPDEBUG` options.

item **type varies by value (optional)**

Passes and/or returns the `HPDEBUG` option indicated by the corresponding *itemnum* parameter. The *itemnum/item* optional parameters must appear in pairs. You can specify any number of option pairs. Any *itemnum* takes precedence over any previously specified duplicate *itemnum*. The following discussion lists the optional *itemnum/item* parameter pairs available to you.

itemnum=1 **Output file number (I32)**

Passes an item value specifying an opened file number to which DEBUG output is sent. The file must be a writeable ASCII file. The item value 1 is valid and specifies that $STDLIST will be used. Default: Use terminal LDEV for sessions and $STDLIST for jobs.

itemnum=2 **Welcome Banner Flag (I32)**

Passes an item value indicating if the Debug welcome banner should be printed. An item value of zero (0) keeps the banner from printing. Any other value causes the banner to print. Default: Print the welcome banner (1).

Discussion

The HPDEBUG intrinsic calls Debug with an optional character array containing Debug commands. If the command list is specified, Debug pushes the commands onto its command stack and executes them.

If no command in the command string causes control to be returned to the calling procedure (that is, a CONTINUE command), the user is left in Debug as long as the process is being run from a session environment. Processes run from a job are not allowed to stop in Debug. If the command string does cause control to return to the calling procedure, any remaining commands are left pending on Debug's command stack to be executed the next time Debug is called.

Refer to the *MPE XL Intrinsics Reference Manual* (32650-90028) for additional discussion of this intrinsic.

Condition Codes

This intrinsic does not return meaningful condition code values. Status information is returned in the optional *status* parameter described above.

Example

The following example is an excerpt from a Pascal program which illustrates a call to the HPDEBUG intrinsic. The commands passed to Debug produce output similar to that of the STACKDUMP intrinsic. The command string contains commands that tell Debug to first open a list file, print a title, produce a stack trace, and finally close the list file and return to the calling routine.

```
PROCEDURE call_hpdebug;

   VAR debug_cmds   : string[255];:
       status       : integer;

   procedure HPDEBUG; intrinsic;

   BEGIN

     debug_cmds := '\list myfile;wl "***STACKDUMP***";tr,dual;list close;c\';

     hpdebug(status, debug_cmds);

     IF (status <> 0) THEN
        error_routine(status, 'HPDEBUG');
   END;
```

HPRESETDUMP Intrinsic

Disarms Debug call which is made during abnormal process terminations.

Callable from: NM

Syntax

```
HPRESETDUMP (status);
```

Parameters

status **32-bit signed integer (optional)**

The status returned by the HPRESETDUMP intrinsic call. The variable is a record containing two 16-bit fields, with the error number in the high-order 16 bits and the intrinsic subsystem number in the low-order 16 bits.

Discussion

The HPRESETDUMP intrinsic disarms the Debug call that is made during abnormal process termination. If the Setdump feature was not previously armed by one of the Setdump intrinsics or commands, this intrinsic has no effect. Only the current process is affected; all other existing processes retain their current Setdump attributes. After this call, any child process of the calling process will not have the Setdump attribute. This intrinsic performs the same function as the RESETDUMP intrinsic. The only difference is the means by which status information is returned.

Refer to the *MPE XL Intrinsics Reference Manual* (32650-90028) for additional discussion of this intrinsic.

Condition Codes

This intrinsic does not return meaningful condition code values. Status information is returned in the optional *status* parameter described above.

Example

The following example is a code fragment from a Pascal program. It declares HPRESETDUMP as an intrinsic and then calls it.

```
PROCEDURE call_hpresetdump;

   VAR status       : integer;

   procedure HPRESETDUMP; intrinsic;

   BEGIN
      HPRESETDUMP(status);

      IF (status <> 0) THEN
         error_routine(status, 'HPRESETDUMP');
   END;
```

HPSETDUMP Intrinsic

Arms a call to Debug which takes place during abnormal process termination.

Callable from: NM

Syntax

```
HPSETDUMP (status, cmdstr);
```

Parameters

status **32-bit signed integer (optional)**

The status returned by the `HPSETDUMP` intrinsic call. The variable is a record containing two 16-bit fields, with the error number in the high-order 16 bits and the intrinsic subsystem number in the low-order 16-bits.

cmdstr **character array (optional)**

A packed array of characters (up to 255 bytes) that contains the `DEBUG` commands to be executed if the process aborts. The first character in the array is recognized as the command delimiter. The last character in the command string must be immediately followed by the same delimiter.

Discussion

The `HPSETDUMP` intrinsic enables automatic execution of a set of Debug commands when a process terminates abnormally (aborts). This intrinsic affects the current process, child process, and any generation grandchild processes subsequently created by the calling process. That is, the Setdump attribute and *cmdstr* is inherited by any new child process and all generations thereafter.

Debug executes the commands in *cmdstr* and sends the output to the standard list file (`$STDLIST`). Any commands which require input generate an error message.

If the process that aborts is being run from a job, the process terminates after executing the command string. If the process is being run from a session, then after the specified command string has been executed, Debug stops to accept interactive commands with I/O performed at the user terminal, contingent upon the following requirements:

- The abort did not occur while in system code, and
- The process entered the abort code through a native mode interrupt. Such aborts are typically caused by arithmetic and code-related traps (refer to the `XARITRAP` and `XCODETRAP` intrinsics).

Note CM programs usually fail these tests.

Once Debug accepts interactive input, the user is free to enter any Debug command. The user may choose to resume the process or have it terminate (see the `CONTINUE` command in chapter 4).

If the cause of the abort is a stack overflow, the command list is ignored and a stack trace is sent to `$STDLIST`, after which the process is terminated with no interactive debugging allowed.

Refer to the *MPE XL Intrinsics Reference Manual* (32650-90028) for additional discussion of this intrinsic.

Condition Codes

This intrinsic does not return meaningful condition code values. Status information is returned in the optional *status* parameter described above.

Example

Assume that a file called ABORTCMD contains a set of Debug commands to be used when a process abort occurs.

A process abort in the following procedure opens a list file, performs a stack trace, executes the commands from the use file, and closes the list file:

```
PROCEDURE myproc{};

VAR
   status      :  integer;
   debug_cmds  :  string[255];

BEGIN
   debug_cmds := '\list errfile;tr,dual;use abortcmd;list close\';
   hpsetdump(status, debug_cmds);

   IF (status <> 0) THEN
      error_routine(status, 'HPSETDUMP');

   .
   . <code in this area is protected with the "setdump" facility>
   .

   hpresetdump(status);

   IF (status <> 0) THEN
      error_routine(status, 'HPRESETDUMP');
END;
```

RESETDUMP Intrinsic

Disarms the Debug call that is made during abnormal process termination

Callable from: NM, CM

Syntax

```
RESETDUMP;
```

Discussion

The RESETDUMP intrinsic disarms the Debug call that is made during abnormal process termination. If the Setdump feature was not previously armed by one of the Setdump intrinsics or commands, this intrinsic has no effect. Only the current process is affected. This intrinsic performs a function identical to the HPRESETDUMP intrinsic. The only difference is the means by which status information is returned.

Refer to the *MPE XL Intrinsics Reference Manual* (32650-90028) for additional discussion of this intrinsic.

Condition Codes

CCE	Request granted.
CCG	Abnormal process termination; Debug call is not currently enabled and remains disabled.
CCL	Not returned by this intrinsic.

Example

The following example is a code fragment from a Pascal program. It declares RESETDUMP as an intrinsic and then calls it.

```
PROCEDURE call_resetdump;

   procedure RESETDUMP; intrinsic;

   BEGIN
      RESETDUMP;
   END;
```

SETDUMP Intrinsic

Arms the Debug call that is made during abnormal process termination.

Callable from: NM, CM

Syntax

```
SETDUMP (flags );
```

Parameters

flags **16-bit unsigned integer (required)**

This parameter is provided for compatibility with MPE V. It is required, but is ignored.

Discussion

The `SETDUMP` intrinsic arms a call to Debug which is made during abnormal process terminations (aborts). If the process aborts, Debug is called with a command string that results in a full stack trace of both the CM and NM data stacks along with a dump of the native mode registers. This output is sent to the standard list device (`$STDLIST`). This intrinsic affects the current process, child process, and any generation grandchild processes subsequently created by the calling process. That is, the Setdump attribute and the default *cmdstr* are inherited by any new child process and all generations thereafter.

If the process that aborts is being run from a job, the process terminates after the stack trace and register dump are performed. If the process is being run from a session, after the stack trace and register dump have been completed, Debug stops to accept interactive commands with I/O performed at the user terminal, contingent upon the following requirements:

- The abort did not occur while in system code, and

- The process entered the abort code through a native mode interrupt. Such aborts are typically caused by arithmetic and code-related traps (see the `XARITRAP` and `XCODETRAP` intrinsics).

Note CM programs usually fail these tests.

Once Debug accepts interactive input, the user is free to enter any Debug command. The user may choose to resume the process or have it terminate (refer to the `CONTINUE` command in chapter 4).

If the cause of the abort is a stack overflow, the command list is ignored and a stack trace is sent to `$STDLIST`, after which the process terminates. No interactive debugging is allowed.

Refer to the `HPSETDUMP` intrinsic for a more flexible version of this intrinsic.

Refer to the *MPE XL Intrinsics Reference Manual* (32650-90028) for additional discussion of this intrinsic.

Condition Codes

CCE	Request granted.
CCG	Abnormal process termination. Debug call is already enabled and remains enabled.
CCL	Not returned by this intrinsic.

Examples

The following example is a code fragment from a Pascal program. It declares `SETDUMP` as an intrinsic and then calls it. The rest of the code in the program is protected by the Setdump facility, unless another routine in the program explicitly turns it off.

```
PROGRAM myprog;

   TYPE bit16 = 0 .. 65535;

   flags : bit16;
```

SETDUMP Intrinsic

```
procedure SETDUMP; intrinsic;

BEGIN
   SETDUMP( flags );

   .
   . <the rest of the program follows >
   .

END.
```

STACKDUMP Intrinsic

Produces a full stack trace.

Callable from: NM, CM

Syntax

```
STACKDUMP (filename, idnumber, flags, selec);
```

Parameters

filename **Byte array (optional)**

An array of characters giving the file name of a new output file to be opened. The name should be terminated by any nonalphanumeric character except a slash (/) or a period (.). The same restrictions for the *formaldesignator* parameter in the `FOPEN` intrinsic apply to this parameter.

idnumber **16-bit integer (optional)**

If the intrinsic fails due to a file system error, the file system specific error number of the failure is returned here. Any value passed into the intrinsic through this parameter is ignored.

flags **16-bit unsigned integer (optional)**

This parameter is provided for compatibility with MPE V. If it is present in the intrinsic call, it is ignored and has no effect.

selec **32-bit integer array by reference (optional)**

This parameter is provided for compatibility with MPE V. If it is present in the intrinsic call, it is ignored and has no effect.

Discussion

The STACKDUMP intrinsic calls Debug to send a stack trace to the standard list file ($STDLIST) or to a new file named in the *filename* parameter. Control then returns to the calling procedure.

Refer to the *MPE XL Intrinsics Reference Manual* (32650-90028) for additional discussion of this intrinsic.

Condition Codes

CCE	Request granted.
CCG	Request denied. An invalid address for the location of the *filename* parameter was detected.
CCL	Request denied. File system error occurred during opening or closing of the file. The specific file system error number is returned in the *idnumber* described above.

Examples

The following example is a code fragment from a Pascal program. First, it prints out the error status and intrinsic name that were passed as parameters. Next, it calls the `STACKDUMP` intrinsic to produce a stack trace. Finally, the process is terminated with a call to the `TERMINATE` intrinsic.

```
PROCEDURE error_routine(status : integer;     { error status }
                        proc   : proc_str);   { Intrinsic name that failed }

   procedure STACKDUMP; intrinsic;
   procedure TERMINATE; intrinsic;

   BEGIN
      writeln(proc, ' returned error status of ', status);

      stackdump;

      terminate;
   END;
```

The next example prompts the user for a file name and then calls the `STACKDUMP` intrinsic to print a stack trace to the specified file.

```
PROCEDURE show_stack;

   VAR fname : string[80];

   procedure STACKDUMP; intrinsic;
```

```
BEGIN
   prompt('Print stack trace to which file: ');
   readln(fname);

   fname := fname + ' ';      { Add terminator character }

   stackdump(fname);
END;
```

STACKDUMP' Intrinsic

Writes a full stack trace to a previously opened file.

Callable from: CM

Syntax

```
STACKDUMP' (filename, idnumber, flags, selec);
```

Parameters

filename **Byte array (required)**

The first byte of this array contains the file number of a previously opened file. The file is used as the output file. The file must have a record length between 32 and 256 CM words, and write access must be allowed for the file.

idnumber **16-bit integer (required)**

If the intrinsic fails due to a file system error, the file system specific error number of the failure is returned here. Any value passed into the intrinsic through this parameter is ignored.

flags **16-bit unsigned integer (optional)**

This parameter is provided for compatibility with MPE V. If it is present in the intrinsic call, it is ignored and has no effect.

selec **32-bit integer array by reference (optional)**

This parameter is provided for compatibility with MPE V. If it is present in the intrinsic call, it is ignored and has no effect.

FINAL TRIM SIZE : 7.5 in x 9.0 in

Discussion

The STACKDUMP' intrinsic writes a full dual stack trace to a previously opened file. The file number of this file is passed to the intrinsic in the first byte of the *filename* parameter.

This intrinsic exists only in the compatibility mode library SL.PUB.SYS. No native mode to compatibility mode switch stub is provided.

Condition Codes

CCE	Request granted.
CCG	Request denied. One of two possible problems causes this condition code. First, an invalid address for the location of the *filename* parameter was detected. Second, the file record size was not between 32 and 256 CM words.
CCL	Request denied. User does not have access to the file number passed in the *filename* parameter.

Example

The following example is a code fragment from a Pascal/V program. It is a procedure which is passed the file number of an already opened file. The procedure then uses the STACKDUMP' intrinsic to have a stack trace printed to the specified file number. Note the use of the Pascal $ALIAS$ directive in declaring the intrinsic.

```
PROCEDURE dump_stack_to_fnum(fnum : shortint);

   TYPE  bit8 = 0..255;

         kludge_record = RECORD
            CASE integer OF
               0 : (byte_1 : bit8;
                    byte_2 : bit8);

               1 : (pac     : packed array[1..2] OF char);
         END;
```

```
VAR   kludge_var : kludge_record;

procedure STACKDUMP_PRIME $alias 'stackdump'''$; intrinsic;

BEGIN
   kludge_var.byte_1 := fnum;        { This assumes that the value of FNUM }
                                     { is no bigger than 8 bits.  This is  }
                                     { a valid assumption.                 }

   stackdump_prime(kludge_var.pac);  { Call STACKDUMP' to produce the      }
                                     { stack trace.                        }
END;
```

FINAL TRIM SIZE : 7.5 in x 9.0 in

4

System Debug Command Specifications

Specifications for the System Debug commands are presented in this chapter in alphabetical order.

Window command specifications are presented in chapter 7, "System Debug Window Commands."

System Debug tools share the same command set. A few commands, however, are inappropriate in either DAT or Debug. These commands are clearly identified as "DAT only" or "Debug only" on the top of the page that defines the command.

Debug only

The following Debug commands cannot be used in DAT:

`B`	All forms of the break command
`BD`	Breakpoint delete
`BL`	Breakpoint list
`C[ONTINUE]`	Continue
`DATAB`	Data breakpoint
`DATABD`	Data breakpoint delete
`DATABL`	Data breakpoint list
`F`	All forms of the `FREEZE` command
`FINDPROC`	Dynamically loads NL library procedure
`KILL`	Kills a process
`LOADINFO`	Displays currently loaded program / libraries
`LOADPROC`	Dynamically loads CM library procedure
`M`	All forms of the modify command
`S[S]`	Single step
`TERM`	Terminal semaphore control
`TRAP`	Arm/Disarm/List Traps
`UF`	All forms of the `UNFREEZE` command

DAT only

The following DAT commands cannot be used in Debug:

`CLOSEDUMP`	Closes a dump file
`DEBUG`	Enters Debug; used to debug DAT
`DPIB`	Displays a portion of the Process Information Block
`DPTREE`	Displays the process tree
`DUMPINFO`	Displays dump file information
`GETDUMP`	Reads in a dump tape to create a dump file
`OPENDUMP`	Opens a dump file
`PURGEDUMP`	Purges a dump file

:

The CI command - Access to the MPE XL command interpreter (CI).

Syntax

```
: [ command ]
```

The `HPCICOMMAND` intrinsic is used to access the MPE XL command interpreter (CI).

Parameters

command The command to execute via the CI. If no command is given, a new version (new process) of the CI is created.

Examples

```
$nmdebug > :showtime
WED, JAN 8, 1986, 1:32 PM
```

The above is typical use of the CI command.

```
$nmdebug > :file t;dev=tape
```

See the note below.

Limitations, Restrictions

Semicolons normally separate commands for System Debug. When the ":" command is entered at the System Debug prompt, however, the entire user command line is passed to the CI. One exception is within macro bodies, where the command line is split at the semicolons.

Every time this command is used, Debug assumes ownership of the Control-Y handler (even if it already owns it).

:

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

=

The calculator command.

Calculates the value of an expression and displays the result in the specified base.

Syntax

```
= expression [base]
```

Parameters

expression The expression to evaluate.

base The desired representation mode for output values:

`%` or `octal`	Octal representation
`#` or `decimal`	Decimal representation
`$` or `hexadecimal`	Hexadecimal representation
`ASCII`	ASCII representation

This parameter can be abbreviated to a single character.

If omitted, the current output base is used. Refer to the `SET` command to change the current output base.

String expressions (of four or fewer characters) are automatically coerced into a numeric value when the display base of octal, decimal, or hexadecimal is specified.

Examples

```
%cmdebug > = 12 + #10 + $a, d
#30
```

What is octal 12 (current input base) plus decimal 10 plus hex a, in decimal?

```
%cmdebug > = 5 + (-2)
```

=

```
%3
```

Negative values that follow immediately after an operator (+, -, *, /) must be placed within parentheses.

```
%cmdebug > = 'ABCD'
'ABCD'
%cmdebug > = 'ABCD',h
$41424344
```

In the second example, the string is coerced into a hexadecimal value.

```
%cmdebug > = [dst 12.100] + [db+4], $
$4820
```

The sum of the contents of data segment 12.100 plus the contents of DB+4, displayed in hexadecimal.

```
%cmdebug > = fopen
SYS %22.4774
```

What is the start address of the CM procedure `FOPEN`? The address is returned as logical code address.

```
%cmdebug > = ?fopen
SYS %22.5000
```

What is the entry point address of the CM procedure `FOPEN`? The question mark is used (CM) to indicate entry point, rather than start address.

```
$nmdebug > = [r12]
$c04
```

The indirect contents of register 12.

```
$nmdebug > = vtor (c.c0000000)
$0020800
$nmdebug > = rtov (20800)
$c.c0000000
```

Translate a virtual address to a real address and then back again.

```
$nmdebug > = 1 << 2
$4
```

The value 1, left-shifted by two bits.

```
$nmdebug > = $1234 band $ff
$34
```

The value $1234, Bit-ANDed with the mask $ff.

```
$nmdebug > = sendio
SYS $a.$219ef0
```

What is the start address of NM procedure `sendio`?

```
$nmdebug > = ?sendio
```

```
=

   SYS $a.$217884
```

What is the address of the export stub for NM procedure `sendio`? Note the different use of "?" in CM and NM. In CM "?" is used for entry address, while in NM "?" is used for export stub.

```
$nmdebug > = strup("super") + 'duper'
"SUPERduper"
```

The calculator accepts string expressions as well as numeric expressions.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

ABORT

Aborts/terminates the current System Debug process.

Syntax

```
ABORT
```

Parameters

none

Examples

```
%cmdebug > ABORT

END OF PROGRAM
:
```

Limitations, Restrictions

If Debug is entered using the `DEBUG` command at the CI, the `ABORT` command causes the current session to be logged off. Use `CONTINUE` to exit from Debug in this case.

If the process holds a SIR (system internal resource) or is "critical," you are not allowed to execute this command.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

ALIAS

Defines an alias (alternative) name for a command or macro.

Syntax

```
ALIAS name command
```

Aliases are useful for defining a new (shorter or longer) name for a command name or macro name. Aliases have higher precedence than command or macro names, and they can therefore be used to redefine (or conceal) commands or macros. When a new alias redefines a command, a warning is generated, indicating that a command has been hidden.

User defined aliases, created with the `ALIAS` command, are classified as *user* aliases. Several predefined aliases (command abbreviations) are automatically generated, and are classified as *predefined* aliases. Refer to the `ALIASLIST` and `ALIASINIT` commands.

Parameters

name The name of the alias (the new name to be used in place of another). Alias names are restricted to 16 characters.

command The command name to be used when the alias name is encountered. This can be any command or macro name. The command name is restricted to 32 characters.

Examples

```
$nmdebug > printtableentrylength 6
$200
$nmdebug > alias tbl printtableentrylength
$nmdebug > tbl 6
$200
```

The above example assumes that a macro called `printtableentrylength` has been defined, and a typical macro invocation is displayed. Since the macro name is long, and difficult to enter, an alias named `TBL` is defined. The shorter alias name can now be used in place of the longer macro name.

```
$nmdebug > alias loop foreach
$nmdebug > loop j '1 2 3' {wl j}
$1
$2
$3
```

Create an alias named `LOOP` that is the same as the `FOREACH` command.

```
$nmdat > macro concealexit { wl "type EXIT to exit."}
$nmdat > alias e concealexit
A command is hidden by this new alias.  (warning #71)
$nmdat > e
type EXIT to exit.
```

In this example, the single character command `e` (for `EXIT`) is protected by an alias, that conceals (hides) the original command. Note that a warning message is generated whenever a command name is concealed by an alias definition.

```
$nmdat > alias one two
$nmdat > alias two three
$nmdat > alias three one
$nmdat > one
Circular ALIAS error.  Recursive ALIAS definition(s).  (error #2445)
```

It is legal for an alias (for example, `one` in the example above) to refer to another alias (`two` in the example above), so long as the chain of aliases does not wrap back onto itself. Recursive aliases are detected, and an error is generated.

```
$nmdat > alias showtime "wl time"
$nmdat > aliasl showtime
alias showtime wl time   /* user
$nmdat > showtime
Unknown command.  (error #6105)
   Command "showtime" was aliased to "wl time".
```

Note that alias command names are restricted to simple command or macro names. In the above example, the command `wl time` was assumed to be the name of a command or macro. Since no match was found in the command or macro table, an error is generated. Macros should be used when more complex command lists or commands with parameters are desired.

Related commands: `ALIASINIT`, `ALIASL`, `ALIASD`.

Limitations, Restrictions

A maximum of 60 alias definitions are currently supported.

The alias `command` (the replacement name) is limited to command and macro names; no parameters or complex command lists are allowed. Refer to the `showtime` example above.

The `ALIASD` command cannot be aliased.

No testing is performed for invalid characters within the *name* or *command* parameters.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

ALIASD[EL]

Deletes the specified alias(es).

Syntax

```
ALIASD[EL] pattern [group]
```

Parameters

pattern The alias name(s) to be deleted.

This parameter can be specified with wildcards or with a full regular expression. Refer to appendix A for additional information about pattern matching and regular expressions.

The following wildcards are supported:

`@`	Matches any character(s).
`?`	Matches any alphabetic character.
`#`	Matches any numeric character.

The following are valid name pattern specifications:

`@`	Matches everything; all names.
`pib@`	Matches all names that start with "pib".
`log2##4`	Matches "log2004", "log2754", and so on.

The following regular expressions are equivalent to the patterns with wildcards that are listed above:

```
`.*`
`pib.*`
`log2[0-9][0-9]4`
```

This parameter must be specified; no default is assumed.

group The type(s) of aliases that are deleted. Aliases are classified as `USER` or `PREDEFINED` aliases. `ALL` refers to both types of aliases.

`U[SER]`	User-defined aliases
`P[REDEFINED]`	Predefined aliases
`A[LL]`	Both user-defined and predefined aliases

By default, only `USER` aliases are deleted. In order to delete a predefined alias, the group `PREDEFINED` or `ALL` must be specified.

Examples

```
$nmdebug > aliasd loop
$nmdebug >
```

Remove the user alias `loop` from the alias table.

```
$nmdebug > aliasd s@ pre
$nmdebug >
```

Delete all predefined aliases that begin with the letter "s".

Related commands: `ALIAS`, `ALIASINIT`, `ALIASLIST`.

Limitations, Restrictions

Numerous System Debug commands are implemented with aliases. If these predefined aliases are deleted, commands you are accustomed to using may not be available. Refer to the `ALIASINIT` command for a complete list of predefined aliases.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

ALIASINIT

Restores the predefined aliases, in case they have been deleted.

Syntax

```
ALIASINIT
```

For a full listing of all predefined aliases, see the example below.

Parameters

none

Examples

```
$nmdebug > aliasd @ all
$nmdebug > aliasinit
$nmdebug > aliasl @
alias aliasdel    aliasd   /* predefined
alias aliaslist   aliasl   /* predefined
alias cmdlist     cmdl     /* predefined
alias deletealias aliasd   /* predefined
alias deleteb     bd       /* predefined
alias deleteerr   errd     /* predefined
alias deletemac   macd     /* predefined
alias deletevar   vard     /* predefined
alias envlist     envl     /* predefined
alias errlist     errl     /* predefined
alias funclist    funcl    /* predefined
alias history     hist     /* predefined
alias listredo    hist     /* predefined
alias loclist     locl     /* predefined
alias macdel      macd     /* predefined
```

```
alias maclist     macl     /* predefined
alias maplist     mapl     /* predefined
alias proclist    procl    /* predefined
alias setalias    alias    /* predefined
alias setenv      env      /* predefined
alias seterr      err      /* predefined
alias setloc      loc      /* predefined
alias setmac      mac      /* predefined
alias setvar      var      /* predefined
alias showalias   aliasl   /* predefined
alias showb       bl       /* predefined
alias showcmd     cmdl     /* predefined
alias showdatab   databl   /* predefined
alias showenv     envl     /* predefined
alias showerr     errl     /* predefined
alias showfunc    funcl    /* predefined
alias showloc     locl     /* predefined
alias showmac     macl     /* predefined
alias showmap     mapl     /* predefined
alias showset     set      /* predefined
alias showsym     syml     /* predefined
alias showvar     varl     /* predefined
alias symfiles    symf     /* predefined
alias symlist     syml     /* predefined
alias trace       tr       /* predefined
alias vardel      vard     /* predefined
alias varlist     varl     /* predefined
$nmdebug >
```

Delete all aliases (user-defined and predefined). `ALIASINIT` is used to restore the predefined aliases. The entire set of predefined aliases is listed.

Related commands: `ALIAS`, `ALIASD`, `ALIASL`.

Limitations, Restrictions

A maximum of 60 alias definitions are currently supported. Therefore, the `ALIASINIT` command may not be able to re-establish all of the predefined aliases if the number of current user aliases is already close to the limit.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

ALIASL[IST]

Lists the currently defined aliases.

Syntax

```
ALIAS[LIST] [pattern] [group]
```

Parameters

pattern The alias name(s) to be displayed.

This parameter can be specified with wildcards or with a full regular expression. Refer to appendix A for additional information about pattern matching and regular expressions.

The following wildcards are supported:

`@`	Matches any character(s).
`?`	Matches any alphabetic character.
`#`	Matches any numeric character.

The following are valid name pattern specifications:

`@`	Matches everything; all names.
`pib@`	Matches all names that start with "pib".
`log2##4`	Matches "log2004", "log2754", and so on.

The following regular expressions are equivalent to the patterns with wildcards that are listed above:

```
'.*'
'pib.*'
'log2[0-9][0-9]4'
```

By default, all alias names are listed, subject to the group specification described below.

group The type of aliases that are to be listed. Aliases are classified as USER or PREDEFINED aliases. ALL refers to both types of alias.

U[SER]	User-defined aliases
P[REDEFINED]	Predefined aliases
A[LL]	Both user-defined and predefined aliases

By default, ALL aliases are deleted. In order to restrict the listing to a single group of aliases, the group USER or PREDEFINED must be specified.

Examples

```
$nmdebug > aliasl del@ p
 alias deletealias aliasd   /* predefined
 alias deleteb     bd       /* predefined
 alias deleteerr   errd     /* predefined
 alias deletemac   macd     /* predefined
 alias deletevar   vard     /* predefined
```

List all predefined aliases that start with "del".

```
$nmdebug > alias quit exit
$nmdebug > alias q quit
$nmdebug > alias bye exit
$nmdebug > aliasl ,user
alias bye  exit   /* user
alias q    quit   /* user
alias quit exit   /* user
```

Define three other command aliases that can be used in place of the EXIT command and list them.

Related commands: ALIAS, ALIASD, ALIASINIT.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

B (break)

Debug only

Privileged Mode: BA, BAX, BS

Break. Sets a breakpoint.

Syntax

```
B   logaddr [:pin|@] [count] [loud] [cmdlist]        Program
BG  logaddr [:pin|@] [count] [loud] [cmdlist]        Group library
BP  logaddr [:pin|@] [count] [loud] [cmdlist]        Account library
BLG logaddr [:pin|@] [count] [loud] [cmdlist]        Logon group lib
BLP logaddr [:pin|@] [count] [loud] [cmdlist]        Logon account lib
BS  logaddr [:pin|@] [count] [loud] [cmdlist]        System library

BU fname logaddr [:pin|@] [count] [loud] [cmdlist]   User library

BV  virtaddr [:pin|@] [count] [loud] [cmdlist]       Virtual address

BA  cmabsaddr [:pin|@] [count] [loud] [cmdlist]      Absolute CST
BAX cmabsaddr [:pin|@] [count] [loud] [cmdlist]      Absolute CSTX
```

The various forms of the BREAK command are used to set process-local and global (system-wide) breakpoints. Only users with privileged mode (PM) capability are allowed to set global breakpoints. Users without PM capability may only specify PINs that are descendant processes (any generation) of the current PIN.

Setting a breakpoint for another process is implemented such that it appears the target process set the breakpoint itself. Therefore, when the target process encounters the breakpoint, it enters Debug with its output directed to the LDEV associated with the target process.

If a breakpoint is set in CM code that has been translated by the Object Code Translator (OCT), Debug automatically sets a NM breakpoint in the closest previous corresponding translated code node point. If more than one CM breakpoint is set within a given node, only one NM breakpoint is set; however, a counter is incremented so the number of corresponding CM breakpoints can be tracked. If a NM breakpoint is set in translated code, no corresponding CM emulated breakpoint is set. Refer to appendix C for a discussion of CM object code translation, node points, and breakpoints in translated CM code.

Parameters

logaddr A full logical code address (`LCPTR`) specifies three necessary items:

- The logical code file (`PROG, GRP, SYS,`, and so on)
- NM: the virtual space ID number (`SID`)
  CM: the logical segment number
- NM: the virtual byte offset within the space
  CM: the word offset within the code segment

Logical code addresses can be specified in various levels of detail:

- As a full logical code pointer (`LCPTR`):

`B procname+20`	procedure name lookups return `LCPTR`s
`B pw+4`	predefined `ENV` variables of type `LCPTR`
`B SYS(2.200)`	explicit coercion to a `LCPTR` type

- As a long pointer (`LPTR`):

  `B 23.2644` *sid.offset* or *seg.offset*

  The logical file is determined based on the command suffix, for example:

  `B` implies `PROG`
  `BG` implies `GRP`
  `BS` implies `SYS`

- As a short pointer (`SPTR`):

  `B 1024` *offset* only

For NM, the short pointer offset is converted to a long pointer using the function `STOLOG`, which looks up the SID of the loaded logical file. This is different from the standard short to long pointer conversion, `STOL`, which is based on the current space registers (SRs).

For CM, the current executing logical segment number and the current executing logical file are used to build a `LCPTR`.

The search path used for procedure name lookups is based on the command suffix letter:

`B`	Full search path: NM: `PROG`, `GRP`, `PUB`, `USER`(s), `SYS` CM: `PROG`, `GRP`, `PUB`, `LGRP`, `LPUB`, `SYS`
`BG`	Search `GRP`, the group library.
`BP`	Search `PUB`, the account library.
`BLG`	Search `LGRP`, the logon group library.
`BLP`	Search `LPUB`, the logon account library.
`BS`	Search `SYS`, the system library.
`BU`	Search `USER`, the user library.

For a full description of logical code addresses, refer to the section "Logical Code Addresses" in chapter 2.

fname The file name of the NM user library. Since multiple NM libraries can be bound with the `XL=` option on a `RUN` command,

```
:run nmprog; xl=lib1,lib2.testgrp,lib3
```

it is necessary to specify the desired NM USER library. For example,

```
BU lib1 204c
BU lib2.testgrp test20+1c0
```

If the file name is not fully qualified, the following defaults are used:

Default account: the account of the program file.
Default group: the group of the program file.

virtaddr The virtual address of NM code.

Virtaddr can be a short pointer, a long pointer, or a full logical code pointer.

Short pointers are implicitly converted to long pointers using the `STOL` (short to long) function.

cmabsaddr A full CM absolute code address specifies three necessary items:

- Either the `CST` or the `CSTX`.
- The absolute code segment number.
- The CM word offset within the code segment.

Absolute code addresses can be specified in two ways:

- As a long pointer (`LPTR`)

`BA 23.2644`	Implicit `CST 23.2644`
`BAX 5.3204`	Implicit `CSTX 5.3204`

- As a full absolute code pointer (`ACPTR`)

`BA CST(2.200)`	Explicit CST coercion
`BAX CSTX(2.200)`	Explicit CSTX coercion
`BAX logtoabs(prog(1.20))`	Explicit absolute conversion

The search path used for procedure name lookups is based on the command suffix letter:

`BA`	`GRP`, `PUB`, `LGRP`, `LPUB`, `SYS`
`BAX`	`PROG`

pin |@ — The process identification number (PIN) of the process for which the breakpoint is to be set. If omitted, the breakpoint is set for the current process. The character "@" can be used to set a global breakpoint at which all processes stop.

count — *Count* has a twofold meaning: it specifies a break every *n*th time the breakpoint is encountered, and it is used to set permanent/temporary breakpoints. If *count* is positive, the breakpoint is permanent. If *count* is negative, the breakpoint is temporary and is deleted as soon as the process breaks at it. For example, a *count* of 4 means break every fourth time the breakpoint is encountered; a *count* of -4 means break on the fourth time, and immediately delete the breakpoint. If *count* is omitted, +1 is used, which breaks every time, permanently.

loud — Either `LOUD` or `QUIET`. If `QUIET` is selected the debugger does not print out a message when the breakpoint is hit. This is useful for performing a command list a great number of times before stopping without being inundated with screen after screen of breakpoint messages. These keywords may be abbreviated as desired. The default is `LOUD`.

cmdlist — A single Debug command or a list of Debug commands that are executed immediately when the breakpoint is encountered. Command lists for breakpoints are limited to 80 characters. (If this is too few characters, write a macro and have the command list invoke the macro.) *Cmdlist* has the form:

```
        CMD1
{ CMD1; CMD2; CMD3; ... }
```

NM Code Examples

```
$nmdebug > loadinfo
nm PROG   GRADES.DEMO.TELESUP            SID = $115
    parm = #0  info = ""
nm GRP    XL.DEMO.TELESUP                SID = $118
nm USER   XL.PUB.SYS                     SID = $f4
nm SYS    NL.PUB.SYS                     SID = $a
cm SYS    SL.PUB.SYS
```

Show the list of loaded files and the space into which they are loaded.

```
$nmdebug > b PROGRAM+270
added: NM    [1] PROG 115.00006a8c PROGRAM+$270
```

Set a breakpoint at the procedure `PROGRAM` plus an offset of $270. This corresponds to a statement in the outer block of the program being debugged. The name and offset were determined by looking at the statement map produced by the Pascal compiler (all language compilers produce similar maps). The expression evaluator found the procedure `PROGRAM` in the program file.

```
$nmdebug > b 6a90
added: NM    [2] PROG 115.00006a90 PROGRAM+$274
```

Break in the program file at offset $6a90. Remember that when only an offset is specified as a logical address for this command, the space (SID) for the program file is assumed. A `STOLOG` conversion (*not* `STOL`) with the "prog" selector is used to accomplish this.

```
$nmdebug > b processstudent,,, {wl "Processing #" r26:"d";c}
added: NM    [3] PROG 115.00005d24 processstudent
```

Set a breakpoint at the procedure called `processstudent` and provide a command list to be executed *each time* the breakpoint is encountered. In this example, we know that the student number being processed is passed to the routine in general register 26. Each time the routine is entered, Debug prints the student number and automatically continue execution of the process.

```
$nmdebug > b nmaddr("processstudent.highscore"),-1
added: NM   T[4] PROG 115.00005b50 processstudent.highscore
```

Set a breakpoint at the nested procedure `highscore` that is contained in the level 1 procedure `processstudent`. The `NMADDR` function is used to specify the breakpoint address since the expression *parent_proc.nested_proc* would not have been recognized by the expression evaluator (*a.b* implies *space.offset*, for example, a long pointer). This breakpoint is a temporary breakpoint, which is automatically deleted after it is encountered. `T[4]` indicates a temporary breakpoint with index number 4.

B (break)

```
$nmdebug > b average
added: NM    [5] GRP 118.00015c88 average

$nmdebug > bg average+4
added: NM    [6] GRP 118.00015c8c average+$4

$nmdebug > b grp(average)+8
added: NM    [7] GRP 118.00015c90 average+$8

$nmdebug > bs average
Missing or invalid logical code address.  (error #1741)
```

Set a breakpoint at the procedure `average`. Notice that the routine was found in the group (`GRP`) library. The `B` command starts searching for symbol names in the program file and continues through all of the loaded library files until a match is found. The second example uses the `BG` command to explicitly restrict the search for symbol names to the group library. The third example shows how the coercion function `GRP` is used to restrict procedure name lookups to the group library. In the fourth example above, the `BS` command is used to restrict the search for procedure names to the system library. The routine `average` was not found in the system library, and so an error was generated.

```
$nmdebug > dc pc
GRP $118.15c88
00015c88  average  0000400e  BREAK    (nmdebug bp)

$nmdebug > wl r2
$15c77

$nmdebug > wl sr4
$118

$nmdebug > b r2
The virtual address specified does not exist.  (error #1407)

$nmdebug > errl
$28:  The virtual address specified does not exist.  (error #1407)
$28:  The virtual address does not exist.  (error #6017)
```

```
$28:        VADDR= 115.15c74
$28:  A pointer was referenced that contained a virtual address outside
of the bounds of an object.
```

The above example starts by showing that Debug has stopped in the group library in the `average` procedure. The `B` command was used to set a breakpoint at the address specified in r2, and this caused the command to fail. Recall that the `B` command assumes that the breakpoint is to be set in the program file when only an offset is provided. The SID for the program file ($115) is retrieved, and a long pointer is generated by performing a `STOLOG` conversion. The resulting address ($115.$15c74) does not exist in the program file; thus an error is generated.

```
$nmdebug > bg r2
added: NM    [3] GRP 118.00015c74 ?average+$8

$nmdebug > bd 3
deleted: NM    [3] GRP 118.00015c74 ?average+$8
```

The BG command is used to set a breakpoint at the offset indicated by the contents of general register 2. This command assumes the breakpoint is to be set in the group library. The SID for the group library ($118) is retrieved, and a long pointer is generated by performing a STOLOG conversion. The resulting address ($118.$15c74) is a valid group library virtual address, and so the breakpoint is set. The address corresponds to the export stub for the average procedure. Refer to the *Procedure Calling Conventions Reference Manual* (09740-90015) for an explanation of the use and purpose of export stubs.

```
$nmdebug > bv r2
added: NM    [3] GRP 118.00015c74 ?average+$8

$nmdebug > bd 3
deleted: NM    [3] GRP 118.00015c74 ?average+$8
```

The BV command is used to set a breakpoint at the offset indicated by general register 2. Unlike the above example, the offset in r2 is converted to a long pointer by performing a STOL conversion. The resulting address (sr4.r2 = $118.$15c74) is a valid group library virtual address, and so the breakpoint is set. A full long pointer is always valid, so the command b 118.r2 also results in the breakpoint being set.

```
$nmdebug > b P_INIT_HEAP
added: NM    [8] USER f4.0012f2b8 p_heap:P_INIT_HEAP

$nmdebug > bu xl.pub.sys U_INIT_TRAPS
added: NM    [9] USER f4.001f9188 U_INIT_TRAPS
```

The above example sets a breakpoint at the procedure P_INIT_HEAP. The routine was found in one of the loaded user libraries (this process only has one loaded user library). The BU command is used in the second example to specify which user library to search when looking for procedure names. The U_INIT_TRAPS routine was found in the user library XL.PUB.SYS and a breakpoint was set.

```
$nmdebug > bs ?FREAD,#100,q,{wl "Read another 100 records";c}
added: NM    |10| SYS a.0074aa34 FREAD
```

Set a breakpoint at the FREAD intrinsic. Every #100 times the routine is called, stop and print out a message. The QUIET option is specified so this operation produces no extra terminal output. The vertical bars in the breakpoint notation indicates that the process does not stop the next time the breakpoint is encountered, since the count is not yet exhausted.

```
$nmdebug > bs trap_handler:@,,,{trace ,ism}
added: NM   @[1] SYS a.00668684 trap_handler
```

Set a system-wide breakpoint in the trap handler. This routine is in the system `NL`. When the breakpoint is hit, perform a stack trace. The "@" indicates that the breakpoint is a *global* breakpoint.

```
$nmdebug > b pw+4
added:  NM  [11] PROG $115.00006984 initstudentrecord+14
```

Break at the address specified by adding 4 to the address of the first line in the program window. In this case, the program window must have been aimed at `initstudentrecord+10`.

```
$nmdebug > bl
NM    [1] PROG 115.00006a8c PROGRAM+$270
NM    [2] PROG 115.00006a90 PROGRAM+$274
NM    [3] PROG 115.00005d24 processstudent
      cmdlist: {wl "Processing #" r26:"d";c}
NM   T[4] PROG 115.00005b50 processstudent.highscore
NM    [5] GRP 118.00015c88 average
NM    [6] GRP 118.00015c8c average+$4
NM    [7] GRP 118.00015c90 average+$8
NM    [8] USER f4.0012f2b8 p_heap:P_INIT_HEAP
NM    [9] USER f4.001f9188 U_INIT_TRAPS
NM   |10| SYS a.0074aa34 FREAD
        [QUIET]  count: 0/64  cmdlist: {wl "Read another 100 records";c}
NM   [11] PROG $115.00006984 initstudentrecord+14
NM   @[1] SYS a.00668684 trap_handler
        [QUIET]  cmdlist: {trace ,ism}
```

Now list all of the breakpoints just set above.

CM Code Examples

```
%cmdebug > loadinfo
cm PROG  GRADES.DEMOCM.TELESUP
    parm = #0  info = ""
cm GRP   SL.DEMOCM.TELESUP
cm SYS   SL.PUB.SYS
nm SYS   NL.PUB.SYS                    SID = $a
```

Show the list of all currently loaded files.

```
%cmdebug > b ?processstudent
added: CM    [1] PROG %   0.1665   ?PROCESSSTUDENT
```

Set a breakpoint at the entry point (indicated by the `?` character) of the procedure `PROCESSSTUDENT`. The expression evaluator found the procedure in the program file in logical segment zero, at an offset of %1665 CM words from the start of the segment procedure.

```
%cmdebug > b 0.1670
added: CM    [2] PROG %   0.1670   PROCESSSTUDENT+%263
```

Set a breakpoint %1670 CM words into the program file's logical segment zero. That address corresponds to the %263rd CM word from the start of the `PROCESSSTUDENT` procedure. Note that this command sets a breakpoint in the program file, no matter where the process was stopped (in the group library for example), since the `B` command implies the program file.

```
%cmdebug > b 1672
added: CM    [3] PROG %   0.1672   PROCESSSTUDENT+%265
```

Set a breakpoint %1672 CM words into the program file. The logical segment number from the current value of `CMPC` is used as the segment number for this command.

```
%cmdebug > b processstudent+14
added: CM   [4] PROG %   0.1421   PROCESSSTUDENT+%14
```

Set a breakpoint %14 CM words into the start of the procedure `PROCESSSTUDENT`. This address corresponds to the first statement of the nested procedure `HIGHSCORE` which is contained in the level 1 procedure `PROCESSSTUDENT`. The correct offset to use for nested procedures is determined by looking at the statement map produced by the Pascal compiler. (All language compilers produce similar maps.) Unfortunately, information about nested procedure names and size is not available for CM programs.

```
%cmdebug > b ob'+40,-3
added: CM   T|5| PROG %   0.40     OB'+%40
```

Set a breakpoint %40 words into the procedure ob' (the outer block of the Pascal program being run). The third time the breakpoint is encountered, stop in Debug and delete the breakpoint. The notation T|5| indicates a temporary breakpoint with index number 2. The vertical bars indicate that the process does not stop the next time the breakpoint is encountered, since the count is not yet exhausted.

```
%cmdebug > b ?average
added: CM    [6] GRP  %   0.13     ?AVERAGE

%cmdebug > bg ?average+4
added: CM    [7] GRP  %   0.17     AVERAGE+%17

%cmdebug > b grp(0.20)
added: CM   [10] GRP  %   0.20     AVERAGE+%20
```

Set a breakpoint at the entry point to the procedure `average`. Notice that the procedure was found in the group (`GRP`) library. The `B` command starts searching for symbol names in the program file and continues through all of the loaded library files until a match is found. The second example uses the `BG` command to explicitly restrict the search for symbol names to the group library. The third example shows how the coercion function `GRP` is used to specify a logical segment in the group library rather than the program file.

```
%cmdebug > bs ?fwrite,#100,q,{wl "Another #100 records written";c}
added: CM    |11| SYS  %  27.4727   ?FWRITE
       NM    |1| TRANS 30.00737fb4 SUSER1:?FWRITE
```

The above example sets a breakpoint at the entry point of the `FWRITE` intrinsic which is located in the system library `SL.PUB.SYS`. Every #100 times the routine is called, stop and print out a message. The `QUIET` option is specified so this operation produces no extra terminal output. `SL.PUB.SYS` has been translated with the Object Code Translator (OCT), and so Debug automatically sets a breakpoint in the translated native mode code. Refer to appendix C for a discussion of CM object code translation, node points, and breakpoints in translated CM code.

```
%cmdebug > bl
CM     [1] PROG %   0.1665   ?PROCESSSTUDENT       SEG'             (CSTX 1)
CM     [2] PROG %   0.1670   PROCESSSTUDENT+%263   SEG'             (CSTX 1)
CM     [3] PROG %   0.1672   PROCESSSTUDENT+%265   SEG'             (CSTX 1)
CM     [4] PROG %   0.1421   PROCESSSTUDENT+%14    SEG'             (CSTX 1)
CM    T|5| PROG %   0.40     OB'+%40               SEG'             (CSTX 1)
       count: 0/3
CM     [6] GRP  %   0.13     ?AVERAGE              SEG'             (CST 112)
CM     [7] GRP  %   0.17     AVERAGE+%17           SEG'             (CST 112)
```

```
CM   [10] GRP  %   0.20     AVERAGE+%20           SEG'             (CST 112)
CM   |11| SYS  %  27.4727   ?FWRITE               SUSER1           (CST 30)
        [QUIET]  count: 0/144  cmdlist: {wl "Another #100 records written";c}
      Corresponding NM bp = 1
```

Now list the breakpoints that were set in the above examples.

Translated Code Examples

```
%cmdebug > bg ?average
added: CM    [1] GRP  %   0.13      ?AVERAGE
       NM    [1] TRANS 3d.0016962c SEG':?AVERAGE
```

Set a breakpoint in the group library at the entry point to the AVERAGE procedure. The group library and program file have been translated by the Object Code Translator (OCT). Debug determined that the code is translated and thus set a CM breakpoint in the emulated code *and* a NM breakpoint in the translated code. Refer to appendix C for a discussion of CM object code translation, node points, and breakpoints in translated CM code.

```
%cmdebug > b ?processstudent
added: CM    [2] PROG %   0.1665   ?PROCESSSTUDENT
       NM    [2] TRANS 48.0000a610 SEG':?PROCESSSTUDENT
```

Set a breakpoint at the entry point to the PROCESSSTUDENT procedure. As in the above example, the code is translated, and so Debug sets two breakpoints.

```
%cmdebug > b cmpc
added: CM    [3] PROG %   0.1672   PROCESSSTUDENT+%265
       NM    [3] TRANS 48.0000a66c SEG':PROCESSSTUDENT+%265

%cmdebug > b cmpc+1
added: CM    [4] PROG %   0.1673   PROCESSSTUDENT+%266
       NM    [3] TRANS 48.0000a66c SEG':PROCESSSTUDENT+%265
```

Set a breakpoint at the current CM program counter. Both the CM emulated and NM translated breakpoints are set. Next, set a breakpoint at the instruction following the current CM program counter. Again, both the CM and NM breakpoints are set. Note that the index number for the NM breakpoint is the same. This is because the two CM breakpoints are contained in the same node. Appendix C provides a description of node points.

```
%cmdebug > nm
$nmdebug > b 20.b940,#100,,{wl "Read another 100 records";c}
added: NM |4| TRANS $20.b940 FSEG:?FREAD
```

Break in space 20 at the indicated offset. Every 100 times the routine is called, stop and print out a message. As with all breakpoint commands, the address typed in is converted to a logical address. In this example, the long to logical (`LTOLOG`) routine is used by the debugger. Space 20 does not correspond to any of the native mode libraries or the program file. It is, however, found to correspond to a translated body of CM code (in this instance, the `FREAD` intrinsic). Note that the corresponding CM emulator breakpoint is *not* set by Debug.

```
%cmdebug > bl
CM    [1] GRP  %   0.13     ?AVERAGE                SEG'          (CST 112)
       Corresponding NM bp = 1
CM    [2] PROG %   0.1665   ?PROCESSSTUDENT         SEG'          (CSTX 1)
       Corresponding NM bp = 2
CM    [3] PROG %   0.1672   PROCESSSTUDENT+%265     SEG'          (CSTX 1)
       Corresponding NM bp = 3
CM    [4] PROG %   0.1673   PROCESSSTUDENT+%266     SEG'          (CSTX 1)
       Corresponding NM bp = 3

%cmdebug > nm
$nmdebug > bl
NM    [1] TRANS 3d.0016962c SEG':?AVERAGE
       CM Ref count = 1
NM    [2] TRANS 48.0000a610 SEG':?PROCESSSTUDENT
       CM Ref count = 1
NM    [3] TRANS 48.0000a66c SEG':PROCESSSTUDENT+%265
       CM Ref count = 2
NM    [4] TRANS 20.0000b940 FSEG:?FREAD
       count: 0/64  cmdlist: {wl "Read another 100 records";c}
       CM Ref count = 1
```

Now list the breakpoints that have been set.

Limitations, Restrictions

You cannot set a breakpoint on a gateway page.

If breakpoints are set for a process other than the current PIN, Debug has no knowledge of the procedure names for the specified process unless the specified process is running the exact same program file.

Having breakpoints set causes slight process overhead. Arming a global breakpoint causes *all* processes to suffer this overhead.

Breakpoints are ignored in the following circumstances:

- While on the ICS.
- While disabled.
- In a "dying" process. (See the `DYING_DEBUG` variable in the `ENV` command discussion.)
- In a job. (See the `JOB_DEBUG` variable in the `ENV` command discussion.)

Breakpoints set in CM translated code (which has been optimized) may not always be hit. In some cases, the optimizer saves an instruction by targeting a branch to the delay slot immediately following a node point. As a result, a breakpoint that was set at the node point is not hit.

Caution Setting global breakpoints must be done with extreme care, and only when debugging requires it. Do not try this on a system under use. A global breakpoint may cause processes to suspend unexpectedly.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

BD

Debug only

Breakpoint delete. Deletes a breakpoint entry specified by index number.

Syntax

```
BD [number | @ [: pin | @] ]
```

The BD command is used to delete process-local breakpoints and global (system-wide) breakpoints. Only users with privileged mode (PM) capability are allowed to view and delete global breakpoints. Users without PM capability may only specify PINs that are descendant processes (any generation) of the current PIN.

When an NM breakpoint set in translated code is deleted, all corresponding CM breakpoints are automatically removed. When a CM breakpoint is deleted, the CM reference counter in the corresponding NM breakpoint (if any) is decremented. If the reference count reaches zero, the NM breakpoint is deleted. Refer to appendix C for a discussion of CM object code translation, node points, and breakpoints in translated CM code.

Parameters

number | @ — The index number of the breakpoint entry that is to be deleted. The character "@" can be used to delete all breakpoint entries.

If the index number is omitted, Debug displays each breakpoint, one at a time, and asks the user if it should be deleted (Y/N?). The following responses are recognized:

Y[E[S]]	Yes, remove the breakpoint.
YES *any_text*	Yes, remove the breakpoint.
N[O]	No, do not remove the breakpoint.

NO *any_text*	No, do not remove the breakpoint.

If any other response is given, the default value NO is assumed.

pin | @ The PIN for the process whose breakpoint entry is to be deleted. Typically this is omitted, and *pin* defaults to the current process.

The character "@" can be used to specify that a global breakpoint is to be deleted.

Examples

```
$nmdebug > bl
NM    [1] PROG 115.00006a8c PROGRAM+$270
NM    [2] PROG 115.00006a90 PROGRAM+$274
NM    [3] PROG 115.00005d24 processstudent
       cmdlist: {wl "Processing #" r26:"d";c}
NM   T[4] PROG 115.00005b50 processstudent.highscore
NM    [5] GRP 118.00015c88 average
NM    [6] GRP 118.00015c8c average+$4
NM    [7] GRP 118.00015c90 average+$8
NM    [8] USER f4.0012f2b8 p_heap:P_INIT_HEAP
NM    [9] USER f4.001f9188 U_INIT_TRAPS
NM   |10| SYS a.0074aa34 FREAD
        [QUIET]  count: 0/64  cmdlist: {wl "Read another 100 records";c}
NM   [11] PROG $115.00006984 initstudentrecord+14
NM   @[1] SYS a.00668684 trap_handler
        [QUIET]  cmdlist: {trace ,ism}
```

Display all breakpoints. Process-local breakpoints are always displayed first, followed by all global breakpoints.

```
$nmdebug > bd 2
deleted: NM    [2] PROG 115.00006a90 PROGRAM+$274
```

Delete process-local breakpoint number 2.

```
$nmdebug > bd
NM    [1] PROG 115.00006a8c PROGRAM+$270 (Y/N) ?
NM    [3] PROG 115.00005d24 processstudent (Y/N) ? y
NM   T[4] PROG 115.00005b50 processstudent.highscore (Y/N) ?
NM    [5] GRP 118.00015c88 average (Y/N) ?
NM    [6] GRP 118.00015c8c average+$4 (Y/N) ? YES
NM    [7] GRP 118.00015c8c average+$4 (Y/N) ? YES
NM    [8] USER f4.0012f2b8 p_heap:P_INIT_HEAP (Y/N) ? YES
NM    [9] USER f4.001f9188 U_INIT_TRAPS (Y/N) ? YES
NM   |10| SYS a.0074aa34 FREAD (Y/N) ?
NM   [11] PROG $115.00006984 initstudentrecord+14 (Y/N) y
NM   @[1] SYS a.00668684 trap_handler (Y/N) ?
```

BD

Display each breakpoint (local first, then global), then ask the user if the breakpoint should be deleted. In this example, process-local breakpoints numbers 3, 6, 7, 8, and 9 are removed.

```
$nmdebug > bl
NM    [1] PROG 115.00006a8c PROGRAM+$270
NM   T[4] PROG 115.00005b50 processstudent.highscore
NM    [5] GRP 118.00015c88 average
NM   |10| SYS a.0074aa34 FREAD
        [QUIET]  count: 0/64  cmdlist: {wl "Read another 100 records";c}
NM   @[1] SYS a.00668684 trap_handler
        [QUIET]  cmdlist: {trace ,ism}
```

List the remaining breakpoints.

```
$nmdebug > bd 1:@
deleted: NM   @[1] SYS a.00668684 trap_handler
```

Delete global breakpoint number 1.

```
$nmdebug > bd @
deleted: NM    [1] PROG 115.00006a8c PROGRAM+$270
deleted: NM   T[4] PROG 115.00005b50 processstudent.highscore
deleted: NM    [5] GRP 118.00015c88 average
deleted: NM   |10| SYS a.0074aa34 FREAD
        [QUIET]  count: 0/64  cmdlist: {wl "Read another 100 records";c}
```

Delete all remaining process-local breakpoints.

Translated Code Examples

```
%cmdebug > bl
CM    [1] GRP  %   0.13     ?AVERAGE               SEG'          (CST 112)
       Corresponding NM bp = 1
CM    [2] PROG %   0.1665   ?PROCESSSTUDENT        SEG'          (CSTX 1)
       Corresponding NM bp = 2
CM    [3] PROG %   0.1672   PROCESSSTUDENT+%265    SEG'          (CSTX 1)
       Corresponding NM bp = 3
CM    [4] PROG %   0.1673   PROCESSSTUDENT+%266    SEG'          (CSTX 1)
       Corresponding NM bp = 3

%cmdebug > nm
$nmdebug > bl
NM    [1] TRANS 3d.0016962c SEG':?AVERAGE
       CM Ref count = 1
NM    [2] TRANS 48.0000a610 SEG':?PROCESSSTUDENT
       CM Ref count = 1
NM    [3] TRANS 48.0000a66c SEG':PROCESSSTUDENT+%265
       CM Ref count = 2
NM    [4] TRANS 20.0000b940 FSEG:?FREAD
       count: 0/64  cmdlist: {wl "Read another 100 records";c}
       CM Ref count = 1
```

Show all of the CM and NM breakpoints. Notice that all of the native mode breakpoints are set in translated code and correspond to the emulated CM code breakpoints.

```
$nmdebug > bd 1
deleted: CM    [1] GRP  $   0.b      ?AVERAGE
deleted: NM    [1] TRANS 3d.0016962c SEG':?AVERAGE
       CM Ref count = 0
```

Delete NM breakpoint number 1. The corresponding CM breakpoint is also deleted. If more than one CM breakpoint corresponds to the NM breakpoint, then all of the CM breakpoints are deleted.

```
$nmdebug > cm
%cmdebug > bd 2
```

```
deleted: NM    [2] TRANS 48.0000a610 SEG':?PROCESSSTUDENT
       CM Ref count = 0
deleted: CM    [2] PROG %   0.1665   ?PROCESSSTUDENT
```

Delete CM breakpoint number 2. The corresponding NM breakpoint is also deleted.

```
%cmdebug > bd 3
deleted: NM    [3] TRANS 48.0000a66c SEG':PROCESSSTUDENT+%265
       CM Ref count = 1
deleted: CM    [3] PROG %   0.1672   PROCESSSTUDENT+%265
```

Delete CM breakpoint number 3. In this example, two CM breakpoints are mapped to one NM breakpoint (indicated by the reference counter). The corresponding NM breakpoint has its CM reference count decremented by one. When the reference count is zero, the NM breakpoint is deleted.

```
%cmdebug > bl
CM    [4] PROG %   0.1673   PROCESSSTUDENT+%266     SEG'             (CSTX 1)
       Corresponding NM bp = 3

%cmdebug > nm
$nmdebug > bl
NM    [3] TRANS 48.0000a66c SEG':PROCESSSTUDENT+%265
       CM Ref count = 1
NM    [4] TRANS 20.0000b940 FSEG:?FREAD
       count: 0/64  cmdlist: {wl "Read another 100 records";c}
       CM Ref count = 1
```

List the remaining CM and NM breakpoints.

Limitations, Restrictions

If breakpoints are listed for a process other than the current PIN, Debug has no knowledge of the procedure names associated with the addresses unless the specified process is running the exact same program file.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the

exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

BL

Debug only

Breakpoint list. Lists breakpoint entries, specified by index number.

Syntax

```
BL [number | @ [: pin | @] ]
```

The `BL` command is used to list process-local and global (system-wide) breakpoints. Global breakpoints are always displayed after the process-local breakpoints. Users without privileged mode (PM) capability are shown only the list of process-local breakpoints. Users without PM capability may only specify PINs that are descendant processes (any generation) of the current PIN.

Parameters

number The index number of the breakpoint entry to display. The symbol "@" can be used to display all entries. If omitted, then all entries are displayed.

pin The PIN for the process whose breakpoint entries are to be displayed. Typically this is omitted, and *pin* defaults to the current process.

The character "@" can be used to indicate global breakpoint(s).

Refer to appendix C for a discussion of CM object code translation, node points, and breakpoints in translated CM code.

Examples

```
$nmdebug > bl
NM    [1] PROG 115.00006a8c PROGRAM+$270
NM    [2] PROG 115.00006a90 PROGRAM+$274
NM    [3] PROG 115.00005d24 processstudent
       cmdlist: {wl "Processing #" r26:"d";c}
NM   T[4] PROG 115.00005b50 processstudent.highscore
NM    [5] GRP 118.00015c88 average
NM    [6] GRP 118.00015c8c average+$4
NM    [7] GRP 118.00015c90 average+$8
NM    [8] USER f4.0012f2b8 p_heap:P_INIT_HEAP
NM    [9] USER f4.001f9188 U_INIT_TRAPS
NM   |10| SYS a.0074aa34 FREAD
        [QUIET]  count: 0/64  cmdlist: {wl "Read another 100 records";c}
NM   [11] PROG $115.00006984 initstudentrecord+14
NM   @[1] SYS a.00668684 trap_handler
        [QUIET]  cmdlist: {trace ,ism}
```

Display all breakpoints. Process-local breakpoints are always displayed first, followed by all global breakpoints. See the Conventions page for a description of breakpoint notation.

```
$nmdebug > bl 3
NM    [3] PROG 115.00005d24 processstudent
       cmdlist: {wl "Processing #" r26:"d";c}
```

Display process-local breakpoint number 3.

```
$nmdebug > bl :@
NM   @[1] SYS a.00668684 trap_handler
        [QUIET]  cmdlist: {trace ,ism}
```

List all of the global breakpoints.

Translated Code Examples

```
%cmdebug > bl
CM    [1] GRP  %   0.13     ?AVERAGE                SEG'           (CST 112)
       Corresponding NM bp = 1
CM    [2] PROG %   0.1665   ?PROCESSSTUDENT         SEG'           (CSTX 1)
       Corresponding NM bp = 2
CM    [3] PROG %   0.1672   PROCESSSTUDENT+%265     SEG'           (CSTX 1)
       Corresponding NM bp = 3
CM    [4] PROG %   0.1673   PROCESSSTUDENT+%266     SEG'           (CSTX 1)
       Corresponding NM bp = 3

%cmdebug > nm
$nmdebug > bl
NM    [1] TRANS 3d.0016962c SEG':?AVERAGE
       CM Ref count = 1
NM    [2] TRANS 48.0000a610 SEG':?PROCESSSTUDENT
       CM Ref count = 1
NM    [3] TRANS 48.0000a66c SEG':PROCESSSTUDENT+%265
       CM Ref count = 2
NM    [4] TRANS 20.0000b940 FSEG:?FREAD
       count: 0/64  cmdlist: {wl "Read another 100 records";c}
       CM Ref count = 1
```

Show all of the CM and NM breakpoints. Notice that the CM breakpoints all have corresponding NM breakpoints. The NM breakpoints show a counter reflecting the number of corresponding CM breakpoints. However, the list of corresponding CM breakpoint numbers is not part of the NM breakpoint listing.

Limitations, Restrictions

If breakpoints are listed for a process other than the current process, Debug has no knowledge of the procedure names associated with the addresses unless the specified process is running the exact same program file.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

CLOSEDUMP

DAT only

Closes a dump file. (See `OPENDUMP` to open a dump.)

Syntax

```
CLOSEDUMP
```

Parameters

none

Examples

```
$nmdat > closedump
$nmdat >
```

Closes the dump file currently opened.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

CM

Enters compatibility mode (cmdat/cmdebug). See the `NM` command.

Syntax

```
CM
```

The command switches from NM (nmdat/nmdebug) to CM (cmdat/cmdebug). If the windows are on, the screen is cleared and the set of windows enabled for cmdebug is redrawn. The command also sets several environment variables. The variables affected and their new values are shown below:

```
ENV  MODE     "CM"
ENV  INBASE   CM_INBASE
ENV  OUTBASE  CM_OUTBASE
```

Parameters

none

Examples

```
$nmdebug > cm
%cmdebug >
```

Switch from nmdebug to cmdebug.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the

exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

CMDL[IST]

Command list. Displays a list of the valid commands for System Debug.

Syntax

```
CMDL[IST] [pattern] [group] [options]
```

This command displays a list of valid commands for System Debug. Several System Debug commands are actually implemented as aliases. Aliases are not displayed with the `CMDL` command; rather, the `ALIASL` command must be used to view them.

Parameters

pattern The command name(s) to be displayed.

This parameter can be specified with wildcards or with a full regular expression. Refer to appendix A for additional information about pattern matching and regular expressions.

The following wildcards are supported:

`@`	Matches any character(s).
`?`	Matches any alphabetic character.
`#`	Matches any numeric character.

The following are valid name pattern specifications:

`@`	Matches everything; all names.
`pib@`	Matches all names that start with "pib".
`log2##4`	Matches "log2004", "log2754".

The following regular expressions are equivalent to the patterns with wildcards that are listed above:

```
'.*'
'pib.*'
```

`'log2[0-9][0-9]4'`

By default, all command names are listed.

group Commands are logically organized in groups. When listed, the commands can be filtered by group, that is, only those commands in the specified group are displayed.

`PROCESS`	Process control
`BREAK`	Breakpoint setting/listing/deleting
`DISPLAY`	Display memory/code/segments
`OBJECTS`	File mapping, Object freezing
`REGISTER`	Display/modification/listing of registers
`STACK`	Stack tracing, level switching
`MODIFY`	Modify memory/code/segments
`SYMBOLIC`	Symbolic file access
`VAR`	Variable definition/listing/deleting
`MACRO`	Macro definition/listing
`FUNC`	Predefined function information
`ENV`	Commands to list/show/alter the environment
`TRANSLATE`	Translate CM addresses to NM address
`CI`	Command Interpreter-related
`IO`	For producing I/O
`DUMP`	Open/close/purge/info on dumps
`ERROR`	Error management
`MISC`	Grab bag
`WINDOW`	Window related
`ALL | @`	All groups

options Any number of the following options can be specified in any order, separated by blanks or commas:

`NAME`	Display command name only (default).
`USE`	Display command syntax, and summary of use.
`NOUSE`	Skip the syntax/summary.
`PARMS`	Display parameter names and types.
`NOPARMS`	Skip parameter display.
`DESC`	Display a general description.
`NODESC`	Skip the description.
`EXAMPLE`	Display an example.
`NOEXAMPLE`	Skip the example.
`ALL | @`	Display everything. Same as:

	NAME USE PARMS DESC EXAMPLE
PAGE	Page eject after each command definition. Useful for paged (listfile) output.
NOPAGE	No special page ejects. (default)

If none of the options above are specified, NAME is displayed by default. If any options are specified, then they are accumulated to describe which fields are printed.

Examples

```
$nmdat > cmdl ,err
cmd ERR         error       nm cm
cmd ERRD        error       nm cm
cmd ERRL        error       nm cm
cmd IGNORE      error       nm cm

Type "WHELP" for a list of the window commands
Type "ALIASL" for a list of the command aliases
```

List all of the commands that deal with error management.

```
$nmdat > cmdl w@
cmd W           io          nm cm
cmd WCOL        io          nm cm
cmd WHELP       window      nm cm
cmd WHILE       ci          nm cm
cmd WL          io          nm cm
cmd WP          io          nm cm
cmd WPAGE       io          nm cm
```

List all of the commands that start with the letter “W”.

```
$nmdat > cmdl w@,ci
cmd WHILE       ci          nm cm
```

List all of the commands that start with the letter “W” and deal with System Debug’s command interpreter. There is only one such command, `WHILE`.

```
$nmdat > cmdl while,,all
cmd WHILE       ci          nm cm

USE:

WHILE condition DO command | {cmdlist}

PARMS:

condition  A logical expression to be repeatedly evaluated.
command    A single command to be executed while CONDITION is true.
cmdlist    A list of commands to be executed while CONDITION is true.

DESC:

The WHILE command evaluates a logical expression and, if TRUE, executes
a command/command list.  The expression is then reevaluated, and the
process continues until the expression is FALSE.

EXAMPLE:

$nmdebug >  while [pc] >> $10 <> $2000 do ss
<Single step until the next Pascal statement number>
```

Provide all information available for the `WHILE` command.

```
$nmdat > cmdl while,,all noexample nodesc
cmd CMDL        ci          nm cm

USE:

WHILE condition DO command | {cmdlist}

PARMS:

condition  A logical expression to be repeatedly evaluated.
command    A single command to be executed while CONDITION is true.
cmdlist    A list of commands to be executed while CONDITION is true.
```

Provide all information available for the `WHILE` command *except* examples and description.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

CMG

Privileged Mode

Displays values in the CMGLOBALS record for a process.

Syntax

```
CMG  [pin]
```

The CMGLOBALS record is an operating system data structure that maintains compatibility mode information.

Parameters

pin The PIN for the process whose CMGLOBALS are to be displayed.

Examples

```
$nmdat > cmg
          dp0 : 0
   dp_scratch : c0105b60
      cm_info : c
      cm_ctrl : 0
    stack_dst : 84
       db_dst : 84
db_3k_offset : 200
       db_sid : 2c4
    db_offset : 400120b0
           dl : 2c4.40012000
            s : 2c4.4001245e
            z : 2c4.40014310
   stack_base : 2c4.40011cb0
  stack_limit : 2c4.40015fff
```

```
          cst : 80000700
         cstx : c6bc8000
         lstt : 0.0
   nrprgmsegs : 0
          dst : 81800000
        bank0 : 80000000
   bank0_size : 10000
       debug  : 0
    mcode_adr : 3ee090
$nmdat >
```

Display the `CMGLOBALS` record for the current PIN.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

C[ONTINUE]

Continues/resumes execution of user program.

Syntax

```
C[ONTINUE]
C[ONTINUE] [IGNORE]
C[ONTINUE] [NOIGNORE]
```

The program executes until a breakpoint is encountered or the program completes.

Used to exit Debug when it was entered via the `DEBUG` command at the CI.

Parameters

`[NO]IGNORE` This parameter is meaningful only in two states. The first is when Debug has stopped due to one of the MPE XL traps defined in the `TRAP` command (`XLIB, XCODE, XARI, XSYS`). The default value is `NOIGNORE`. If you wish to have the trap ignored (pretend it never happened), you must use the `IGNORE` option.

The second state is when the debugger has stopped due to a `SETDUMP` command. That is, the process is about to be killed by the trap handler and Debug has been called. If one just continues from this state, the process is terminated. If the `IGNORE` option is specified, the process is relaunched as if the error did not occur. It is up to the user to update registers and the process stack as appropriate to enable the process to continue correctly.

Examples

```
%cmdebug > c
```

Limitations, Restrictions

The CONTINUE command cannot be used from within macro bodies that are invoked as a function.

This command resumes execution of your program or the CI if you entered the debugger with a DEBUG command. If you wish to abort your program or session, use the ABORT command.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

D (display)

Privileged Mode: DA, DCS, DCA, DZ, DSEC

Displays the contents of the specified address.

Syntax

`DA`	*offset*	[*count*]	[*base*]	[*recw*]	[*bytew*]	`ABS relative`
`DD`	*dst.off*	[*count*]	[*base*]	[*recw*]	[*bytew*]	`CM data segment`
`DDB`	*offset*	[*count*]	[*base*]	[*recw*]	[*bytew*]	`DB relative`
`DS`	*offset*	[*count*]	[*base*]	[*recw*]	[*bytew*]	`S relative`
`DQ`	*offset*	[*count*]	[*base*]	[*recw*]	[*bytew*]	`Q relative`
`DC`	*logaddr*	[*count*]	[*base*]	[*recw*]	[*bytew*]	`Program file`
`DCG`	*logaddr*	[*count*]	[*base*]	[*recw*]	[*bytew*]	`Group library`
`DCP`	*logaddr*	[*count*]	[*base*]	[*recw*]	[*bytew*]	`Account library`
`DCLG`	*logaddr*	[*count*]	[*base*]	[*recw*]	[*bytew*]	`Logon group lib`
`DCLP`	*logaddr*	[*count*]	[*base*]	[*recw*]	[*bytew*]	`Logon account lib`
`DCS`	*logaddr*	[*count*]	[*base*]	[*recw*]	[*bytew*]	`System library`
`DCU`	*fname logaddr*	[*count*]	[*base*]	[*recw*]	[*bytew*]	`User library`
`DCA`	*cmabsaddr*	[*count*]	[*base*]	[*recw*]	[*bytew*]	`Absolute CST`
`DCAX`	*cmabsaddr*	[*count*]	[*base*]	[*recw*]	[*bytew*]	`Absolute CSTX`
`DV`	*virtaddr*	[*count*]	[*base*]	[*recw*]	[*bytew*]	`Virtual`
`DZ`	*realaddr*	[*count*]	[*base*]	[*recw*]	[*bytew*]	`Real memory`
`DSEC`	*ldev.off*	[*count*]	[*base*]	[*recw*]	[*bytew*]	`Secondary store`

Parameters

offset
: `DA`, `DDB`, `DQ`, `DS` only.

  The CM word offset that specifies the relative starting location of the area to be displayed.

dst.off
: `DD` only.

  The data segment number and CM word offset that specifies the starting location of the area to be displayed.

logaddr DC, DCG, DCP, DCLG, DCLP, DCS, DCU only.

A full logical code address (LCPTR) specifies three necessary items:

- the logical code file (PROG, GRP, SYS, and so on)
- NM: the virtual space ID number (SID)
  CM: the logical segment number
- NM: the virtual byte offset within the space.
  CM: the word offset within the code segment.

Logical code addresses can be specified in various levels of detail:

- as a full logical code pointer (LCPTR)

DC procname+20	procedure name lookups return LCPTRs
DC pw+4	predefined ENV variables of type LCPTR
DC SYS(2.200)	explicit coercion to a LCPTR type

- as a long pointer (LPTR)

  DC 23.2644 *sid.offset* or *seg.offset*

  The logical file is determined based on the command suffix, for example:

  DC implies PROG
  DCG implies GRP
  DCS implies SYS

- as a short pointer (SPTR)

  DC 1024 *offset* only

For NM, the short pointer offset is converted to a long pointer using the function STOLOG, which looks up the SID of the loaded logical file. This is different from the standard short to long pointer conversion, STOL, which is based on the current space registers (SRs).

For CM, the current executing logical segment number and the current executing logical file are used to build an `LCPTR`.

The search path used for procedure name lookups is based on the command suffix letter:

`DC`	Full search path: NM: `PROG`, `GRP`, `PUB`, `USER`(s), `SYS` CM: `PROG`, `GRP`, `PUB`, `LGRP`, `LPUB`, `SYS`
`DCG`	Search `GRP`, the group library.
`DCP`	Search `PUB`, the account library.
`DCLG`	Search `LGRP`, the logon group library.
`DCLP`	Search `LPUB`, the logon account library.
`DCS`	Search `SYS`, the system library.
`DCU`	Search `USER`, the user library.

For a full description of logical code addresses, refer to the section "Logical Code Addresses" in chapter 2.

fname `DCU` only.

The file name of the NM USER library. Since multiple NM libraries can be bound with the `XL=` option on a `RUN` command,

```
:run nmprog; xl=lib1,lib2.testgrp,lib3
```

it is necessary to specify the desired NM user library. For example,

```
DCU lib1 204c
DCU lib2.testgrp test20+1c0
```

If the file name is not fully qualified, then the following defaults are used:

Default account: the account of the program file.
Default group: the group of the program file.

cmabsadr `DCA`, `DCAX` only.

A full CM absolute code address specifies three necessary items:

- Either the `CST` or the `CSTX`
- The absolute code segment number
- The CM word offset within the code segment.

Absolute code addresses can be specified in two ways:

- As a long pointer (`LPTR`)

`DCA 23.2644`	Implicit `CST` 23.2644
`DCAX 5.3204`	Implicit `CSTX` 5.3204

- As a full absolute code pointer (`ACPTR`)

`DCA CST(2.200)`	Explicit `CST` coercion
`DCAX CSTX(2.200)`	Explicit `CSTX'` `coercion \DCAX` `logtoabs(prog(1.20))`\Explicit absolute conversion

The search path used for procedure name lookups is based on the command suffix letter:

`DCA`	`GRP, PUB, LGRP, LPUB, SYS`
`DCAX`	`PROG`

virtaddr
: `DV` only. The virtual address to be displayed.

  Virtaddr can be a short pointer, a long pointer, or a full logical code pointer.

realaddr
: DZ only.

  The real mode HP Precision Architecture memory address to be displayed.

ldev.off
: `DSEC` only.

  The logical device number (LDEV) and offset (in bytes) of the data on disk to be displayed.

count	`DA`, `DC@` (CM), `DD`, `DDB`, `DS`, `DQ`: The number of CM 16-bit words to be displayed. `DC@` (NM), `DV`, `DZ`, `DSEC`: The number of NM 32-bit words to be displayed. If omitted, then a single value is displayed.
base	The desired representation mode for output values: `%` or `OCTAL` — Octal representation `#` or `DECIMAL` — Decimal representation `$` or `HEXADECIMAL` — Hexadecimal representation `ASCII` — ASCII representation `BOTH` — Numeric and ASCII together `CODE` — Disassembled code representation `STRING` — Packed ASCII representation This parameter can be abbreviated to as a single character. By default, and for the numeric portion of `B[OTH]`, the current output base is used. Display code commands (`DC@`) automatically set the base to `CODE`, unless another base is explicitly specified. Note that the address portion of the display is always formatted using the current output base (see `ENV OUTBASE` and the `SET` command), not the specified base parameter.
recw	The number of words to be displayed per line. Large requests may cause lines to wrap around on the terminal, but may be appropriate for offline listings, based on the `ENV` variable `LIST_WIDTH`. By default, either 4 or 8 words will be displayed per line, based on the command, count, and base. When the base CODE is selected, disassembled code is always displayed one word per line.
bytew	The width in bytes of the displayed values. Values can be displayed as 1 byte — Single bytes (8 bits) 2 bytes — CM (16-bit words)

4 bytes	NM (32-bit words) / CM double-words

If omitted, values are displayed as CM words (2) or NM words (4), based on the current mode (CM/NM) and the specified command.

This parameter is ignored for display code commands (DC@).

Examples

```
%cmdebug > dd 77.0
DST %77.0       % 000655
```

Display DST 77.0. By default, one word is displayed in the current output base, octal.

```
%cmdebug > dd 77.0,20
DST %77.0
%0      % 000655 000012 000000 000000 000000 000000 000000 000000
%10     % 000000 000000 041515 023511 047111 052111 040514 020040
```

Display DST 77.0 for %20 words. By default, the data is displayed in the current output base, octal, at eight words per line.

```
%cmdebug > dd 77.0,20,a
DST %77.0
%0      ASCII .. .. .. .. .. .. .. ..
%10     ASCII .. .. CM 'I NI TI AL
```

Display DST 77.0 for %20 words in ASCII. The two character ASCII representations for each word are displayed, separated by blanks. Dots (".") are displayed for nonprintable characters.

```
%cmdebug > dd 77.0,20,b
DST %77.0
%0      % 000655 000012 000000 000000  .. .. .. ..
%4      % 000000 000000 000000 000000  .. .. .. ..
%10     % 000000 000000 041515 023511  .. .. CM 'I
%14     % 047111 052111 040514 020040  NI TI AL
```

Display DST 77.0 for %20 words. Display both numeric and ASCII data together. By default, four words are displayed per line.

```
%cmdebug > dd 77.0,100,a,12
DST %77.0
%0      ASCII .. .. .. .. .. .. .. .. .. ..
%12     ASCII CM 'I NI TI AL        . .. ..
%24     ASCII MI X' PA RM           . .. ..
%36     ASCII LO AD                 . .. ..
```

```
%50     ASCII GE TS IR              . .. ..
%62     ASCII RE LS IR              . .. ..
%74     ASCII FR EE 'P RI
```

Display `DST 77.0`, for %100 words, in ASCII, in a width of %12 words per line.

```
%cmdebug > dd 77.0,100,s,12
DST %77.0       "...................."
DST %77.12      "CM'INITIAL     ....."
DST %77.24      "MIX'PARM       ....."
DST %77.36      "LOAD           ....."
DST %77.50      "GETSIR         ....."
DST %77.62      "RELSIR         ....."
DST %77.74      "FREE'PRI"
```

Display DST 77.0 for %100 words, as a string, in a width of %12 CM words = #10 CM words = 20 characters per line.

```
%cmdebug > dd 77.0,20,h,6,1
DST %77.0
%0       $ 01 ad 00 0a 00 00 00 00 00 00 00 00
%6       $ 00 00 00 00 00 00 00 00 43 4d 27 49
%14      $ 4e 49 54 49 41 4c 20 20

%cmdebug > dd 77.0,20,h,6,2
DST %77.0
%0       $ 01ad 000a 0000 0000 0000 0000
%6       $ 0000 0000 0000 0000 434d 2749
%14      $ 4e49 5449 414c 2020

%cmdebug > dd 77.0,20,h,6,4
DST %77.0
%0       $ 01ad000a 00000000 00000000 00000000 00000000 434d2749
%14      $ 4e495449 414c2020 20202000 930c0000 4d495827 5041524d
%30      $ 20202020 20202000 00000000 4c4f4144
```

Display DST 77.0, for 20 words, in hexadecimal.

Display the data as bytes (1), CM 16-bit words (2), and NM 32-bit words (4).

Note that the offset addresses are displayed in octal (the current output base), while the data is displayed in hexadecimal, as requested.

```
$nmdebug > dsec 1.0,4,a
SEC $1.0         ASCII ..HP ESYS ..]@ ....
```

Display secondary storage at the disk address `1.0` (LDEV=1, byteoffset=0). Display four words in ASCII. This example displays a portion of the volume label.

```
%cmdebug > da %1114,3,a
ABS+%1114   ASCII 82 04  9

%cmdebug > da %1474,3,a
ABS+%1474   ASCII  9 82 04
```

Two examples that display CM ABS relative. Both examples display three words in ASCII.

ABS is CM Bank 0 low core memory. CM SYSGLOB starts at ABS+%1000.

The first example displays the SEL release ID in the form: *uu ff vv*.

The second example displays the MPE XL system version ID in the form: *vv uu ff*.

```
$nmdat > wl pc
SYS $a.728304
$nmdat > wl vtor(pc)
$c18304
$nmdat > dz tr0+((vtor(pc)>>$b)*$10),4
REAL $00603500 $ 80000000 0000000a 00728000 02400000
```

The logical code address of PC is SYS $a.728304, which translates to real memory address c18304.

This example displays the 4-word PDIR entry in real memory for the page that contains PC.

Display real memory (DZ) at the address TR0 (start of PDIR) plus the offset to entry, which is calculated by right-shifting the real address of PC by $b (to determine page number), and then multiplying by $10 since each 4-word PDIR entry is $10=#16 bytes long.

Examples of Code Displays

```
$nmdebug > dcs sendio+18,7
SYS $a.219f08
00219f08  sendio+$18  6bd83d69  STW      24,-332(0,30)
00219f0c  sendio+$1c  4bda3d51  LDW      -344(0,30),26
00219f10  sendio+$20  081a0241  OR       26,0,1
00219f14  sendio+$24  081e025f  OR       30,0,31
00219f18  sendio+$28  34180050  LDO      40(0),24
00219f1c  sendio+$2c  ebfe174d  BL       ?ldm_completion+$1e4,31
00219f20  sendio+$30  37d93dc1  LDO      -288(30),25
```

Display code in the NM system library, starting at `sendio+18`, for seven words. By default, the display code commands use the `CODE` radix and display formatted lines of disassembled code.

```
$nmdebug > dcs sendio+18,7,h
SYS $a.219f08   $ 6bd83d69 4bda3d51 081a0241 081e025f
SYS $a.219f18   $ 34180050 ebfe174d 37d93dc1
```

Display code in the system library, starting at `sendio+18`, for seven words in hexadecimal. By default, four words are displayed per line.

```
%cmdebug > dcs lsearch+11,10
SYS %12.20262
%020262:  LSEARCH+%11            051401  S.  STOR  Q+1
%020263:  LSEARCH+%12            000600  ..  ZERO, NOP
%020264:  LSEARCH+%13            151607  ..  LDD   Q-7
%020265:  LSEARCH+%14            041605  C.  LOAD  Q-5
%020266:  LSEARCH+%15            041604  C.  LOAD  Q-4
%020267:  LSEARCH+%16            031105  2E  PCAL ?LSEARCH'
%020270:  LSEARCH+%17            013712  ..  BRE   P+%12
%020271:  LSEARCH+%20            031107  2G  PCAL ?TRANS'XDST'TO'L
```

Display code starting at `lsearch+11`, for %10 words. The procedure is located in the CM system library, `SL.PUB.SYS`.

Listing Disassembled Code to a File

The following example demonstrates how to dump disassembled code into a file. The example is explained command by command, based on the command numbers that appear within the prompt lines.

Command %10 opens an offline list file with the name codedump. All Debug input and output is recorded into this file, including the code we intend to display.

Command %11 sets the environment variable term_loud to FALSE. This prevents subsequent Debug output from being displayed on the terminal. We capture the output in the list file (codedump), but we do not want the output on the terminal.

Command %12 contains the desired display code command. We display %20 words of disassembled code, starting at the entry point address ?fopen.

Command %13 closes (and saves) the current list file (codedump).

Command %14 uses the SET DEFAULT command to effectively reset the environment variable term_loud back to TRUE. Debug output once again is displayed on the terminal.

Command %15 issues an MPE XL CI command PRINT CODEDUMP to display the newly created list file with the disassembled code. Note the additional Debug commands that were captured in the list file.

```
%10 (%53) cmdebug > list codedump
%11 (%53) cmdebug > env term_loud false
%12 (%53) cmdebug > dc ?fopen,20
%13 (%53) cmdebug > list close
%14 (%53) cmdebug > set def
%15 (%53) cmdebug > :print codedump

Page: 1     DEBUG/XL A.01.00    WED, FEB 23, 1987  11:42 AM

%11 (%53) cmdebug > env term_loud false
%12 (%53) cmdebug > dc ?fopen,20
SYS %22.5000
%005000:  ?FOPEN                   170404  ..  LRA   P-4
```

```
%005001:  FOPEN+%5              030400  1.  SCAL  0
%005002:  FOPEN+%6              000600  ..  ZERO, NOP
%005003:  FOPEN+%7              051451  S)  STOR  Q+%51
%005004:  FOPEN+%10             140060  .0  BR    P+%60
%005005:  FOPEN+%11             140003  ..  BR    P+3
%005006:  ?FSOPEN               170412  ..  LRA   P-%12
%005007:  FOPEN+%13             030400  1.  SCAL  0
%005010:  FOPEN+%14             021001  ".  LDI   1
%005011:  FOPEN+%15             051451  S)  STOR  Q+%51
%005012:  FOPEN+%16             140052  .*  BR    P+%52
%005013:  FOPEN+%17             140003  ..  BR    P+3
%005014:  ?FJOPEN               170420  ..  LRA   P-%20
%005015:  FOPEN+%21             030400  1.  SCAL  0
%005016:  FOPEN+%22             021002  ".  LDI   2
%005017:  FOPEN+%23             051451  S)  STOR  Q+%51
%13 (%53) cmdebug > list close
```

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

DATAB

Debug only

Privileged Mode

Sets a data breakpoint.

Syntax

```
DATAB virtaddr [:pin|@] [byte_count] [count] [loudness] [cmdlist]
```

Data breakpoints "break" when the indicated address is written to. The debugger stops at the instruction that is about to perform the write operation.

The `DATAB` command is used to set process-local and global (system-wide) data breakpoints.

Setting a breakpoint for another process is implemented so that it appears the target process set the breakpoint itself. Therefore, when the target process encounters the breakpoint, it enters Debug with its output directed to the LDEV associated with that process.

Parameters

virtaddr — The virtual address at which to set the data breakpoint.

Virtaddr can be a short pointer, a long pointer, or a full logical code pointer.

pin |@ — The process identification number (PIN) of the process for which the breakpoint is to be set. If omitted, the breakpoint is set for the current process. The character "@" can be used to set a global breakpoint at which all processes stop.

byte_count — *Byte_count* specifies the number of bytes to "protect" with the data breakpoint. If no value is given, one byte is assumed.

count — *Count* has a twofold meaning: it specifies to break every *n*th time the breakpoint is encountered, and it is used to set permanent/temporary breakpoints.

If *count* is positive, the breakpoint is permanent. If *count* is negative, the breakpoint is temporary and is deleted as soon as the process attempts to modify the protected address. For example, a *count* of 4 means break every fourth time the protected address range is modified; a *count* of -4 means break on the fourth time, and immediately delete the breakpoint. If *count* is omitted, +1 is used, which breaks every time the address range is written to, permanently.

loudness Either LOUD or QUIET. If QUIET is selected the debugger does not print out a message that the breakpoint has been hit. This is useful for performing a command list a great number of times before stopping without being inundated with screen after screen of breakpoint messages. These keywords may be abbreviated as desired. The default value is LOUD.

cmdlist A single Debug command or a list of Debug commands that are executed immediately when the breakpoint is encountered. Command lists for breakpoints are limited to 80 characters. (If this is too few characters, write a macro and have the command list invoke the macro). *Cmdlist* has the form:

```
CMD1

{ CMD1; CMD2; CMD3; ... }
```

Examples

```
$ nmdebug > datab dp+c14,8
added:    [1] 49.40150c68 for 8 bytes
```

Set a data breakpoint at DP+c14. (We will assume it's a global variable.) Protect 8 bytes starting at that address.

```
$ nmdebug > datab r24,c4,-1
added:   T[2] 49.401515d4 for c4 bytes
```

Set a temporary data breakpoint at the address pointed to by general register 24. For this example we assume that r24 contains a pointer to the user's dynamic heap space. Protect c4 bytes starting at that address. The breakpoint is a temporary breakpoint (that is, it is deleted after it is encountered for the first time).

```
$ nmdebug > databl
     [1] 49.40150c68 for 8 bytes
    T[2] 49.401515d4 for c4 bytes
         count 0/1
```

Now list the data breakpoints we have just set.

Limitations, Restrictions

Keep in mind that the architecture supports data breakpoints on a page basis only. Anything more granular requires substantial software intervention.

Caution Data breakpoints on process stacks are not supported, and setting breakpoints there may crash the system.

Breakpoints set in the global data area of a user's stack are safe as long as the page containing the global data contains only global data (that is, the process does not use that page for stacking procedure call frames or local data).

Setting data breakpoints at addresses on a process stack can severely degrade performance of the process.

Data breakpoints are ignored in the following circumstances:

- While on the ICS (interrupt control stack).
- While disabled.
- In a "dying" process (See `ENV DYING_DEBUG`).
- In a job (See `ENV JOB_DEBUG`).

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

DATABD

Debug only

Privileged Mode

Deletes a data breakpoint entry specified by index number.

Syntax

```
DATABD [number | @ [: pin | @] ]
```

The `DATABD` command is used to delete process-local data breakpoints and global (system-wide) data breakpoints.

Parameters

number | @ — The index number of the data breakpoint entry that is to be deleted. The character "@" can be used to delete all breakpoint entries.

If the index number is omitted, Debug displays each breakpoint, one at a time, and asks the user if it should be deleted (Y/N?). The following responses are recognized:

`Y[E[S]]`	Yes, remove the breakpoint.
`YES` *any_text*	Yes, remove the breakpoint.
`N[O]`	No, do not remove the breakpoint.
`NO` *any_text*	No, do not remove the breakpoint.

If any other response is given, the default value `NO` is assumed.

pin | @ — The PIN for the process whose data breakpoint entry is to be deleted. Typically this is omitted, and *pin* defaults to the current process.

The character “@” can be used to specify that a global breakpoint is to be deleted.

Examples

```
$ nmdebug > databl
    [1] 49.40150c68 for 8 bytes
   T[2] 49.401515d4 for c4 bytes
        count 0/1
   @[1]  c.c1040480 for 4 bytes
        cmdlist: {WL "pib data breakpoint was hit"}
```

List the data breakpoints that exist.

```
$ nmdebug > databd
    [1] 49.40150c68 for 8  bytes      (Y/N) ?
   T[2] 49.401515d4 for c4 bytes      (Y/N) ?
   @[1]  c.c1040480 for 4  bytes      (Y/N) ? y
```

Display each breakpoint and ask the user if the breakpoint should be deleted. In this example, the global breakpoint is deleted.

```
$ nmdebug > databd 1
deleted:  [1]  49.40150c68 for 8 bytes
```

Delete data breakpoint number 1.

```
$ nmdebug > databl
   T[2] 49.401515d4 for c4 bytes
        count 0/1
```

List the data breakpoints that remain.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

DATABL

Debug only

Privileged Mode

Lists data breakpoint entries, specified by index number.

Syntax

```
DATABL [number | @ [: pin | @] ]
```

The DATABL command is used to list process-local and global (system-wide) data breakpoints. Global data breakpoints are always displayed after the process-local data breakpoints.

Parameters

number The index number of the data breakpoint entry to display. The symbol "@" can be used to display all entries. If omitted, all entries are displayed.

pin The PIN number for the process whose data breakpoint entries are to be displayed. Typically this is omitted, and *pin* defaults to the current process.

The character "@" can be used to indicate global data breakpoint(s).

Examples

```
$ nmdebug > databl
    [1] 49.40150c68 for 8 bytes
   T[2] 49.401515d4 for c4 bytes
        count 0/1
   @[1]  c.c1040480 for 4 bytes
```

DATABL

```
          cmdlist: {WL "pib data breakpoint was hit"}
```

Display all data breakpoints. Process-local breakpoints are always displayed first, then global breakpoints are displayed.

```
$ nmdebug > databl 1
    [1]  49.40150c68 for 8 bytes
```

Display data breakpoint number 1.

```
$ nmdebug > databl @:@
   @[1]  c.c1040480 for 4 bytes
        cmdlist: {WL "pib data breakpoint was hit"}
```

Display all of the global data breakpoints.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

DEBUG

DAT only

Privileged Mode

`DEBUG` command—access to DEBUG XL.

Syntax

```
DEBUG
```

Parameters

none

Examples

```
$nmdat > debug
DEBUG XL A.00.00

DEBUG Intrinsic at: 401.000b431c do_the_command+2c4
$1 ($38) nmdebug >
```

Limitations, Restrictions

The `DEBUG` command is generally useful only to the developer of DAT.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

DELETE*xxx*

Delete various items. These are predefined aliases for other commands.

Syntax

```
DELETEB        alias for  BD
DELETEALIAS    alias for  ALIASD
DELETEERR      alias for  ERRD
DELETEMAC      alias for  MACD
DELETEVAR      alias for  VARD
```

See the `ALIASINIT` command.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

DEMO

Privileged Mode

Adds/deletes/lists terminals used for demonstrating System Debug.

Syntax

```
DEMO
DEMO LIST
DEMO ADD    ldevs
DEMO DELETE ldevs
```

The `DEMO` command is used for giving demonstrations of System Debug. With this command, the user is able to enslave up to 50 terminals. Each of the enslaved terminals receives all input and output generated by System Debug. Output generated by the CI through the use of the `":"` command or `CIGETVAR` and `CIPUTVAR` functions is not sent to the enslaved terminals.

Please read and heed the warnings listed in "Limitations, Restrictions."

Parameters

`DEMO`
`DEMO LIST` — List the terminal LDEV's that currently are receiving System Debug I/O. Both command forms are identically supported.

`DEMO ADD` — This keyword tells System Debug to add the following LDEVs to the list of terminals to receive a copy of all System Debug I/O.

`DEMO DELETE` — This keyword tells System Debug to remove the following LDEVs from the list of terminals that receive a copy of all System Debug I/O.

ldevs — A list of terminal LDEV numbers (logical device numbers), separated by blanks or commas. A note of caution: remember that the LDEV numbers are interpreted using the current input base for System Debug.

Examples

```
$nmdat > demo
No demonstration terminals are defined

$nmdat > demo add #200 #201 #205 #206

$nmdat > demo list
DEMO LDEVS (#): 200 201 205 206
```

First, check to see if any demonstration LDEVs have been specified. Next, add four LDEVs to the list of terminals to receive a copy of DAT's input and output stream. As soon as the `DEMO ADD` command is processed, the indicated terminals begin receiving I/O. Finally, display the list of demonstration terminals.

Limitations, Restrictions

A total of 50 demonstration LDEVs are supported.

The functionality is implemented with low-level I/O routines. I/O is done directly to the LDEV. No attempt is made to lock or obtain ownership of the LDEV before sending data to it. Nonpreemptive I/O is used when sending data to the LDEVs. Therefore, if a read is pending at the LDEV (For example, the CI prompt), System Debug blocks until the pending read is satisfied. It is good practice to free up the LDEVs that will be used during a demonstration by issuing the `:RESTORE` command at each terminal (do not REPLY to the resulting tape request). This removes any pending I/O from the LDEV. When the demonstration is finished, break out of the RESTORE process and issue an `ABORT` command.

No validation of LDEV numbers is performed. If you give an *ldev*, then no matter what the value is, System Debug tries to write to it!

The same LDEV may be specified more than once, in which case the LDEV is sent a copy of any I/O for each occurrence in the list of LDEVs.

DEMO

The Control-S/Control-Q/stop keys suspend output only for the master terminal (that is, the one where the demonstration is being run). All of the enslaved terminals continue to receive output as an uninterrupted flow.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

DIS

Disassembles a single NM or CM assembly instruction, based on the current mode.

Syntax

```
DIS nmword [virtaddr]
DIS cmword1 [cmword2] [cmlogaddr]
```

The `DCx` (display code) commands can be used to display a block of code at a specified address. The program windows also display disassembled code.

Parameters

nmword The Precision Architecture instruction to disassemble. All disassembled values are in decimal unless otherwise indicated.

cmword1 The CM HP 3000 instruction to disassemble.

cmword2 A second CM HP 3000 instruction to disassemble for double-word instructions.

virtaddr If a virtual address is given, this value is used when computing branch addresses. That is, "disassemble this instruction as if it were at the indicated address." A valid virtual address results in branch targets being printed as a procedure name plus offset. If this value is omitted, branch targets always appear as numeric values.

cmlogaddr If a CM logical address is specified, the address is used to compute the targets of CM `PCAL` instructions.

Cmlogaddr must be a full CM logical code address (`LCPTR`).

For example,

`CMPC` Current CM program counter

`CMPW+4`	Top of CM program window + 4
`PROG(2.102)`	Program file logical seg 2 offset 102
`fopen+102`	CM procedure `fopen` + %102 (assumes CM mode)
`cmaddr('fopen')+%102`	CM procedure `fopen` + %102 (NM or CM mode)

Examples

```
$nmdebug > dis 6bc23fd9
STW      2,-20(0,30)
```

This NM example disassembles the NM word $6bc23fd9 into the `STW` instruction.

```
$nmdebug > dis e84001d8
BL       $000000f4,2
$nmdebug > dis e84001d8, a.4adeb4
BL       test_proc+$68,2
```

This NM example disassembles the word $e84001d8 into a `BL` instruction. In the second command, the virtual address of the instruction is specified, and the disassembler is able to compute and to display the effective procedure name target of the branch.

```
%cmdebug > dis 41101
LOAD  DB+%101
```

This CM example disassembles the single CM word %41101 into the `LOAD DB+%101` instruction.

```
%cmdebug > dis 20477 43
LDDW  SDEC=1
```

This CM example disassembles the two CM words, %20477 and %43, into the `LDDW SDEC=1` instruction.

```
%cmdat > dis 31163
PCAL %163
%cmdat > dis 31163,,sys(25.0)
PCAL ?SWITCH'TO'NM'
%cmdat > dis 31163,,sys(1.0)
PCAL ?ATTACHIO
```

These CM examples involve the CM `PCAL` instruction. In the first example, `31163` is recognized as the `PCAL` instruction, but the STT number is invalid for the current CM segment. In the second example, the instruction is disassembled as if it were found in CM logical segment `SYS %25`, and the resulting destination of the `PCAL` is displayed as `?SWITCH'TO'NM`. The third example indicates that within CM logical segment SYS 1, the resulting target of a PCAL %163 is ?ATTACHIO.

```
%cmdat > var n 1
%cmdat > while 1 do {w "stt: " n:"w3" "  " ;dis 31000+n; var n n+1}
stt:   %1  PCAL ?TERMINATE
stt:   %2  PCAL ?TERMINATE
stt:   %3  PCAL ?ABORTJOB
stt:   %4  PCAL ?ACTIVATE
stt:   %5  PCAL ?ADOPT
stt:   %6  PCAL ?ONENET'ADOPT
stt:   %7  PCAL ?CREATEPROCESS
stt:  %10  PCAL ?EXEC'TERMINATE
stt:  %11  PCAL ?GET'PLFD'TBLPTR
stt:  %12  PCAL ?GETORIGIN
stt:  %13  PCAL ?GETPRIORITY
stt:  %14  PCAL ?GETPROCID
stt:  %15  PCAL ?GETPROCINFO
stt:  %16  PCAL ?JSM'TO'CI'PIN
stt:  %17  PCAL ?KILL
stt:  %20  PCAL ?PROCINFO
stt:  %21  PCAL ?PROCTIME
stt:  %22  PCAL ?SET'JSM'TIME'LI
stt:  %23  PCAL ?SET'PLFD'TBLPTR
stt:  %24  PCAL ?SUSPEND
stt:  %25  PCAL ?XCONTRAP
stt:  %26  PCAL ?NM'BREAKCONTROL
stt:  %27  PCAL ?SETSERVICE
stt:  %30  PCAL ?REQUESTSERVICE
stt:  %31  PCAL ?RESETCONTROL
stt:  %32  PCAL ?CAUSEBREAK
stt:  %33  PCAL ?CAUSEBREAK'
stt:  %34  PCAL ?BRK'IN'BREAK
stt:  %35  PCAL ?BRK'ABORT
stt:  %36  PCAL ?BRK'RESUME

control-Y encountered
%cmdat >
```

This example demonstrates how a simple loop can be used to display the targets for each STT entry within the current CM segment. Since we know that `%31000` is the `PCAL` instruction, we simply add the desired STT number and use the `DIS` command to display the target entry point name. Control-Y is used to terminate the loop.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

DO

Reexecutes a command from the command stack.

Syntax

```
DO [cmd_string ]
DO [history_index]
```

`DO`, entered alone, reexecutes the most recent command.

Parameters

cmd_string	Execute the most recent command in the history stack that commences with *cmd_string*. For example, `do wh` could be used to match the most recent WHILE statement.
history_index	The history stack index of the command that is to be executed. A negative index can be used to specify a command relative to the current command. For example, -2 implies the command used two commands ago.

Examples

```
%cmdebug > do w
%cmdebug > wl 2+4
%6
```

Execute the most recent command that started with “w”.

Limitations, Restrictions

Upon initial entry into System Debug, the command stack is empty, since no prior command has been executed. If the `DO` command is entered as the first command, an empty command is reexecuted. This is effectively the same as entering a blank line.

The MPE XL command interpreter allows an edit string to be specified on the `DO` command line. This feature is not supported in System Debug.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

DPIB

DAT only

Display data from the process identification block (PIB) for a process. You can use `DPIB` in both native mode and compatibility mode.

Syntax

```
DPIB  [pin]
```

Parameters

pin The process identification number for the process whose PIB values are to be displayed. If no *pin* is specified, the current *pin* is used.

Examples

```
%cmdebug > dpib 2

PIN: 20   Pid:  0000002000000001   Process state: 1   Space ID:  000002c4

PCB    : 80001b40  PCBX    : 40011cb0  PIBX    : 83980000  CMGLB   : 83980000
Parent : 80e0db18  Sibling : 00000000  Child   : 00000000  JSMAIN  : 80e0d5c0
```

Display the PIB values for PIN 2.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the

exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

DPTREE

DAT only

Prints out the process tree starting at the given PIN.

Syntax

```
DPTREE [pin]
```

Parameters

pin The process identification number (PIN) where the process tree display starts. If omitted, PIN 1 (the first PIN in all process trees) is assumed, and the entire process tree is printed.

Examples

```
$nmdat > dptree

 1 ( PROGEN.PUB.SYS )
    2 ( LOAD.PUB.SYS )
    3 ( .. )
    4 ( .. )
    5 ( .. )
    6 ( LOG.PUB.SYS )
    7 ( SYSMAIN.PUB.SYS )
    9 ( SESSION.PUB.SYS )
       a ( JSMAIN.PUB.SYS )
          15 ( CI.PUB.SYS )
       16 ( JSMAIN.PUB.SYS )
          17 ( CI.PUB.SYS )
             12 ( FCOPY.PUB.SYS )
    8 ( JOB.PUB.SYS )
       b ( JSMAIN.PUB.SYS )
```

```
      c ( DIAGMON.DIAG.SYS )
         d ( RUNPROG.DIAG.SYS )
            e ( MEMLOGP.DIAG.SYS )
         f ( RUNPROG.DIAG.SYS )
            10 ( LOGGER.DIAG.SYS )

$nmdat >
```

Prints out the entire process tree.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

DR

Displays contents of the CM or NM registers.

Syntax

```
DR [cm_register] [base]
DR [nm_register] [base]
```

Parameters

cm_register The CM register to be displayed. This can be the:

DB	The stack base relative word offset of DB.
DBDST	The DB data segment number.
DL	The DL register word offset, DB relative.
CIR	The current instruction register.
CMPC	The full logical CM program counter address.
MAPDST	The CST expansion mapping data segment number.
MAPFLAG	The CST expansion mapping bit.
Q	The Q register word offset, DB relative.
S	The S register word offset, DB relative.
SDST	The CM stack data segment number.
STATUS	The CM status register.
X	The X (index) register.

If *cm_register* is omitted, all of the above CM registers are displayed.

nm_register The NM register to be displayed.

If no value is provided, all NM registers are displayed (excluding the floating-point registers). The `ENVL ,FP` command displays all of the floating-point registers at once.

To fully understand the use and conventions for the various registers, refer to the *Precision Architecture and Instruction*

Reference Manual (09740-90014) and *Procedure Calling Conventions Reference Manual* (09740-90015). (These may be ordered as a set with the part number 09740-64003.) The *Procedure Calling Conventions Reference Manual* is of particular importance for understanding how the language compilers utilize the registers to pass parameters, return values, and hold temporary values.

The following tables list the native mode registers available within System Debug. Many registers have aliases through which they may be referenced. Alias names in *italics* are not available in System Debug.

Access rights abbreviations are listed below. PM indicates that privileged mode (PM) capability is required.

`d`	Display access
`D`	PM display access
`m`	Modify access
`M`	PM modify access

The following registers are known as the *General Registers*.

Name	Alias	Access	Description
`R0`	*none*	d	A constant 0
`R1`	*none*	dm	General register 1
`R2`	*none*	dm	Used to hold RP at times
`R3`	*none*	dm	General register 3
⋮			
`R22`	*none*	dm	General register 22
`R23`	`ARG3`	dm	Argument register 3
`R24`	`ARG2`	dm	Argument register 2
`R25`	`ARG1`	dm	Argument register 1
`R26`	`ARG0`	dm	Argument register 0
`R27`	`DP`	dM	Global data pointer
`R28`	`RET1`	dm	Return register 1
`R29`	`RET0`	dm	Return register 0
	`SL`	dm	Static link
`R30`	`SP`	dM	Current stack pointer
`R31`	*MRP*	dm	Millicode return pointer

The following registers are pseudo-registers. They are not defined in the Precision Architecture, but are terms used in the procedure calling conventions document and by the language compilers. They are provided for convenience. They are computed based on stack unwind information. They may not be modified.

Name	Alias	Access	Description
`RP`	*none*	d	Return pointer (not the same as R2)
`PSP`	*none*	d	Previous stack pointer

The following registers are known as the *Space Registers*. Registers SR4 through SR7 are used for short pointer addressing:

Name	Alias	Access	Description
`SR0`	*none*	dm	Space register 0
`SR1`	*SARG*	dm	Space register argument
	SRET	dm	Space return register
`SR2`	*none*	dm	Space register 2
`SR3`	*none*	dm	Space register 3
`SR4`	*none*	dM	Process local code space (tracks PC space)
`SR5`	*none*	dM	Process local data space
`SR6`	*none*	dM	Operating system data space 1
`SR7`	*none*	dM	Operating system data space 2

The following registers are known as the *Control Registers*. They contain system state information.

Name	Alias	Access	Description
CR0	RCTR	dM	Recovery counter
CR8	PID1	dM	Protection ID 1 (16 bits)
CR9	PID2	dM	Protection ID 2 (16 bits)
CR10	CCR	dM	Coprocessor configuration (8 bits)
CR11	SAR	dm	Shift amount register (5 bits)
CR12	PID3	dM	Protection ID 3 (16 bits)
CR13	PID4	dM	Protection ID 4 (16 bits)
CR14	IVA	dM	Interrupt vector address
CR15	EIEM	dM	External interrupt enable mask
CR16	ITMR	dM	Interval timer
CR17	PCSF	dM	PC space queue front
none	PCSB	dM	PC space queue back
CR18	PCOF	dM	PC offset queue front
none	PCSB	dM	PC offset queue back
none	PCQF	dM	PC queue (PCOF.PCSF) front
none	PCQB	dM	PC queue (PCOB.PCSB) back
none	PC	dM	PCQF with priv bits set to zero.
none	PRIV	dM	Low two order bits (30,31) of PCOF.
CR19	IIR	dM	Interrupt instruction register
CR20	ISR	dM	Interrupt space register
CR21	IOR	dM	Interrupt offset register
CR22	IPSW	dM	Interrupt processor status word
	PSW	dM	Processor status word
CR23	EIRR	dM	External interrupt request register
CR24	TR0	dM	Temporary register 0
⋮			
CR31	TR7	dM	Temporary register 7

Note

The *Precision Architecture and Instruction Reference Manual* refers to the PC (*program counter*) registers as the IA (*instruction address*) registers. This manual will use the PC mnemonic when referring to the IA registers.

The following registers are floating-point registers. If a machine has a floating-point coprocessor board, these values are from that board. If no floating-point hardware is present, the operating system emulates the function of the hardware; in that case these are the values from floating-point emulation.

Name	Alias	Access	Description
FP0	*none*	dm	FP register 0
FP1	*none*	dm	FP register 1
FP2	*none*	dm	FP register 2
FP3	*none*	dm	FP register 3
FP4	*FARG0*	dm	FP argument register 0
	FRET	dm	FP return register
FP5	*FARG1*	dm	FP argument register 1
FP6	*FARG2*	dm	FP argument register 2
FP7	*FARG3*	dm	FP argument register 3
FP8	*none*	dm	FP register 8
⋮			
FP15	*none*	dm	FP register 15
FPSTATUS	*none*	dm	FP status reg (left half of FP0)
FPE1	*none*	dm	FP exception reg 1 (right half of FP0)
FPE2	*none*	dm	FP exception reg 2 (left half of FP1)
FPE3	*none*	dm	FP exception reg 3 (right half of FP1)
FPE4	*none*	dm	FP exception reg 4 (left half of FP2)
FPE5	*none*	dm	FP exception reg 5 (right half of FP2)
FPE6	*none*	dm	FP exception reg 6 (left half of FP3)
FPE7	*none*	dm	FP exception reg 7 (right half of FP3)

base Specifies the base used to display the register data.

% or OCTAL	Octal representation
# or DECIMAL	Decimal representation
$ or HEXADECIMAL	Hexadecimal representation
ASCII	ASCII representation

This parameter can be abbreviated to as little as a single character.

Examples

```
%cmdebug > dr
DBDST=%132    DB=%1000   X=%102    STATUS=%140075=(MItroc CCG 075)
SDST=%132     DL=%650    Q=%1006   S=%1007   CMPC=PROG %12.2046
SEG =%12      P=%2046    CIR=%000700   MDST=%0
```

Display the contents of all CM registers.

```
%cmdebug > dr status
STATUS=%022002=(miTRoC CCE 002)
```

Display the contents of the CM status register.

```
$nmdebug > dr

R0 =00000000 00464800 005a6e48 00000000 R4 =00000000 00000000 00000000 00000000
R8 =00000000 00000000 00000000 00000000 R12=00000000 00000000 00000000 00000000
R16=00000000 00000000 00000000 0000002a R20=00000006 00007fff ffff8000 400524a8
R24=400524a0 00000400 40052058 c0080008 R28=00000000 00000000 40052520 0000003f

IPSW=0006ff0f=jthlnxbCVmrQPDI PRIV=0000 SAR=0010 PCQF=a.5a6e48   a.5a6e4c

SR0=0000000a 00000057 00000017 00000000 SR4=0000000a 00000057 0000000a 0000000a
TR0=007ea040 0080a040 0000000a 007727c0 TR4=40052848 400526a8 00bba1e0 00bba228

PID1=0020=0010(W) PID2=0000=0000(W)     PID3=0000=0000(W) PID4=0000=0000(W)
RCTR=ffffffff ISR=00000057 IOR=4005250c IIR=6bc23fd9 IVA=001cb000 ITMR=5b8b1e69
EIEM=ffffffff EIRR=00000000 CCR=0000
```

Display all NM registers.

```
$nmdebug > dr pcqb
PCQB=0000000a.0021d7b8
```

Display the contents of "pcq back".

```
$nmdebug > dr pid2
PID2=$0004=0002(W)
```

Display the contents of protection ID register number 2.

Limitations, Restrictions

Floating-point registers are displayed as 64-bit long pointers. No interpretation of the data is attempted.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

DUMPINFO

DAT only

Displays dump file information.

Syntax

```
DUMPINFO [options]
```

Parameters

options This parameter specifies what information is to be displayed. If no option is given, `STATE` is assumed. The following list shows the valid options:

`STATE`	Display the last active PIN and the state of the system at the time the dump was taken.
`DIRECTORY`	Display the dump file directory.
`MAP`	Display a map of all secondary store addresses dumped.
`TABLES`	Display the basic machine characteristics, such as memory size, register pointers, and address translation tables location.
`CACHE`	Display internal cache statistics.
`ALL`	Display all the above information.

Examples

```
$nmdat > DUMPINFO

Dump Title: SA 2559 on KC (8/29/88 9:40)
Last PIN  : 34 - On ICS -- Dispatcher running

$nmdat >
```

Display the dump title (entered by the dump operator) and the machine state at the time the dump was taken.

```
$nmdat> DUMPINFO DIR

Dump file set  . . . . . . D7054.DUMP.CMDEBUG
Dumped OS  . . . . . . . . MPE-XL (99999X B.09.22)
Dump tape creator  . . . . SOFTDUMP (99999X A.00.02)
Dump disc file creator . . DAT/XL ( X.09.00)
Tape format ID . . . . . . 9.00.00
Tape creation date . . . . THU, MAY 16, 1991,  3:23 PM
Tape compression . . . . . 36% (RLE)
Dump disc format ID  . . . B.01.00

 NAME    LDEV  DESC     BYTES     MBYTES    BYTES RESTORED  (All decimal)

                 DUMP DIRECTORY (All Values Decimal)
                 -----------------------------------
 NAME    LDEV  DESC     BYTES     MBYTES    BYTES RESTORED  COMPRESSION

PIM00                        4096     0.0        4096, 100%
MEMDUMP                  50331648    48.0    50331648, 100%         61%
VM001       1    66      41013248    39.1    41013248, 100%         79%
VM002       2     3        585728     0.6      585728, 100%         82%
VM003       3     2         61440     0.1       61440, 100%         84%
VM004       4   209      17227776    16.4    17227776, 100%         82%
VM014      14     3        585728     0.6      585728, 100%         83%

Dump disc file space reduced by 71% due to LZ data compression.

$nmdat >
```

Display the dump file directory.

```
$nmdat > dumpinfo tables

Logical page size: 00001000    Memory size      : 03000000
Hash table adress: 00744200    Hash table length: 00040000
PDIR table adress: 006e4200    PDIR table length: 00060000
REALGLOB  address: 00788000    ICS address      : 009cf000
TCB table address: 009f7000    Current TCB adr  : 00a000a0
```

DUMPINFO

```
$nmdat >
```

Display the basic machine characteristics.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

ENV

Assigns a new value to one of the predefined environment variables.

Syntax

```
ENV var_name [=] var_value
```

The environment variables allow control and inspection of the operation of System Debug.

Parameters

var_name The name of the environment variable to set.

var_value The new value for the variable, which can be an expression.

The environment variables are logically organized in the following groups:

`(cmd)`	Command related
`(cmreg)`	Compatibility mode registers
`(const)`	Predefined constants
`(fpreg)`	Native mode floating-point registers
`(io)`	Input/output related
`(limits)`	Limits
`(misc)`	Miscellaneous
`(nmreg)`	Native mode registers
`(system)`	System-wide Debug registers
`(state)`	All nmreg + cmreg + fpreg registers
`(win)`	Window

Access rights abbreviations are listed below. PM indicates that privileged mode (PM) capability is required.

d	Display access (`DR` command)

D	PM display access (DR command)
m	Modify access (MR command)
M	PM modify access (MR command)
r	Read access
R	PM read access
w	Write access
W	PM write access

Two names separated by a hyphen indicate a range of names. For example, ARG0 - ARG3 implies the full range: ARG0, ARG1, ARG2, and ARG3.

The Environment Variables - Sorted by Group

The following table lists all environment variables, arranged by their logical groups. A full alphabetically-sorted listing and description of each variable can be found following this table.

const - constants

```
const   r    FALSE            : BOOL
const   r    TRUE             : BOOL
```

cmd - command related

```
cmd     rw   AUTOIGNORE       : BOOL
cmd     rw   AUTOREPEAT       : BOOL
cmd     rw   CMDLINESUBS      : BOOL
cmd     rw   CMDNUM           : U32
cmd     rw   ECHO_CMDS        : BOOL
cmd     rw   ECHO_SUBS        : BOOL
cmd     rw   ECHO_USE         : BOOL
cmd     rw   ERROR            : S32
cmd     r    MACRO_DEPTH      : U16
cmd     rw   MULTI_LINE_ERRS  : U16
cmd     rw   NONLOCALVARS     : BOOL
cmd     rw   TRACE_FUNC       : U16
```

io - input/output

```
io      rw   CM_INBASE        : STR
io      rw   CM_OUTBASE       : STR
```

ENV

```
io    r     COLUMN            : U16
io    rW    CONSOLE_IO        : BOOL     (Debug only)
io    rw    FILL              : STR
io    rw    FILTER            : STR
io    rw    HEXUPSHIFT        : BOOL
io    rw    INBASE            : STR
io    rw    JUSTIFY           : STR
io    rw    LIST_INPUT        : BOOL
io    rw    LIST_PAGELEN      : U16
io    r     LIST_PAGENUM      : U16
io    rw    LIST_PAGING       : BOOL
io    rw    LIST_TITLE        : STR
io    rw    LIST_WIDTH        : U16
io    rw    NM_INBASE         : STR
io    rw    NM_OUTBASE        : STR
io    rw    OUTBASE           : STR
io    rw    PROMPT            : STR
io    rw    TERM_KEEPLOCK     : BOOL    (Debug only)
io    rW    TERM_LDEV         : U16     (Debug only)
io    rw    TERM_LOCKING      : BOOL    (Debug only)
io    rw    TERM_LOUD         : BOOL
io    rw    TERM_PAGING       : BOOL
io    rw    TERM_WIDTH        : U16
```

limits - limits for macros and variables

```
limits  rw    MACROS             : U16
limits  r     MACROS_LIMIT       : U16
limits  rw    VARS               : U16
limits  r     VARS_LIMIT         : U16
limits  rw    VARS_LOC           : U16
limits  r     VARS_TABLE         : U16
```

misc - miscellaneous

```
misc    rW    CCODE              : STR     (Debug only)
misc    rw    CHECKPSTATE        : BOOL
misc    r d   CPU                : U16
misc    rW    CSTBASE            : LPTR
misc    r     DATE               : STR
misc    r     DISP               : BOOL
misc    rW    DSTBASE            : LPTR
misc    rw    DUMPALLOC_LZ       : U16
misc    rw    DUMPALLOC_RLE      : U16
misc    r     DUMP_COMP_ALGO     : STR
misc    r     ENTRY_MODE         : STR
misc    rW    ESCAPECODE         : U32     (Debug only)
misc    r     EXEC_MODE          : STR
misc    rw    GETDUMP_COMP_ALGO  : STR
misc    r     ICSNEST            : U16
misc    r     ICSVA              : LPTR
misc    r     LASTPIN            : U16
misc    rw    LOOKUP_ID          : STR
misc    r     MODE               : STR
misc    r d   MONARCHCPU         : U16
misc    rw    MPEXL_TABLE_VA     : LPTR
misc    r     PIN                : U16
misc    rW    PRIV_USER          : BOOL
misc    r     PROGNAME           : STR
misc    r d   PSEUDOVIRTREAD     : BOOL
misc    rw    PSTMT              : U16
misc    rw    QUIET_MODIFY       : BOOL
misc    rw    SYMPATH_UPSHIFT    : BOOL
misc    r     SYSVERSION         : STR
```

ENV

```
misc    r     TIME              : STR
misc    r     VERSION           : STR
```

win - window

```
win     rw    CHANGES          : STR
win     rw    CMPW             : LCPTR
win     r     LW               : SADDR
win     rw    MARKERS          : STR
win     r     NMPW             : LCPTR
win     r     PW               : LCPTR
win     r     PWO              : SPTR
win     r     PWS              : U32
win     r     SHOW_CCTL        : BOOL
win     r     VW               : LPTR
win     r     VWO              : SPTR
win     r     VWS              : U32
win     rw    WIN_LENGTH       : U32
win     rw    WIN_WIDTH        : U32
win     r     ZW               : U32
```

cmreg - compatibility mode regs

```
cmreg   r dm  CIR              : S16
cmreg   r dm  CMPC             : LCPTR
cmreg   r dm  DB               : S16
cmreg   r dm  DBDST            : S16
cmreg   r dm  DL               : S16
cmreg   r d   MAPDST           : S16
cmreg   r d   MAPFLAG          : S16
cmreg   r dm  Q                : S16
cmreg   r dm  S                : S16
cmreg   r dm  SDST             : S16
cmreg   r dm  STATUS           : S16
cmreg   r dm  X                : S16
```

nmreg - native mode regs

```
nmreg   r dm  ARG0 - ARG3       : U32
nmreg   r dM  CCR               : U16
nmreg   r dm  CR0               : U32
nmreg   r dm  CR8 - CR31        : U32
nmreg   r dm  DP                : U32
nmreg   r dM  EIEM              : U32
nmreg   r dM  EIRR              : U32
nmreg   r dM  IIR               : U32
nmreg   r dM  IOR               : U32
nmreg   r dM  IPSW              : U32
nmreg   r dM  ISR               : U32
nmreg   r dM  ITMR              : U32
nmreg   r dM  IVA               : U32
nmreg   r dm  PC                : LPTR
nmreg   r dm  PCOB              : U32
nmreg   r dm  PCOF              : U32
nmreg   r dm  PCQB              : LPTR
nmreg   r dm  PCQF              : LPTR
nmreg   r dm  PCSB              : U32
nmreg   r dm  PCSF              : U32
nmreg   r dM  PID1 - PID4       : U16
nmreg   r dM  PRIV              : BOOL
nmreg   r d   PSP               : U32
nmreg   r dM  PSW               : U32
nmreg   r d   R0                : U32
nmreg   r dm  R1 - R31          : U32
nmreg   r dM  RCTR              : U32
nmreg   r dm  RET0              : U32
nmreg   r dm  RET1              : U32
nmreg   r d   RP                : U32
nmreg   r dm  SAR               : U16
nmreg   r dm  SL                : U32
nmreg   r dm  SP                : U32
nmreg   r dm  SR0 - SR7         : U32
nmreg   r dM  TR0 - TR7         : U32
```

fpreg - floating point regs

```
fpreg   r dM  FP0 - FP15        : LPTR    (until S64 is supported)
fpreg   r dM  FPE0 - FPE7       : U32
fpreg   r dM  FPSTATUS          : U32
```

system - system wide debug

```
system  rW    CONSOLE_DEBUG     : BOOL    (Debug only)
system  rW    DYING_DEBUG       : BOOL    (Debug only)
system  rW    JOB_DEBUG         : BOOL    (Debug only)
```

state - process state

The *state* variables consist of all `NMREG`, `CMREG`, and `FPREG` variables.

The Environment Variables - Sorted Alphabetically

The following table lists all predefined environment variables. Each variable description displays on the first line the variable name and type, group name in parentheses, and access rights, for example:

name **TYPE (group) access [*]**

Environment variable description

Those variables flagged with a "*" have their value reset to their default value if the `SET DEFAULT` command is issued.

`ARG0 - ARG3` **U32 (nmreg) r dm**

NM argument registers. These registers are used by the language compilers for parameter passing. (Alias for R26 - R23)

`AUTOIGNORE` **BOOL (cmd) rw ***

Setting `AUTOIGNORE` is equivalent to using the `IGNORE LOUD` command before every command. When `AUTOIGNORE` is set, System Debug ignores errors (that is, the `ERROR` variable contains a negative value). Among other things, this means that System Debug continues processing USE files, macros, and looping constructs even though an error occurs while doing so. (Refer to the `IGNORE` command.) The default for this variable is FALSE.

`AUTOREPEAT` **BOOL (cmd) rw**

Controls the automatic repetition of the last command whenever a lone carriage return is entered. Setting `AUTOREPEAT` allows repetitive operations (such as single stepping or PF) to be automatically executed by pressing Return. This variable may also be altered with the `SET CRON` and `SET CROFF` commands. The default value for the `AUTOREPEAT` variable is FALSE.

`CCODE` **STR (misc) rW**

Condition code. This value is captured on entry to Debug. It is restored when the debugger resumes the process. Since

Debug itself causes the condition code for the process to change, it is necessary to cache the original value. The following string literals are valid: "CCE", "CCG", "CCL".

`CCR` **U16 (nmreg) r dM**

NM coprocessor configuration register. (Alias for CR10)

`CHANGES` **STR (win) rw**

Selects the type of video enhancement used to flag window values modified since the last command. The following string literals are valid: "INVERSE", "HALFINV", "BLINK", "ULINE", and "FEABLE". Note that this is a string variable; thus, literals must be quoted. The default value is "HALFINV".

CHECKPSTATE	**BOOL (misc) rw**
	If FALSE, inhibits validation of the process state when performing the following functions: PIB, PIBX, PCB, PCBX, CMG, CMSTACKBASE, CMSTACKDST, CMSTACKLIMIT, NMSTACKBASE and NMSTACKLIMIT.
CIR	**U16 (cmreg) r dm**
	CM current instruction register.
CMDLINESUBS	**BOOL (cmd) rw**
	Setting CMDLINESUBS enables command line substitutions (for example, expanding the "|" character in-line). When macro bodies use command line substitutions, it is sometimes desirable to disable CMDLINESUBS while reading the macro definitions in from a USE file. (Refer to the ECHO_SUBS variable). The default for this variable is TRUE.
CMDNUM	**U32 (cmd) rw**
	The current command number is maintained as a running counter. This value is displayed as part of the default prompt string.
CMPC	**LCPTR (cmreg) r**
	The full logical code address for CM, based on the current logical code file, logical segment number, and offset.
CMPW	**LCPTR (win) r**
	The address (as a logical code address) where the CM program window is aimed.
CM_INBASE	**STR (io) rw**
	The current CM input conversion base. When in cmdebug, all values entered are assumed to be in this base unless otherwise specified. The following values are allowed: % or OCTAL # or DECIMAL

$ or HEXADECIMAL

The names may be abbreviated to a single character. The default value is % (octal). Refer to the SET command for an alternate method of setting this variable.

CM_OUTBASE **STR (io) rw ***

The current CM output display base. The following values are allowed:

% or OCTAL
# or DECIMAL
$ or HEXADECIMAL

The names may be abbreviated to a single character. The default value is % (octal). Refer to the SET command for an alternate method of setting this variable.

COLUMN	**U16 (io) rw** The current character position in the user's output buffer. The position is advanced by the `W` and `WCOL` commands (or by the `C` directive in a format specification). Refer to the `W` command for details.
CONSOLE_DEBUG	**BOOL (system) rW** If this system-wide flag is set, all processes entering the debugger for the first time automatically have their debug I/O performed at the system console with the system console I/O routines. Processes that have already entered Debug and have established a debugging environment are not affected by this variable. When this variable is set, the `CONSOLE_IO` variable is set to TRUE for all processes entering Debug for the first time. Setting `CONSOLE_DEBUG` is useful when doing system debugging. If global breakpoints have been set, all of the I/O can be directed to one terminal by setting this variable. The default value is FALSE. This variable is not available in DAT.
CONSOLE_IO	**BOOL (io) rW** If set, the current process uses the system console I/O routines to perform Debug I/O. No other processes are affected by this command. Note that this variable has precedence over the `TERM_LDEV` variable. System processes and jobs entering Debug (assuming the `JOB_DEBUG` environment variable was set), has this variable set to TRUE upon entry to the debugger. The default value is FALSE. This variable is not available in DAT.
CPU	**U16 (misc) r d** The CPU number of the processor that is being examined.
CR0	**U32 (nmreg) r dm**

NM control register 0 (alias for RCTR). Debug uses this value while single stepping.

CR8 - CR31 **U32 (nmreg) r dm**

NM control registers. These registers have the following aliases and names (for descriptions of their usage, refer to the *Precision Architecture and Instruction Reference Manual* (09740-90014)):

Register	Alias		Description
CR0	RCTR		Recovery counter
CR8	PID1		Protection ID 1
CR9	PID2		Protection ID 2
CR10	CCR		Coprocessor configuration register
CR11	SAR		Shift amount register
CR12	PID3		Protection ID 3
CR13	PID4		Protection ID 4
CR14	IVA		Interrupt vector address
CR15	EIEM		External interrupt enable mask
CR16	ITMR		Interval timer
CR17	PCSF		PC space queue front
CR18	PCOF		PC offset queue front
CR19	IIR		Interrupt instruction register
CR20	ISR		Interrupt space register
CR21	IOR		Interrupt offset register
CR22	IPSW	PSW	Interrupt processor status word
CR23	EIRR		External interrupt request register
CR24	TR0		Temporary register 0
⋮			
CR31	TR7		Temporary register 7

Refer to the PID environment variable entry for a detailed description of the format of PID registers.

Refer to the IPSW environment variable entry for a detailed description of the format for the PSW register.

CSTBASE **LPTR (misc) rW**

The virtual address of the CST table.

DATE **STR (misc) r**

The current date string in the form 'WED, OCT 14, 1951'.

`DB` **U16 (cmreg) r dm**

The CM DB register.

`DBDST` **U16 (cmreg) r dm**

The CM DB DST number.

DISP **BOOL (misc) r**

A Boolean value that indicates whether or not the dispatcher is currently running. This value is always FALSE in Debug.

DL **U16 (cmreg) r dm**

The CM DL register.

DP **U32 (nmreg) r dm**

NM global data pointer register. (Alias for R27)

DSTBASE **LPTR (misc) rW**

The virtual address of the CM DST table.

DUMPALLOC_LZ **U16 (misc) rw**

Determines the percentage of disk space DAT will preallocate before restoring a dump encoded with LZ data compression. The percentage is relative to the space required to contain a fully uncompressed dump. This means if you normally expect your dumps to be compressed by 60%, setting DUMPALLOC_LZ to 40 should preallocate enough disk space to contain the entire dump.

DUMPALLOC_RLE **U16 (misc) rw**

Similar to DUMPALLOC_LZ, except that it applies to dumps encoded with RLE data compression.

DUMP_COMP_ALGO **STR (misc) r**

Set to the data compression algorithm used by the currently opened dump. Possible values are:

"NONE" The dump is not compressed.
"RLE" The dump is RLE-compressed.
"LZ" The dump is LZ-compressed.

DYING_DEBUG **BOOL (system) rW**

When a process is being killed, its state is said to be "dying." Once a process is in this state, Debug normally ignore all breakpoints, traps, and so on. If this system-wide variable is set to TRUE, Debug stops for all events even if the process is dying. This is useful to operating system developers only. It is possible to cause system failures if this variable is turned on and breakpoints are set at inappropriate locations. The default value for this variable is FALSE.

This variable is not available in DAT.

ECHO_CMDS **BOOL (cmd) rw ***

When ECHO_CMDS is set, each command (other than those executed within macros) is echoed just prior to its execution. The default value for this variable is FALSE.

ECHO_SUBS **BOOL (cmd) rw ***

When ECHO_SUBS is set, and CMDLINESUBS is enabled, command line substitutions are displayed as they are performed. In the following example, the first line displays the location of the substitution and the second line displays the result after the substitution has taken place. The default value for this variable is FALSE.

```
subs > fv a.c0341450 "|symfile :student_record"
                      /\
done > fv a.c0341450 "gradtyp:student_record"
```

ECHO_USE **BOOL (cmd) rw ***

When ECHO_USE is set, each command line that is read in from a use file is echoed (along with the name of the USE file), prior to its execution. The USE file name is used as the prompt. The default value for this variable is FALSE.

EIEM **U32 (nmreg) r dm**

NM external interrupt enable mask. (Alias for CR15)

EIRR **U32 (nmreg) r dM**

The NM external interrupt request register. (Alias for CR23)

ENTRY_MODE **STR (misc) r**

This variable contains either "NM" or "CM". For Debug, it indicates whether you entered stopped in cmdebug or nmdebug. For DAT, it just tracks the MODE variable.

ERROR **S32 (cmd) rw**

The ERROR variable contains the most recent error number. It is cleared on entry to any user-defined macro. Refer to the IGNORE command, the ENV variable AUTOIGNORE, and the "Error Handling" section in Chapter 2 for additional error handling information. Note that only negative values constitute errors. Positive values are warnings.

ESCAPECODE	**U32 (misc) rW**
	This is the last ESCAPECODE value that was stored for the process at the moment Debug was entered. This variable is restored when the debugger resumes execution of the process. Since Debug itself causes the escape code for the process to change, it is necessary to cache the original value.
	This variable is not available in DAT.
EXEC_MODE	**STR (misc) r**
	This variable contains either "NM" or "CM". It indicates the execution mode of the current process. This value is obtained from the TCB (operating system data structure). This value does not necessarily match the ENTRY_MODE variable.
FALSE	**BOOL (const) r**
	The constant FALSE.

`FILL` **STR (io) rw ***

This variable determines how leading zeros in right-justified data (refer to `JUSTIFY` variable) are output from the Display commands and in the windows. This variable may take on one of two quoted literal values: "BLANK" (show leading zeros as blanks) or "ZERO" (show leading zeros as zeros). The default value is "ZERO".

`FILTER` **STR (io) rw ***

All output, with the exception of error messages and the prompts, passes through a final filtering process. Those lines that match the value in the `FILTER` variable are displayed and the rest are discarded. By default, `FILTER` is initialized to the blank string (&'&', &"&", or) that matches all output. `FILTER` can be set to a regular expression for the purpose of pattern matching. For example, the following shows how to find the pattern "123" in memory. Only a line that contains "123" *anywhere* in the line is displayed. Note that `FILTER` is displayed as part of the default prompt.

```
$6 ($10) nmdat > env FILTER 123
$7 ($10) nmdat 123> dv a.c0000000, 4000
$ VIRT a.c0001020   $ 40020330  4002033c  40012348  c0002342
$ VIRT a.c0001238   $ c0062344  ffffffff  fffffec2  00000004
$ VIRT a.c0003240   $ 00000001  0000cf42  40012362  000000bc
$8 ($10) nmdat 123> env filter ''
$9 ($10) nmdat >
```

Three lines of output were matched. The pattern "123" has been highlighted in the example to help point out where the pattern was found in the line. Notice that one of the lines contained the pattern as part of the address displayed by the `DV` command. We could use a fancier regular expression to have just those lines with a "123" in the *data* part of the output be displayed. In the following example, the regular expression translates into "Match those lines that start with a dollar sign (`^$`), are followed by any number of any characters (.*), that are followed by a dollar sign and a space ($), and followed by any number of any character (.*), and finally followed by characters 123 (123)."

```
$a ($10) nmdat > env FILTER '^$.*$ .*123'
$b ($10) nmdat ^$.*$ .*123> dv a.c0000000, 4000
$ VIRT a.c0001020   $ 40020330  4002033c  40012348  c0002342
$ VIRT a.c0003240   $ 00000001  0000cf42  40012362  000000bc
$c ($10) nmdat ^$.*$ .*123> set def
$d ($10) nmdat >
```

Note that only those lines with "123" as part of the data output by the DV command were matched and displayed. For additional information on how to specify regular expressions, refer to appendix A.

`FP0-FP15` **LPTR (fpreg) r dm**

NM floating-point registers 0-15. The 64 bits of these registers are presented as long pointers until System Debug supports 64-bit integers.

`FPE1-FPE7` **S32 (fpreg) r dm**

NM floating-point exception registers 1-7. These registers are extracted from FP0-FP3. That is, `FPE1` is an alias for the right 32 bits of FP0, `FPE2` is an alias for the left 32 bits of FP1, and so on. (Refer to the *Precision Architecture and Instruction Reference Manual* (09740-90014).)

`FPSTATUS` **U32 (fpreg) r dm**

NM floating-point status register. (Alias for the left 32 bits of FP0.)

`GETDUMP_COMP_ALGO` **STR (misc) r**

Determines the data compression algorithm to be used when creating a new dump disk file with the `GETDUMP` command. This algorithm may be different from the one used on the dump tape. Possible values are:

`""` or `"DEFAULT"`	Use the best algorithm supported by the current version of DAT.
`"TAPE"`	Use the same algorithm used on the dump tape.
`"NONE"`	Don't compress the dump.
`"RLE"`	Use RLE compression on the disk file.
`"LZ"`	Use LZ compression on the disk file.

`HEXUPSHIFT` **BOOL (io) r ***

If TRUE, all hex output is displayed in uppercase; otherwise it is displayed in lowercase. The default is FALSE, lowercase.

`ICSNEST`	**U16 (misc) r**
	The current ICS nest count as found in the base of the ICS. This value is always 0 for Debug.
`ICSVA`	**LPTR (misc) r**
	The virtual address for the base of the ICS.
`IIR`	**U32 (nmreg) r dM**
	NM interrupt instruction register. (Alias for CR19)

`INBASE` **STR (io) rw ***

The current input conversion radix, which is based on the current mode. Values entered are assumed to be in this radix unless otherwise specified. This variable tracks `NM_INBASE` and `CM_INBASE` dependent upon the `MODE` variable. The following values are allowed:

`%` or `OCTAL`
`#` or `DECIMAL`
`$` or `HEXADECIMAL`

The names may be abbreviated to 1 character.

The default is based on the current mode (NM or CM). Refer to the `SET` command for an alternate method of setting this variable.

`IOR` **U32 (nmreg) r dM**

NM interrupt offset register. (Alias for CR21)

`IPSW` **U32 (nmreg) r dM**

NM interrupt processor status word (alias for CR22 and PSW). Debug may set or alter the "R" bit while single stepping, as well as the "T" bit if the `TRAP BRANCH ARM` command has been issued.

This register has the following format:

```
                1 1 1 1 1 1 1              2     2 2 2 3 3
 0            7 8 9 0 1 2 3 4 5 6          4     7 8 9 0 1
-----------------------------------------------------------------
|J|          |T|H|L|N|X|B|C|V|M|     C/B       |     |R|Q|P|D|I|
-----------------------------------------------------------------
```

`J`	Joint instruction and data TLB misses/page faults pending
`T`	Taken branch trap enabled
`H`	Higher-privilege transfer trap enable
`L`	Lower-privilege transfer trap enable
`N`	Instruction whose address is at front of PC queue is nullified
`X`	Data memory break disable
`B`	Taken branch in previous cycle
`C`	Code address translation enable
`V`	Divide step correction
`M`	High-priority machine check disable
`C/B`	Carry/borrow bits
`R`	Recovery counter enable
`Q`	Interruption state collection enable
`P`	Protection ID validation enable
`D`	Data address translation enable
`I`	External, power failure, & low-priority machine check interruption enable

System Debug displays this register in two formats:

```
IPSW=$6ff0b=jthlnxbCVmrQpDI
```

The first value is a full 32-bit integer representation of the register. The second format shows the value of the special named bits. An uppercase letter means the bit is ON while a lowercase letter indicates the bit is OFF.

ISR **U32 (nmreg) r dM**

NM interrupt space register. (Alias for CR20)

ITMR **U32 (nmreg) r dM**

NM interval timer register. (Alias for CR16)

IVA **U32 (nmreg) r dM**

NM interrupt vector address. (Alias for CR14)

JOB_DEBUG **BOOL (system) rW**

A system wide flag that enables the debugging of jobs. The default value is FALSE; any process attempting to access Debug in a job has that request ignored (with the exception of the `HPDEBUG` intrinsic, which will execute a command string but not stop in Debug). If this variable is set, and a job does call Debug, upon entry the `CONSOLE_IO` variable is set to TRUE and the `TERM_LDEV` variable is set to the console port (LDEV 20).

This variable is available only in Debug.

JUSTIFY **STR (io) rw ***

This variable controls the form justification used when numeric values are displayed in the windows or from the Display commands. This variable may take on one of two quoted literal values: "LEFT" or "RIGHT". When right-justified, values can be blank or zero filled (refer to the `FILL` variable). Decimal values are always left-justified in windows, despite this setting. The default value is "RIGHT".

LAST_PIN **U16 (misc) r**

For DAT, this is the last PIN that was running at dump time (as found in SYSGLOB). For Debug, this variable is the PIN on whose stack the debugger is running.

LIST_INPUT **U16 (io) rw**

When `LIST_INPUT` is set, all user input lines are written into any currently opened list file (refer to the `LIST` command). When `ECHO_USE` is set, those lines that are input from the USE file are always displayed to the list file, even if `LIST_INPUT` is disabled. The default value is TRUE.

LIST_PAGELEN **U16 (io) rw ***

The page length (in lines) of the list file (refer to the LIST command). The default page length is #60. If the LIST_PAGING environment variable is set, a page eject is placed in the list after every LIST_PAGELEN lines.

LIST_PAGENUM **U16 (io) r**

The current page number of the list file (refer to the LIST command). When a list file is opened, this variable is reset to 1. The default LIST_TITLE uses this value as part of the page title written to each page.

LIST_PAGING **BOOL (io) r ***

When LIST_PAGING is set, output to the list file (refer to the LIST command) is paged (based on LIST_PAGELEN). In addition, the LIST_TITLE is written at the top of each new page. The default value for this variable is TRUE.

LIST_TITLE **STR (io) rw ***

When the LIST_PAGING variable is enabled, this LIST_TITLE is written to the top of each new page in the list file (refer to the LIST command). The default LIST_TITLE is displayed below, followed by the output it produces:

```
'"Page: " list_pagenum:"d" "   " version "   " date "   " time'
```

```
Page: 1    DAT-XL 9.00.00    FRI, FEB 13, 1987  2:22 PM
```

The variables in the title are evaluated each time the title is written to the list file.

LIST_WIDTH **U16 (io) rw ***

The width (in number of characters) to be used for the list file (refer to the LIST command). This number must be in the range 1-132, and is 80 characters by default. Lines written to the list file that are longer than the LIST_WIDTH length are not truncated; instead they are split, with the extra data placed on the following line.

LOOKUP_ID **STR (misc) rw ***

This variable is used by the expression evaluator in determining where to look up NM procedure names. Refer to the "Procedure Name Symbols" section in chapter 2 "User Interfaces" for additional details. It may take on any of the following values:

UNIVERSAL	Search exported procedures in the System Object Module symbols.
LOCAL	Search non-exported procedures in the System Object Module symbols.
NESTED	Search nested procedures in the System Object Module symbols.
PROCEDURES	Search local or exported procedures in the System Object Module symbols.
ALLPROC	Search local/exported/nested procedures in the System Object Module symbols.
EXPORTSTUB	Search export stubs in the System Object Module symbols.
DATAANY	Search exported or local data System Object Module symbols.
DATAUNIV	Search exported data System Object Module symbols.
DATALOCAL	Search local data System Object Module symbols.
LSTPROC	Search exported level 1 procedures in the LST.
LSTEXPORTSTUB	Search export stubs in the LST.
ANY	Search for any type of symbol in the System Object Module symbols.

The default is LSTPROC. Note that it is noticeably slower to look up symbols from the System Object Module symbol table. For additional information, see the section "Procedure Names" in chapter 2, the PROCLIST command, and the NMADDR function.

`LW` **SADDR (win) r**

The secondary address where the LDEV window is aimed. The value returned is interpreted as *ldev.offset*.

`MACROS` **U16 (limits) rw**

The `MACROS` variable controls the size of the macro table, and must be changed (from the default size) before any macros are created. The `MACROS` limit is automatically increased to the nearest prime number, which must be less than or equal to `MACROS_LIMIT`.

MACROS_LIMIT **U16 (limits) r**

MACROS_LIMIT is a compile time constant that defines the absolute maximum size of the macro table. The product must be recompiled and redistributed to increase this absolute capacity.

MACRO_DEPTH **U16 (cmd) r**

MACRO_DEPTH tracks the current nested call level for macros. A depth of 1 implies the macro was invoked from the user interface. A depth of 2 implies that the current macro was called by another macro, and so on.

MAPDST **U16 (cmreg) r**

This variable contains the mapping DST number for CM CST expansion.

MAPFLAG **U16 (cmreg) r**

MAPFLAG indicates the mapping of the current CM segment, running under CST expansion. If MAPFLAG = 0, the current CM segment is logically mapped. If MAPFLAG = 1, the current CM segment is physically mapped.

MARKERS **STR (win) rw ***

The MARKERS variable selects the type of video enhancement which is used to flag stack markers in the CM Q (frame) and S (stack) windows. The following string literals are valid: "INVERSE", "HALFINV", "BLINK", "ULINE", and "FEABLE". The default is "ULINE".

MODE **STR (misc) r** This variable contains either "NM" if you are in NMDebug, or "CM" if in cmdebug.

MONARCHCPU **U16 (misc) r d**

This variable contains the number of the Monarch processor.

MPEXL_TABLE_VA **U16 (misc) rw**

ENV

This variable contains the address of the table used by the `MPEXL` command. Initially the address is set to NIL (0.0). The first invocation of the MPEXL command will correctly replace the NIL value with the actual table address. If any (non-NIL) virtual address is written into this variable, then the MPEXL comand will honor this address and use it to attempt access to the MPEXL table.

`MULTI_LINE_ERRS` **U16 (cmd) rw ***

When a user's multiple line input contains an error, it is sometimes desirable to limit the quantity of error output generated. In particular this variable controls how much of the user's original input line is displayed in the error message:

1	Display the single input line that contains the error.
2	Display all lines up to and including the line with the error.
3	Display all input lines (up to, including and after) the error.

The default value is 2. Any value larger than 3 is interpreted as a 3.

`NMPW` **LCPTR (win) r**

The logical code address where the NM program window is aimed.

NM_INBASE **STR (io) rw ***

The current NM input conversion base. When in NMDebug, all values entered are assumed to be in this base unless otherwise specified. The following values are allowed:

% or OCTAL
# or DECIMAL
$ or HEXADECIMAL

The names may be abbreviated to as little as a single character.

The default value is $ (hex). Refer to the SET command for an alternate method of setting this variable.

NM_OUTBASE **STR (io) rw ***

When in NM (nmdat or nmdebug), all numbers printed will be this base, unless otherwise indicated (refer to the SET command). The following values are allowed:

% or OCTAL
# or DECIMAL
$ or HEXADECIMAL

The names may be abbreviated to as little as a single character.

The default value is $ (hex). Refer to the SET command for an alternate method of setting this variable.

NONLOCALVARS **BOOL (cmd) rw**

When NONLOCALVARS is FALSE (default), macro bodies can only reference local variables that are declared locally within the current macro. When NONLOCALVARS is TRUE, a macro body can reference a local variable within another macro that called it. Setting this variable is useful when a macro is too large for the current macro size restrictions and must be broken into several pieces. The first piece can call the subsequent pieces without passing all of the local variables as parameters.

OUTBASE **STR (io) rw ***

This variable tracks NM_OUTBASE and CM_OUTBASE dependent upon the MODE variable. The following values are allowed:

% or OCTAL
# or DECIMAL
$ or HEXADECIMAL

The names may be abbreviated to as little as 1 character.

The default is based on the current mode (NM or CM). Refer to the SET command for an alternate method of setting this variable.

`PC` **LPTR (nmreg) r dm**

NM program counter register as a logical code address. This value is composed of data taken from CR17 (PCSF) and CR18 (PCOF). The privileged bits from CR18 (bits 30, 31) are masked out (that is, they are set to zero).

`PCOB` **U32 (nmreg) r dm**

NM program counter *offset* (next in pipeline queue).

`PCOF` **U32 (nmreg) r dm**

NM program counter *offset* (first in pipeline queue).

`PCQB` **LPTR (nmreg) r dm**

NM program counter *sid.offset* (next in pipeline queue). (Alias for CR18)

`PCQF` **LPTR (nmreg) r dm**

NM program counter *sid.offset* (first in pipeline queue). (Alias for CR17)

`PCSB` **U32 (nmreg) r dm**

NM program counter *sid* (next in pipeline queue).

`PCSF` **U32 (nmreg) r dm**

NM program counter *sid* (first in pipeline queue).

`PID1 - PID4` **U16 (nmreg) r dM**

NM protection ID registers. (Alias for CR8, CR9, CR12, CR13.) The format of the PID registers is as follows:

```
                              1 1                             3
0                             5 6                             1
---------------------------------------------------------------
|         <reserved>          |     Protection ID          |WD|
---------------------------------------------------------------
```

`<reserved>` The top 16 bits are undefined for this register.

`Protection ID`	The protection ID number.
`WD`	Write disable bit (1 = read only, 0 = write enabled)

System Debug displays these registers in two formats:

```
PID1=030e=0187(W)
```

The first value is the register as a 16-bit value. The second form is the original 16-bit register shifted right by 1 bit followed by the value of the write disable bit. The `(W)` indicates the WD bit is off. That is, write capability is enabled. When the WD bit is on, an `(R)` is displayed indicating Read access.

PIN **U16 (misc) r**

The current process identification number (PIN). Note that this variable changes when one uses the PIN command. PIN 0 (zero) indicates that the dispatcher is running. (Refer to the variable LAST_PIN.)

PRIV **U16 (nmreg) r dM**

Current privilege level (low two bits of PCOF).

PRIV_USER **BOOL (nmreg) r rW**

This variable is TRUE if the user running Debug has privileged mode (PM) capabilities. If set, the user has access to all privileged commands within Debug. Privileged users may alter the value of this variable if desired to supply a "safe" environment.

In DAT, this variable is always TRUE.

PROGNAME **STR (misc) r**

This variable contains the name of the tool that is being run. It is either 'dat' or 'debug'.

PROMPT **STR (io) rw**

Current user prompt. It is defined as a quoted string with the same syntax and options as the WL command. The default prompt is:

```
'cmdnum " (" pin ")   " mode progname " " filter "> "'
```

The variables in the prompt are evaluated each time the prompt is displayed.

PSEUDOVIRTREAD **BOOL (misc) r d**

This variable is TRUE if the last virtual access came from a pseudomapped file. Otherwise, the access came from virtual memory.

PSP **U32 (nmreg) r d**

	Previous SP. This is not really a register; it is computed based on the current SP and size of the current frame.
`PSTMT`	**BOOL (misc) rw ***
	When `PSTMT` is set, the NM disassembler interprets certain LDIL instructions as statement numbers, as generated by some of the language compilers. The default value is TRUE.
`PSW`	**U32 (nmreg) r dM**
	Processor status register (alias for IPSW and CR22). Refer to the `IPSW` environment variable for a complete description of this variable.
`PW`	**LCPTR (win) r**
	The address (as a logical code address) where the (current) program window is aimed.
`PWO`	**SPTR (win) r**
	The offset where the (current) program window is aimed.

`PWS`	**U32 (win) r**
	The SID (NM) or SEG (CM) where the (current) program window is aimed.
`Q`	**U16 (cmreg) r dm**
	This is the CM Q register. The value in this register is relative to the CM DB register.
`QUIET_MODIFY`	**U16 (io) rw ***
	When this variable is FALSE (the default value), all modifications to registers and memory cause the current value of the item to be displayed. If the variable is set to TRUE, all modifications are performed quietly. Quiet modifications are useful in macros and breakpoint command lists.
`R0`	**U32 (nmreg) r d**
	NM register 0; the constant 0 (zero).
`R1 - R31`	**U32 (nmreg) rwdm**
	NM general registers. Many of these registers have aliases. Refer to the `DR` command for a complete list.
`RCTR`	**U32 (nmreg) r dM**
	NM recovery counter register. (Alias for CR0)
`RET0`	**U32 (nmreg) r dm**
	NM return register 0 (alias for R28). This register is used by the language compilers to return function results.
`RET1`	**U32 (nmreg) r dm**
	NM return register 1 (alias for R29). This register is used by the language compilers to return function results.
`RP`	**U32 (nmreg) r d**
	NM return pointer. This value is determined based on stack unwind information. It may be the contents of R2 or

	it may be the return address stored somewhere in the NM stack. Note that `RP` is not an alias for R2.
`S`	**U16 (cmreg) r dm**
	CM S (stack) register. The value in this register is relative to the CM DB register.
`SAR`	**U16 (nmreg) r dm**
	NM shift amount register. (Alias for SR11)
`SDST`	**U16 (cmreg) r dm**
	DST number of the CM stack.

SL **U32 (nmreg) r dm**

NM static link register. (Alias for R29)

SP **U32 (nmreg) r dm**

NM stack pointer register. (Alias for R30)

SR0 - SR7 **U32 (nmreg) r dM**

NM space registers 0 - 7.

STATUS **U16 (cmreg) r dm**

CM status register. This register has the following format:

```
                     1 1 1 1 1 1
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
---------------------------------
|M|I|T|R|O|C|CC |  Segment #    |
---------------------------------
```

M bit	1 if program is privileged 0 if program is in user mode
I bit	1 if external Interrupts are enabled 0 if not
T bit	1 if user Traps are enabled 0 if not
R bit	1 if right stack operation pending 0 if left stack operation pending
O bit	1 if Overflow bit set (not set if user traps enabled) 0 if not
C bit	1 if Carry bit set 0 if not
CC bits	01 if CCL (This is the condition code value) 10 if CCE 00 if CCG

System Debug display this register with two formats:

```
STATUS=%100030=(Mitroc CCG 030)
```

The first value is the full 16-bit integer representation of the register. The second format shows the value of the special named bits. An uppercase letter means the bit is on while a lowercase letters indicates the bit is off.

The segment number has various interpretations. For non-CST expansion systems, this is an absolute segment number. For CST expansion systems, refer to the *MPE V/E Tables Manual* for details on its interpretation.

`SYMPATH_UPSHIFT` **BOOL (misc) rw**

TRUE if path specifications used by symbolic formatting should be upshifted. This should be FALSE if a symbol file originated with a case-sensitive language, such as C. Note that this variable affects only those symbols entered in System Debug commands and functions, *not* those in symbol files.

`SYSVERSION` **STR (nmreg) r**

The version of the operating system (as found in SYSGLOB).

This variable is currently a null string in DAT.

`TERM_KEEPLOCK` **BOOL (io) rw**

If this variable is set, the terminal semaphore is not released when the process is resumed by Debug. The default for this variable is FALSE. If the process dies, the terminal semaphore is automatically released. If the `TERM NEXT` command is issued or the value of `TERM_LOCKING` is changed, this variable is reset to FALSE.

This variable is available only in Debug.

`TERM_LDEV` **U16 (io) rW**

This variable contains the logical device number (LDEV) to use for I/O. Debug determines this value by looking up the LDEV for the session.

If the `ENV` command is used to alter this value, Debug attempts to allocate the indicated LDEV. If the LDEV is already allocated (that is, in use by another session), an error status is returned. If the user has privileged mode (PM) capabilities, the allocation check may be bypassed by specifying a negative LDEV. In this case, all security and validity checking is bypassed. Non-Preemptive send_io calls are done to the specified LDEV without question.

When Debug is entered from a job (this is possible when the `HPDEBUG` intrinsic is used), this variable is not used.

Rather, Debug performs I/O to the job's standard list file ($STDLIST).

If the JOB_DEBUG system wide variable is set, when a process being run in a job enters Debug, this variable is set to the console port (LDEV 20) and the CONSOLE_IO variable is set to TRUE.

Note that the CONSOLE_IO environment variable has precedence over TERM_LDEV.

Note A privileged procedure exists that allows the user to enter Debug and specify the initial value of this variable. The name of the routine is debug_at_ldev. It takes one parameter, the LDEV.

This variable is not available in DAT.

TERM_LOCKING **BOOL (io) rw**

If this variable is set (the default value), the debugger will perform "terminal locking" (with a semaphore) to ensure that only one debug process can use a terminal at any given time. This prevents multiple prompts from appearing on the screen when debugging multiple processes at the same terminal. The `TERM` command may then be used to control which process owns the semaphore. If this variable is not set, no terminal locking is performed.

The `TERM_LDEV` variable is not used to determine which semaphore to attempt to lock; rather, the session number is used for this purpose. There is one semaphore per session. If a process enters Debug with its I/O from the system console (that is, the `CONSOLE_IO` variable was set to TRUE at entry), a single console semaphore is used.

Altering the value of the `CONSOLE_IO` variable or the `TERM_LDEV` variable does _not affect which semaphore is used for terminal locking.

This variable is not available in DAT.

TERM_LOUD **BOOL (io) rw ***

If this variable is clear, all output to the terminal is suppressed with the exception of prompts and error messages. This is useful when listing large amounts of data to a list file so that you do not see it on your screen. The default for this variable is TRUE.

TERM_PAGING **BOOL (io) rw ***

If this variable is set, all output is paged. That is, after each full screen of output, System Debug pauses. At that point the user is prompted with the question `"MORE?"`. Any response that does not begin with the letter "Y" or "y" will cause the user to be returned to the System Debug prompt (any pending output is flushed). This variable may also be set with the `SET MOREON`/`SET MOREOFF` commands. The default value is FALSE.

ENV

TERM_WIDTH	**U16 (io) rw ***
	This is the number of characters to print per line. The default is set at 79. Any output line longer than this value is split with the remainder placed on the next line.
TIME	**STR (misc) r**
	The current time of day in the format: "5:25 PM".
TR0 - TR7	**U32 (n*eg) r dM**
	NM "temp" registers (alias for CR24..CR31).

`TRACE_FUNC` **U16 (cmd) rw**

Setting this variable allows you to observe function calls and their parameters. The current values and meanings are:

0	Trace is off.
1	Trace EXIT from functions.
2	Trace ENTRY and EXIT from functions.
3	Trace function PARAMETERS as well as ENTRY and EXIT.

`TRUE` **BOOL (const) r**

The constant "TRUE".

`VARS` **U16 (limits) rw**

The VARS limit determines the maximum number of variables that can be defined by the VAR command. The VARS limit must be set (changed from the default) before the first variable is defined. The VARS limit is automatically increased to the nearest prime number. The combined sum of the VARS and VARS_LOC limits must be less than or equal to the value of VARS_LIMIT.

`VARS_LIMIT` **U16 (limits) r**

VARS_LIMIT is the compile time constant that defines the absolute maximum size of the variable table. The product must be recompiled and redistributed to increase this absolute capacity. The combined sum of the VARS and VARS_LOC limits must be less than or equal to the value VARS_LIMIT.

`VARS_LOC` **U16 (limits) rw**

The VARS_LOC limit determines the maximum number of local variables that can be defined. Local variables are explicitly defined by the LOC command, and are implicitly defined for macro parameters. The VARS_LOC limit must be set before any local variable is defined. The combined sum of the VARS and VARS_LOC limits must be less than the value VARS_LIMIT.

VARS_TABLE **U16 (limits) rw**

VARS_TABLE tracks the total number of entries in the variable table, which is defined to be the sum of variables VARS plus VARS_LOC. The VARS_TABLE size must always be less than or equal to VARS_LIMIT.

VERSION **STR (misc) r**

The version ID of the program, for example, "DAT XL A.00.00".

VW	**LPTR (win) r**
	The virtual address where the current virtual window is aimed.
VWO	**SPTR (win) r**
	The *offset* portion for the virtual address where the current virtual window is aimed.
VWS	**U32 (win) r**
	The *sid* portion for the virtual address where the current virtual window is aimed.
WIN_LENGTH	**U32 (io) rw ***
	Specifies the number of lines available on the display terminal. The default value is #24. Values grater than or less than the actual number of terminal lines may cause unpredictable screen output.
WIN_WIDTH	**U32 (io) rw ***
	Specifies the number of columns available on the display terminal. The default value is #80. Modification of this value is permitted, but the value is ignored.
X	**U16 (c*eg) r dm**
	The CM X (index) register.
ZW	**U32 (win) r**
	The real address where the Z window is aimed.

Examples

```
%cmdebug > env autoignore true
```

Set the environment variable `AUTOIGNORE` to TRUE.

```
$nmdebug > env cmdlinesubs true
```

ENV

Set the variable `CMDLINESUBS` to TRUE. This enables command line substitutions, that may have been disabled while macros were being read in from a file.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

ENVL[IST]

Displays the current values for environment variables.

Syntax

```
ENVL[IST] [pattern] [group] [options]
```

Parameters

pattern The name of the environment variable(s) to be listed.

This parameter can be specified with wildcards or with a full regular expression. Refer to Appendix A for additional information about pattern matching and regular expressions.

The following wildcards are supported:

`@`	Matches any character(s).
`?`	Matches any alphabetic character.
`#`	Matches any numeric character.

The following are valid name pattern specifications:

`@`	Matches everything; all names.
`pib@`	Matches all names that start with "pib".
`log2##4`	Matches "log2004", "log2754", and so on.

The following regular expressions are equivalent to the patterns with wildcards that are listed above:

```
`.*`
`pib.*`
`log2[0-9][0-9]4`
```

By default, all variables are listed.

ENVL[IST]

group The environment variables are logically organized in groups. When listed, the variables can be filtered by group; that is, only those variables in the specified group is displayed.

CONST	Predefined constants
CMD	Command-related
IO	Input/output-related
MISC	Miscellaneous
WIN	Window
SYSTEM	System-wide Debug registers
C*EG	Compatibility mode registers
N*EG	Native mode registers
FPREG	Native mode floating-point registers
STATE	Same as C*EG N*EG FPREG
NOSTATE	Same as CONST CMD IO MISC WIN SYSTEM (default)
ALL | @	All groups

If the group name is omitted, NOSTATE is used by default.

options Any number of the following options can be specified in any order, separated by blanks:

NAME	Display variable name only
USE	Display a one-line summary
NOUSE	Skip the summary
DESC	Display a general description
NODESC	Skip the description
EXAMPLE	Display an example
NOEXAMPLE	Skip the example
ALL | @	Display everything, Same as:

NAME USE DESC EXAMPLE

If none of the options above are specified, NAME is displayed by default. If any options are specified, they are accumulated to describe which fields are printed.

Examples

```
$nmdat > envl, win
win     rw    CHANGES           : STR   = 'HALFINV'
win     r     CMPW              : LCPTR = SYS $15.0
win     r     LW                : SADDR = SADDR $1.0
win     rw    MARKERS           : STR   = 'ULINE'
win     r     NMPW              : LCPTR = SYS $a.702d6c
win     r     PW                : LCPTR = SYS $a.702d6c
win     r     PWO               : SPTR  = $00702d6c
win     r     PWS               : U32   = $a
win     rw    SHOW_CCTL         : BOOL  = FALSE
win     r     VW                : LPTR  = $0.0
win     r     VWO               : SPTR  = $00000000
win     r     VWS               : U32   = $0
win     r     ZW                : U32   = $0
```

Display all window-related environment variables.

ENVL[IST]

```
$nmdat > envl m@
cmd      r     MACRO_DEPTH        : U16   = $0
win      rw    MARKERS            : STR   = 'ULINE'
misc     r     MODE               : STR   = 'nm'
cmd      rw    MULTI_LINE_ERRS    : U16   = $2
```

Display all environment variables that begin with the letter "m".

```
$nmdat > envl vw,,all
win      r     VW                 : LPTR  = $0.0

DESC:

The virtual address where the current virtual window is aimed.
```

Display the environment variable `VW` and all related information associated with that variable.

```
$nmdat > env term_loud 0
$nmdat > list envinfo
$nmdat > envl @,,all page
$nmdat > list close
$nmdat > env term_loud 1
```

Create a list file with complete information on all of the environment variables. The list file is paged with one environment variable description per page.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

ERR

Pushes a user error message onto the error command stack.

Syntax

```
ERR errmsg
```

The ERR command is typically used within user defined macros.

Parameters

errmsg The error message that is to be pushed onto the error stack. This message must be entered as a string expression (that is, a quoted string literal, a string function or macro result).

Examples

```
$nmdat > err "Illegal negative parameter value"
```

Push a custom user error message onto the error stack.

Limitations, Restrictions

The error stack is implemented as a ring, with a total of 10 elements.

Note that the ERROR environment variable is not set by this command.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

ERRD[EL]

Deletes all errors on the error stack (reset the stack).

Syntax

```
ERRD[EL]
```

Parameters

none

Examples

```
$nmdat > errd
```

Reset the error stack.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

ERRL[IST]

Error list. Lists the most recent error(s) on the error stack.

Syntax

```
ERRL[IST] [ALL]
```

Parameters

ALL — By default, only the most recent (set) of errors are displayed. If the special option ALL is specified, all sets of errors are displayed.

Examples

```
$nmdat > dv a.234e0
Display error.  Check ERRLIST for details.  (error #3800)

$nmdat > errl
$47:  Display error.  Check ERRLIST for details.  (error #3800)
$47:  data read access error  (error #805)
$47:    READ_CMWORD   bad address: $ VIRT a.234e0
$47:  No dump file set is opened  (error #5083)
```

Display error information from the error stack about the last error. Useful additional error information is often available in the error stack. In this example, we see that several error lines were stacked for command number $47. The display command failed because no dump has been opened.

ERRL[IST]

```
$nmdat > errl all

$47:  Display error.  Check ERRLIST for details.  (error #3800)
$47:  data read access error  (error #805)
$47:    READ_CMWORD   bad address: $ VIRT a.234e0
$47:  No dump file set is opened  (error #5083)

$22:  Error evaluating a predefined function.  (error #4240)
$22:    function is"vtor"
$22:    wl vtor(pc)
                   ^
$22:  Virtual-to-real translation failed.  (error #6013)

$1f:  Unknown topic for HELP.  (error #1488)

$1c:  This command is invalid for this program.  (error #6115)
$1c:       Program: DAT
$1c:    mv a.c00012c4
          ^

$17:  File system error opening an old file.  (error #1302)
$17:    NONEXISTENT PERMANENT FILE  (FSERR 52) [LOADMACS]
```

Display all entries in the error stack. Multiple stacked errors are displayed, along with the command numbers that caused the errors. Errors are recorded for commands $47, $22, $1f, $1c, and $17.

Limitations, Restrictions

The error stack is implemented as a ring, with a total of 10 elements.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

E[XIT]

Exits/resumes execution of user program.

Syntax

```
E[XIT]      Same as CONTINUE (in Debug)
E[XIT]      Exit program       (in DAT)
```

Same as the C[ONTINUE] command in Debug. For DAT, this command exits the DAT program.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

Fx (format)

Formats a specified data structure.

Syntax

```
FT          path ft_options

FV virtaddr path fv_options
```

`FT` = format data structure with type information.

`FV` = format data structure with data starting at *sid.off*.

Parameters

virtaddr	`FV` only. The virtual address of the data to be formatted. *Virtaddr* can be a short pointer, a long pointer, or a full logical code pointer.
path	A path specification, as described in chapter 5, "Symbolic Formatting/Symbolic Access".
ft_options	These options are for the FT command only. The `MAP` option causes a location map to be printed for components of complex structures such as records or arrays. `MAP` Include a location map. `NOMAP` Do not include a location map (default).
fv_options	These options are for the `FV` command only. `PAC` Print packed array of chars as a string of characters. `NOPAC` Print packed array of chars as an array index followed by the element value. `PAB` Print packed array of boolean as a bit string.

`NOPAB`	Print packed array of boolean as an array index followed by the element value.
`ARCH`	For selected MPE/XL architect types, print the data in the "expected" fashion.
`NOARCH`	Do no special formatting for MPE/XL architected types.

If no options are given, the default set is:

```
PAC  PAB  ARCH
```

The known types given special treatment with the `ARCH` option are:

```
VA_TYPE
SHORT_VA_TYPE
CONVERT_PTR_TYPE
```

Examples

```
$nmdebug > symopen gradtyp.demo
```

Opens the symbolic data type file `gradtyp.demo`. It is assumed that the Debug variable *addr* contains the address of a `StudentRecord` data structure in virtual memory. The following code fragment is from this file:

```
CONST         MINGRADES    = 1;          MAXGRADES    = 10;
              MINSTUDENTS  = 1;          MAXSTUDENTS  = 5;

TYPE
   GradeRange     = MINGRADES . . MAXGRADES;
   GradesArray    = ARRAY [ GradeRange  ] OF integer;
   Class          = ( SENIOR,  JUNIOR,  SOPHOMORE,  FRESHMAN );
   NameStr        = string[8];
   StudentRecord  = RECORD
                       Name         : NameStr;
                       Id           : integer;
                       Year         : Class;
                       NumGrades    : GradeRange;
                       Grades       : GradesArray;
                    END;
```

FT (Format Type) Examples

```
$nmdebug > FT "StudentRecord"

RECORD
   NAME      : NAMESTR ;
   ID        : INTEGER ;
   YEAR      : CLASS ;
   NUMGRADES: GRADERANGE ;
   GRADES    : GRADESARRAY ;
END
```

Display the structure of `StudentRecord`.

```
$nmdebug > FT "StudentRecord" MAP
```

```
RECORD
   NAME     : NAMESTR ;   ( 0.0 @ 10.0 )
   ID       : INTEGER ;   ( 10.0 @ 4.0 )
   YEAR     : CLASS ;   ( 14.0 @ 1.0 )
   NUMGRADES: GRADERANGE ;   ( 15.0 @ 1.0 )
   GRADES   : GRADESARRAY ;   ( 18.0 @ 28.0 )
END ;
RECORD Size: 40 bytes
```

Display the structure of `StudentRecord` and print a component map.

F*x* (format)

```
$nmdebug > FT "StudentRecord.grades"
ARRAY [ GRADERANGE  ] OF INTEGER

$nmdebug > FT "graderange"
1 .. 10

$nmdebug > FT "maxgrades"
INTEGER
```

Display various types. Notice that structure name is not limited to a simple type or constant name; rather, it may consist of any composite structure name.

FV (Format Virtual) Examples

The following examples assume that debug variable `data` contains the virtual address of a data structure corresponding to the type `StudentArray`.

Before looking at `FV` examples, let's take a look at the data for student number 1 the "old fashioned way" (with the `DV` command):

```
$nmdebug > dv data,10
$ VIRT 7b8.40200010 $ 00000004 42696c6c 00000000 00000000
$ VIRT 7b8.40200020 $ 00000001 00040000 0000002d 00000041
$ VIRT 7b8.40200030 $ 0000004e 00000042 00000000 00000000
$ VIRT 7b8.40200040 $ 00000000 00000000 00000000 00000000

$nmdebug > dv data,6,a
$ VIRT 7b8.40200010 A .... Bill .... .... .... ....
```

This is what the first few words of the `StudentArray` data looks like in virtual memory.

```
$nmdebug > fv data "StudentRecord"

RECORD
   NAME      : 'Bill'
   ID        : 1
   YEAR      : SENIOR
   NUMGRADES : 4
   GRADES    :
      [ 1 ]: 2d
      [ 2 ]: 41
      [ 3 ]: 4e
      [ 4 ]: 42
      [ 5 ]: 0
      [ 6 ]: 0
      [ 7 ]: 0
      [ 8 ]: 0
      [ 9 ]: 0
      [ a ]: 0
END
```

This is what the first element of the `StudentArray` data looks like when formatted as if it were a `StudentRecord`.

```
$nmdebug > fv data "StudentRecord.Name"

'Bill'

$nmdebug > fv data "StudentRecord.Year"

SENIOR

$nmdebug > fv data "StudentRecord.Grades[3]"

4e
```

MPE XL Operating System Examples

We can also look at individual items of a data structure as the above examples depict.

```
$nmdebug > symopen symos.pub.sys
$nmdebug > fv pib(pin) "pib_type.cm_global"
c79c0000
```

Open the operating system symbolic file. Format the data in the `cm_global` field of the PIB for the current PIN. It is a short pointer.

```
 $nmdebug > fv pib(pin) "pib_type.cm_global^"
PACKED RECORD
   CM_DP0          : 0
   CM_DP_SCRATCH   : c0105d40
   CM_INFO         :
         CM_INFO_INT : c
   CM_CTRL         :
         CM_CTRL_INT : 0
   CM_STACK_DST    : ac
   CM_DB_DST       : ac
   CM_DB_3K_OFFSET : 200
   CM_DB_SID       : 7d4
   CM_DB_OFFSET    : 400110b0
   CM_DL           : CONVERT_PTR_TYPE( 7d4.40011000  )
   CM_S            : CONVERT_PTR_TYPE( 7d4.400110be  )
   CM_Z            : CONVERT_PTR_TYPE( 7d4.40015ed0  )
   CM_STACK_BASE   : CONVERT_PTR_TYPE( 7d4.40010cb0  )
   CM_STACK_LIMIT  : CONVERT_PTR_TYPE( 7d4.40020fff  )
   CM_CST          : 80000700
   CM_CSTX         : 0
   CM_LSTT         : CONVERT_PTR_TYPE( 0.0  )
   CM_NRPGMSEGS    : 0
   CM_DST          : 81400000
   CM_BANK0        : 80000000
   CM_BANK0_SIZE   : 10000
   CM_DEBUG        : 0
   CM_MCODE_ADR    : 484228
   CM_RESVD6       : 0
```

```
    CM_RESVD5        : 0
    CM_RESVD4        : 0
    CM_RESVD3        : 0
    CM_RESVD2        : 0
    CM_RESVD1        : 0
END
```

Format the data in the `cm_global` field of the PIB for the current PIN. That is, format what the pointer points to.

```
$nmdebug > fv pib(pin) "pib_type.cm_global^.cm_info"
CRUNCHED RECORD
      CM_INFO_INT : c
END
```

Format the data in the `cm_info` record of the `cm_global` record.

```
$nmdebug > ft "pib_type.cm_global^.cm_info"
CRUNCHED RECORD
 CASE BOOLEAN OF
      TRUE: ( CM_INFO_INT: SEM_LOCK_TYPE );
      FALSE: ( SPLITSTACK : BIT1 ;
              SINGLE_STEP: BIT1 ;
              CNTRL_Y    : BIT1 ;
              SCRATCH1   : BIT5 );
END
```

Format the type for the `acm_info` record contained in the `cm_global` record. We see that the record has an invariant case structure. By default, the formatter takes the first invariant structure found.

```
$nmdebug > fv pib(pin) "pib_type.cm_global^.cm_info,false"
CRUNCHED RECORD
      SPLITSTACK  : 0
      SINGLE_STEP : 0
      CNTRL_Y     : 0
      SCRATCH1    : c
END
```

Format the data for the `cm_info` record contained in the `cm_global` record. Note that we asked for a specific case invariant.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the

exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

F*mm* (freeze)

Debug only

Privileged Mode

Freezes a code segment, data segment, or virtual address (range) in memory.

Syntax

`FC`	*logaddr*	[*bytelength*]	`Program file`
`FCG`	*logaddr*	[*bytelength*]	`Group library`
`FCP`	*logaddr*	[*bytelength*]	`Account library`
`FCLG`	*logaddr*	[*bytelength*]	`Logon group library`
`FCLP`	*logaddr*	[*bytelength*]	`Logon account library`
`FCS`	*logaddr*	[*bytelength*]	`System library`
`FCU`	*fname logaddr*	[*bytelength*]	`User library`
`FCA`	*cmabsaddr*		`CM absolute CST`
`FCAX`	*cmabsaddr*		`CM absolute CST`
`FDA`	*dstoff*		`CM data segment`
`FVA`	*virtaddr*	[*bytelength*]	`Virtual address`

Parameters

logaddr A full logical code address (`LCPTR`) specifies three necessary items:

- the logical code file (`PROG`, `GRP`, `SYS`, and so on).
- NM: the virtual space ID number (SID).
  CM: the logical segment number.
- NM: the virtual byte offset within the space.
  CM: the word offset within the code segment.

Logical code addresses can be specified in various levels of detail:

- As a full logical code pointer (`LCPTR`):

`FC procname+20`	Procedure name lookups return `LCPTR`s.
`FC pw+4`	Predefined `ENV` variables of type `LCPTR`.
`FC SYS(2.200)`	Explicit coercion to a `LCPTR` type.

FINAL TRIM SIZE : 7.5 in x 9.0 in

- As a long pointer (`LPTR`):

  `FC 23.2644` *sid.offset* or *seg.offset*

  The logical file is determined based on the command suffix:

  `FC` implies `PROG`.
  `FCG` implies `GRP`.
  `FCS` implies `SYS`, and so on.

- As a short pointer (`SPTR`):

  FC 1024 *offset* only

  For NM, the short pointer offset is converted to a long pointer using the function `STOLOG`, which looks up the SID of the loaded logical file. This is different from the standard short to long pointer conversion, `STOL`, which is based on the current space registers (`SRs`).

  For CM, the current executing logical segment number and the current executing logical file are used to build a `LCPTR`.

The search path used for procedure name lookups is based on the command suffix letter:

`FC`	Full search path: NM: `PROG`, `GRP`, `PUB`, `USER`(s), `SYS`. CM: `PROG`, `GRP`, `PUB`, `LGRP`, `LPUB`, `SYS`.
`FCG`	Search `GRP`, the group library.
`FCP`	Search `PUB`, the account library.
`FCLG`	Search `LGRP`, the logon group library.
`FCLP`	Search `LPUB`, the logon account library.
`FCS`	Search `SYS`, the system library.
`FCU`	Search `USER`, the user library.

For a full description of logical code addresses, refer to the section "Logical Code Addresses" in chapter 2.

cmabsaddr A full CM absolute code address specifies three necessary items:

- Either the CST or the CSTX.
- The absolute code segment number.
- The CM word offset within the code segment.

Absolute code addresses can be specified in two ways:

- As a long pointer (`LPTR`):

`FCA 23.2644`	Implicit `CST` 23.2644
`FCAX 5.3204`	Implicit `CSTX` 5.3204

- As a full absolute code pointer (`ACPTR`):

`FCA  CST(2.200)`	Explicit `CST` coercion
`FCAX CSTX(2.200)`	Explicit `CSTX` coercion
`FCAX logtoabs(prog(1.20))`	Explicit absolute conversion

The search path used for procedure name lookups is based on the command suffix letter:

`FCA`	`GRP, PUB, LGRP, LPUB, SYS`
`FCAX`	`PROG`

fname The file name of the NM USER library. Since multiple NM libraries can be bound with the `XL=` option on a `:RUN` command,

```
:run nmprog; xl=lib1,lib2.testgrp,lib3
```

it is necessary to specify the desired NM user library. For example,

```
FCU lib1 204c
FCU lib2.testgrp test20+1c0
```

If the file name is not fully qualified, the following defaults are used:

Default account: the account of the program file.
Default group: the group of the program file.

dstoff
: A data segment address (specified as `DST.OFFSET`) of the data segment to be frozen in memory. The segment remains frozen until it is explicitly unfrozen (see `UDA` command).

virtaddr
: The starting virtual address of the page(s) that are to be frozen in memory. The pages remain frozen until they are explicitly unfrozen (see `UVA` command). *Virtaddr* can be a short pointer, a long pointer, or a full logical code pointer.

bytelength
: This parameter is valid only when in nmdebug. It indicates the desired number of bytes to be frozen. Based on the starting virtual address and the specified bytelength, the appropriate number of virtual pages are frozen. If omitted, the default is four bytes. The implementation of this command dictates that the smallest unit that is actually frozen is one page of virtual memory. That is, if you say 1 byte, the whole page on which that byte resides is made resident.

Examples

```
%cmdebug > fc cmpc
```

Freeze the current CM code segment, as indicated by the CM logical address CMPC.

```
%cmdebug > fcs sys(12.0)
```

Freeze CM logical code segment SYS 12.

```
$nmdebug > fva 22.104, #1000
```

Freeze 1000 bytes starting at virtual address 22.104.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

FINDPROC

Debug only

Dynamically loads a specified NM procedure from any NM library.

Syntax

```
FINDPROC procedurename library_file [ [NO]IGNORECASE]
```

This command dynamically loads a NM procedure from any NM library. The complete executable System Object Module containing the named procedure is loaded. This command is implemented by calling the `HPGETPROCPLABEL` intrinsic. (Refer to the *MPE XL Intrinsics Reference Manual* (32650-90028) for additional information.) If no error message is printed, the user can assume the command succeeded. The `LOADINFO` command may be used to verify that the library was loaded.

Parameters

procedurename	The name of the procedure to be loaded.
library_file	Any valid NM library file from which the procedure is to be loaded.
`IGNORECASE`	If `IGNORECASE` is specified, a case-insensitive search is performed for the program file. The default is `NOIGNORECASE`.

Examples

```
$nmdebug > findproc libsort testlib.test
$nmdebug >
```

Dynamically load the procedure libsort from the file `TESTLIB.TEST`

Limitations, Restrictions

This routine functions by calling the `FINDPROC` intrinsic. Refer to the *MPE XL Intrinsics Reference Manual* (32650-90028) for additional information.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

FOREACH

Each time a `FOREACH` command is executed, *name* is set to the next expression value in *value_list* prior to the execution of *cmdlist*. Execution ends when there are no more expression values in the *value_list*.

Syntax

```
FOREACH  name  value_list      command

FOREACH  name  value_list    { cmdlist }
```

Parameters

name — The name for the control variable that is set to the next expression value in *value_list*. A local variable is declared automatically, and it can be referenced with the *cmdlist*.

An optional type specification can be appended to the variable name, in order to restrict/convert the values in the list to a specific desired type:

```
foreach j:S16 '1 2 3+4 5' {wl j }
```

If the type specification is omitted, the type `ANY` is assumed.

value_list — This is a quoted string (or string variable) that contains a list of values (expressions). The *cmdlist* is evaluated once for every expression in the list. The list may contain string and or numeric expressions.

command
cmdlist — A single command (or command list) that is executed for each value in *value_list*.

Examples

```
%cmdebug > foreach j '1 2 3 "MOM" date 12.330' wl j
$1
$2
$3
MOM
WED. SEPT 3, 1986
$12.00000330
```

A local variable j is assigned each of the expression values in the value list string, and the specified command references the current value of j in order to write its value.

```
$nmdebug > foreach j '6  -2  "a" + "b"  3 +4' {wl j}
$4
"ab"
$7
```

This example shows that full expression values are evaluated within the value list.

```
$nmdebug > var nums '"1" "2" "3"'
$nmdebug > var lets '"A" "B" "C"'
$nmdebug > foreach l lets { foreach n nums {wl l n }}
A1
A2
A3
B1
B2
B3
C1
C2
C3
```

This is an example of nested FOREACH commands that use string variables for their value lists.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

FPMAP

Reinitializes CM FPMAP symbolic procedure name access.

Syntax

```
FPMAP
```

Initialization of CM FPMAP symbolic procedure names is automatic in System Debug.

The FPMAP command is typically used to "pick up" new libraries that have been dynamically loaded (through LOADPROC or SWITCH intrinsics) since the original program execution.

The FPMAP command inspects the CM program file and all currently loaded CM libraries in order to locate the necessary FPMAP records.

Examples

```
%cmdebug > fpmap
```

Re-initialize CM symbolic access for FPMAP records.

Limitations, Restrictions

The CM program file and libraries must have been prepared with the Segmenter's FPMAP option.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

FUNCL[IST]

Function list. Displays information about the predefined functions.

Syntax

```
FUNCL[IST] [pattern]  [group]  [options]
```

Parameters

pattern The name(s) of the function(s) to be displayed. This parameter can be specified with wildcards or with a full regular expression. Refer to appendix A for additional information about pattern matching and regular expressions.

The following wildcards are supported:

`@`	Matches any character(s).
`?`	Matches any alphabetic character.
`#`	Matches any numeric character.

The following are valid name pattern specifications:

`@`	Matches everything; all names.
`pib@`	Matches all names that start with "pib".
`log2##4`	Matches "log2004", "log2754", and so on.

The following regular expressions are equivalent to the patterns with wildcards that are listed above:

```
'.*'
'pib.*'
'log2[0-9][0-9]4'
```

By default, all functions are displayed.

group The functions are logically divided into groups, and they can be displayed, filtered by group name.

`COERCION`	Coercion functions.
`UTILITY`	General utility functions.
`ADDRESS`	Address manipulation functions.
`PROCESS`	Process data structure address functions.
`PROCEDURE`	Procedure name/length/entry/path functions.
`STRING`	String manipulation functions.
`SYMBOLIC`	Symbolic access functions.
`ALL | @`	Display all groups.

By default, all groups are displayed.

options Any number of the following options can be specified in any order, separated by blanks:

`NAME`	Display function name and result type.
`USE`	Display a short summary of use.
`NOUSE`	Skip the use summary.
`PARMS`	Display parameter names, types, default values.
`NOPARMS`	Skip parameter displays.
`DESC`	Display a general description.
`NODESC`	Skip the description.
`EXAMPLE`	Display the example.
`NOEXAMPLE`	Skip the example.
`ALL | @`	Display everything. Same as: `NAME USE PARMS DESC EXAMPLE`
`PAGE`	Page eject after each function definition. Useful for paged (listfile) output.
`NOPAGE`	No special page ejects.

If none of the options above are specified, the `NAME` is displayed by default. If any options are specified, they are accumulated to describe which fields are printed.

Examples

```
%cmdebug > funcl
```

List all functions.

```
%cmdebug > funcl @node
func CMNODE                   : LPTR    ADDRESS
func CMTONMNODE               : LPTR    ADDRESS
func NMNODE                   : LPTR    ADDRESS
func NMTOCMNODE               : LPTR    ADDRESS
```

List all functions (in all groups) that match the pattern "@node".

```
$nmdebug > funcl cm@ procedure
func CMADDR              : LCPTR     PROCEDURE
func CMBPADDR            : LCPTR     PROCEDURE Not in: dat sat
func CMBPINDEX           : U16       PROCEDURE Not in: dat sat
func CMBPINSTR           : U16       PROCEDURE Not in: dat sat
func CMENTRY             : LPTR      PROCEDURE
func CMPROC              : STR       PROCEDURE
func CMPRCLEN            : U16       PROCEDURE
func CMSEG               : STR       PROCEDURE
func CMSTART             : LCPTR     PROCEDURE
```

List all functions, in the group `PROCEDURE`, that start with "CM".

Note Some functions are not available in all programs. For example, the three breakpoint functions above, are flagged as NOT being available in DAT or SAT (since breakpoints are not supported in these programs).

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the

exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

GETDUMP

DAT only

Reads in a dump tape and creates a dump file.

Syntax

```
GETDUMP dumpfile [ ldevlist ]
GETDUMP dumpfile [ DIR ]
```

This command is used to restore the contents of a tape created by the DUMP utility onto disk. Once restored, the dump must be opened by the `OPENDUMP` command for access by the DAT program. A tape request for *dumptape* is generated; a message appears on the system console informing the operator of the request.

In order to conserve the disk space used to store a dump, DAT is capable of applying one of several data compression algorithms to reduce the required storage. Normally, DAT selects the algorithm which is known to produce the greatest compression, but other algorithms may be selected based on the setting of the environmental variable `GETDUMP_COMP_ALGO`. This variable may be set to a specific algorithm, or to the value `"TAPE"`. This special setting instructs DAT to use the same algorithm used by DUMP when the tape was produced. While this setting may not result in minimal disk space consumption, it will optimize `GETDUMP` performance, since the dump tape data will never have to be recompressed with a different algorithm.

Before data on a dump tape are copied to disk, DAT will preallocate a certain amount of disk space in order to avoid running out of this resource in the middle of a `GETDUMP`. The amount of space preallocated is controlled by the environmental variables `DUMPALLOC_RLE` and `DUMPALLOC_LZ`. One of these two variables will be used depending on the data compression algorithm applied to the dump disk file.

See the `ENV` command for further information about the environmental variables mentioned above.

Parameters

dumpfile	The name of the dump file to be created. Dump file names are limited to a maximum of five characters. All files related to the dump are given names composed of this name followed by a three-character mnemonic indicating the file contents.
ldevlist	A list of secondary-store LDEVs to be read from the dump. If no list is given, all LDEVs on the dump are read.
`DIR`	This option indicates that only the dump tape directory should be read and displayed, along with an estimate of the amount of disk space required to restore the dump. However, the dump itself is not restored. The use of the `DIR` option requires a dummy file parameter to be supplied, even though no disk files are created.

Examples

```
$nmdat > getdump examp dir

Please mount dump volume #1.

SA 2559 on KC (8/29/88 9:40)
Tape created by SOFTDUMP 99999X A.00.00
MPE-XL A.11.10 dumped on MON, AUG 29, 1988,  9:39 AM

     Dump Tape Contents
     ------------------

   PIM00       4.0 Kbytes
   MEMDUMP    48.0 Mbytes
   VM001      39.1 Mbytes
   VM002       0.6 Mbytes
   VM003       0.1 Mbytes
   VM004      16.4 Mbytes
   VM014       0.6 Mbytes

This dump will require approximately 62.1 Mbytes (#257913 sectors)
```

```
of disc space.

$nmdat >
```

The above example displays the directory of a dump tape and an estimate of the amount of disk space required to restore the dump.

```
$nmdat > getdump examp

Please mount dump volume #1.

SA 2559 on KC (8/29/88 9:40)
Tape created by SOFTDUMP 99999X A.00.00
MPE-XL A.11.10 dumped on MON, AUG 29, 1988,  9:39 AM

     Dump Tape Contents
     ------------------

   PIM00       4.0 Kbytes
   MEMDUMP    48.0 Mbytes
   VM001      39.1 Mbytes
   VM002       0.6 Mbytes
   VM003       0.1 Mbytes
   VM004      16.4 Mbytes
   VM014       0.6 Mbytes

This dump will require approximately 62.1 Mbytes (#257913 sectors)
of disc space.

Please stand by for disc space allocation.

                             0        100%
Loading tape file PIM00   :  +....+....+
Loading tape file MEMDUMP :  +....+....+
Loading tape file VM001   :  +....+....+
Loading tape file VM002   :  +....+....+
Loading tape file VM003   :  +....+....+
Loading tape file VM004   :  +....+....+
Loading tape file VM014   :  +....+....+
```

```
Please stand by while dump pages are posted to disk.

Dump disc file space reduced by 60% due to LZ data compression.
$nmdat >
```

The above example creates the dump file `EXAMP`. DAT keeps the user informed as to how much of the dump has been read in by printing a dot every time it transfers 10% of each file in the dump file from tape to disk. When the dump has been fully restored, the amount of disk space saved due to data compression is displayed.

Limitations, Restrictions

DUMP stores data on dump tapes in compressed form. Prior to DAT A.01.18, dumps were restored on disk in expanded form, possibly resulting in extremely large dump files. As of DAT A.01.18 and later versions, the `GETDUMP` command restores dumps in compressed form, often resulting in a significant savings in disk space when compared to uncompressed dumps. These versions of DAT are also able to access (with `OPENDUMP`) uncompressed dumps restored by previous DAT versions.

`GETDUMP` always creates at least one file when restoring a dump, known as the MEM file. Its name is made up of the dump file name followed by "MEM". Uncompressed dump files use separate files for storing data dumped from secondary store (LDEVs) and Processor Internal Memory (PIM), while compressed dumps are usually restored entirely within the MEM file.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

H[ELP]

Displays online help messages for System Debug.

Syntax

```
H[ELP] [topic] [options]
```

The `HELP` command is used to obtain help information about any command, window command, user macro, user variable, function, environment variable, and so on. Some items may fall into more than one category. For example, `S` is the single step command *and* the CM S register. In such cases, the help entries for all defined items are displayed.

Refer to the `WHELP` command for an overview of window commands.

Parameters

topic The topic for which help is desired. Help is available for a single:

- Command name.
- Environment variable name.
- Predefined function name.
- Macro name.
- User variable name.

Use the `CMDLIST`, `ENVLIST`, `FUNCLIST`, `MACLIST`, and `VARLIST` commands to see all of the names that are defined for each respective class listed above.

options The options available depend upon the class of the topic. In general, the following options are available:

`USE/NOUSE`	Short summary of usage.
`PARMS/NOPARMS`	Information about parameters.
`DESC/NODESC`	General description.

EXAMPLE/NOEXAMPLE	Examples.
ACCESS/NOACCESS	Access rights information.
ALL	Everything.

The following table indicates which combination of topics/options are valid (invalid options are ignored).

	USE	PARMS	DESC	EXAMPLE
Commands	YES	YES	YES	YES
ENV variables	NO	NO	YES	NO
Functions	YES	YES	YES	YES
Macros	YES	YES	YES	YES
User variables	NO	NO	NO	NO

Examples

```
$nmdat > help dc
"dc" is a NUMBER, and a COMMAND name.


cmd DC          display     nm cm

USE:

DC  logaddr  [count]  [base]  [recw]  [bytew]

PARMS:

logaddr  The logical code address of the first byte of code to be
         displayed.  Short pointers are treated as program file off-
         sets (NM) or offsets in the currently executing code segment
         (CM).  Long pointers are unambiguous in NM, but are treated
         as a CM program file seg.offset in CM.
count    The number of words to be displayed (default = 1).
base     The desired output base/mode of representation:
         OCT, %     Octal.
```

```
         DEC, #     Decimal.
         HEX, $     Hexadecimal.
         ASCII      Character output, separated at word boundaries.
         BOTH       Both numeric (current output base) and ASCII.
         CODE       Disassembled code.
         STRING     Continuous character output.
recw     The number of words to be displayed per line when the code
         is not disassembled.  Defaults are 4 (CM) and 8 (NM).
bytew    The width in bytes of the displayed values when the code is
         not disassembled.  Used to determine the output spacing, and
         may be 1, 2 (CM default) or 4 (NM default).
```

DESC:

The DC (Display Code) command displays CM or NM program file code. Library code may also be displayed based on the type of the LOGADDR parameter (e.g., GRP(1.70), SYS(1.40)), or by using the appropriate Display Code command variant (e.g., DCG, DCS, and so on.). By default, disassembled code is displayed one instruction per line.

EXAMPLE:

```
$ nmdebug > dc FOPEN,4
SYS $a.3714f8
003714f8  FOPEN     6bc23fd9  STW      2,-20(0,30)
003714fc  FOPEN+$4  37de00d0  LDO      104(30),30
00371500  FOPEN+$8  6bda3ee9  STW      26,-140(0,30)
00371504  FOPEN+$c  67d93ee5  STH      25,-142(0,30)
```

Display the help entry for the DC command. Notice that the two characters “DC” are a valid hexadecimal literal, so the help facility reports that fact.

```
$nmdat > help dc, desc

"dc" is a NUMBER, and a COMMAND name.

cmd DC          display     nm cm
```

```
DESC:

The DC (Display Code) command displays CM or NM program file code.
Library code may also be displayed based on the type of the LOGADDR
parameter (e.g., GRP(1.70), SYS(1.40)), or by using the appropriate
Display Code command variant (e.g., DCG, DCS, and so on.).  By default,
disassembled code is displayed one instruction per line.

$nmdat >
```

Display the help entry for the DC command but only show the command description.

```
$nmdat > help 123
"123" is a NUMBER.
```

Display the help text for the number "123".

Limitations, Restrictions

Topical help (for example, general help with expressions, breakpoints, and so on.) is not supported.

Help for the window commands do not contain help text broken down by USE, PARMS, DESC, and EXAMPLEs.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

HIST[ORY]

Displays the history command stack.

Syntax

```
HIST[ORY] option
```

Parameters

option The history stack can be displayed three ways:

ABS	With absolute command numbers. Default.
REL	With relative command numbers.
UNN	Without command numbers.

Examples

```
%nmdebug > hist
$1 = 1836/4 + 12
$2 ddb+224,20
$3 = [s-12]
$4 c
$5 ss
$6 while [s] <> 0 do ss
$7 dr status
$8 ss
```

By default, the history stack is displayed with absolute command numbers.

```
%nmdebug > hist unn
= 1836/4 + 12
ddb+224,20
= [s-12]
c
ss
```

```
while [s] <> 0 do ss
dr status
ss
```

Display the history stack without command numbers. This option allows the history to be written into a file in a form suitable for use as command file input at a later time.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

IF

If *condition* evaluates to TRUE, then execute all commands in *cmdlist*, else execute all commands in *cmdlist2*.

Syntax

```
IF condition THEN command

IF condition THEN { cmdlist }

IF condition THEN command1 ELSE command2

IF condition THEN { cmdlist } ELSE command2

IF condition THEN command1 ELSE { cmdlist2 }

IF condition THEN { cmdlist } ELSE { cmdlist2 }
```

Parameters

condition	A logical expression to be evaluated.
command *cmdlist*	A single command (or command list) that is executed if *condition* evaluates to TRUE.
command2 *cmdlist2*	A single command (or command list) that is executed if *condition* evaluates to FALSE.

Note that in nested `IF-THEN-ELSE` clauses, the first `ELSE` clause *always* matches the first `IF` clause. This is different from the conventions of most compilers, and it may not be intuitive. Explicit use of {*cmdlists*} is recommended in these nested cases.

Examples

```
%cmdebug > if [q-3]>[db+4] then c
```

If the contents of Q−3 are greater than the contents of DB+4, then continue.

```
$nmdebug > if (length>20) and (pcsf=a) then {wl "GOT IT"; c}
```

If the value of the variable `length` is greater than 20, and the contents of the predefined variable `pcsf` equals $a, then execute the following from the command list: print the string "GOT IT", then continue.

```
$nmdat > if 1 then {if 0 then wl "wee" else wl "willy"} else wl "wonka"
willy
```

This example shows a nested `IF-THEN-ELSE` clause within a *cmdlist* clause.

Limitations, Restrictions

The interpreter does not parse or analyze the contents of the clauses prior to their execution. Based on the value of the condition, the `THEN` or `ELSE` clause is be executed, and the other clause disregarded.

This implies that the clauses may be syntactically illegal, but the errors are not discovered until they are executed.

Note that in the following examples, entire clauses are bogus, but not detected:

```
$nmdebug > if TRUE then wl "good" else XXXXXXXXXXXXXXXXXX
good

$nmdebug > if FALSE then XXXXXXXXXXXXXXXX else wl "good"
good
```

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

IGNORE

Protects the next command (list) from error bailout.

Syntax

```
IGNORE option
```

The IGNORE command protects the following command, or command list, from aborting due to a detected error condition. Unless protected by the IGNORE command, a command list or subsequent macro commands are aborted/flushed as soon as any error occurs.

A special option, QUIET, causes error messages that occur within a protected command list to be suppressed.

This is similar to the MPE V/E CONTINUE command used in job and command files. See the environment variable AUTOIGNORE.

Parameters

option The user can choose to display/suppress error messages that occur during the command (list) that is protected by the IGNORE command. Two options are supported:

LOUD	Display error messages (default)
QUIET	Suppress error messages

Examples

```
%nmdebug > {wl 111; wl 22q; wl 333; wl 444}
$111
Expected a number, variable,function, or procedure (error #3720)
  undefined operator is:"22q"
```

In this example, an error causes the rest of a command list to be aborted, since it is not protected by the `IGNORE` command. As a result, the command that prints the value ($333) is never executed.

```
%nmdebug > ignore; {wl 111; wl 22q; wl 333; wl 444}
$111
Expected a number, variable,function, or procedure (error #3720)
  undefined operator is:"22q"
$333
$444
```

In this example, the `IGNORE` command is used to protect the entire command list that follows it. Even though the second command in the list produces an error, execution of the rest of the list continues. By default, the option `LOUD` is assumed, and all resulting error messages are displayed.

```
%nmdebug > ignore quiet; {wl 111; wl 22q; wl 333; wl 444}
$111
$333
$444
```

In this example, the `IGNORE QUIET` command is used to protect the command list that follows it AND to suppress all error messages. Note that the error encountered when attempting to write the value "22" is silently ignored, and the command list execution continues.

```
%nmdebug > ignore quiet; use unwind
```

In this example, the `IGNORE QUIET` command is used to protect the execution of all commands found within the USE file `unwind`. If this use file uses additional USE files, the commands in those additional USE files are also protected.

```
%nmdebug > ignore quiet; printsum (200 tablesize("mytable"))
```

In this example, the `IGNORE QUIET` command is used to protect the following command that invokes a macro named `printsum`. All commands within this macro are protected. In addition, all commands within the macro function `tablesize` are protected.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

INIT*xx*

Privileged Mode

Initialize registers from a specified location.

Syntax

```
INITNM virtaddr [ISM |PIMREAL |PIMVIRTUAL]
INITCM virtaddr [ISM |PIMREAL |PIMVIRTUAL]

INITNM TCB
INITCM TCB | CMG | REGS
```

This command is for use by experienced DAT users and internals specialists to initialize DAT when a dump is corrupted. The command is also provided for the experienced Debug user.

For the `INITNM` command, the NM register set is loaded from the specified location. It is assumed that the location contains data in the form of an interrupt stack marker (ISM) which is the default, or in the form of processor internal memory (PIM). Not all of the machine's registers are found in an ISM. If this is the structure being used, those registers not stored in the ISM are retrieved from the save state area in the dump (or from the running machine in Debug).

For the `INITCM` command, the CM register set is loaded from one of several locations depending upon the option specified. Four possibilities exist:

- The emulator/translator is not running, and the CM state for the process is stored in the CMGLOBALS area of the PIB. The `CMG` option is used in this case.
- The emulator/translator is running, in which case the CM state is maintained in the native mode registers. In this case the virtual address of an interrupt stack marker (ISM) or processor internal memory record (PIM)

containing the emulator/translator's native mode register set should be given so that the CM state may be extracted from the registers.

- The state of the emulator/translator is stored in the task control block (TCB). As in the PIM and ISM case above, the register data found is used to set up the CM state.
- The user desires to construct the CM state from scratch. To do this, the user must place into the current NM register set (using the `MR` command) values that correspond to the state of an active emulator/translator. The appropriate values are then extracted from the register set to build the CM state. The `REGS` options allows this to be done.

Parameters

virtaddr — Any valid expression specifying the virtual address of an interrupt stack marker (ISM) or a processor internal memory (PIM) record. The type of structure is indicated by one of the following optional parameters:

`ISM`	The data is an interrupt stack marker (default).
`PIMVIRTUAL`	The data is processor internal memory format.
`PIMREAL`	The data is processor internal memory format, but the address is a real memory address. If a full virtual address is given, the offset part is used as the real memory address.

`TCB` — This parameter indicates that the register save state in the task control block (TCB) for the current PIN should be used for initialization. The register save state in the TCB is in the form of an *interrupt_marker_type*.

`CMG` — This parameter indicates that the CM registers should be initialized based on CMGLOBALS area in the process information block (PIB) of the current process.

`REGS` — This parameter indicates that the CM registers should be initialized based on the current NM regs. The NM regs are interpreted as containing values used by the emulator/translator.

Examples

```
$ nmdebug > initnm 0.tcb(20)
```

Initialize the native mode registers from the indicated virtual address.

```
% cmdebug > initcm 40153014
```

Initialize the CM registers from the interrupt marker that starts at address 40153014. The process was most likely in the emulator (or else the CM state would be stored in the CMGLOBALS area of the PIB).

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

KILL

Debug only

Privileged Mode

Issues a request to process management to kill the specified process.

Syntax

```
KILL pin
```

Parameters

pin The process identification number (PIN) to be killed. If you are a privileged user, you may specify any PIN. If you are not privileged, you may specify any PIN that is a child of the process making this request.

Examples

```
$nmdat > kill 8
```

Tell process management to kill PIN 8.

Limitations, Restrictions

This routine is implemented by calling the process management `KILL` routine. That routine does not kill a process until it is out of system code and is no longer critical. Debug waits until the request can be completed.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the

exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

LEV

Sets the current environment to the specified stack level in the stack markers.

Syntax

```
LEV [number]
LEV [number] [interrupt_level]
```

The `LEV` command changes the current environment to the environment at the specified stack level.

All commands accurately reflect the register values that are in effect a level change. Windows also reflect the new level values.

If the `CONTINUE` or `SS` command in Debug is issued after changing levels, an implicit `LEV 0` is performed.

If any error is encountered during a level change, the environment is automatically set to stack level 0.

The following algorithm is used to set level *n* on the CM stack:

WHILE lev <> desired level DO

Get previous stack marker.
Set Q based on delta-Q in marker.
Set S to Q-4.
Set X based on X in marker.
Set STATUS based on status marker.
Set CMPC based on status and P offset in marker.
Set CIR based on fetch from new value of CMPC.

The following algorithm is used to set level *n* on the NM stack:

Get current frame info (based on unwind info);

WHILE lev <> desired level DO
Restore entry save registers (based on frame unwind info);

Get previous frame (based on unwind info);
IF frame is an interrupt stack marker (ISM) THEN
  Restore RP, SP, DP, SR4, SR5, SR0, PCQ from the ISM
ELSE
  Set RP, SP, DP, SR4, to new values from the stack;
  Restore call save registers (based on unwind info);

Parameters

number The stack level number at which the environment should be set.

interrupt_level The interrupt level number at which the environment should be set. If this parameter is omitted, the current interrupt level is assumed.

This parameter is valid only for NM.

Examples

```
%cmdebug > tr
      PROG %   0.1421   PROCESSSTUDENT+14     (mITroc CCG)  SEG'
*  0) PROG %   0.2004   PROCESSSTUDENT+377    (mITroc CCG)  SEG'
   1) PROG %   0.253    OB'+253               (mITroc CCG)  SEG'
   2) SYS  %  25.0      ?TERMINATE            (MItroc CCG)  CMSWITCH''

%cmdebug >  dr cmpc
CMPC=PROG %0.1421

      %cmdebug > lev 2
```

First use TR to list the stack trace in order to decide which level is desired. The current value of CMPC is then displayed. Next the stack level is set to level 2.

```
%cmdebug > tr
      PROG %   0.1421   PROCESSSTUDENT+14     (mITroc CCG)  SEG'
   0) PROG %   0.2004   PROCESSSTUDENT+377    (mITroc CCG)  SEG'
   1) PROG %   0.253    OB'+253               (mITroc CCG)  SEG'
*  2) SYS  %  25.0      ?TERMINATE            (MItroc CCG)  CMSWITCH''
```

```
%cmdebug > dr cmpc
CMPC=PROG %0.253
```

The above stack trace reveals that the level has been changed to stack level two (note the asterisk). The current value of CMPC is also displayed and confirms that the registers have been correctly updated as well.

```
$nmdebug > tr,ism
     PC=a.006777fc trap_handler
* 0) SP=40221338 RP=a.002a1fec conditional+$ac
  1) SP=40221338 RP=a.000a5040 hpe_interrupt_marker_stub
 --- Interrupt Marker
  2) SP=402211e8 RP=25d.00015134 small_divisor+$8
 --- End Interrupt Marker Frame ---

     PC=25d.00015134 small_divisor+$8
  0) SP=402211e8 RP=25d.00015d38 average+$b0
  1) SP=402211e8 RP=25d.00015c74 ?average+$8
       export stub: 25c.00005d98 processstudent+$74
  2) SP=40221180 RP=25c.00006b1c PROGRAM+$300
  3) SP=40221100 RP=25c.00000000
     (end of NM stack)
```

Show a native mode stack trace that contains an interrupt marker.

```
$nmdebug > lev 1,1
$nmdebug > tr,ism
     PC=25d.00015134 small_divisor+$8
  0) SP=402211e8 RP=25d.00015d38 average+$b0
* 1) SP=402211e8 RP=25d.00015c74 ?average+$8
       export stub: 25c.00005d98 processstudent+$74
  2) SP=40221180 RP=25c.00006b1c PROGRAM+$300
  3) SP=40221100 RP=25c.00000000
     (end of NM stack)
```

Use the LEV command to set the environment to stack level 1, interrupt level 1. A stack trace confirms that the environment has been correctly changed.

Limitations, Restrictions

You must be at stack level 0 in order to modify any registers.

For native mode code, if you are in procedure entry or exit code, this command may not function properly. For example, if the user is stopped in entry code, callee save registers have not been saved and therefore are restored incorrectly. Other scenarios exist.

LEV

If the environment for the CM stack is set to a level that is a switch marker, no values for CMPC and CIR are available.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

LIST

Controls the recording of input and output to a list file.

Syntax

```
LIST

LIST  [filename]

LIST  [ON ]
LIST  [OFF]

LIST  [CLOSE]
```

All Debug input/output is recorded to an open, active list file. This includes the prompt, user command input, and all resulting output, with the exception of window displays and updates. Users typically use the list file to record Debug output to a file for later reference or printing.

`LIST`, entered alone, displays the state of the list file, including the file name, if open, and current status (ON/OFF).

`LIST` *filename* opens the specified file and activates (turns ON) the list file. If another list file was already opened, it is first closed (saved), before the new file is opened.

`LIST ON` and `LIST OFF` can be used to activate/deactivate the currently opened list file. The file remains open (pending), but Debug output is *not* recorded if the list file is `OFF`.

`LIST CLOSE` closes (saves) the current opened list file.

Parameters

filename The file name for the list file that is to be opened. If the file already exists, it is automatically purged (without warning), and reopened new.

If omitted, the status of the current list file is displayed.

Examples

```
%cmdebug > list junk1
```

Open a new list file named `junk1` and activate it (ON). All Debug input/output is automatically recorded in this file until it is explicitly deactivated (`LIST OFF`) or closed (`LIST CLOSE`).

```
%cmdebug > list off
%cmdebug > dq-40, 200
%cmdebug > list on
```

Temporarily disable the list file, while we display 200 Q-relative words, then enable the list file again.

```
%cmdebug > list close
```

Close (and save) the current list file. Auto-listing is now off.

Limitations, Restrictions

Unless a file equation is used, the list file is opened as follows:

CCTL, FIXED, ASCII, 20000 Records.

The record size is based on the `LIST_WIDTH` environment variable.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

LISTREDO

Displays the history command stack.

Syntax

```
LISTREDO                    alias for HIST[ORY]
```

LISTREDO is a predefined alias for the HIST[ORY] command.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

LOADINFO

Debug only

Lists information about the currently loaded program and libraries.

Syntax

```
LOADINFO
```

For Debug, this command displays the list of files that are loaded by the current process. Both CM and NM libraries and program files are included in the list. This list is automatically updated as the process dynamically loads NM and CM libraries.

For DAT and SAT, this command displays the list of files for which symbol name and address information is available. In most cases, this consists of the system libraries (NL.PUB.SYS and SL.PUB.SYS). In addition, any files that were loaded by the loader as "dumpworthy" files are included in this list.

For all of the tools, any file mapped in with the XL command has an entry in this loaded file list as well. It is therefore possible to have several entries with the same space ID (SID) in the list. (Refer to the XL command for additional details).

Parameters

none

Examples

```
$ nmdebug > loadinfo
nm  PROG   TEST4.TEST.QA              SID=$23
    parm=#2  info=""
nm  GRP    XL.TEST.QA                 SID=$1d
```

```
nm  USER   LIB1.TESTLIBS.QA             SID=$26
nm  USER   LIB2.TESTLIBS.QA             SID=$27
nm  SYS    NL.PUB.SYS                   SID=$a
cm  GRP    SL.TEST.QA
```

Assume that a typical NM program is being executed. Display the currently loaded program and library files.

```
% cmdebug > loadinfo
cm  PROG   PFLIGHT.MODEL.DESIGN
    parm=#3  info="wind 5, clouds2"
cm  GRP    SL.MODEL.DESIGN
cm  PUB    SL.PUB.DESIGN
cm  SYS    SL.PUB.SYS
nm  GRP    XL.PUB.SYS                   SID=$1c
nm  SYS    NL.PUB.SYS                   SID=$a
```

Assume that a typical CM program is being executed. Display the currently loaded program and library files.

Limitations, Restrictions

If the INFO string is longer than 255 characters, it is not displayed.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

LOADPROC

Debug only

Dynamically loads a specified CM procedure from a logically specified CM library selector.

Syntax

```
LOADPROC procedurename libselect
```

Parameters

procedurename	The name of the procedure to be loaded.
libselect	The logical library from which the procedure is to be loaded. The library selector must be specified from the following keyword list: `GRP` Group library (program group) `PUB` Account library (program group) `LGRP` Group library (logon group) `LPUB` Account library (logon group) `SYS` System library

Examples

```
%cmdebug > loadproc mysort pub
```

Dynamically load the procedure `mysort` from `PUB` (the account library).

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

LOC

Defines a local variable within a macro body.

Syntax

```
LOC   var_name [:var_type] [=] var_value
```

The `LOC` command can only be executed within a macro.

Local variables are known *only* to the macro in which they are defined. The environment variable `NONLOCALVARS` may be changed so that local variables are accessible to any macro called after a local variable has been defined. (Refer to the `ENV` command).

Local variables are automatically deleted when the macro in which the variable was defined finishes execution.

Parameters

var_name The name of the local variable being defined. Names must begin with an alphabetic character and are restricted to thirty-two (32) characters, that must be alphanumeric or an underscore (_), an apostrophe ('), or a dollar sign ($). Longer names are truncated (with a warning). Names are case insensitive.

var_type The type of the local variable. The following types are supported:

`STR`	String
`BOOL`	Unsigned 16 bit
`U16`	Unsigned 16 bit
`S16`	Signed 16 bit
`U32`	Unsigned 32 bit
`S32`	Signed 32 bit

S64	Signed 64 bit
SPTR	Short pointer
LPTR	Long pointer
PROG	Program logical address
GRP	Group library logical address
PUB	Account library logical address
LGRP	Logon group library logical address
LPUB	Logon account library logical address
SYS	System library logical address
USER	User library logical address
TRANS	Translated CM code virtual address

If the type specification is omitted, the type is assigned automatically, based on *var_value*.

The optional *var_type* allows the user to explicitly specify the desired internal representation for *var_value* (that is, signed or unsigned, 16-bit or 32-bit) for this particular assignment only. It does *not* establish a fixed type for the lifetime of this variable. A new value of a different type may be assigned to the same local variable (name) by a subsequent `LOC` command.

var_value — The new value for the variable, which can be an expression. An optional equal sign "=" can be inserted before the variable value.

Examples

```
$nmdat > loc temp a.c000243c
```

Define local variable `temp` to be the address a.c000243c. By default, this variable is of type `LPTR` (long pointer), based on the value.

```
$nmdebug > loc count=1c
```

Define local variable `count` to be the value 1c.

```
$nmdebug > loc s1:str="this is a string"
```

Define local variable `s1` to be of type `STR` (string) and assign the value "this is a string".

```
nmdat > mac sum(p1 p2) {loc temp p1+p2; loclist; ret temp}
```

```
nmdat > wl sum (1 2)
var temp  : U16 = $3
var loc p2    : U16 = $2
var loc p1    : U16 = $1
$3
```

This example shows how the `LOCLIST` command, when executed as part of a macro body, displays all currently defined local variables. Note that the macro parameters appear as local variables. Local variables are always listed in the reverse order that they were created.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

LOCL[IST]

Lists the local variables that are defined with a macro.

Syntax

```
LOCL[IST] [pattern]
```

Parameters

pattern The name of the local variable(s) to be listed.

This parameter can be specified with wildcards or with a full regular expression. Refer to appendix A for additional information about pattern matching and regular expressions.

The following wildcards are supported:

`@`	Matches any character(s).
`?`	Matches any alphabetic character.
`#`	Matches any numeric character.

The following are valid name pattern specifications:

`@`	Matches everything; all names.
`pib@`	Matches all names that start with "pib".
`log2##4`	Matches "log2004", "log2754", and so on.

The following regular expressions are equivalent to the patterns with wildcards that are listed above:

```
`.*`
`pib.*`
`log2[0-9][0-9]4`
```

By default, all local variables are listed.

Examples

```
nmdat > mac sum(p1 p2) {loc temp p1+p2; loclist; ret temp}
nmdat > wl sum (1 2)
var temp  : U16 = $3
var loc p2    : U16 = $2
var loc p1    : U16 = $1
$3
```

This example shows how the LOCLIST command, when executed as part of a macro body, displays all currently defined local variables. Note that the macro parameters appear as local variables. Local variables are always listed in the reverse order that they were created.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

LOG

Controls the recording of user input to the logfile.

Syntax

```
LOG

LOG [filename]

LOG [ON   ]
LOG [OFF  ]
LOG [CLOSE]
```

All Debug user input can be recorded to the log file. The log file can be used as a playback file.

`LOG`, entered alone, displays the state of the log file, including the file name, if open, and the current status (ON/OFF).

`LOG` *filename* opens the specified file and activates (turns on) the log file. If another log file is already opened, it is first closed (saved) before the new file is opened. This command does an implicit `LOG ON`

`LOG ON` and `LOG OFF` can be used to activate/deactivate-activate the currently opened log file. The file remains open (pending), but Debug input is *not* recorded if the log file is OFF.

`LOG CLOSE` closes (saves) the current opened log file. Note that this command is written to the log file. Executing this command without a log file has no effect.

Parameters

filename The file name for the logfile that is to be opened. If the file already exists, it is automatically purged (without warning), and reopened new. This command performs an implicit `LOG ON`.

If omitted, the status of the current log file is displayed.

Examples

```
%cmdebug > log logfile
```

Open a new logfile named `logfile` and start logging to it.

```
%cmdebug > log close
```

Close (and save) the current logfile. Auto-logging is now off.

Limitations, Restrictions

Unless a file equation is used, the list file is opened as the following:

CCTL, FIXED, ASCII, 10000 Records, 80 byte record width.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

M (modify)

Debug only

Privileged Mode: MA, MD, MCS, MZ, MSEC

Modifies the contents of the specified number of words at the specified address.

Syntax

```
MA   offset   [count] [base] [newvalue(s)]      ABS relative
MD   dst.off  [count] [base] [newvalue(s)]      Data segment
MDB  offset   [count] [base] [newvalue(s)]      DB relative
MS   offset   [count] [base] [newvalue(s)]      S relative
MQ   offset   [count] [base] [newvalue(s)]      Q relative

MC   logaddr  [count] [base] [newvalue(s)]      Program file (default)
MCG  logaddr  [count] [base] [newvalue(s)]      Group library
MCP  logaddr  [count] [base] [newvalue(s)]      Account library
MCLG logaddr  [count] [base] [newvalue(s)]      Logon group
MCLP logaddr  [count] [base] [newvalue(s)]      Logon account
MCS  logaddr  [count] [base] [newvalue(s)]      System library
MCU  fname logaddr  [count] [base] [newvalue(s)] User library
MCA  cmabsaddr  [count] [base] [newvalue(s)]    Absolute CST
MCAX cmabsaddr  [count] [base] [newvalue(s)]    Absolute CSTX

MV   virtaddr   [count] [base] [newvalue(s)]    Virtual
MZ   realaddr   [count] [base] [newvalue(s)]    Real memory

MSEC ldev.off   [count] [base] [newvalue(s)]    Secondary store
```

By default, the current value is displayed. The ENV variable QUIET_MODIFY can be used to suppress the display of the current value.

Parameters

offset `MA`, `MDB`, `MQ`, `MS` only. The CM word offset that specifies the relative starting location of the area to be modified.

logaddr

`MC`, `MCG`, `MCP`, `MCLG`, `MCLP`, `MS`, `MCU` only. A full logical code address (`LCPTR`) specifies three necessary items:

- The logical code file (`PROG`, `GRP`, `SYS`, and so on.).
- NM: the virtual space ID number (SID).
  CM: the logical segment number.
- NM: the virtual byte offset within the space.
  CM: the word offset within the code segment.

Logical code addresses can be specified in various levels of detail:

- As a full logical code pointer (`LCPTR`):

`MC procname+20`	Procedure name lookups return `LCPTR`s.
`MC pw+4`	Predefined `ENV` variables of type `LCPTR`.
`MC SYS(2.200)`	Explicit coercion to a `LCPTR` type.

- As a long pointer (`LPTR`):

  `MC 23.2644` *sid.offset* or *seg.offset*

  The logical file is determined based upon the command suffix. For example:

  `MC` implies `PROG`
  `MCG` implies `GRP`
  `MCS` implies `SYS`, and so on

- As a short pointer (`SPTR`):

  `MC 1024` *offset* only

  For NM, the short pointer offset is converted to a long pointer using the function `STOLOG`, which looks up the SID of the loaded logical file. This is different from the standard short to long pointer conversion, `STOL`, which is based on the current space registers (SRs).

  For CM, the current executing logical segment number and the current executing logical file are used to build a `LCPTR`.

The search path used for procedure name lookups is based on the command suffix letter:

`MC`	Full search path: NM: `PROG`, `GRP`, `PUB`, `USER`(s), `SYS` CM: `PROG`, `GRP`, `PUB`, `LGRP`, `LPUB`, `SYS`
`MCG`	Search `GRP`, the group library.
`MCP`	Search `PUB`, the account library.
`MCLG`	Search `LGRP`, the logon group library.
`MCLP`	Search `LPUB`, the logon account library.
`MCS`	Search `SYS`, the system library.
`MCU`	Search `USER`, the user library.

For a full description of logical code addresses, refer to the section "Logical Code Addresses" in Chapter 2.

fname

`MCU` only. The file name of the NM user library. Since multiple NM libraries can be bound with the `XL=` option on a `RUN` command,

```
:run nmprog; xl=lib1,lib2.testgrp,lib3
```

it is necessary to specify the desired NM user library. For example:

```
MCU lib1 204c
MCU lib2.testgrp test20+1c0
```

If the file name is not fully qualified, the following defaults are used:

Default account: the account of the program file.
Default group: the group of the program file.

cmabsaddr

`MCA`, `MCAX` only. A full CM absolute code address specifies three necessary items:

- Either the CST or the CSTX.
- The absolute code segment number.
- The CM word offset within the code segment.

Absolute code addresses can be specified in two ways:

- As a long pointer (`LPTR`):

`MCA 23.2644`	Implicit `CST` 23.2644
`MCAX 5.3204`	Implicit `CSTX` 5.3204

- As a full absolute code pointer (`ACPTR`):

`MCA CST(2.200)`	Explicit `CST` coercion
`MCAX CSTX(2.200)`	Explicit `CSTX` coercion
`MCAX logtoabs(prog(1.20))`	Explicit absolute conversion

The search path used for procedure name lookups is based on the command suffix letter:

`MCA`	`GRP`, `PUB`, `LGRP`, `LPUB`, `SYS`
`MCAX`	`PROG`

virtaddr

`MV` only. The virtual address to be modified.

Virtaddr can be a short pointer, a long pointer, or a full logical code pointer.

realaddr MZ only. The real mode memory address to be modified.

ldev.off — `MSEC` only. The logical device number (LDEV) and byte offset of the data on disk to be displayed. This address is entered in the form *ldev.byteoffset*.

count — `MA`, `MC`, `MD`, `MDB`, `MS`, `MQ`: The number of CM 16-bit words to be modified.

`MC`, `MV`, `MZ`: The number of NM 32-bit words to be modified.

If omitted, a single line of values is modified.

base — The desired representation mode for output values:

`%` or `OCTAL`	Octal representation
`#` or `DECIMAL`	Decimal representation
`$` or `HEXADECIMAL`	Hexadecimal representation
`ASCII`	ASCII representation

This parameter can be abbreviated to as little as a single character.

If omitted, the current output base is used.

newvalue(s) — The new values for the specified locations. Specified new values are automatically assigned to the locations until the new values are exhausted. If the new values are omitted, or if they run out, Debug prompts for the remaining new values. To retain the original value, simply press (Return). The character dot "." can be entered to abort the modification loop. All locations modified before the dot is encountered are permanently changed.

Examples

```
$nmdebug > mv sp-2c,,,4
$ Virt 21.40050780 = '....'  $e7        := 4
```

Modify value at SP-2c, replacing it with $4.

```
%cmdebug > md 1.64,6,h
$ DST 1.34   = "v4"  $7634 := %111
$ DST 1.35   = ".."  $5    :=        (retain original value)
$ DST 1.36   = ".."  $fffa := $c0
```

M (modify)

```
$ DST 1.37   = ".."  $fff0 := 1234
$ DST 1.38   = ".."  $0    := .
current/remaining modifications aborted at user request
```

Modify 6 words starting at DST 1.64. Display values (and addresses) in hex.

DST 1.34 is assigned a new value of %111.
DST 1.35 retains its original value of %5.
DST 1.36 is assigned a new value of $c0.
DST 1.37 is assigned a new value of 1234.
Dot "." terminates modifications.

The modifications for DST 1.34 through 1.37 have been successfully completed.

```
%cmdebug > mq-30,6
% Q-30      = ".P"  %27120  := "AB"
% Q-27      = "UB"  %52502  := 'CD'
% Q-26      = ".S"  %27123  := u16("EF")
% Q-25      = "YS"  %54523  :=
% Q-24      = ".."  %177772 := [q-2]
% Q-23      = ".."  %7      := !s + (1000-[db+22]/2)
```

Modify 6 words starting at Q-%30. The current values are displayed in ASCII and octal (current output base).

Q-30 is assigned the (implicitly coerced) integer value of “AB”.
Q-27 is assigned the implicitly coerced) integer value of ’CD’.
Q-26 is assigned the explicitly coerced unsigned 16-bit integer value of “EF”.
Q-25 is left unchanged.
Q-24 is assigned the contents of Q-2.
Q-23 is assigned the value of the S register + (1000 - the contents of DB+22 divided by 2).

Limitations, Restrictions

When CM code has been translated, modification of the original object code has no effect. The NM translated code must be modified.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

MAC[RO]

Defines a macro.

Syntax

```
MAC[RO] name {body}
MAC[RO] name [ (parameters) ] {body}
MAC[RO] name [ (parameters) ] [options] {body}
```

Macros are a body of commands that are executed (invoked) by *name*. Macros can have optional parameters.

Macros can be executed as if they were commands.

Macros can also be invoked as functions within expressions to return a value.

Macro definitions can include three special options in order to specify a version number (`MACVER`), a help string (`MACHELP`), and a keyword string (`MACKEY`). See the `MACLIST` command.

Reference counts are maintained for macros. Each time a macro is invoked, the reference count for the macro is incremental. (Refer to the `MACREF` and `MACLIST` commands.)

Two special commands are provided to assist with the debugging and support of macros. See the `MACECHO` and `MACTRACE` commands.

The entire set of currently defined macros can be saved into a binary file for later restoration. (Refer to the `STORE` and `RESTORE` commands.)

Parameters

name The name of the macro that is being defined. Names must begin with an alphabetic character and are restricted to thirty-two (32) characters, that must be alphanumeric, or "_", or "' ", or "$". Longer names are truncated (with a warning). Names are case insensitive.

All macros are functions that can be used as operands within expression to return a single value of a specified type.

A default macro return value can optionally be specified directly following the macro name. The *return_type* must be preceded by a colon. The default *return_value* must be preceded by an equal sign, and can be entered as an expression. Below is a syntax of a macro call, followed by examples:

```
macro name [:return_type] [= return_value]
```

For example:

```
macro getnextptr:s16 = -1              {body}
macro tblname = "UNDEF"                {body}
macro tblsize:u32 = max * entrylen     {body}
macro fmtstring:str                    {body}
```

If the default macro *return_value* is not specified, one is assigned automatically, based on the type of the macro. The following table lists the default *return_values* that are based on the macro's *return_type*:

Macro Return Type	Default Return Value
BOOL	FALSE
U16, S16, U32, S32, SPTR	0
LPTR	0.0
CPTR class	0.0 (based on type)
STR	' ' (null string)

By default, a macro is assigned the return value of 0 as a signed 32-bit number.

(*parameters*) Macros can optionally have a maximum of five declared parameters. Parameter definitions are declared within parentheses, separated by blanks or commas.

(*parm1def parm2def, parm3def, parm4def parm5def*)

Parameter names have the same restrictions as macro names. Names must begin with an alphabetic character and are restricted to thirty-two (32) characters, that must be alphanumeric, or an underscore (_), a single quotation ('or'), or a dollar sign ($). Longer names are truncated (with a warning). Names are case insensitive.

Each parameter definition can include an optional *parmtype* declaration that must follow after a colon. In addition, a default initial value for the parameter can optionally be specified, preceded by an equal sign. The initial value can be an expression. Below is a syntax of a parameter description, followed by examples:

(*parmname1* [:*parmtype1*] [=*parm_default_value1*] , ..

```
( addr:sptr=c000104c, len=0, count=20 )
( p1:u32=$100, p2=40-!count  p3:str="totals")
```

When a macro is invoked, a local variable is declared for each parameter, just as if the following command(s) had been entered:

```
LOC parmname1 :type1= default1
LOC parmname2 :type2= default2  ... etc.
```

Parameters are referenced within the macro body in the same manner that local variables are referenced. The parameter name can be preceded by an optional exclamation mark (!) to avoid ambiguity.

When execution of the macro body is completed, the local variables declared for the parameters are automatically deleted.

{*body*} The macro body is a single command, or a list of commands, entered between curly braces. Multiple commands must be separated by semicolons. The commands in this body are executed whenever the macro is invoked. For example:

```
            { CMD }
{ CMD1; CMD2; CMD3; .. CMDn }
```

Unterminated command lists, introduced by the left curly brace, can span multiple lines without the use of the continuation character (&) between lines. Additional command lines are automatically digested as part of the *cmdlist* until the closing right brace is detected.

```
{ CMD1;
  CMD2;
  CMD3;
      ...
  CMDn }
```

The `RETURN` command is used within the macro body to return a specified value and to exit the macro immediately. If a `RETURN` command is not supplied within the macro body, the macro exits when all commands have been executed, and the default return value is used.

options Special macro options can be specified following the parameter declarations that precede the macro body. Any number of these options can be specified in any order. Each option is specified as a keyword, followed by a (case sensitive) string value:

`MACVER` = *version_string*
`MACKEY` = *keyword_string*

MACHELP = *help_string*

The following are typical valid declarations for macro options:

MACVER = 'A.00.01'
MACKEY = "PROCESS PIN PARENT"
MACHELP = "Returns the pin number of the parent process"

By default, the null string (' ') is assigned for unspecified options.

Examples

```
$nmdat > macro showtime {wl 'The current time is: ' time}
$nmdat > showtime
The current time is:  2:14 PM
```

This example demonstrates a simple macro that executes a single command. The new macro, named `showtime`, is defined and then executed as if it were a command. The macro body, in this case a simple write command, is executed, and the current time is displayed. This macro has no parameters.

```
$nmdat > macro starline (num:u16=#20) {
{$1} multi > while num > 0 do {
{$2} multi >   w '*';
{$2} multi >   loc num num -1 };
{$1} multi > wl }

$nmdat > starline (5)
*****

$nmdat > starline (#60)|
************************************************************

$nmdat > starline
********************

$nmdat > starline (-3)
Parameter type incompatibility.  (error #4235)
  expected the parameter "num:U16"  for "starline"
  starline (-3)
             ^
 Error during macro evaluation.  (error #2115)
```

This example defines a macro named `starline` that prints a line of stars. The number of stars is based on the macro parameter `num` that is typed (unsigned 16-bit), and has a default value of decimal twenty.

The macro is entered interactively across several lines. The unterminated left curly brace causes the interpreter to enter *multi-line mode*. The prompt changes to indicate that the interpreter is waiting for additional input. The nesting level, or depth of unterminated curly braces, is displayed as part of the prompt.

The macro `starline` is called with the parameter `5`, and a line of five stars is printed. The macro is called again to print a line with sixty stars. In the third invocation no parameter value is specified, so the default value of twenty stars is used.

The fourth and final call displays the parameter type checking, which is performed for typed macro parameters. In this example a negative number of stars are requested, and the interpreter indicates that the parameter is invalid.

```
$nmdat > mac fancytime {starline(#30); showtime; starline(#30)}
$nmdat > fancytime
******************************
The current time is:  2:17 PM
******************************
```

In this example a new macro named `fancytime` is defined. This new macro calls the two previously defined macros in order to produce a fancy display of the time.

Macros can include calls to other macros. The contents of macro bodies are not inspected when macros are defined. Therefore one macro can include a call to another macro before it is defined.

```
%nmdebug > mac printsum (p1,p2=0) {wl "the sum is " p1+p2}
%nmdebug > printsum (1 2)
the sum is $3
%nmdebug > printsum 3 4
the sum is $7
%nmdebug > printsum 5
the sum is $5
```

Defines macro `printsum` that prints the sum of the two parameters `p1` and `p2`. Note how the parameters are referenced as simple local variables within the macro body. When a macro is used as a command, parentheses around parameters are optional. Also note how the default value (0) is used for the omitted optional parameter `p2`.

```
%cmdebug > mac is (p1="DEBUG",p2:str="GNARLY") {wl p1 "is very" p2.}
%cmdebug > is ("MPE" 'mysterious')
MPE is very mysterious.
%cmdebug > is ("mpe")
mpe is very GNARLY.
%cmdebug > is
DEBUG is very GNARLY.
```

These examples demonstrate simple typed parameters with default values. The default values are used whenever optional parameters are omitted.

```
%nmdat > mac double (p1) { return p1*2 }
```

```
%nmdat > wl double(2)
$4
%nmdat > wl double(1+2)+1
$7
```

Defines macro `double` as a function with one parameter `p1`. The `RETURN` command is used to return the functional result of twice the input parameter. Note how the macro is used as a function, as an operand in an expression.

```
%nmdat > mac triple (p1:INT) { return p1*3 }
%nmdat > wl triple(2)
$6
%nmdat > wl triple (double (1+2))
$12
```

Macro function `triple` is similar to macro function `double` defined above. Note that macros (used as functions) can be nested within expressions.

```
$nmdebug > { macro factorial=1 (n)
{$1} multi >  machelp = 'Returns the factorial for parameter "n"'
{$1} multi >  mackey  = 'FACTORIAL UTILITY ARITH TEST'
{$1} multi >  macver = 'A.01.00'
{$1} multi >  { if n <= 0
{$2} multi >    then return
{$2} multi >    else if n > 10
{$2} multi >         then { wl "TOO BIG"; return}
{$2} multi >         else return n * factorial(n-1)
{$2} multi >  }
{$1} multi > }

$nmdebug > wl factorial(0)
$1
$nmdebug > wl factorial(1)
$1
$nmdebug > wl factorial(2)
$2
$nmdebug > wl factorial(3)
$6
$nmdebug > wl factorial(123)
TOO BIG
$1
```

This example defines a macro function named `factorial` that has a default return value of 1. A help string, keyword string, and version string are included in the macro definition.

Note that the macro definition was preceded by a left curly brace in order to enter *multi-line mode*. This allowed the options to be specified on separate lines, before the left curly brace for the macro body.

This macro calls itself recursively, but protects against runaway recursion by testing the input parameter against an upper limit of ten.

Discussion - Macro Parameters

Assume that the following macro is defined.

```
$nmdat > { macro double( num=$123, loud=TRUE)
{$1} multi > { if loud
{$2} multi >   then wl 'the double of ', num, ' = ', num*2;
{$2} multi >   return num*2}
{$1} multi > }
$nmdat >
```

This macro has two optional parameters: `num` that defaults to the value 123, and `loud` that defaults to TRUE.

The macro is written in a manner that allows it to be invoked as a function to return a value that is the double of the input parameter. The second parameter controls the display of an output line, and therefore this macro might also be used as a command to calculate a value and display the result. When invoked as a command, the returned value is simply ignored.

The following examples illustrate the rules governing the specification of macro parameters for macros invoked as functions and for macros invoked as commands.

Macro Functions

For macros invoked as a function, parameters *must* be specified within parentheses as a parameter list. The same convention applies to parameters passed to any of the System Debug standard functions. Optional parameters can be implicitly omitted if a comma is used as a parameter place holder. When all parameters are optional and are to be omitted, the parentheses around the empty parameter list can be omitted.

```
$nmdat > wl double(1,false)
$2

$nmdat > wl double(,false)
$246

$nmdat > wl double ()
the double of $123 = $246
```

```
$246

$nmdat > wl double
the double of $123 = $246
$246
```

Macro Commands

For macros invoked as commands, parameter(s) can be specified without parentheses, in the same manner that System Debug commands are normally used.

Unlike normal System Debug commands, however, parentheses can be used to surround a parameter list for a macro command. If the first parameter to a macro command requires a parenthesized expression, an ambiguity arises. In this case, parentheses should be used around the entire parameter list.

Just as with macro functions, optional parameters can be implicitly omitted if a comma is used as a parameter place holder.

```
$nmdat > double 1
the double of $1 = $2

$nmdat > double (2)
the double of $2 = $4

$nmdat > double 3 true
the double of $3 = $6

$nmdat > double ( (1+2)*3 )
the double of $9 = $12

$nmdat > double
the double of $123 = $246

$nmdat > double 6,false
$nmdat >
```

Limitations, Restrictions

Refer to ENV MACROS and ENV MACROS_LIMIT. These environment variables determine the number of macros that can be created.

Current limit of 32 characters in a macro name or macro parameter name.

Current limit of five parameters per macro.

Macro parameters are passed by value. Parameter values are not changed.

The total length of an entire macro definition is limited by the maximum supported string length, that is currently 2048 characters. See the STRMAX function.

The System Debug interpreter maintains an internal command stack for general command execution, including the execution of macros. The command stack is large enough to support the useful nesting of macros, including simple recursive macros. Command stack overflow is possible, however, and when detected, results in an error message and the immediate termination of the current command line execution. Following command stack overflow, the stack is reset, the prompt is displayed, and normal command line interpretation resumes.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

MACD[EL]

Macro delete. Deletes the specified macro definition(s).

Syntax

```
MACD[EL] pattern
```

Parameters

pattern The name(s) of the macro(s) to be deleted.

This parameter can be specified with wildcards or with a full regular expression. Refer to appendix A for additional information about pattern matching and regular expressions.

The following wildcards are supported:

`@`	Matches any character(s).
`?`	Matches any alphabetic character.
`#`	Matches any numeric character.

The following are valid name pattern specifications:

`@`	Matches everything; all names.
`pib@`	Matches all names that start with "pib".
`log2##4`	Matches "log2004", "log2754", and so on.

The following regular expressions are equivalent to the patterns with wildcards that are listed above:

```
'.*'
'pib.*'
'log2[0-9][0-9]4'
```

Examples

```
%cmdebug > macd test2
```

Delete the macro named `test2`.

```
%cmdebug > macd format@
```

Delete all macros that match the pattern "format@".

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

MACECHO

Controls the "echoing" of each macro command line prior to its execution.

Syntax

```
MACECHO pattern [level]
```

Parameters

pattern The name(s) of the macro(s) for which echoing is to be enabled/disabled.

This parameter can be specified with wildcards or with a full regular expression. Refer to appendix A for additional information about pattern matching and regular expressions.

The following wildcards are supported:

`@`	Matches any character(s).
`?`	Matches any alphabetic character.
`#`	Matches any numeric character.

The following are valid name pattern specifications:

`@`	Matches everything; all names.
`pib@`	Matches all names that start with "pib".
`log2##4`	Matches "log2004", "log2754", and so on.

The following regular expressions are equivalent to the patterns with wildcards that are listed above:

```
'.*'
'pib.*'
'log2[0-9][0-9]4'
```

level Echoing can be enabled or disabled (default). The following values are valid:

MACECHO

`0`	Disabled (default).
`1`	Enabled.

Examples

```
$nmdat > macl @ all
macro driver
   machelp = 'This macro calls macros "triple", "min",  and  "inc" in order' +
             'to demonstrate the MACECHO, MACREF, and MACTRACE commands'
{ loc one 1;
  loc two 2;
  wl min ( triple(two) inc(one) )
}
macro inc
 ( num : ANY  )
   machelp = 'returns the increment of "num"'
{ loc temp num;
  loc temp temp + 1;
  return temp
}
macro min
 ( parm1 : ANY  ,
   parm2 : ANY  )
   machelp = 'returns the min of "parm1" or "parm2"'
{ if parm1 < parm2
  then return parm1
  else return parm2
}
macro triple
 ( input : ANY  )
   machelp = 'triples the parameter "input"'
{ return input *3
}
```

Assume that the macros listed above have been defined. A few of the macros use local variables inefficiently, for the purpose of demonstration.

```
$nmdat > driver
$2
```

When a macro is called, the commands in the macro body are typically executed silently. They are not displayed as they are being executed. In this example, macro `driver` executes silently, and only the expected macro output is displayed.

```
$nmdat > macecho driver 1
$nmdat > driver
  driver > loc one 1
  driver > loc two 2
  driver > wl min ( triple(two) inc(one) )
$2
```

In this example, echoing is enabled for macro `driver`. Then, when the macro is executed, each command line in the macro body is displayed just prior to the execution of that line.

```
$nmdat > macecho min 1
$nmdat > driver
  driver > loc one 1
  driver > loc two 2
  driver > wl min ( triple(two) inc(one) )
     min > if parm1 < parm2 then return parm1 else return parm2
     min > return parm2
$2
```

In this example, echoing is enabled for macro `min`, in addition to macro `driver` which remains enabled from above. Command lines are displayed for both macros. Notice that the command lines for macro `min` are indented, since it is called by macro `driver`. At each nested level of macro invocation, an additional three blanks are added as indentation.

```
$nmdat > macecho @ 1
$nmdat > driver
  driver > loc one 1
  driver > loc two 2
  driver > wl min ( triple(two) inc(one) )
     triple > return input *3
     inc > loc temp num
     inc > loc temp temp + 1
     inc > return temp
     min > if parm1 < parm2 then return parm1 else return parm2
     min > return parm2
$2
```

In this example, echoing is enabled for all ("@") currently defined macros. Each command line, for every macro, is displayed before the command line is executed.

```
$nmdat > macecho @
$nmdat > driver
$2
```

In this example, echoing is disabled for all macros. Since the *level* parameter is not specified, the default of disabled is assumed. Execution of the macro `driver` is silent once again.

MACECHO

```
$nmdat > macecho min 1
$nmdat > driver
     min > if parm1 < parm2 then return parm1 else return parm2
     min > return parm2
$2
$nmdat > macl @ echo
macro min  echo
```

In this example, echoing is enabled for macro `min`. The command lines for macro `min` are displayed, indented. The `MACLIST` command is used to display all macros that currently have `ECHO` enabled, and macro `min` is indicated.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

MACL[IST]

Macro list. Lists the specified macro definition(s).

Syntax

```
MACL[IST] [pattern]  [options]
```

Macros are always listed in alphabetical order.

Parameters

pattern

The name(s) of the macro(s) to be listed.

This parameter can be specified with wildcards or with a full regular expression. Refer to appendix A for additional information about pattern matching and regular expressions.

The following wildcards are supported:

`@`	Matches any character(s).
`?`	Matches any alphabetic character.
`#`	Matches any numeric character.

The following are valid name pattern specifications:

`@`	Matches everything; all names.
`pib@`	Matches all names that start with "pib".
`log2##4`	Matches "log2004", "log2754", and so on.

The following regular expressions are equivalent to the patterns with wildcards that are listed above:

```
'.*'
'pib.*'
'log2[0-9][0-9]4'
```

By default, all macros are listed.

options

Display Options

Special options can be specified to control the level of detail that is presented for each macro definition.

Any number of the following options can be specified in any order, separated by blanks:

`NAME`	Display the macro name, type. (Default value)
`PARMS`	Display parameter names, types, default values.
`NOPARMS`	Skip parameter display.
`BODY`	Display the macro body as a string.
`FMTBODY`	Format the macro body command lines.
`NOBODY`	Skip body display.
`VER`	Display the `MACVER` string.
`NOVER`	Skip version display.
`KEY`	Display the `MACKEY` string.
`NOKEY`	Skip keyword display.
`HELP`	Display the `MACHELP` string.
`NOHELP`	Skip help display.
`ALL | @`	Display all fields. Same as: `NAME PARMS FMTBODY VER KEY HELP`.
`PAGE`	Page eject after each macro definition. Useful for paged (list file) output.
`NOPAGE`	No special page ejects. (Default)

If none of the options above are specified, `NAME` is displayed by default. If any options are specified, they are accumulated to describe which fields are printed.

Filter Options

The following options can be used to further restrict which macro definitions are printed, based on keyword and version matching:

`KEY=`*keyword* Display only those macros that contain the specified *keyword* in their `MACKEY` keyword string.

`VER=`*version* Display only those macros that contain the specified *version* in their `MACVER` version string.

MACL[IST]

The parameters *keyword* and *version* are entered as a single word, or a quoted text string. The interpreter will search for an exact occurrence of the pattern within the specified string. Keyword and version comparisons are case sensitive.

`REF`	Display the macro reference counts.
`ECHO`	Display only macros that have `ECHO` set.
`TRACE`	Display only macros that have `TRACE` set.

These three special filter options are used to display macro reference counts, and to display those macros that have special macro debugging enabled. When any of these three options are specified, only the macro names are displayed (that is, implicit `NOPARMS`, `NOBODY`, `NOHELP`, `NOKEY`, `NOVER`). A special page of examples for these options is provided.

Refer to the `MACECHO`, `MACTRACE`, and `MACREF` commands.

Examples

```
$nmdat > macl
macro cmpin_db                        : PTR/LPTR = $0.0
macro cmport_context                  : PTR/LPTR = $0.0
macro cmport_dst                      : INT/U16 = $0
macro cmport_name                     : INT/U16 = $0
macro cmport_record                   : PTR/LPTR = $0.0
macro config_device_ldev
macro config_device_path
macro config_memory
macro console_ldev
macro convert_string                  : STR/STR =
macro delete_blanks                   : STR/STR =
macro event_ci_history
macro event_footprint
macro event_io_trace
macro event_process
macro event_process_errors
macro file_in_use
macro first_entry                     : PTR/LPTR = $0.0

control-Y encountered
$nmdat >
```

The `MACLIST` command, when entered without parameters, lists all currently defined macros in alphabetically sorted order. By default, only the macro names, and default return value and type (if declared) are displayed.

Note that Control-Y can be used to interrupt any `MACLIST` command.

```
$nmdat > macl fs_disc_alloc parms
macro fs_disc_alloc : PTR/LPTR = $0.0
 ([pin_num    : INT / U16 = $0] ,
   fnum       : INT  ,
  [detail     : INT / U16 = $5] ,
  [error_parm : STR = 'pad'] )
```

Display the `PARMS` (parameters) for macro `fs_disc_alloc_parms`

MACL[IST]

```
$nmdat > macl fs_table all nobody
macro fs_table : UNKN/U16 = $0
 ( entry_ptr  : PTR  ,
   table      : STR  ,
  [detail     : INT / U16 = $1] ,
  [field_name : STR = ] )
   machelp = 'Print the table and optionally returns the field value'
   mackey  = 'MXFS HP Q_FS_X_NM EL FS TABLE PLFD GDPD GUFD LACB PACB MVT' +^S
             'FMAVT AFT FLAB'
   macver  = 'A.00.01'
```

For the macro `fs_table`, display all macro attributes, except for the macro body (`NOBODY`). The macro parameters, help string, keywords string, and version string are displayed.

```
$nmdat > macl @sem@
macro pm_semaphores                  : PTR/LPTR = $0.0
macro rm_build_semaphore_wait_list : STR/STR =
macro rm_sem_blocked_proc          : STR/STR =
macro rm_sem_deadlock              : STR/STR =
macro rm_sem_owner                 : INT/U16 = $0
macro rm_semaphore
macro rm_semaphore_info            : UNKN/U16 = $0
macro xm_semp
```

List all macros that match the pattern "@sem@". By default, only the names of the macros are displayed. Note that default types and return values are displayed for those macros that have specified defaults.

```
$nmdat > macl `.*port_.*`
macro cmport_context     : PTR/LPTR = $0.0
macro cmport_dst         : INT/U16 = $0
macro cmport_name        : INT/U16 = $0
macro cmport_record      : PTR/LPTR = $0.0
macro global_port_name   : STR/STR =
macro io_ioldm_port_fv
macro io_port_data       : UNKN/U16 = $0
macro port_data          : PTR/LPTR = $0.0
macro port_global        : INT/U16 = $0
macro port_message       : PTR/LPTR = $0.0
```

```
macro port_record        : PTR/LPTR = $0.0
macro ui_job_port_msg    : UNKN/U16 = $0
macro ui_jsmain_port_msg : UNKN/U16 = $0
```

List all macros that match the regular expression pattern ".*port_.*". By default, only the macro names (and default return values/types) are displayed.

```
$nmdat > macl @timer@ help
macro format_timer_msg
   machelp = 'Formats the timer request list entrys message.'

macro io_timer_list
   machelp = 'Formats the timer request list.'

macro start_timer
   machelp = 'Sets variable cpustart to current value of HPCPUSECS CI' +
             'variable.'

macro stop_timer
   machelp = 'Sets variable cputime to current value of HPCPUSECS CI' +
             'variable - variable cpustart.'

macro timer
   machelp = 'Times events and then prints elapsed cpu time.'
```

List all macros that match the pattern "@timer@", and display the MACHELP string for each macro.

```
$nmdat > macl @ key=CHAIN
macro io_data_chain         : UNKN/U16 = $0
macro io_getnext_data_chain : PTR/LPTR = $0.0
```

List all macros, but only if the pattern CHAIN can be located within the macro's keyword string, defined with the MACKEY option. By default, only the names of the macros are displayed.

```
$nmdat > macl @ key=CHAIN help
macro io_data_chain         : UNKN/U16 = $0
   machelp = 'Print or returns the specified field form the data chain' +
             'record.'

macro io_getnext_data_chain : PTR/LPTR = $0.0
   machelp = 'Returns the address of the next data chain entry '+
             'associated with the specified I/O request'
```

List all macros, but only if the keyword CHAIN can be located within the macro's keyword string, defined with the MACKEY option. Display the macro name and the MACHELP string for those macros.

```
$nmdat > macl @ key=GUFD key
macro fs_addr          : PTR/LPTR = $0.0
   mackey  = 'MXFS HP Q_FS_X_NM EL FS FILENAME FILE ADDRESS GUFD'

macro fs_fname_nm      : STR/STR =
   mackey  = 'MXFS HP Q_FS_X_NM EL FS FNAME GUFD'

macro fs_fname_to_gufd : PTR/LPTR = $0.0
   mackey  = 'MXFS HP Q_FS_X_NM EL FS GUFD GLOBAL UNIQUE FILE DESCRIPTOR'

macro fs_gufd          : PTR/LPTR = $0.0
   mackey  = 'MXFS HP Q_FS_X_NM EL FS GUFD PLFD'

macro fs_table         : UNKN/U16 = $0
   mackey  = 'MXFS HP Q_FS_X_NM EL FS PLFD GDPD GUFD LACB PACB MVT' +
             'FMAVT AFT FLAB'

macro fs_ufid_str      : STR/STR =
   mackey  = 'MXFS HP Q_FS_X_NM EL FS GUFD UFID STR'

macro fs_ufid_to_gufd  : PTR/LPTR = $0.0
   mackey  = 'MXFS HP Q_FS_X_NM EL FS UFID TO GUFD'
```

List all macros, but only those that contain the keyword `GUFD` within the macro's keyword string, defined with the `MACKEY` option. List the names and the keyword string for those macros.

```
$nmdat > macl fs_fname_to_gufd all
macro fs_fname_to_gufd : PTR/LPTR = $0.0
 ( filename : STR  )
   machelp = 'Returns the address of the GUFD for the specified filename'
   mackey  = 'MXFS HP Q_FS_X_NM EL FS GUFD GLOBAL UNIQUE FILE DESCRIPTOR FILE'
   macver  = 'A.00.01'
{ loc save_error_action error_action;
  loc vsod_hdr        = kso_pointer (kso_number ('kso_vs_od_gu_fd_header'));
  loc entry_size      = symval (vsod_hdr, 'tbl_hdr.' +  'hdr_entry_size');
  loc vsod_rec_size   = symlen ('!vs_som:vs_od_type');
  ignore quiet;
```

```
  loc first_entry_ptr = first_entry (vsod_hdr);
  if error <> 0
  then return NMNIL;
  loc max_entry_ptr = first_entry_ptr +  symval (vsod_hdr, 'tbl_hdr.' + 'hdr_rs^
rc_block.body_current_size') -  vsod_rec_size;
  loc filename = strup(filename);
  loc vsod_ptr = first_entry_ptr;
  var error_action = 'pa';
  while vsod_ptr < max_entry_ptr do
    { loc gufd_ptr = vsod_ptr + vsod_rec_size;
      loc fname = fs_fname_nm (gufd_ptr);
      if fname = filename
      then { var error_action = save_error_action;
             return gufd_ptr
           };
      loc vsod_ptr = vsod_ptr + entry_size
    };
  var error_action = save_error_action;
  stderr (HP_FILENAME_NOT_FOUND, 'fs_fname_to_gufd', filename);
  return NMNIL
}
```

Display macro `fs_fname_to_gufd`. Since the `ALL` option is specified, all macros attributes are displayed, including the name, parameters, help, version, and the full formatted body.

This is a typical macro from the DAT Macros package.

Examples of the ECHO, REF, and TRACE options

```
$nmdat > macl format@ ref
macro format            ref = 0
macro format_job        ref = 1
macro format_raw_table  ref = 0
macro format_timer      ref = 3
```

Display the `REF` (reference counts) for all macros that match the pattern "format@". Macro `format_job` has been called one time, and macro `format_timer` has been called three times.

```
$nmdat > macl @ trace
```

```
macro get_disp_wait_event  trace = 3
macro get_element          trace = 1
macro get_entry_ptr        trace = 3
macro get_sublist          trace = 3
macro get_table_info       trace = 3
macro kso_number           trace = 1
macro kso_pointer          trace = 2
```

List all macros for which the `MACTRACE` command has been used to enable tracing of the macro execution. The trace level number is displayed.

```
$nmdat > maclist @ echo
macro kso_number   echo
macro kso_pointer  echo
macro port_data    echo
```

List all macros for which the `MACECHO` command has been used to enable the echoing of each macro command line during macro execution.

```
$nmdat > macl @ trace echo all
macro kso_number   echo  trace = 1
macro kso_pointer  echo  trace = 2
```

List all macros that have tracing and echoing enabled. Note that only the macro names, and the echo and trace information is displayed, even though the `ALL` option was requested.

The keywords `ECHO`, `REF`, and `TRACE` restrict the output display to macro names and the selected option(s). Parameters, keywords, help strings, versions, and macro bodies are not listed when any one of these three options are specified on the `MACLIST` command.

Listing Macros to a File

The following example demonstrates how to produce a paged listing of all currently defined macros, formatted to a file, one macro per page. The example is explained command by command, based on the command numbers that appear within the prompt lines.

```
%10 (%53) cmdat > list macros
%11 (%53) cmdat > env term_loud false
%12 (%53) cmdat > maclist @ all page
%13 (%53) cmdat > list close
%14 (%53) cmdat > set def
```

- Command `%10` opens an offline list file, named `MACROS`. All System Debug input and output is recorded into this file, including the code we intend to display.
- Command `%11` sets the environment variable `term_loud` to FALSE. This prevents subsequent System Debug output from being displayed on the terminal. We capture the output in the list file (`macros`), but we do not want to watch all of the output on the terminal.
- Command `%12` contains the `MACLIST` command. All attributes of all currently defined macros are displayed. The `PAGE` option causes each macro to start on a new page. The list file contains CCTL (carriage control) information for the paging.
- Command `%13` closes (and saves) the current list file (`macros`).
- Command `%14` uses the `SET DEFAULT` command to effectively reset the environment variable `term_loud` back to TRUE. System Debug output is once again displayed on the terminal.

Limitations, Restrictions

Macros listed into a file are not currently formatted in a style that allows the macro to be redefined by reading the file back in as a USE file.

The macro pretty printer attempts to format the macro body in a reasonable manner. Occasionally, the formatting includes extra blank lines, usually as a result of unnecessary semicolons within the original macro body.

MACL[IST]

When macros are defined, all comments are removed, and the macro body is stored in compressed form. The `MACLIST` command does not display the original form of the macro body.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

MACREF

Resets the reference count to zero for the specified macro(s).

Syntax

```
MACREF pattern
```

Reference counts are maintained for macros. Each time a macro is invoked, the reference count for the macro is incremental.

Current reference counts can be displayed with the `MACLIST` command.

This `MACREF` command is used to reset macro reference counts.

Parameters

pattern The name(s) of the macro(s) for which the reference counts are to be reset to zero.

This parameter can be specified with wildcards or with a full regular expression. Refer to appendix A for additional information about pattern matching and regular expressions.

The following wildcards are supported:

`@`	Matches any character(s).
`?`	Matches any alphabetic character.
`#`	Matches any numeric character.

The following are valid name pattern specifications:

`@`	Matches everything; all names.
`pib@`	Matches all names that start with "pib".
`log2##4`	Matches "log2004", "log2754", and so on.

The following regular expressions are equivalent to the patterns with wildcards that are listed above:

```
'.*'
'pib.*'
'log2[0-9][0-9]4'
```

Examples

```
$nmdat > macl @ all
macro driver
   machelp = 'This macro calls macros "triple", "min",  and  "inc" in order' +
             'to demonstrate the MACECHO, MACREF, and MACTRACE commands'
{ loc one 1;
  loc two 2;
  wl min ( triple(two) inc(one) )
}
macro inc
 ( num : ANY  )
   machelp = 'returns the increment of "num"'
{ loc temp num;
  loc temp temp + 1;
  return temp
}
macro min
 ( parm1 : ANY  ,
   parm2 : ANY  )
   machelp = 'returns the min of "parm1" or "parm2"'
{ if parm1 < parm2
  then return parm1
  else return parm2
}
macro triple
 ( input : ANY  )
   machelp = 'triples the parameter "input"'
{ return input *3
}
```

Assume that the macros listed above have been defined. A few of the macros use local variables inefficiently, for the purpose of demonstration.

```
$nmdat > macl @ ref
macro driver  ref = #0
macro inc     ref = #0
macro min     ref = #0
macro triple  ref = #0
```

MACREF

The MACLIST command is used to display the current reference counts for all macros. At this point, the reference counts for all macros are zero.

```
$nmdat > wl inc(4)
$5
$nmdat > wl min(inc(3) inc(0))
$1
$nmdat > macl @ ref
macro driver  ref = #0
macro inc     ref = #3
macro min     ref = #1
macro triple  ref = #0
```

A few macros are invoked, then the MACLIST command is used again to display the current reference counts. Macro inc has been called three times, and macro min has been called one time.

```
$nmdat > macref inc
$nmdat > macl @ ref
macro driver  ref = #0
macro inc     ref = #0
macro min     ref = #1
macro triple  ref = #0
```

The MACREF command is used to reset the reference count for macro inc. The MACLIST command is used to verify that the count has been successfully reset.

```
$nmdat > driver
$2
$nmdat > macl @ ref
macro driver  ref = #1
macro inc     ref = #1
macro min     ref = #2
macro triple  ref = #1
```

Macro driver is invoked, then the reference counts are checked again.

```
$nmdat > macref @
$nmdat > macl @ ref
macro driver  ref = #0
macro inc     ref = #0
```


FINAL TRIM SIZE : 7.5 in x 9.0 in

```
macro min     ref = #0
macro triple  ref = #0
```

The reference counts for *all* macros are reset to zero.

Limitations, Restrictions

The macro reference count is incremental at macro entry, after parameter type checking, but before actual execution of the macro body. The actual macro execution may result in errors and be terminated. Reference counts, therefore, indicate the number of times the macro has been called (not the number of times that the macro has been successfully executed to completion).

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

MACTRACE

Controls the "tracing" of macro execution.

Syntax

```
MACTRACE pattern [level]
```

It is possible to enable/disable the observation of entry/exit of macros, along with input parameter values and functional return values.

Parameters

pattern The name(s) of the macro(s) that are to be traced.

This parameter can be specified with wildcards or with a full regular expression. Refer to appendix A for additional information about pattern matching and regular expressions.

The following wildcards are supported:

`@`	Matches any character(s).
`?`	Matches any alphabetic character.
`#`	Matches any numeric character.

The following are valid name pattern specifications:

`@`	Matches everything; all names.
`pib@`	Matches all names that start with "pib".
`log2##4`	Matches "log2004", "log2754", and so on.

The following regular expressions are equivalent to the patterns with wildcards that are listed above:

```
‘.*‘
‘pib.*‘
‘log2[0-9][0-9]4‘
```

level The level of macro "tracing" detail.

Four increasing levels are supported:

1	All tracing is disabled. (Default)
2	Macro entry is displayed.
3	Macro entry and exit are displayed.
4	Macro entry, input parameter values, macro exit, and functional return values are displayed.

Examples

```
$nmdat > macl @ all
macro driver
   machelp = 'This macro calls macros "triple", "min",  and  "inc" in order' +
             'to demonstrate the MACECHO, MACREF, and MACTRACE commands'
{ loc one 1;
  loc two 2;
  wl min ( triple(two) inc(one) )
}
macro inc
 ( num : ANY  )
   machelp = 'returns the increment of "num"'
{ loc temp num;
  loc temp temp + 1;
  return temp
}
macro min
 ( parm1 : ANY  ,
   parm2 : ANY  )
   machelp = 'returns the min of "parm1" or "parm2"'
{ if parm1 < parm2
  then return parm1
  else return parm2
}
macro triple
 ( input : ANY  )
   machelp = 'triples the parameter "input"'
{ return input *3
```

}

Assume that the macros listed above have been defined. A few of the macros use local variables inefficiently, for the purpose of demonstration.

```
$nmdat > driver
$2
```

Macros normally execute silently, as they invoke commands, and often other macros. In this example, macro `driver` is invoked, and this macro calls several other macros. Since macro tracing is not enabled for any of these macros, execution proceeds silently.

```
$nmdat > mactrace inc 3
$nmdat > driver
--> enter macro: inc
--> parms macro: inc
 ( num : ANY = $1  )
<-- exit  macro: inc  : U16 = $2
$2
```

The `MACTRACE` command is used to enable macro tracing for macro `inc` at trace level 3. Now, every time macro `inc` is invoked, trace information is displayed. Since the trace level for this macro is set to level 3, entry into the macro is displayed, along with the parameter value(s) at entry, and exit from the macro is displayed, along with the function return value.

```
$nmdat > macl @ trace
macro inc  trace = 3
```

The `MACLIST` command is used to display all macros that have tracing enabled (level >= 1). Macro `inc` is shown to have tracing enabled at level 3.

```
$nmdat > mactrace @ 3
$nmdat > driver
--> enter macro: driver
--> enter macro: min
--> enter macro: triple
--> parms macro: triple
 ( input : ANY = $2  )
<-- exit  macro: triple  : U16 = $6
--> enter macro: inc
```

```
--> parms macro: inc
 ( num : ANY = $1  )
<-- exit  macro: inc  : U16 = $2
--> parms macro: min
 ( parm1 : ANY = $6  ,
   parm2 : ANY = $2  )
<-- exit  macro: min  : U16 = $2
$2
<-- exit  macro: driver
```

In this example, macro tracing is set to level 3 for all macros.

```
$nmdat > mactrace @
```

Tracing is disabled for all macros.

Limitations, Restrictions

none

Caution	The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

MAP

Opens a file and maps it into a usable virtual address space.

Syntax

```
MAP filename [option]
```

The `MAP` command allows a file to be accessed (displayed or modified) in virtual space by other System Debug commands. This command is useful for analyzing dump files generated by subsystems that are not part of the dump created by the DUMP utility.

Parameters

filename The file name of the file to map into usable address space.

option Read or read/write access can be explicitly requested, a filecode can be specified, and a virtual offset set be specified. Multiple options can be specified for a single `MAP` command.

`READACCESS`	Open the file for read access only (default). Users with PM capability can still write to the file (file system feature).
`WRITEACCESS`	Open the file for read/write access. Standard file system security checking is performed while opening the file.
`FILECODE` *value*	Privileged files cannot be accessed without providing the numeric file code associated with the file. This keyword/value pair allows privileged users to map in these privileged files.

	Remember that file codes are thought of as negative decimal numbers.
`OFFSET` *value*	Map the file, in starting at the specified virtual byte offset. The default offset is 0.

Examples

```
$nmdebug > map DTCDUMP
1  DTCDUMP.DUMPUSER.SUPPORT     1000.0  Bytes = 43dc
```

Open the file DTCDUMP and assign it to the virtual object in space $1000. It is mapped to file index number 1. Use this number to UNMAP the file.

```
$nmdebug > map DATA2 off c0004c00
2  DATA2.DUMPUSER.SUPPORT     1000.1c004c00. Bytes = 2340
```

Map the file DATA2 at a specified virtual offset of $c0004c00.

Related commands: MAPLIST, UNMAP.

Related functions: MAPINDEX, MAPVA, MAPSIZE.

Limitations, Restrictions

A maximum of ten files can be mapped in at any one time.

It is not currently possible to map a file if it is already open and loaded for execution. Refer to the HPFOPEN intrinsic description in the *MPE XL Intrinsic Reference Manual* (32650-90028) for additional details.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

MAPL[IST]

Lists the specified file(s) that have been opened with the `MAP` command.

Syntax

```
MAPL[IST] [pattern]
```

Parameters

pattern The file name(s) of the mapped files to be listed.

If no file name is given, all currently mapped files are displayed.

This parameter can be specified with wildcards or with a full regular expression. Refer to appendix A for additional information about pattern matching and regular expressions.

The following wildcards are supported:

`@`	Matches any character(s).
`?`	Matches any alphabetic character.
`#`	Matches any numeric character.

The following are valid name pattern specifications:

`@`	Matches everything; all names.
`pib@`	Matches all names that start with "pib".
`log2##4`	Matches "log2004", "log2754", and so on.

The following regular expressions are equivalent to the patterns with wildcards that are listed above:

```
`.*`
`pib.*`
`log2[0-9][0-9]4`
```

Examples

```
$nmdebug > maplist
1  DTCDUMP.DUMPUSER.SUPPORT     1000.0  Bytes = 43dc
2  DTCDUMP2.DUMPUSER.SUPPORT    1001.0  Bytes = c84
3  MYFILE.MYGROUP.MYACCT        1005.0  Bytes = 1004

$nmdebug > mapl myfile
3  MYFILE.MYGROUP.MYACCT        1005.0  Bytes = 1004
```

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

MODD

DAT ONLY

Modification delete. Deletes a modification entry specified by index number.

Syntax

$$\texttt{MODD} \begin{bmatrix} index \\ \texttt{@} \end{bmatrix}$$

The MODD command is used to delete a modification which has been applied to an opened dump.

Parameters

index The index number of the modification entry which is to be deleted.

@ @, the wildcard character, can be used to delete all currently defined entries.

Examples

```
$nmdat > modl
Current TEMPORARY dump modification(s):
1)  VIRT  $b.80b4f300
    REAL  $a80300          $70ff4e74  "p.Nt"     (orig: $8119e000    "....")
2)  REAL  $1d654           $ffffffff  "...."     (orig: $0           "....")
3)  SEC   $1.a552000       $20c0104   "...."     (orig: $20b0104     "....")
$nmdat > modd 1
$nmdat > modl
Current TEMPORARY dump modification(s):
2)  REAL  $1d654           $ffffffff  "...."     (orig: $0           "....")
3)  SEC   $1.a552000       $20c0104   "...."     (orig: $20b0104     "...."
```

Deletes the temporary dump modification entry at index number 1.

Caution The output format of all DAT/DEBUG commands is subject to change without notice. Programs which are developed to postprocess DAT/DEBUG output should not depend on the exact format (spacing, alignment, number of lines, upper or lower case, or spelling) of any DAT/DEBUG command output.

MODL

DAT ONLY

Modification list. Lists current dump modifications.

Syntax

```
MODL [ index ]
     [ @     ]
```

The `MODL` command is used to list all current modifications which have been applied to an opened dump.

Parameters

index The index number of the modification entry to display.

@ The wildcard symbol "@" can be used to display all entries.

If no parameter is entered, the default is that all entries are displayed.

Examples

In the following examples, three different types of dump modifications are applied and then all three modifications are listed.

```
$nmdebug > bl

$nmdat > mv 80b4f300
VIRT $b.80b4f300 = "...."  $8119e000 := 70ff4e74
Added TEMPORARY dump modification.  Use MODL to list, MODD to delete.
1)  VIRT $b.80b4f300
    REAL $a80300             $70ff4e74 "p.Nt"   (orig: $8119e00 "....")
```

```
 $nmdat > mz 1d654
 REAL $0001d654 = "...."  $0 := -1
 Added TEMPORARY dump modification.  Use MODL to list, MODD to delete.
 2)  REAL $1d654              $ffffffff "...."   (orig: $0       "....")

$nmdat > msec vtos(a.0)
SEC $1.a552000   = "...."  $20b0104 := 20c0104
Added TEMPORARY dump modification.  Use MODL to list, MODD to delete.
3)  SEC  $1.a552000          $20c0104  "...."   (orig: $20b0104  "....")

$nmdat > modl
Current TEMPORARY dump modification(s):
1)  VIRT  $b.80b4f300
    REAL  $a80300            $70ff4e74 "p.Nt"   (orig: $8119e000 "....")
2)  REAL  $1d654             $ffffffff "...."   (orig: $0        "....")
3)  SEC   $1.a552000         $20c0104  "...."   (orig: $20b0104  "....")
```

Caution The output format of all DAT/DEBUG commands is subject to change without notice. Programs which are developed to postprocess DAT/DEBUG output should not depend on the exact format (spacing, alignment, number of lines, upper or lower case, or spelling) of any DAT/DEBUG command output.

Limitations, Restrictions

none

MPEXL

PRIVILEGED MODE

Displays information about the files which were used to build the operating system SOM portion of the NL.PUB.SYS for MPE XL.

Syntax

```
MPEXL [fileset] [optionstring] [outputfile]
```

During the build and integration process for MPE XL, valuable submittal build line version numbers, date and time, submittal number, source control version numbers, and user-defined keywords are maintained for every single file which is used to construct the MPE XL operating system SOM portion of NL.PUB.SYS. This information is compressed into a compact table and inserted into the library when it is linked.

The `MPEXL` command can be used to extract this stored information and display the data in a readable format. Since this command is supported in all debugger products, it is possible to inspect the current system (DEBUG), a dumped system (DAT), and a dead system (SAT).

By default, the information line for a specified file is displayed as follows:

```
DHEADNM.HPESTD  A.35.26  19890126:1325  M03678 R4.2.1.4 'integration NewHasH'
```

The names and formats of each file are as follows:

```
File     Group      VUF         Date        MSN         Rev           User
FFFFFFFF.GGGGGGGG  V.UU.FF  YYYYMMDD:HHMM  Mnnnnn  R[n[.n[.n]]]  'user keywords'
```

All field lengths and formats are fixed, except for the source control `Rev` number and any `User` keywords which may be defined within the file. These two fields are of variable length.

By default, an information line for each file is displayed on the terminal. Special options are supported which allow the user to enable or disable the display of individual fields within the line. Additional options can be used as filters. For example, you can request to see only those files which are integrated into a particular build line, which were submitted after a certain date, and which contain a specified user keyword.

Output can be directed to a file instead of to the terminal. You can use the output files as input data for other existing tools, such as **awk, sort, grep,** MPE SORT. Use the backslash delimiter (\) as a field separator, as shown in the following example:

```
TDEBUG \CMDEBUG \A.35.21\19890202:0925\M03724\R4.24\'DaT BuG SaT'\
```

Parameters

fileset Specifies the set of files to be listed. The default, @.@, produces a listing of all files used to build the operating system SOM portion of NL.PUB.SYS. You can select the file(s) to be listed by using the fully or partially qualified form for fileset:

```
filename[.groupname]
```

You can use wildcards. Each of the wildcard characters counts toward the eight-character limit for the file and group names.

- # Use the wildcard symbol "#" to specify a single numeric character.
- ? Use the wildcard symbol "?" to specify a single alphanumeric character.
- @ Use the wildcard symbol "@" to specify zero or more alphanumeric characters. By itself, "@" represents all the members of a set.

optionstring You can specify a string of options. Two different types of options (*field display* and *field filter*) are supported, and you can specify either or both in a single option string. You can enter the option string as a string literal (surrounded by single or double quotes), or as a string variable or expression. The following is an example of an *optionstring*:

```
"+FGVMR m>1246 m<=4672 u='NewHash'"
```

Field Display Options: The display of each individual field can be selectively enabled or disabled. By default, all fields are displayed; however, on the terminal, some data may not show up on the screen. Terminal lines are automatically truncated to the current value of the environment variable *ENV TERM_WIDTH*, which defaults to 79 characters.

Fields are specified by the first letter of the field name:

F	`File`	file name
G	`Group`	group name
V	`VUFG`	V.UU.FF of submittal build line
D	`Date`	date when last submitted
M	`MSN`	master submittal number
R	`Rev`	revision number (source control)
U	`User`	user-defined keywords within the file

Two different forms of field display selection are supported, but only one form can be used at a time within the option string; they cannot be mixed. The two different forms are:

−	the minus sign, followed immediately by any number of field name letters, will disable the display of the specified fields. The letters are conceptually subtracted from the default set which displays all fields.
+	the plus sign, followed immediately by any number of field name letters, will enable the display of the specified fields. The fields are conceptually added to the empty set of fields. Therefore, only the specified fields following the plus signs will be displayed.

For example, the two following option strings are valid and equivalent:

−DU	disables `Date` and `User`.
+FGVMR	enables `File`, `Group`, `Ver`, `Msn`, and `Rev`.

Field Filter Options: The *fileset* parameter supports filtering based on file name. The *field filter* options allow additional filtering, based on the values of individual fields.

The form of a filter option is:

```
<Field-Letter><Relational-Operator><Value>
```

The field letter must be followed immediately by the relational operator, which must be followed immediately by the value. The field value must be entered in the supported format, which is the same as the way it is displayed.

The following table lists the field abbreviation letter, supported relational operators for that field, and the supported field value formats:

Field	Relational Op.						Value (Supported Format)
V	<=	<	=	>	>=	<>	V.UU.FF
D	<=	<	=	>	>=	<>	YYYYMMDD[:HHMM]
M	<=	<	=	>	>=	<>	[M]nnnnn
R	<=	<	=	>	>=	<>	[R][n[.n[.n[n]]]
U[i]			=			<>	word
U[i]			=			<>	"string of chars"
Where: i = Ignore Case							

The following is an example of a valid *field filter* option:

```
"D>19880623"
```

Ranges of values can be defined by using two separate conditions for the same field. For example, the following example limits the MSN value to the range $100 \leq M \leq 1999$. The two separate conditions are ANDed together.

```
"m>=1000 M<=1999"
```

Filtering based on user-defined keywords is slightly different. The equal sign "=" is used to match a user keyword value. The value can be an unquoted single word (terminated by the next blank or comma), or a quoted string of characters. In both cases, if the specified keyword pattern is found anywhere within the keyword string(s) defined with the file, then comparison succeeds.

By default, keyword matching is case sensitive, unless the `Ui` (ignore case) form is explicitly used.

When more than one user keyword filter option is specified, they are effectively ORed together.

The following is a valid option string which contains multiple user keywords.

```
"D<19890100 u=VSM ui=NewHash ui='new hash'"
```

This would match `VSM`, `newhash`, `New Hashing`, or `NEW Hashed`; it would not match `vsm` or `NEW FMT`.

outputfile When the optional *outputfile* filename is specified, information lines are written into the file, and not to the terminal.

The output file will be built with fixed-length, ASCII records (NOCCTL). The file record length is automatically set to a length larger than the longest line. This guarantees that no data will be lost.

If the file already exists, an error message is printed. The file must be explicitly deleted by the user using `PURGE`.

The product SAT does not support this parameter since it is not possible to create files in SAT.

Examples

```
$nmdebug  > mpex 1
DPDA    .ACTIVITY  0.00.00  19870817:1917  M00000 R ''
EACTIV  .ACTIVITY  0.00.00  19870817:1917  M00000 R ''
OACTIV  .ACTIVITY  1.34.12  19881014:1500  M02933 R ''
PACTIV  .ACTIVITY  0.00.00  19870817:1917  M00000 R ''
DAFIFUNC.AFIDAM    A.41.00  19890317:1912  M04190 R ''
DDAMMSGS.AFIDAM    1.34.04  19880831:1308  M02464 R ''
OAFIMGR .AFIDAM    A.41.00  19890317:1912  M04190 R ''
TDAMTABL.AFIDAM    A.41.00  19890317:1912  M04190 R ''
XABORTPF.AFIDAM    1.34.04  19880831:1308  M02464 R ''
XAFIMGR .AFIDAM    1.34.04  19890317:1912  M02855 R ''
XAFITERM.AFIDAM    1.34.04  19880831:1308  M02464 R ''
XCONXFIG.AFIDAM    1.34.04  19880831:1308  M02464 R ''
XDAMDATE.AFIDAM    A.41.00  19890317:1912  M04190 R ''
XDIAGLOG.AFIDAM    1.34.04  19880831:1308  M02464 R ''
XDMAREQS.AFIDAM    A.41.00  19890317:1912  M04190 R ''
XEVENT  .AFIDAM    1.34.04  19880831:1308  M02464 R ''
XLKDIAGS.AFIDAM    1.34.04  19880831:1308  M02464 R ''
XMSGRJCT.AFIDAM    1.34.04  19880831:1308  M02464 R ''
DAHLDATA.AHOSTDM   0.00.00  19871016:2141  M00000 R ''
DALHERRS.AHOSTDM   A.02.00  19880126:1605  M01269 R ''
OAHOSTDM.AHOSTDM   A.02.00  19880126:1605  M01269 R ''
TALHMSGS.AHOSTDM   0.00.00  19871016:2141  M00000 R ''
TALHPORT.AHOSTDM   A.02.00  19800126:1605  M01269 R ''
TALHPROC.AHOSTDM   0.00.00  19871016:2141  M00000 R ''
TALHREQ .AHOSTDM   0.00.00  19871016:2141  M00000 R ''
TINTRO  .AHOSTDM   X.3B.02  19871211:2058  M00773 R ''
XABEVENT.AHOSTDM   X.3B.02  19871211:2058  M00773 R ''
XABORT  .AHOSTDM   X.3B.02  19871211:2058  M00773 R ''
XALHMAIN.AHOSTDM   0.00.00  19871016:2141  M00000 R ''
XBADMSG .AHOSTDM   0.00.00  19871016:2141  M00000 R ''
control-y encountered
```

By default, information is displayed for all files (@.@). Use CTRL Y to interrupt the display to the terminal.

In the following example, information for all files (@.@) is written into the file "versions". This file includes the backslash delimiter between each field and is suitable as input for AWK:

```
$nmdebug > mpexl @.@,,versions
```

In the following example, information will be displayed for all files which begin with the letter T in the group AHOSTDM.

```
$nmdebug . mpexl t@.ahostdm
TALHMSGS.AHOSTDM   0.00.00  19871016:2141  M00000 R ''
TALHPORT.AHOSTDM   A.02.00  10990126:1605  M01269 R ''
TALHPROC.AHOSTDM   0.00.00  19871016:2141  M00000 R ''
TALHREQ .AHOSTDM   0.00.00  19871016:2141  M00000 R ''
TINTRO  .AHOSTDM   X.3B.02  19871211:2058  M00773 R ''
```

MPEXL

In the following example, information is displayed for all X-files in the group AFIDAM. The option string disables the display of the Version, Rev, and User keyword fields. The remaining enabled fields (File, Group, Date, and MSN) are displayed.

```
$nmdebug > mpexl x@.afidam '-vru'
XABORTPF.AFIDAM   19880831:1308  M02464
XAFIMGR .AFIDAM   19890317:1912  M02855
XAFITERM.AFIDAM   19880831:1308  M02464
XCONXFIG.AFIDAM   19880831:1308  M02464
XDAMDATE.AFIDAM   19890317:1912  M04190
XDIAGLOG.AFIDAM   19880831:1308  M02464
XDMAREQS.AFIDAM   19890317:1912  M04190
XEVENT  .AFIDAM   19880831:1308  M02464
XLKDIAGS.AFIDAM   19880831:1308  M02464
XMSGRJCT.AFIDAM   19880831:1308  M02464
```

Three examples follow. The first two examples display all of the files whose Master Submittal Number equals M4129 *or* M4295. The third example does not display anything, since MSN filters are ANDed, and no file can have both MSN 4129 *and* MSN 4295.

```
$nmdebug > mpexl ,'m=4129';
EFEQINFO.FSMPEII   A.41.00  19890323:1640 M04129 R ''
XFEQINFO.FSMPEII   A.41.00  19890323:1640 M04129 R ''

$nmdebug > mpexl @,'m=4295'
TCIOPORT.CIOCAM   A.41.02   19890331:1650 M04295 R ''
XRELEASE.CIOCAM   A.41.02   19890331:1650 M04295 R ''
XTIMER  .CIOCAM   A.41.02   19890331:1650 M04295 R ''

$nmdebug > mpexl @,'m=4129 m=4295'
```

The following example displays all files in the group NLIO whose master submittal numbers fall in the range 4320 to 4330, inclusive (4320 $\leq$ M $\leq$ 4330):

```
$nmdebug > mepxl @.nlio 'm>=4320 m<=4330'
OUTIL   .NLIO   A.42.02  19890404:1833 M04322 R ''
```

The following displays all X-files which were submitted on March 3. 1989.

```
$nmdebug > mpexl x@,'d=19890331'
XRELEASE.CIOCAM   A.41.02  19890331:1650 M04295 R ''
XTIMER  .CIOCAM   A.41.02  19890331:1650 M04295 R ''
```

MPEXL

The following displays the filename, gorup name, build version and submittal date for all files which were submitted after April 1, 1989.

```
$nmdebug > mpexl ,"+fgvd d>=19890401"
AVER    .ASMVER  A.41.10  19890502:1906
HGRPTBL .ASMVER  A.41.00  19890404:1922
HVER    .ASMVER  A.41.00  19890404:1922
HVERFILE.ASMVER  A.41.00  19890404:2041
DBUILDID.MAIN    A.41.10  19890428:1238
HBUILDID.MAIN    A.41.10  19890428:1238
OUTIL   .NLIO    A.41.02  19890404:1833
IODFAULT.SYSG    A.41.02  19890404:1812
```

The following displays all files in the group VSM which were accepted into submittal build line A.41.00.

```
$nmdebug > mpexl @.vsm 'v=A.41.00'
FVSM    .VSM      A.41.00  19890317:1813 M04018  1.4 ''
XBTINS  .VSM      A.41.00  19890317:1808 M04017 R1.4 ''
XBTSCH  .VSM      A.41.00  19890317:1813 M04018 R1.4 ''
XVDEBUG .VSM      A.41.00  19890317:1813 M04018 R1.4 ''
XVUTIL  .VSM      A.41.00  19890317:1921 M04187 R1.1.1.1 ''
```

The following displays all files which meet all the following critera:

- they are in the group VSM
- they were accepted into submittal build line A.41.00
- their source control revision is not equal to 1.4.

```
$nmdebug >  mpexl @.vsm 'v=A.41.00 r<>1.4'
XVUTIL  .VSM      A.41.00  19890317:1921 M04187 R1.1.1.1 ''
```

The following displays the filename, date, and buld line version for all X-files which were acepted into build A.41.00 or later, submitted after March 1, 1989.

```
$nmdebug > mpexl x@.vsm '+fdv d>19890301 v>=A.41.00'
XBTINS     A.41.00  19890317:1808
XBTSCH     A.41.00  19890317:1813
XVDEBUG    A.41.00  19890317:1813
XVUTIL     A.41.00  19890317:1921
```

MPEXL

The following displays all files which were accepted into submittal build line A.42.01 after the date March 28, 1989.

```
$nmdebug > mpexl ,'v=a.41.02 d>19890328'
TCIOPORT.CIOCAM   A.41.02   19890331:1650 M04295 R ''
XRELEASE.CIOCAM   A.41.02   19890331:1650 M04295 R 'MP'
XTIMER  .CIOCAM   A.41.02   19890331:1650 M04295 R 'MP'
OUTIL   .NLIO     A.41.02   19890404:1833 M04322 R ''
OPMEACC .SAT      A.41.02   19890331:1742 M04306 R5.3 'MP SaT'
IODFAULT.SYSG     A.41.02   19890404:1812 M04314 R ''
OSYS1   .SYSTEM   A.41.02   19890331:1658 M04289 R ''
TSYS    .SYSTEM   A.41.02   19890331:1658 M04289 R ''
```

The following displays all files which were accepted into submittal build line A.42.01 after the date March 28, 1989. In addition, the files must contain either the user keyword "sat" (ignoring case) *or* the user keyword "MP" (case sensitive).

```
$nmdebug > mpexl ,"v=a.41.02 d>19890328 ui='sat' u=MP"
XRELEASE.CIOCAM   A.41.02   19890331:1650 M04295 R 'MP'
XTIMER  .CIOCAM   A.41.02   19890331:1650 M04295 R 'MP'
OPMEACC .SAT      A.41.02   19890331:1742 M04306 R5.3 'MP SaT'
```

The following displays all files in the group SAT whose source control revision number is greater than 5.1.

```
$nmdebug > mpexl @.sat 'r>5.1'
ONMMPEXL.SAT    A.41.00  19890327:1500 M04260 R5.2 'BuG DaT SaT'
OPMEACC .SAT    A.41.02  19890331:1742 M04306 R5.3 'SaT'
```

The following lists all C-files which were accepted after March 1, 1989, and which contain the user keyword "SaT" (case sensitive).

```
$nmdebug > mpexl c@ 'd>19890301 u=SaT'
CCMUTLC  .CCDAT    A.41.00  19890323:1519 M04043 R5.1 'BuG DaT SaT'
CCMUTLC  .CCSAT    A.41.00  19890323:1522 M04044 R5.1 'BuG DaT SaT'
```

The following displays the file name, group name, revision number, and the user keyword(s) for the file AVER.ASMVER. Note that the closing single quote is not displayed. This indicates that some user keyword information has been truncated.

```
$nmdebug > mpexl aver.asmver '+fgru'
AVER  .ASMVER  R1.1 'BuildID_Built A.41.10 Date_Built TUE MAY 2 1989 7.06 PM
```

Caution The output format of all DAT/DEBUG commands is subject to change without notice. Programs which are developed to postprocess DAT/DEBUG output should not depend on the exact format (spacing, alignment, number of lines, upper or lower case, or spelling) of any DAT/DEBUG command ouput.

Limitations Restrictions

Several (non-restrictive) limits are defined for the field filter options:

- The total number of all field filter options for the fields `Date`, `VUG`, `MSN`, and `Rev` must be eight or less. This limit allows a pair of filters (such as a range) to be defined for each field.
- Impossible non-overlapping ranges can be requested without warning or error. No lines will match if mutually exclusive conditions are defined for the same field. For example, the range "`M<=100 M>=200`" is impossible. You should use two separate command invocations if you want to display all lines that are less than 100 and all lines that are greater than 200.
- The total number of all USER field filter options is limited to eight.
- In field filter options, the DATE values are forced into the format YYYYMMDD[:HHMM], but the individual values are not checked for validity. For example, the filter "D>19890235" is accepted as input, even though the DD portion has an impossible day of the month, 35, and no matches will ever be found.
- In field filter options, the REV and USER values are limited to a maximum of 32 characters.
- The total length of the *optionstring* is limited to 256 characters.

MPSW

Privileged Mode

Modifies the NM processor status word (PSW).

Exercise a bit of care with this command.

Syntax

```
MPSW bit_string
```

Parameters

bit_string A string of characters that indicates which bits in the PSW are to be modified. The letters listed below represent individual fields: lower case implies turn the bit off, and uppercase implies turn the bit on. All unreferenced bits remain unchanged. All named bits with the exception of the “C/B” bits may be altered with this command. The IPSW has the following format:

```
                 1 1 1 1 1 1 1              2     2 2 2 3 3
 0          7 8 9 0 1 2 3 4 5 6             4     7 8 9 0 1
----------------------------------------------------------------
|J|         |T|H|L|N|X|B|C|V|M|     C/B      |     |R|Q|P|D|I|
----------------------------------------------------------------
```

J Joint instruction and data TLB misses/page faults pending
T Taken branch trap enabled
H Higher-privilege transfer trap enable
L Lower-privilege transfer trap enable
N Instruction whose address is at front of PC queue is nullified
X Data memory break disable

MPSW

B	Taken branch in previous cycle
C	Code address translation enable
V	Divide step correction
M	High-priority machine check disable
C/B	Carry/Borrow bits
R	Recovery counter enable
Q	Interruption state collection enable
P	Protection ID validation enable
D	Data address translation enable
I	External, power failure, & low-priority machine check interruption enable

System Debug displays this register in two formats:

```
IPSW=$6ff0b=jthlnxbCVmrQpDI
```

The first value is a full 32-bit integer representation of the register. The second format shows the value of the special named bits. An uppercase letter means that the bit is on while a lowercase letter indicates that the bit is off.

Examples

```
%nmdebug > dr psw
PSW=0006ff0f=jthlnxbCVmrQPDI
%nmdebug > mpsw p
%nmdebug > dr psw
PSW=0006ff0b=jthlnxbCVmrQpDI
```

Turn OFF the protection ID validation enable bit in the IPSW.

```
$nmdat > mpsw CD
$nmdat >
```

Enable code and data translation. System Debug windows are affected by these two bits.

Limitations, Restrictions

Nmdebug alters the “R” bit while single stepping and the “T” bit when the TRAP BRANCH command is used.

The system dispatcher enforces fixed settings for several key bits. For example, if the “I” bit is turned off with this command, the dispatcher sets it back on when this process is launched.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

MR

Modifies the contents of the specified CM or NM register.

Syntax

```
MR cm_register [newvalue]
MR nm_register [newvalue]
```

By default, the current register value is displayed. The `ENV` variable `QUIET_MODIFY` can be used to suppress the display of the current value.

Parameters

cm_register The CM register whose contents are to be modified. This can be:

`DB`	The stack base relative word offset of DB.
`DBDST`	The DB data segment number.
`CIR`	The current instruction register.
`CMPC`	The full logical CM program counter address. ■ Only the offset part can be modified. ■ CIR will also be modified.
`Q`	The Q register word offset, DB relative.
`S`	The S register word offset, DB relative.
`SDST`	The stack data segment number.
`STATUS`	The CM status register. ■ The segment number portion cannot be modified.
`X`	The X (index) register.

Note CM registers can *not* be modified when the user initially entered Debug in NM (nmdebug).

nm_register The NM register whose contents are to be modified.

Note NM registers can *not* be modified when the user initially entered Debug in CM (cmdebug).

Modifying PC modifies PCOF and PCSF. It sets PCOB to PCOF+4 and to PCSF. The original priv bits are retained. That is, when PC is modified, the priv bits are unaffected.

To fully understand the use and conventions for the various registers, refer to the *Precision Architecture and Instruction Reference Manual*(09740-90014) and *Procedure Calling Conventions Reference Manual* (09740-90015). (These may be ordered as a set with Part Number 09740-64003.) The procedure calling conventions manual is of particular importance for understanding how the language compilers utilize the registers to pass parameters, return values, and hold temporary values. The following tables list the NM registers available within System Debug. Many registers have aliases through which they may be referenced. Alias names in *italics* are not available in System Debug.

Access rights abbreviations are listed below. `PM` indicates that privileged mode (`PM`) capability is required.

`d` Display access
`D` PM display access
`m` Modify access
`M` PM modify access

The following registers are known as the *General Registers*.

Name	Alias	Access	Description
R0	*none*	d	A constant 0
R1	*none*	dm	General register 1
R2	*none*	dm	Used to hold RP at times
R3	*none*	dm	General register 3
⋮			
R22	*none*	dm	General register 22
R23	ARG3	dm	Argument register 3
R24	ARG2	dm	Argument register 2
R25	ARG1	dm	Argument register 1
R26	ARG0	dm	Argument register 0
R27	DP	dM	Global data pointer
R28	RET1	dm	Return register 1
R29	RET0	dm	Return register 0
	SL	dm	Static link
R30	SP	dM	Current stack pointer
R31	*MRP*	dm	Millicode return pointer

The following registers are pseudo registers. They are not defined in the Precision Architecture, but are terms used in the Procedure Calling Conventions document and by the language compilers. They are provided for convenience. They are computed based on stack unwind information. They may not be modified.

Name	Alias	Access	Description
RP	*none*	d	Return pointer (not the same as R2)
PSP	*none*	d	Previous stack pointer

The following registers are known as the *Space Registers*. They are used for short pointer addressing:

Name	Alias	Access	Description
`SR0`	*none*	dm	Space register 0
`SR1`	*SARG*	dm	Space register argument
	SRET	dm	Space return register
`SR2`	*none*	dm	Space register 2
`SR3`	*none*	dm	Space register 3
`SR4`	*none*	dM	Process local code space(tracks PC space)
`SR5`	*none*	dM	Process local data space
`SR6`	*none*	dM	Operating system data space 1
`SR7`	*none*	dM	Operating system data space 2

The following registers are known as the *Control Registers*. They contain system state information:

Name	Alias	Access	Description
CR0	RCTR	dM	Recovery counter
CR8	PID1	dM	Protection ID 1 (16 bits)
CR9	PID2	dM	Protection ID 2 (16 bits)
CR10	CCR	dM	Coprocessor configuration (8 bits)
CR11	SAR	dm	Shift amount register (5 bits)
CR12	PID3	dM	Protection ID 3 (16 bits)
CR13	PID4	dM	Protection ID 4 (16 bits)
CR14	IVA	dM	Interrupt vector address
CR15	EIEM	dM	External interrupt enable mask
CR16	ITMR	dM	Interval timer
CR17	PCSF	dM	PC space queue front
none	PCSB	dM	PC space queue back
CR18	PCOF	dM	PC offset queue front
none	PCSB	dM	PC offset queue Back
none	PCQF	dM	PC queue (PCOF.PCSF) front
none	PCQB	dM	PC queue (PCOB.PCSB) back
none	PC	dM	PCQF with priv bits set to zero
none	PRIV	dM	Low two order bits (30,31) of PCOF.
CR19	IIR	dM	Interrupt instruction register
CR20	ISR	dM	Interrupt space register
CR21	IOR	dM	Interrupt offset register
CR22	IPSW	dM	Interrupt processor status word
	PSW	dM	Processor status word
CR23	EIRR	dM	External interrupt request register
CR24	TR0	dM	Temporary register 0
⋮			
CR31	TR7	dM	Temporary register 7

Note

the *Precision Architecture and Instruction Reference Manual* (09740-90014) refers to the PC (*program counter*) registers as the IA (*instruction address*) registers. This manual will use the PC mnemonic when referring to the IA registers.

The following registers are floating-point registers. If a machine has a floating-point coprocessor board, these values are from that board. If no floating-point hardware is present, the operating system emulates the function of the hardware, in which case these are the values from floating-point emulation.

Name	Alias	Access	Description
`FP0`	*none*	dm	FP register 0
`FP1`	*none*	dm	FP register 1
`FP2`	*none*	dm	FP register 2
`FP3`	*none*	dm	FP register 3
`FP4`	*FARG0*	dm	FP argument register 0
	FRET	dm	FP return register
`FP5`	*FARG1*	dm	FP argument register 1
`FP6`	*FARG2*	dm	FP argument register 2
`FP7`	*FARG3*	dm	FP argument register 3
`FP8`	*none*	dm	FP register 8
⋮			
`FP15`	*none*	dm	FP register 15
`FPSTATUS`	*none*	dm	FP status reg (left half of FP0)
`FPE1`	*none*	dm	FP exception reg 1 (right half of FP0)
`FPE2`	*none*	dm	FP exception reg 2 (left half of FP1)
`FPE3`	*none*	dm	FP exception reg 3 (right half of FP1)
`FPE4`	*none*	dm	FP exception reg 4 (left half of FP2)
`FPE5`	*none*	dm	FP exception reg 5 (right half of FP2)
`FPE6`	*none*	dm	FP exception reg 6 (left half of FP3)
`FPE7`	*none*	dm	FP exception reg 7 (right half of FP3)

newvalue

The new value for the register can optionally be supplied on the command line. If the new value was omitted, Debug displays the old value, and prompts for the new value. To retain the original value, just hit return.

When a register is modified, the actual machine registers are not changed until the process is resumed. That is, the new value is recorded and takes effect when Debug is exited using the `CONTINUE` or `EXIT` commands. Furthermore the value is applied only to the PIN being debugged. This is true of all but several special registers that are expected to remain constant during the life of MPE XL. The list of these registers follows:

`sR6`	
`sR7`	
`tr0-tr7`	Alias for cr24 - cr31
`cCr`	Alias for cr10
`iVa`	Alias for cr14

`eIem`	Alias for cr15
`eIrr`	Alias for cr23

When one of these registers is modified, the new value takes effect *immediately*. Since these registers are global across all processes, all other users are affected by the change.

Examples

```
%cmdebug > mr cmpc
CMPC=PROG %0.01754 := prog(0.1762)
```

Modify the contents of the CM program counter. Only the offset portion of the CM logical address can be modified. It is not possible to change the logical segment number portion.

Note that this also modifies `CIR`, the current instruction register.

```
%cmdebug > mr x 0
X=000123 := 0
```

Zero the X register.

```
$nmdebug > mr pc pc + 4
pc=0021d7b4 := 0021d7b8
```

Advance the PC (this changes pcq front and pcq back).

```
$nmdebug > mr ret0  [psp-20]
r28=00000001 := 00ef2340
```

Modify return register `0` (r28) to be the contents of the address specified by psp-20.

Limitations, Restrictions

The PC register can not be modified unless the user has privileged mode.

When CM code has been translated, and is executing translated, modification of the CM registers may result in an undefined/undesirable state.

Refer to appendix C for a discussion of CM object code translation, node points, and breakpoints in translated CM code.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

NM

Enters native mode (nmdat / nmdebug). See the `CM` command.

Syntax

```
NM
```

The command switches from CM (cmdat/cmdebug) to NM (nmdat/nmdebug). If the windows are on, the screen is cleared and the set of windows enabled for nmdebug are redrawn. The command also sets several environment variables. The variables affected and their new values are shown below:

```
ENV  MODE     "NM"
ENV  INBASE   NM_INBASE
ENV  OUTBASE  NM_OUTBASE
```

Parameters

none

Examples

```
%cmdebug > nm
$nmdebug >
```

Switch from cmdebug to nmdebug.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the

exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

OPENDUMP

DAT only

Opens a dump file.

Syntax

```
OPENDUMP file
```

This command opens the specified dump file previously restored to disk by the `GETDUMP` command. An implicit `DUMPINFO STATE` command is then performed to show the user the state of the dump. If another dump file is already open when this command is entered, it is closed automatically first.

Parameters

file	The name of the dump file to be opened. Dump file names are limited to a maximum of five characters.

Examples

```
$nmdat > opendump EXAMP

Dump Title: SA 2559 on KC (8/29/88 9:40)
Last Pin: 34

$nmdat >
```

Opens the dump file `EXAMP`.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

PAUSE

Pauses (puts to sleep) a process for the specified number of seconds.

Syntax

```
PAUSE n
```

Parameters

n	The number of seconds the process is to be suspended. Negative values are treated the same as positive ones.

Examples

```
$nmdebug > pause #10
```

Suspend the process for (decimal) 10 seconds.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

PIN

Privileged Mode

Switches the process-specific pointers and registers to allow the examination of process related information.

Syntax

```
PIN [pin] [ANYSTATE]
```

Parameters

pin — The process identification number (PIN). If omitted, the current process that was active at dump time is used. If no process was active at dump time, a PIN of zero is used (A PIN of 0 refers to the dispatcher).

`ANYSTATE` — If the keyword `ANYSTATE` is specified, the current state of the process for *pin* is not verified before the process switch occurs. If this keyword is absent, the current state of the process for *pin* must be "alive" for the command to succeed.

Examples

```
$nmdat > pin 8
```

Switches the process pointers and the registers to PIN 8.

Limitations, Restrictions

The current implementation of this command for Debug is to take the process state as last stored in its task control block (TCB). The NM symbol names for the process will not be known.

Warning **In Debug, switching to another PIN does not cause that process to suspend execution. As a result, subsequent use of certain other Debug commands, such as `TRACE`, may not work properly, and may even cause the system to crash. In order to prevent the possibility of a system failure, the PIN should first be suspended, as with the (Break) key or the `:BREAKJOB` command, before using the `PIN` command in debug.**

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

PROCLIST

Lists the specified NM symbols in the specified NM executable library.

Syntax

```
PROCLIST [pattern] [lstfile] [lookup_id] [detail] [outputfile]
```

The values printed by this command are the values found in the symbol table that is searched. This command does not perform any form of symbol location fixups. The addresses printed for most data symbols must be relocated relative to DP to be useful.

Parameters

pattern The symbol names(s) that are to be listed. The pattern match is performed on the symbol *name* only. That is:

parent_name.symbol_name	For nested procedures.
symbol_name	For all other symbols.

For procedure symbols, only the procedure part is used (file name and module are excluded from the pattern match).

This parameter can be specified with wildcards or with a full regular expression. Refer to appendix A for additional information about pattern matching and regular expressions.

The following wildcards are supported:

`@`	Matches any character(s).
`?`	Matches any alphabetic character.
`#`	Matches any numeric character.

The following are valid name pattern specifications:

`@`	Matches everything; all names.
`pib@`	Matches all names that start with "pib".

`log2##4` Matches "log2004", "log2754", and so on.

The following regular expressions are equivalent to the patterns with wildcards that are listed above:

```
'.*'
'pib.*'
log2[0-9][0-9]4'
```

By default, all symbols are listed.

lstfile — The name of the executable library for which to list the symbols (program or library). If the parameter is not given, the program file being executed is assumed. The address printed is the entry point of the procedure (not the start of the procedure).

lookup_id — Specifies which symbols to list. If *lookup_id* is not specified, `PROCEDURES` is assumed. Refer to the "Procedure Name Symbols" section in chapter 2 for additional details.

`PRESORTED`	List System Object Module symbols Debug sorted for use in windows and TR.
`UNIVERSAL`	List exported procedures in the System Object Module.
`LOCAL`	List nonexported procedures in the System Object Module.
`NESTED`	List nested procedures in the System Object Module.
`PROCEDURES`	List local or exported procedures in the System Object Module.
`ALLPROC`	List local/exported/nested procedures in the System Object Module.
`EXPORTSTUB`	List export stubs in the System Object Module.
`DATAANY`	List exported and local data in the System Object Module.
`DATAUNIV`	List exported data in the System Object Module.
`DATALOCAL`	List local data in the System Object Module.
`LSTPROC`	List exported level 1 procedures in the LST.
`LSTEXPORTSTUB`	List export stubs in the LST.
`ANY`	List for any type of symbol in the System Object Module.

detail — This parameter specifies the level of detail given when listing the symbols. The default value is `0` which lists the address and name of the symbol. Negative values are converted to positive ones. Any value larger than the maximum defined detail level functions as if the actual maximum detail level has been entered.

`0`	List symbol address and name.
`1`	Same as `0` but print symbol type, scope, residency bits.
`2`	Same as `1` but print address of symbol record.

The abbreviations used for the output are summarized below. Refer to the Object Module Definition document for detailed descriptions and definitions of the terms.

The following keywords determine the symbol type:

ABS	Absolute constant.
DATA	Normal initialized data.
CODE	Unspecified code.
PRIPROG	Primary program entry point.
SECPROG	Secondary program entry point.
ENTRY	Any code entry point.
STORAGE	Storage. The value of the symbol is not known.
STUB	Either an import or parameter relocation stub.
MODULE	Source module name.
SYMEXT	Symbol extension record.
ARGEXT	Argument extension record.
MILLI	Millicode subroutine.
DISOCT	Disabled translated CM code.
MILXTRN	External millicode subroutine.

The following terms determine the symbol scope:

UNSAT	Unsatisfied, import request not satisfied.
EXTERN	External, import request linked to symbol in another module.
LOCAL	Local, not exported for outside use.
UNIV	Universal, exported for outside use.

The following values determine the parameter check level (CHECK):

0	No checking.
1	Check symbol type descriptor only.
2	Level 1, plus check number of arguments passed.
3	Level 2, plus check type of each argument.

The following values determine the execution level required to call this entry point (XLEAST):

0,1,2,3	The minimum execution level needed.

The following letters indicate the value of various bits associated with each symbol. An uppercase letter indicates the bit is "on", while a lowercase letter means the bit is "off".

`Q|q`	"Must qualify" bit.
`F|f`	"Initially frozen" bit.
`R|r`	"Memory resident" bit.
`C|c`	"Is common" bit.
`D|d`	"Duplicate common" bit.

outputfile If this parameter is given, the symbols are sent to the indicated file rather than to the terminal screen.

Examples

```
$nmdebug > proclist
4d5.58db     $START$
4d5.6b58     $UNWIND_START
4d5.6bc8     $UNWIND_END
4d5.6be0     $RECOVER_START
4d5.6be0     $RECOVER_END
4d5.58bf     ?$START$
4d5.5b53     processstudent.highscore
4d5.5c3f     processstudent.lowscore
4d5.5d27     processstudent
4d5.6073     initstudentrecord
4d5.681f     PROGRAM
4d5.681f     _start
4d5.5937     ?PROGRAM
4d5.5957     ?_start
4d5.5000     lr_na_unk
4d5.5004     $find_alignment
4d5.5084     $more_na
4d5.5028     $bigger_but_still_small
4d5.5024     $b_out
4d5.5018     $b_loop
4d5.5048     $wordloop

control-Y encountered
$nmdebug >
```

The above example lists all of the symbols for the current program file (GRADES.DEMO.TELESUP). The file contains many symbols, including millicode routines added to the program file by the Link Editor. The output was interrupted by striking the Control-Y key.

```
$nmdebug > proclist processstudent@,,allproc
4d5.5b53     processstudent.highscore
4d5.5c3f     processstudent.lowscore
4d5.5d27     processstudent
```

List all procedures that start with the string “processstudent”.

```
$nmdebug > procl ,,nested
4d5.5b53     processstudent.highscore
4d5.5c3f     processstudent.lowscore

$nmdebug > procl ,,nested,1
CODE    LOCAL    check: 0 xl: 3 qfrcd  4d5.5b53     processstudent.highscore
CODE    LOCAL    check: 0 xl: 3 qfrcd  4d5.5c3f     processstudent.lowscore
```

The above examples print only the nested procedures. A detail level value of 1 was specified in the second example.

```
$nmdebug > proclist `^a`,xl.demo
4d8.15c8b    average
```

Show all procedures in `XL.DEMO` that start with the letter "a". Notice the use of regular expressions (see appendix A) for the pattern matching string.

```
$nmdebug > procl ,,datauniv
4d5.40000008 $global$
4d5.40000008 $dp$
4d5.40000160 $PFA_C_START
4d5.40000160 $PFA_C_END
4d5.40000160 output
4d5.400003a8 input
```

```
$nmdebug > proclist ,,data,1
DATA    UNIV     check: 0 xl: 0 qfrcd  4d5.40000008 $global$
DATA    UNIV     check: 0 xl: 0 qfrcd  4d5.40000008 $dp$
DATA    UNIV     check: 0 xl: 0 qfrcd  4d5.40000160 $PFA_C_START
DATA    UNIV     check: 0 xl: 0 qfrcd  4d5.40000160 $PFA_C_END
DATA    UNIV     check: 1 xl: 0 qfrcd  4d5.40000160 output
DATA    UNIV     check: 1 xl: 0 qfrcd  4d5.400003a8 input
DATA    LOCAL    check: 0 xl: 3 qfrcd  4d5.5730     L$5
DATA    LOCAL    check: 0 xl: 3 qfrcd  4d5.5780     L$8
DATA    LOCAL    check: 0 xl: 0 qfrcd  4d5.40000008 M$1
DATA    LOCAL    check: 0 xl: 3 qfrcd  4d5.5850     L$2
```

The `PROCLIST` command can also be used to list data symbols that are present in the System Object Module directory.

```
$nmdebug > proclist @FOPEN@,nl.pub.sys
a.3f8140   FOPEN
a.374428   HPFOPEN
a.2ea29b   P__FOPENERR
```

The final example requests a list of all procedures in the system NL that have the uppercase letters "FOPEN" in their name.

Limitations, Restrictions

Unless a file equation is used, the size of the output file defaults to 20000 records of 80 bytes each.

The `LSTPROC` and `LSTEXPORTSTUB` options are not implemented.

A `PROCLIST` for CM procedures and symbols is not implemented.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

PSEUDOMAP

Logically maps a local file into virtual memory, utilizing symbol information in library/program files.

Syntax

```
PSEUDOMAP local_file space_id [loaded_fname] [offset]
```

The `PSEUDOMAP` command is used to fill in parts of virtual memory that are not accessible in a dump. When a file is mapped using `PSEUDOMAP`, the file appears to be loaded in virtual memory at the specified location. When portions of this virtual memory cannot be read from the dump, corresponding locations from the `PSEUDOMAP`ped file are read instead.

The `PSEUDOMAP` command is also used to provide access to procedure name symbol information stored in local native mode program files or executable libraries. When one of these files is mapped into memory its symbols are preprocessed. The file is then inserted into the list of loaded files (see the `LOADINFO` command). If the specified space ID is not already part of the list of loaded files, it is added at the end of the list, but before the entry for `NL.PUB.SYS`. If the space ID is already present, the entry is inserted just before the entry with the same space ID.

Any attempt to convert an address in the specified space ID to a symbol name uses the symbol information in the `PSEUDOMAP`ped file. The process of converting a symbol name to an address involves scanning the list of loaded files, checking each one in turn for the symbol name of interest. If the loaded file list contains more than one entry for a space ID (as created by this command), only the first one in the list is searched.

Related commands: `MAPLIST`, `UNMAP`

Parameters

local_file The name of the local program/library file from which to obtain symbol information.

space_id Associate symbols from *local_file* with this space. Any attempt to convert a symbol address in this space to an address uses the local file for symbol name lookups.

loaded_fname Bind this file name to all symbols from space *space_id*. All of the commands and functions that deal with file names (for example, the NMPATH function and NM program window) use this file name any time a file name is to be associated with a space ID.

offset Associate *local_file* with this offset within the space.

Examples

```
$nmdebug > wl FOPEN
SYS $a.3e1130

$nmdebug > map nl.build
1  NL.BUILD.CMDEBUG   4ef.0 Bytes = c5f600

$nmdebug > xl nl.build 4ef nl.pub.sys
Preprocessing NL.BUILD.CMDEBUG, please wait ... Done

$nmdebug > dc FOPEN 3
USER $4ef.4c5138
004c5138  FOPEN         6bc23fd9  STW      2,-20(0,30)
004c513c  FOPEN+$4      37de00d0  LDO      104(30),30
004c5140  FOPEN+$8      4bdf3f09  LDW      -124(0,30),31
```

We start by seeing that the FOPEN routine is found in the SYS library at \$a.3e1130. Next we use the map command to map a local copy of a new version of the NL into memory. (It gets mapped at space \$4ef.) We then use the PSEUDOMAP command to obtain access to the symbols in the new copy of NL. Finally, we use the DC command to display the first few words of the FOPEN procedure as found in the new NL (NL.BUILD.CMDEBUG).

Remember that the `PSEUDOMAP` command only provides access to symbol information. In order to display data in a file, the `MAP` command must be used.

```
($22) nmdat > dptree 22
22 (CI.PUB.SYS)

($22) nmdat > tr
     PC=a.000d87f8 enable_int+$20
* 0) SP=40224ac8 RP=a.001cfda8 notify_dispatcher.block_current_process+$268
  1) SP=40224ac8 RP=a.001d0dcc notify_dispatcher+$2b0
  2) SP=40224a10 RP=a.00291b94 wait_for_active_port+$e0
  3) SP=40224828 RP=a.00292324 receive_from_port+$22c
  4) SP=402247c0 RP=a.002c51ec extend_receive+$41c
  5) SP=402246d0 RP=a.002b5d30 rendezvousio.get_specific+$11c
  6) SP=402245b0 RP=a.002b5fb4 rendezvousio+$19c
  7) SP=40224510 RP=a.002b2398 attachio+$5e0
  8) SP=40224308 RP=a.002ad690 ?attachio+$8
       export stub: a.0061575c arg_regs+$28
  9) SP=40224050 RP=a.005984bc nm_switch_code+$9b4
  a) SP=40223f20 RP=a.0042a5bc SWT_RETURN
     (switch marker frame)
  b) SP=40223bc0 RP=a.00597274 switch_to_cm+$8c4
  c) SP=402239d0 RP=a.007499b8 tm_cms_type_mgr+$8bc
  d) SP=40223668 RP=a.0072ee44 FREAD+$3c8
  e) SP=40221780 RP=a.006ac858 readcmd+$1dc
  f) SP=40221560 RP=a.006abcc8 ?readcmd+$8
       export stub: 74.00006274
 10) SP=402211d8 RP=74.000068e0
 11) SP=40221178 RP=74.00007450
 12) SP=40221130 RP=74.00000000
     (end of NM stack)
```

The current PIN ($22) is the program `CI.PUB.SYS`. In DAT, we do a stack trace, but we observe that the symbols for the program file are not part of the stack trace.

```
($22) nmdat > loadinfo
nm SYS   NL.PUB.SYS                    SID = $a
cm SYS   SL.PUB.SYS
```

```
($22) nmdat > xl ci.abuild00.official 74 ci.pub.sys
Preprocessing CI.ABUILD00.OFFICIAL, please wait ... Done

($22) nmdat > loadinfo
nm USER  CI.PUB.SYS                    SID = $74
nm SYS   NL.PUB.SYS                    SID = $a
cm SYS   SL.PUB.SYS
($22) nmdat >
```

A quick check of our loaded files reveals that DAT does not know about the symbols for CI.PUB.SYS. We now use the PSEUDOMAP command to open a local copy of the program file from which symbol information can be gleaned. A final check of the loaded file information shows that CI.PUB.SYS has successfully been added to the list.

Note that the stack trace code works because the unwind descriptors for CI.PUB.SYS happen to be present in the dump. This is usually not the case (unless the file was loaded as a “dumpworthy” file).

```
($22) nmdat > tr
     PC=a.000d87f8 enable_int+$20
* 0) SP=40224ac8 RP=a.001cfda8 notify_dispatcher.block_current_process+$268
  1) SP=40224ac8 RP=a.001d0dcc notify_dispatcher+$2b0
  2) SP=40224a10 RP=a.00291b94 wait_for_active_port+$e0
  3) SP=40224828 RP=a.00292324 receive_from_port+$22c
  4) SP=402247c0 RP=a.002c51ec extend_receive+$41c
  5) SP=402246d0 RP=a.002b5d30 rendezvousio.get_specific+$11c
  6) SP=402245b0 RP=a.002b5fb4 rendezvousio+$19c
  7) SP=40224510 RP=a.002b2398 attachio+$5e0
  8) SP=40224308 RP=a.002ad690 ?attachio+$8
       export stub: a.0061575c arg_regs+$28
  9) SP=40224050 RP=a.005984bc nm_switch_code+$9b4
  a) SP=40223f20 RP=a.0042a5bc SWT_RETURN
     (switch marker frame)
  b) SP=40223bc0 RP=a.00597274 switch_to_cm+$8c4
  c) SP=402239d0 RP=a.007499b8 tm_cms_type_mgr+$8bc
  d) SP=40223668 RP=a.0072ee44 FREAD+$3c8
  e) SP=40221780 RP=a.006ac858 readcmd+$1dc
```

```
 f) SP=40221560 RP=a.006abcc8 ?readcmd+$8
      export stub: 74.00006274 ci_cmd_io+$34
10) SP=402211d8 RP=74.000068e0 main_ci+$a0
11) SP=40221178 RP=74.00007450 PROGRAM+$218
12) SP=40221130 RP=74.00000000
    (end of NM stack)
```

We again do a stack trace; this time the symbols for the program file show up.

```
$nmdat > loadinfo
nm SYS   NL.PUB.SYS                     SID = $a
cm SYS   SL.PUB.SYS

$nmdat > xl nl.build a nl.pub.sys
Preprocessing NL.BUILD.CMDEBUG, please wait ... Done

$nmdat > loadinfo
nm SYS   NL.PUB.SYS                     SID = $a
nm SYS   NL.PUB.SYS                     SID = $a
cm SYS   SL.PUB.SYS
$nmdat >
```

We start by looking at our list of loaded files in DAT. We then proceed to map in a local copy of an NL. Notice that there are now two entries for NL.PUB.SYS in the loaded file list both at space $a. Attempts to look up symbols in space $a use the first entry in the table (which corresponds to the file mapped with the PSEUDOMAP command). Likewise, attempts to perform a name to address lookup for a symbol searches only the first NL.PUB.SYS entry.

Limitations, Restrictions

Information required to perform stack traces (the unwind tables) are also part of program files and executable libraries. When a file is opened with this command, we should be utilizing the unwind tables found there. This functionality is not implemented.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the

exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

PURGEDUMP

DAT only

Purges a dump file.

Syntax

```
PURGEDUMP dumpfile
```

Parameters

dumpfile The name of the dump file to be deleted.

Examples

```
%cmdat > purgedump EXAMP
```

Purge dump file `EXAMP`.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

REDO

Reexecutes a command from the history command stack after optionally editing the command.

Syntax

```
REDO [cmd_string  ]
REDO [history_index]
```

System Debug uses the same `REDO` editing commands as the `REDO` command supported by the MPE XL Command Interpreter. Please refer to the *MPE XL Commands Reference Manual* (32650-90003) for specific details about editing commands.

Parameters

cmd_string Redo the most recent command in the history stack that commences with *cmd_string*. For example, `redo wh` can be used to match the most recent while statement.

history_index The history stack index of the command that is to be redone.

A negative index can be used to specify a command relative to the current command. For example, -2 implies the command used two commands ago.

`REDO`, entered alone, redoes the most recent command.

Examples

```
%cmdebug > redo dq
dq-176,20
    r4
dq-146,20
```

Redo the most recent command that started with "dq".

Limitations, Restrictions

Upon initial entry into System Debug, the command stack is empty, since no prior command has been executed. If the `REDO` command is entered as the command, a blank command is provided for editing.

The MPE XL Command Interpreter allows an edit string to be specified on the `REDO` command line. This feature is not supported in System Debug.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

REGLIST

Lists the registers into a file in USE file format.

Syntax

```
REGLIST [filename]
```

Parameters

filename The name of the file into which the registers are listed.

Examples

```
$nmdebug > reglist  rsave
$nmdebug >
```

List the contents of the registers into the file `rsave`. You can use the `USE` command later to restore the state of the registers.

Limitations, Restrictions

`REGLIST` dumps only the NM register set.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

RESTORE

Restores macros or variables from a file that was previously created by the `STORE` command.

Syntax

```
RESTORE MACROS    filename
RESTORE VARIABLES filename
```

The `RESTORE` command quickly restores saved macros or variables from a binary file that was created by the `STORE` command.

Based on the selector (`MACROS` or `VARIABLES`), all currently defined macros or variables are immediately discarded, and are *replaced entirely* by the contents of the `STORE` file.

The current limits (as set by `ENV MACROS` or `ENV VARS` and `ENV VARS_LOC`) are automatically changed to the limits that were in effect at the time the `STORE` file was created.

After the `RESTORE`, macros or variables can be referenced, created, listed, or deleted in the normal manner.

Parameters

`MACROS`	Specifies that macros are to be restored. This keyword can be abbreviated and entered in uppercase or lowercase.
`VARIABLES`	Specifies that variables are to be restored. This keyword can be abbreviated and entered in uppercase or lowercase.
filename	The name of the file (previously built by the `STORE` command) from which the macros or variables are to be restored.

Examples

```
$nmdat > store var savevar
$nmdat > vard @
$nmdat > restore var savevar
```

Stores the currently defined variables into the file SAVEVAR. All variables are deleted, then the RESTORE command is used to restore them all again.

Related command: STORE.

Related ENV variables: MACROS, VARS, VARS_LOC.

Limitations, Restrictions

STORE/RESTORE are currently very version dependent.

If the internal versions of macros, variables, or storage management change, it may not be possible to RESTORE from a file that was stored with earlier versions of STORE. An error is generated.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

RET[URN]

Exits from a macro, optionally returning a specified value.

Syntax

```
RET[URN] [value]
```

The RETURN command can be used only within a macro.

When the RETURN command is encountered, a value is returned, and the macro execution is immediately terminated. Additional commands within the macro that follows an executed RETURN command are never executed.

Parameters

value	The value to be returned by the macro. If *value* is not specified, the default macro return value is returned.

Examples

```
$nmdebug > macro test=$123 (p1) {if p1 < 10 then return p1 else ret}
$nmdebug > wl test(3)
$3
$nmdebug > wl test(45)
$123
```

A macro named test is defined with a default return value of $123.

When the macro is called with the parameter of 3, the parameter is less than $10, so the parameter value is returned.

In the second call, because $45 is larger than 10, the default macro return value $123 is returned.

RET[URN]

```
$nmdebug > return 33
The RETURN command must be used within a macro body.  (error #1449)
```

The `RETURN` command can be used only within a macro.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

SET

Sets new values for a select subset of all user configurable options.

Syntax

```
SET

    ┌ O[CT] | % ┐ ┌ IN  ┐
SET │ D[EC] | # │ └ OUT ┘
    └ H[EX] | $ ┘

    ┌ CRON  ┐
SET └ CROFF ┘

    ┌ MOREON  ┐
SET └ MOREOFF ┘

SET [DEF[AULT]]
```

The SET command allows a simplified method of setting a few of the many environment variables. See the ENV command for more information.

The SET command entered alone, without parameters, displays all current settings.

Parameters

O[CT] | %
D[EC] | #
H[EX] | $
: Set the current default input conversion base and the current output display base to octal, decimal, or hexadecimal.

IN | OUT
: The input conversion base and the output display base can be individually set to different values. For example:

```
SET OCT IN
SET $ OUT
```


FINAL TRIM SIZE : 7.5 in x 9.0 in

SET

This sets octal for input, hex for output.

If `IN` and `OUT` are omitted, *both* input and output bases are set to the specified base.

CRON | CROFF	CRON (carriage return on) and CROFF (carriage return off) control the automatic repetition of the last typed command whenever a lone carriage control is entered. (This option is for compatibility with prior versions of Debug; see the new ENV AUTOREPEAT.) SET CRON is the same as ENV AUTOREPEAT TRUE. SET CROFF is the same as ENV AUTOREPEAT FALSE.
MOREON | MOREOFF	MOREON (terminal paging on) and MOREOFF (terminal paging off) control the automatic paging of terminal output. SET MOREON is the same as ENV TERM_PAGING TRUE. SET MOREOFF is the same as ENV TERM_PAGING FALSE.
DEF[AULT]	Resets the following ENV variables to their default values indicated below:

env autoignore	FALSE
env changes	"halfinv"
env cm_inbase	%
env cm_outbase	%
env cmdlinesubs	TRUE
env echo_cmds	FALSE
env echo_subs	FALSE
env echo_use	FALSE
env fill	"zero"
env filter	' '
env hexupshift	FALSE
env justify	"right"
env list_paging	TRUE
env list_pagelen	#60
env list_title	&

```
    '"Page: " list_pagenum:"d" "   " version "  "
date " " time'
```

env list_width	#80
env lookup_id	"LSTPROC"
env markers	"uline"
env multi_line_errs	2

```
env nm_inbase       $
env nm_outbase      $
env pstmt           TRUE
env term_loud       TRUE
env term_paging     FALSE
env term_width      #79
```

Examples

```
$nmdat > SET
```

Display all current settings.

```
%cmdebug > set hex out
```

Set output display base to hexadecimal.

```
%cmdebug > set %
```

Set both input and output bases to octal.

```
$nmdat > set def
```

Set default values.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

SET*xxx*

The `SET`*xxx* commands are predefined aliases for other commands.

Syntax

```
SETALIAS    alias for  ALIAS
SETENV      alias for  ENV
SETERR      alias for  ERR
SETLOC      alias for  LOC
SETMAC      alias for  MAC
SETVAR      alias for  VAR
```

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

SHOW*xxx*

The `SHOW`*xxx* commands are predefined aliases for other commands.

Syntax

```
SHOWALIAS    alias for  ALIASL
SHOWB        alias for  BL
SHOWCMD      alias for  CMDL
SHOWDATAB    alias for  DATABL
SHOWENV      alias for  ENVL
SHOWERR      alias for  ERRL
SHOWFUNC     alias for  FUNCL
SHOWLOC      alias for  LOCL
SHOWMAC      alias for  MACL
SHOWMAP      alias for  MAPL
SHOWSET      alias for  SET
SHOWSYM      alias for  SYML
SHOWVAR      alias for  VARL
```

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

S, SS

Single steps.

Syntax

```
S[S]  [num_instrs] [ L[OUD] | Q[UIET] ]
```

This command single steps the specified number of instructions. If the user attempts to single step into the system NL or SL (or any portion of code he/she does not have access to view), Debug stops single stepping and free-runs the process (for example, proceed as if the `CONTINUE` command had been issued). For native mode processes, Debug stops processing as soon as it returns from the inaccessible code. For compatibility mode processes, the process continues to run until it encounters a breakpoint.

Parameters

num_instrs	The number of instructions to be executed. If omitted, a single instruction is executed. Negative values are converted to positive values.
`L[OUD] | Q[UIET]`	If `LOUD` is specified, the address where the process stopped is printed. If `QUIET` is specified, no message is displayed. The default is `LOUD`.

Examples

```
%cmdebug > s
%cmdebug >
```

Single step to the next instruction.

```
%cmdebug > ss 5 l
Step to: PROG %0.172
```

```
%cmdebug >
```

Step 5 instructions "loudly", that is, print the ending address.

```
$nmdebug > s #20 l
Step to: 115.00005f0c processstudent+$1e8
$nmdebug >
```

Step 20 instructions, and print the address when stopped.

Limitations, Restrictions

The single step command cannot be used within a macro that is invoked as a function.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

STORE

Stores the currently defined macros or variables to a file.

Syntax

```
STORE MACROS    filename
STORE VARIABLES filename
```

The `STORE` command quickly saves macros or variables to a binary file. At a later point, the `RESTORE` command can be used to restore these saved macros or variables.

The current limits (as set by `ENV MACROS` or `ENV VARS` and `ENV VARS_LOC`) are automatically saved in the `STORE` file, and is reestablished when this file is restored with the `RESTORE` command.

Parameters

`MACROS` Specifies that macros are to be stored. This keyword can be abbreviated and entered in uppercase or lowercase.

`VARIABLES` Specifies that variables are to be stored. This keyword can be abbreviated and entered in uppercase or lowercase.

filename The file name where the macros or variables are to be stored.

Examples

```
$nmdat > store mac savemac
$nmdat > macd @
$nmdat > restore mac savemac
```

Stores the currently defined macros into the file `SAVEMAC`. All macros are deleted, then the `RESTORE` command is used to restore them all again.

Related command: `RESTORE`

Related ENV variables : MACROS, VARS and VARS_LOC

Limitations, Restrictions

STORE and RESTORE are currently very version dependent.

If the internal versions of macros, variables, or storage management changes, it may not be possible to restore from a file that was stored with earlier versions of the STORE command. An error is generated.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

SYMCLOSE

Closes a symbolic data type file that was opened with the SYMOPEN command.

Syntax

```
SYMCLOSE symname
```

Parameters

symname The symbolic name of the symbolic data type file that was assigned at open time.

Examples

```
$ nmdat > symfiles
OS         SYMOS.PUB.SYS
GRADTYP    GRADTYPE.DEMO.TELESUP

$nmdat > symclose SYMOS

$nmdat >
```

Closes the file SYMOS.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

SYMF[ILES]

Lists all open symbolic data type files and their symbolic names.

Syntax

```
SYMF[ILES]
```

Parameters

none

Examples

```
$ nmdat > symf
OS          SYMOS.PUB.SYS
GRADTYP     GRADTYPE.DEMO.TELESUP
```

List all the symbolic data type files currently opened by the program.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

SYMINFO

Lists information/dump data for an opened symbolic data type file.

Syntax

```
SYMINFO [symname] [option] [offset] [length]
```

This command is generally only useful to System Debug developers and people debugging the contents of the symbolic data type files.

Parameters

symname The symbolic name under which the symbolic data type file is referenced. If the symbolic name is omitted, then the last file which was opened with `SYMOPEN` is selected.

option One of the following options can be specified. If none is specified, `HEADER` is assumed.

- `HEADER` Display info about the System Object Module header within the symbolic data type file.
- `SOM` Display data in the System Object Module portion of the symbolic data type file at the indicated offset and length.
- `LST` Display data in the LST portion of the symbolic data type file at the indicated offset and length.

offset For the `SOM` and `LST` options, this parameter specifies the byte offset within the System Object Module or LST area of the file where to begin dumping data. The default value is 0.

length For the `SOM` and `LST` options, this parameter specifies how many bytes to dump. The default value is 16. All length values are rounded to the next highest multiple of 16.

Examples

```
$nmdebug > syminfo

 Som file name: SYMOS.PUB.SYS  Symname: SYMOS
 Som file length: 006735e0  Som offset: 00004000  Som length: 0066f5e0
 Sp dir loc:     00007000  Sp dir len:     00000003
 Sub sp dir loc: 00000138  Sub sp dir len: 00000019
 String loc:     0000706c  String len:     00000298
 DEBUG space:2
 Header: 000150e0  00000010 Subsp_index: 14
 GNTT:   000150f0  00001280 Subsp_index: 15
 LNTT:   00016370  00101310 Subsp_index: 16
 SLT:    00117680  00014f38 Subsp_index: 17
 VT:     0012c5b8  00543028 Subsp_index: 18
 Debug header info: 0000004a 0000004a 00000000 00002a2f

 Const Lookup table: 0064b45c  0001c9f0
 Type Lookup table:  00667e4c  00007780
```

Show the header (default) information for the most recently accessed symbolic file.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

SYML[IST]

Lists information for the specified symbol name in an opened symbolic data type file.

Syntax

```
SYML[IST] [pattern] [symname] [option]
```

Parameters

pattern The symbol names that are to be listed.

This parameter can be specified with wildcards or with a full regular expression. Refer to appendix A for additional information about pattern matching and regular expressions.

The following wildcards are supported:

`@`	Matches any character(s).
`?`	Matches any alphabetic character.
`#`	Matches any numeric character.

The following are valid name pattern specifications:

`@`	Matches everything; all names.
`pib@`	Matches all names that start with "pib".
`log2##4`	Matches "log2004", "log2754", and so on.

The following regular expressions are equivalent to the patterns with wildcards that are listed above:

```
`.*`
`pib.*`
`log2[0-9][0-9]4`
```

By default, all symbols are listed.

symname The symbolic name under which the symbolic data type file is referenced. If the parameter is not given, the symfile last accessed is used.

option A keyword to further specify the operation:

CONST	Display the constant names that match the given pattern. If the constant is a simple type, display its value.
TYPES	Display the type names that match the given pattern.
ALL	Display both type and constant names (default).

Examples

```
$nmdebug > SYMLIST @,GRADTYP

CLASS                                  TYPE    ENUMERATED TYPE
GRADERANGE                             TYPE    SUBRANGE
GRADESARRAY                            TYPE    ARRAY
NAMESTR                                TYPE    STRING
STUDENTRECORD                          TYPE    RECORD
MAXGRADES                              CONST   INTEGER       $a
MAXSTUDENTS                            CONST   INTEGER       $5
MINGRADES                              CONST   INTEGER       $1
MINSTUDENTS                            CONST   INTEGER       $1
```

Print out the all type and constant declarations for the symfile GRADTYP.

```
$nmdebug > SYMLIST gr@
GRADERANGE                             TYPE    SUBRANGE
GRADESARRAY                            TYPE    ARRAY

$nmdebug > SYML `GRADES`
GRADESARRAY                            TYPE    ARRAY
MAXGRADES                              CONST   INTEGER       $a
MINGRADES                              CONST   INTEGER       $1
```

SYML[IST]

```
$nmdebug > SYML max@,,const
MAXGRADES                           CONST   INTEGER      $a
MAXSTUDENTS                         CONST   INTEGER      $5
```

Print out various subsets from the symfile 'GRADTYP'.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

SYMOPEN

Opens a symbolic data type file and sets up pointers to the symbolic debug records.

Syntax

```
SYMOPEN filename [symname]
```

The SYMOPEN command must be used to open a symbolic data type file before the symbolic formatting command and functions can be used.

Parameters

filename The file name of the symbolic data type file. The file must contain symbolic debug records.

symname The symbolic name under which the symbolic data type file is referenced in the formatter commands. If this parameter is omitted, the file name will be used as the symbolic name.

Examples

```
$nmdat > symopen SYMOS.PUB.SYS OS

$nmdat >
```

Open the symbolic file SYMOS.PUB.SYS and assign the symbolic name OS to it.

Limitations, Restrictions

Before a symbolic data type file is ready to be opened with `SYMOPEN`, ensure that the following steps have been followed:

1. The types must be compiled with the `$SYMDEBUG 'xdb'$` option.
2. The program containing the types must have at least one statement.
3. The relocatable library generated by the compiler must be run through `LINKEDIT`.
4. The program file generated by `LINKEDIT` must be run through PXDB.
5. The modified program file generated by PXDB must be prepared with `SYMPREP` in DAT or Debug.
6. The program file (symbolic data type file) is now ready to be opened with `SYMOPEN`.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

SYMPREP

Prepares a program file containing symbolic debug information to be used by the symbolic formatter/symbolic access facility. Files modified through the use of this command are referred to as symbolic data type files.

Syntax

```
SYMPREP {filename}
```

Parameters

filename The name of the program file name to be preprocessed. (Required)

Limitations, Restrictions

Before a program file is ready to be prepared with `SYMPREP`, be sure that the following steps have been followed:

1. The types must be compiled with the `$SYMDEBUG 'xdb'$` option.
2. The program containing the types must have at least one statement.
3. The relocatable library generated by the compiler must be run through `LINKEDIT`.

The modified program file generated by `PXDB` is now ready to be `SYMPREP`ed by DAT or Debug, after which it may be opened with `SYMOPEN`.

To use this command, you must be logged on to the same account where the symbolic file resides.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the

exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

Example

The following example preprocesses the program file GRADTYP.DEMO.TELESUP.

```
$nmdat > symprep gradtyp.demo.telesup


Preprocessing GRADTYP.DEMO.TELESUP
Building constant symbol dictionary ...
Sorting ...
Build type symbol dictionary ...
Sorting ...
Constructing new SOM file ...
GRADTYP.DEMO.TELESUP preprocessed.

$nmdat >
```

T (translate)

Privileged Mode: TCA, TCS

Translates the specified CM address to a virtual address.

Syntax

```
TA     offset          ABS - Bank0
TD     dst.off         Data segment
TDB    offset          DB relative
TS     offset          S relative
TQ     offset          Q relative

TC     cmlogaddr       Program file
TCG    cmlogaddr       Group library
TCP    cmlogaddr       Account library
TCLG   cmlogaddr       Logon group library
TCLP   cmlogaddr       Logon account library
TCS    cmlogaddr       System library

TCA    cmabsaddr       Absolute CST
TCAX   cmabsaddr       Absolute CSTX
```

Parameters

offset — `TA`, `TDB`, `TQ`, `TS` only. The CM word offset that specifies the relative CM address to be translated.

dseg.off — `TC`, `TD` only. The data segment and word offset to be translated.

cmlogaddr — `TC`, `TCG`, `TCP`, `TCLG`, `TCLP`, `TCS` only. A full logical code address (`LCPTR`) specifies three necessary items:

- The CM logical code file (`PROG`, `GRP`, `SYS`, and so on).

- The CM logical segment number.
- The CM word offset within the code segment.

Logical code addresses can be specified in various levels of detail:

- As a full logical code pointer (LCPTR):

`TC procname+20`	Procedure name lookups return LCPTRs.
`TC pw+4`	Predefined ENV variables of type LCPTR.
`TC SYS(2.200)`	Explicit coercion to a LCPTR type.

- As a long pointer (`LPTR`):

  `TC 23.2644` *seg.offset*

  The logical file is determined based on the command suffix. For example:

  `TC` implies `PROG`.
  `TCG` implies `GRP`.
  `TCS` implies `SYS`, and so on.

- As a short pointer (`SPTR`):

  `TC 1024` *offset* only

  The currently executing logical segment number and the currently executing logical file are used to build a `LCPTR`.

The search path used for procedure name lookups is based on the command suffix letter:

`TC`	Full search path: `CM: PROG, GRP, PUB, LGRP, LPUB, SYS.`
`TCG`	Search `GRP`, the group library.
`TCP`	Search `PUB`, the account library.
`TCLG`	Search `LGRP`, the logon group library.
`TCLP`	Search `LPUB`, the logon account library.
`TCS`	Search `SYS`, the system library.
`TCU`	Search `USER`, the user library.

For a full description of logical code addresses, refer to the section "Logical Code Addresses", in chapter 2.

cmabsaddr `TCA, TCAX` only. A full CM absolute code address specifies three necessary items:

- Either the `CST` or the `CSTX`.
- The absolute code segment number.
- The CM word offset within the code segment.

Absolute code addresses can be specified in two ways:

- As a long pointer (`LPTR`):

  `TCA 23.2644` Implicit `CST` 23.2644

```
TCAX 5.3204    Implicit CSTX 5.3204
```

- As a full absolute code pointer (ACPTR):

`TCA  CST(2.200)`	Explicit CST coercion.
`TCAX CSTX(2.200)`	Explicit CSTX coercion.
`TCAX logtoabs(prog(1.20))`	Explicit absolute conversion.

The search path used for procedure name lookups is based on the command suffix letter:

TCA	GRP, PUB, LGRP, LPUB, SYS
TCAX	PROG

Examples

```
%cmdebug > td 1.100
% DST 1.100       VIRT $b.40011630
```

Translate data segment 1.100 to a virtual address.

```
%cmdebug > ta 2000
% ABS+2000      VIRT $a.80000800
```

Translate ABS+2000 to a virtual address.

```
$nmdebug > tcs %22.%5007
SYS %  22.5007    =  CST %  23.5007    =  VIRT $21.6ed0e
FOPEN+%13  (XLSEG11)
start: %4774   entry: %5000   proclen: %626   seglen: %31454
Translator Node Addresses:
CM prev: SYS %22.5006      NM prev: TRANS $21.6afd5c
CM next: SYS %22.5010      NM next: TRANS $21.6afd74
```

Translate CM logical address SYS %22.5007.

```
%cmdebug > tc fgetkeyinfo+1146
SYS %  32.2031    =  CST %  33.2031    =  VIRT $21.a4c32
FGETKEYINFO+%1146  (KSAMSEG1)
start: %663   entry: %702   proclen: %2145   seglen: %37204
Translator Node Addresses:
```

```
CM prev: SYS %32.2030      NM prev: TRANS $21.7da7a0
CM next: SYS %32.2034      NM next: TRANS $21.7da7c4
```

Translate CM logical address `fgetkeyinfo+1146`.

Refer to appendix C for a discussion of CM object code translation, node points, and breakpoints in translated CM code.

Limitations, Restrictions

All information that is displayed in a `TC` (translate code) display can be obtained programmatically, except for the CM segment length.

There is no way to obtain the virtual address of `ABS` relative addresses programmatically.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

TERM

Debug only

Controls the synchronization of several debug processes on a single terminal.

Syntax

```
TERM
TERM LIST
TERM NEXT
```

Terminal locking allows multiple processes to use a single terminal for debugging without confusion.

`TERM LIST` shows information about processes waiting for the terminal semaphore.

`TERM NEXT` grants the terminal to the process at the head of the waiting list.

Exiting, continuing, and stepping from the debugger perform an implicit `TERM NEXT` command.

Parameters

`TERM` `TERM LIST`	Lists information about processes waiting to enter the debugger for the current session.
`TERM NEXT`	If we own the terminal semaphore, release it and allow the next process waiting for it to enter the debugger. Our process is then queued at the end of the list for the semaphore.

Related environment variables: `TERM_LOCKING`.

Examples

```
$(3b) nmdebug > = 2 + 2
$4

PIN 4c is waiting to enter Debug

$(3b) nmdebug > term list
Current term owner: 3b  Next pin: 1a  # Waiting pins: 2
```

A processes has just notified us that it is waiting to enter Debug. We then list information about the waiting PINS. We see that there are two PINs waiting and the first PIN in the queue is 1a.

```
$(3b) nmdebug > term next

PIN 3b is waiting to enter Debug

$(1a) nmdebug > term list
Current term owner: 1a  Next pin: 4c  # Waiting pins: 2
```

We gave away the semaphore and let the next PIN into Debug (PIN 1a). This placed us (PIN 3b) at the end of the queue. We next listed information about the waiting PINs and see that PIN 4c has moved to the front of the queue.

Limitations, Restrictions

Due to the implementation of semaphores, Debug cannot list all of the PINs in the queue, just the first one and a count.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

TR[ACE]

Displays a stack trace.

Syntax

```
TR[ACE] [level] [options]
```

The `TR` command produces a trace of the procedures active on the current PIN's stack. The command is mode sensitive. If the user is in cmdebug, a trace of the compatibility mode stack is produced, if in nmdebug, a trace of the native mode stack is printed. An interleaved stack trace of both CM and NM stacks is produced by using the `DUAL` option.

If the current stack is the NM interrupt control stack (ICS), when the base of the ICS is reached, System Debug automatically switches to the stack of the last running process and continues the stack trace. This feature in no way implies that the routines on the ICS were invoked on behalf of the last running process. If the dispatcher is currently running, there is no last running process, so the stack trace stops when the base of the ICS is found.

Parameters

level The desired maximum depth for the stack trace. If *level* is omitted, the entire depth of the stack is traced.

options Any combination of the following options may be specified:

`DUAL`	Display both NM and CM stack markers, interleaved across switch markers.
`SINGLE`	Display a single stack marker at the specified level.
`UNWIND`	Display formatted stack unwind descriptor information.
`FULL`	Display a fully detailed stack trace.

ISM	Trace across interrupt markers.

NM Examples

```
$nmdebug > tr
     PC=115.00005b50 processstudent.highscore
* 0) SP=40221180 RP=115.00005f0c processstudent+$1e8
  1) SP=40221180 RP=115.00006b1c PROGRAM+$300
  2) SP=40221100 RP=115.00000000
     (end of NM stack)
```

Display an entire NM stack trace. The first line indicates the address the PC register points to. Each stack level is formatted, starting from the top of stack and working down the depth of the stack. Level numbers are indicated on the left; an asterisk marks the current level. (Refer to the LEV command.)

```
$nmdebug > tr
     PC=a.0074da24 FWRITE
* 0) SP=40221260 RP=a.00748150 ?FWRITE+$8
       export stub: f4.0012d044 P_FLUSHLINE+$54
  1) SP=40221260 RP=f4.00139560 P_WRITELN+$20
  2) SP=40221200 RP=f4.00139630 P_WRITELN+$9c
  3) SP=402211c8 RP=f4.0013950c ?P_WRITELN+$8
       export stub: 115.00005e30 processstudent+$10c
  4) SP=40221180 RP=115.00006b1c PROGRAM+$300
  5) SP=40221100 RP=115.00000000
     (end of NM stack)
```

The above example shows a stack trace that contains a call from the program file to a user library, and from the user library to the system NL. Transitions between libraries are performed through the use of export stubs. (Refer to the *Procedure Calling Conventions Reference Manual* (09740-90015) for a description of export stubs.)

```
$nmdebug > tr,unw
     PC=115.00005b50 processstudent.highscore
* 0) SP=40221180 RP=115.00005f0c processstudent+$1e8

   Can't Unwind: 0  Entry-FR: 00  Call_FR: 00       Region: Normal
```

```
     Millicode: 0  Entry-GR: 00  Call_GR: 00   Frame-size: 6 (dbl words)
Large-Frame-R3: 0  Save-SRs: 00  Save-SP:  0     Save-MRP: 0
      Save-SR0: 0   Cleanup:  0  Save-RP:  0  Args-stored: 1
Interrupt-Mrkr: 0

 1) SP=40221180 RP=115.00006b1c PROGRAM+$300

  Can't Unwind: 0  Entry-FR: 00  Call_FR: 00       Region: Normal
     Millicode: 0  Entry-GR: 03  Call_GR: 00   Frame-size: 10 (dbl words)
Large-Frame-R3: 0  Save-SRs: 00  Save-SP:  1     Save-MRP: 0
      Save-SR0: 0   Cleanup:  0  Save-RP:  1  Args-stored: 1
Interrupt-Mrkr: 0

 2) SP=40221100 RP=115.00000000

  Can't Unwind: 0  Entry-FR: 00  Call_FR: 00       Region: Normal
     Millicode: 0  Entry-GR: 00  Call_GR: 00   Frame-size: c (dbl words)
Large-Frame-R3: 0  Save-SRs: 00  Save-SP:  1     Save-MRP: 0
      Save-SR0: 0   Cleanup:  0  Save-RP:  1  Args-stored: 0
Interrupt-Mrkr: 0

    (end of NM stack)
```

Native mode stack trace relies on the presence of unwind descriptors as produced by the language compilers. Without these information blocks, a stack trace would not be possible. The UNWIND option is used to display the unwind descriptor associated with each procedure. (Refer to the *Procedure Calling Conventions Reference Manual* (09740-90015) for a description of unwind descriptors.)

```
 $nmdebug > tr,f
      PC=a.0074da24 NL.PUB.SYS/FWRITE
 * 0) SP=40221260 RP=a.00748150 ?FWRITE+$8
      DP=c0200008 PSP=40221260 PCPRIV=0
        export stub:
  f4.0012d044 XL.PUB.SYS/P_FLUSHLINE+$54
   1) SP=40221260 RP=f4.00139560 P_WRITELN+$20
      DP=40200648 PSP=40221200 PCPRIV=3
   2) SP=40221200 RP=f4.00139630 P_WRITELN+$9c
```

```
    DP=40200648 PSP=402211c8 PCPRIV=3
 3) SP=402211c8 RP=f4.0013950c ?P_WRITELN+$8
    DP=40200648 PSP=40221180 PCPRIV=3
      export stub: 115.00005e30 GRADES.DEMO.TELESUP/processstudent+$10c
 4) SP=40221180 RP=115.00006b1c PROGRAM+$300
    DP=40200008 PSP=40221100 PCPRIV=3
 5) SP=40221100 RP=115.00000000
    DP=40200008 PS
P=402210a0 PCPRIV=3
    (end of NM stack)
```

A FULL stack trace displays the value of DP, PSP and the privilege level (0-3 for each level in the stack).

```
$nmdebug > tr 2,single
  2) SP=40221200 RP=f4.00139630 P_WRITELN+$9c
```

Display only stack level 2.

```
$nmdebug > tr
     PC=a.006777fc trap_handler
* 0) SP=40221338 RP=a.002a1fec conditional+$ac
  1) SP=40221338 RP=a.000a5040 hpe_interrupt_marker_stub
 --- Interrupt Marker


$nmdebug > tr,ism
     PC=a.006777fc trap_handler
* 0) SP=40221338 RP=a.002a1fec conditional+$ac
  1) SP=40221338 RP=a.000a5040 hpe_interrupt_marker_stub
 --- Interrupt Marker
  2) SP=402211e8 RP=25d.00015134 small_divisor+$8
 --- End Interrupt Marker Frame ---

     PC=25d.00015134 small_divisor+$8
  0) SP=402211e8 RP=25d.00015d38 average+$b0
  1) SP=402211e8 RP=25d.00015c74 ?average+$8
       export stub: 25c.00005d98 processstudent+$74
  2) SP=40221180 RP=25c.00006b1c PROGRAM+$300
  3) SP=40221100 RP=25c.00000000
```

```
     (end of NM stack)
$nmdebug >
```

In the above example, the first stack trace encounters an interrupt marker and stops tracing. The second stack trace uses the ISM option to continue tracing past the interrupt marker. The interrupt that caused the interrupt marker to be generated was caused by a divide by zero in the small_divisor routine.

CM Examples

```
%cmdebug > tr
      PROG %   0.1421   PROCESSSTUDENT+14      (mITroc CCG)  SEG'
*  0) PROG %   0.2004   PROCESSSTUDENT+377     (mITroc CCG)  SEG'
   1) PROG %   0.253    OB'+253                (mITroc CCG)  SEG'
   2) SYS  %  25.0      ?TERMINATE             (MItroc CCG)  CMSWITCH
```

Display a CM stack trace. The first line indicates the address CMPC points to. Each stack marker is formatted, starting from the top of stack and working down the depth of the stack. Level numbers are indicated on the left; an asterisk marks the current level. (Refer to the LEV command.)

```
%cmdebug > tr,f
      PROG %   0.1421   PROCESSSTUDENT+14      (CSTX 1)  SEG'
      X=22750   P=1421    Status=(mITroc CCG 301)  DeltaQ=13670
*  0) PROG %   0.2004   PROCESSSTUDENT+377     (CSTX 1)  SEG'
      X=6       P=2004    Status=(mITroc CCG 301)  DeltaQ=14
   1) PROG %   0.253    OB'+253                (CSTX 1)  SEG'
      X=36      P=253     Status=(mITroc CCG 301)  DeltaQ=10
   2) SYS  %  25.0      ?TERMINATE             (CST 26)  CMSWITCH
      X=0       P=0       Status=(MItroc CCG 026)  DeltaQ=4
```

The above examples specifies the FULL option to display the value of the X, P, and status registers, and the DELTA-Q value.

Translated Code Examples

```
Break at: NM    [1] TRANS 24.00854ea4 PASCAL'LIBRARY2:?P'WRITESTR
$nmdebug > tr ,dual
       PC=24.00854ea4 PASCAL'LIBRARY2:?P'WRITESTR
NM* 0) SP=40221290 RP=a.0067320c outer_block+$e8
NM  1) SP=402210a0 RP=a.00000000 inx_A0000+$14
     (end of NM stack)
```

The above example shows Debug stopping at a breakpoint. The breakpoint was set in SL.PUB.SYS at the entry point to the P'WRITESTR routine. Since the system SL is translated, Debug set two breakpoints (one in the CM emulated code and one in the translated NM code). The NM translated code breakpoint is encountered, and so Debug stops.

A stack trace reveals that the process is indeed stopped at the entry point to P'WRITESTR, but no other recognizable markers appear. This is because translated code does not actually switch to CM mode, so no switch markers exist to enable the DUAL option to function. However, the CM stack is maintained as if the code were being run by the emulator. Switching to cmdebug and performing a stack trace reveals this.

```
$nmdebug > cm
%cmdebug > tr
      SYS  %  36.15626  ?P'WRITESTR            (mITroc CCG)  PASCAL'LIBRARY2
*  0) PROG %   0.1737   PROCESSSTUDENT+%332    (mITroc CCG)  SEG'
   1) PROG %   0.253    OB'+%253               (mITroc CCG)  SEG'
   2) SYS  %  25.0      ?TERMINATE             (MItroc CCG)  CMSWITCH
```

The above trace shows all of the CM procedures that are active on the stack. Remember, the CM stack is maintained even if the code is running translated.

Dual Mode Examples

```
$nmdat > tr,d
       PC=a.000a4838 enable_int+$20
NM* 0) SP=40201ce0 RP=a.0013cdf0 notify_dispatcher.block_current_process+$294
NM  1) SP=40201ce0 RP=a.0013deec notify_dispatcher+$34c
NM  2) SP=40201c88 RP=a.001dc964 wait_for_active_port+$ec
NM  3) SP=40201c10 RP=a.001dd680 receive_from_port+$450
NM  4) SP=40201bc0 RP=a.00228514 extend_receive+$4d8
NM  5) SP=40201b28 RP=a.00218bdc rendezvousio.get_specific+$194
NM  6) SP=40201a78 RP=a.00218ec8 rendezvousio+$13c
NM  7) SP=40201a08 RP=a.0020f274 attachio.perform_io+$f8
NM  8) SP=402018c8 RP=a.00210414 attachio.terminal_functions+$fac
NM  9) SP=40201838 RP=a.00214d40 attachio+$2e4
NM  a) SP=402017e0 RP=a.0020e3bc ?attachio+$8
         export stub: a.003e30e4 arg_regs+$28
NM  b) SP=402015c8 RP=a.0044db34 nm_switch_code+$f30
NM  c) SP=40201498 RP=a.000a09b0 cm_swtnm_call+$8
       (switch marker frame)
   CM       SYS  %  27.253    SWITCH'TO'NM'+%4       (Mitroc CCG)  SUSER1
   CM *  0) SYS  %  27.253    SWITCH'TO'NM'+%4       (Mitroc CCG)  SUSER1
   CM    1) SYS  %  25.7765   ATTACHIO+%325          (Mitroc CCG)  CMSWITCH
   CM    2) SYS  %  22.17700  DEALLOCATE+%30         (Mitroc CCG)  XLSEG11
   CM    3) SYS  %   3.5540   F'CLOSE'+%4321         (MitroC CCG)  FSSEG3
   CM    4) switch marker                            (Mitroc CCG)
NM  d) SP=40201208 RP=a.000a07bc ?CM_SWITCH+$30
         export stub: a.0044c3e4 switch_to_cm+$c30
NM  e) SP=40201018 RP=a.006f3c84 fclose_nm+$74c
NM  f) SP=40200db0 RP=a.006e62a8 FCLOSE+$368
NM 10) SP=40200aa8 RP=a.0036a0b0 fs_proc_term+$a4
NM 11) SP=40200a00 RP=a.00197550 terminate_process+$318
NM 12) SP=40200948 RP=a.00326fb0 TERMINATE+$28
NM 13) SP=40200668 RP=a.00326a2c ?TERMINATE+$8
         export stub: a.003e30e4 arg_regs+$28
NM 14) SP=40200638 RP=a.0044db34 nm_switch_code+$f30
NM 15) SP=40200508 RP=a.000a09b0 cm_swtnm_call+$8
       (switch marker frame)
   CM    5) SYS  %  27.253    SWITCH'TO'NM'+%4       (MITroc CCG)  SUSER1
```

```
   CM    6) SYS  %  25.5      TERMINATE+%5           (MITroc CCG)  CMSWITCH
   CM    7) PROG %   0.244                           (mITroc CCE)
   CM   10) SYS  %  25.0      ?TERMINATE             (MItroc CCG)  CMSWITCH
NM 16) SP=40200278 RP=a.0030d868 outer_block+$144
NM 17) SP=40200088 RP=a.00000000
     (end of NM stack)
$nmdat >
```

The above example shows an interleaved NM and CM stack trace.

Limitations, Restrictions

The DUAL option is ignored if the current mode is not the same as the original entry mode. (Refer to the ENV ENTRY_MODE command.)

When CM code has been translated, it is not possible to obtain dual mode stack traces. The NM and CM stacks may be traced individually, however.

People debugging the operating system need to be aware of the following limitation. If an interrupt handler is running that has interrupted code running in CM mode, dual stack trace is incorrect. In addition, not all of the CM stack may be shown.

Native mode stack trace depends on the presence and accuracy of unwind descriptors in the program file and libraries to trace stacks. If these descriptors are not present, corrupted, or not correctly sorted, System Debug may produce incorrect stack traces.

DAT is only able to trace the part of the NM stack that corresponds to code in NL.PUB.SYS. If by chance the unwind descriptors of the code that called the NL routines are resident, the stacked procedure calls are displayed all the way to the base of the stack. The names of the procedures in other libraries and program files are not known to DAT.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

TRAP

Debug only

Arms/disarms/lists various traps that are monitored by Debug.

Syntax

```
TRAP [LIST]
TRAP [trap-name] [option]
```

Parameters

trap-name Traps can be classified into several classes. The trap names for each class are presented together. In general, this parameter specifies which trap to arm, disarm, or list. Only enough characters to make the name recognizable are required.

Hardware Traps

These are traps that are documented in the Precision Architecture Control Document (ACD). They are trapped directly by the hardware.

`BRANCH` The `BRANCH` trap is the taken branch trap. Any time a branch instruction is executed the debugger stops.

MPE XL X-Traps

These traps correspond to the MPE XL user intrinsics of similar name. (Refer to the *MPE XL Intrinsics Reference Manual* (32650-90028) for descriptions of the each of these traps.) By arming these traps, the debugger obtains control of the process before the system trap mechanism. You may have the system ignore the trap (pretend it never happened) or process it as if the debugger had not been notified.

To have the trap ignored use the `C[ontinue] IGNORE` command.

Typing `C[ontinue]` or `C[ontinue] NOIGNORE` causes the trap subsystem to process the trap as if Debug has not been notified.

`XARITHMETIC`	The trap mask indicating the cause of the trap is displayed.
`XCODE`	The code trap number is displayed.
`XLIBRARY`	Not implemented.
`XSYSTEM`	Not implemented.

Refer to the *MPE XL Intrinsics Reference Manual* (32650-90028) for a description of the format of the various trap masks and codes displayed by Debug when one of the above traps is encountered.

Trace Traps

The currently defined trace events are based on compiler generated breakpoints. These breakpoints are inserted into the code by the compilers only if the symbolic debug compiler option is used. If the debugger arms any of these events, it stops at the indicated event.

BEGIN_PROCEDURE	Stop at the entry to procedure.
END_PROCEDURE	Stop at the exit from procedure.
LABELS	Stop at all labels.
STATEMENTS	Stop at each source statement (requires compiler support).
EXIT_PROGRAM	Stop at the program exit point.
ENTER_PROGRAM	Stop at the program entry point.
TRACE_ALL	All of the trace events.

option Three options are supported. If none is given, LIST is assumed.

LIST	List the current setting of the trap(s).
ARM	Arm the indicated trap(s).
DISARM	Disarm the indicated trap(s).

Examples

```
$nmdebug > trap list
XLIBRARY         DISABLED
XARITHMETIC      DISABLED
XSYSTEM          DISABLED
XCODE            DISABLED
BRANCH           DISABLED
BEGIN_PROCEDURE  DISABLED
END_PROCEDURE    DISABLED
LABELS           DISABLED
STATEMENTS       DISABLED
ENTER_PROGRAM    DISABLED
EXIT_PROGRAM     DISABLED
```

List the status of all the defined traps (initial status is disabled).

TRAP

```
$nmdebug > trap branch arm
```

Arm the branch taken trap and the arithmetic traps.

```
$nmdebug > trap
XLIBRARY          DISABLED
XARITHMETIC       DISABLED
XSYSTEM           DISABLED
XCODE             DISABLED
BRANCH            DISABLED
BEGIN_PROCEDURE   DISABLED
END_PROCEDURE     ARMED
LABELS            DISABLED
STATEMENTS        DISABLED
ENTER_PROGRAM     DISABLED
EXIT_PROGRAM      DISABLED
```

Show the status of the traps.

```
$nmdebug > c
Branch Taken at: 6a8.00005d84 processstudent+$60
 to: 6a8.000056b8 lr_wa_10

$nmdebug > c
Branch Taken at: 6a8.00005708 lr_wa_1+$8
 to: 6a8.00005d88 processstudent+$64

$nmdebug > c
Branch Taken at: 6a8.00005d94 processstudent+$70
 to: 6a8.00005990 ?_start+$3c

$nmdebug > c
Branch Taken at: 6a8.000059ac ?_start+$58
 to: a.fff7b004

$nmdebug > c
Branch Taken at: a.fff7b024
 to: 730.00015c6c ?average
```

The above example shows the use of the branch taken trap. Every time any form of branch instruction is executed, Debug stops just before the branch occurs.

```
$nmdebug > trap xari arm
$nmdebug > trap xari list
XLIBRARY         ARMED

$nmdebug > c
XARI Trap at: 730.00015d38 average+$b0
trap mask = 00000002

$nmdebug > wl pc,#13
GRP $730.15d38
$nmdebug > dc pc-20,#13
GRP $730.15d18
00015d18  average+$90  b6b60802  ADDIO    1,21,22
00015d1c  average+$94  6bd63f81  STW      22,-64(0,30)
00015d20  average+$98  e81f1f77  B,N      average+$58
00015d24  average+$9c  20000009  ** Stmt  9
00015d28  average+$a0  4bc13ee9  LDW      -140(0,30),1
00015d2c  average+$a4  b4390fff  ADDIO    -1,1,25         /* Trap occurred in
00015d30  average+$a8  ebff0595  BL       divoI,31        /* <-- this routine.
00015d34  average+$ac  4bda3f89  LDW      -60(0,30),26
00015d38  average+$b0  4bdf3ed9  LDW      -148(0,30),31   /* <-- PC is here
00015d3c  average+$b4  6bfd0000  STW      29,0(0,31)
00015d40  average+$b8  e840c000  BV       0(2)
00015d44  average+$bc  37de3f31  LDO      -104(30),30

$nmdebug > dr r29
R29=$0

$nmdebug > mr r29 4
R29=$0 := $4

$nmdebug > c ignore
```

The above example starts by arming the `XARI` trap. The process is allowed to run. During execution, an arithmetic trap was detected. Debug stops to allow the user to inspect the state of the process. After viewing the code, it can be seen that the trap occurred in the `divoI` millicode routine. By analyzing the trap mask it is determined that the trap was caused by attempting to divide by zero. The millicode divide routine returns the result of its operation in general register 29.

After looking at the source code, the bug in the program was discovered. It was determined that at this point in process execution, the result of the divide should have been "4". The millicode return register is updated with the correct value. The continue command with the `IGNORE` option is issued to resume the process as if the trap never happened. (If the `IGNORE` option had been specified, the process would have been terminated by the trap subsystem.)

Limitations, Restrictions

The `XLIBRARY` and `XSYSTEM` trace traps are not implemented.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

UF

Debug only

Unfreezes a code segment, data segment, or virtual address (range) in memory.

Syntax

```
UFC    logaddr  [bytelength]           Program file
UFCG   logaddr  [bytelength]           Group library
UFCP   logaddr  [bytelength]           Account library
UFCLG  logaddr                         Logon group library
UFCLP  logaddr                         Logon account library
UFCS   logaddr  [bytelength]           System library
UFCU   fname  logaddr [bytelength]     User library

UFCA   cmabsaddr                       Absolute CST
UFCAX  cmabsaddr                       Absolute CSTX

UFDA   dst.off                         CM data segment

UFVA   virtaddr [bytelength]           Virtual address
```

These unfreeze commands actually decrement a system freeze count. The segment or pages may remain frozen if their freeze count is still positive.

Parameters

logaddr A full logical code address (`LCPTR`) specifies three necessary items:

- The logical code file (`PROG`, `GRP`, `SYS`, and so on).
- NM: the virtual space ID number (SID).
  CM: the logical segment number.

- NM: the virtual byte offset within the space.
  CM: the word offset within the code segment.

Logical code addresses can be specified in various levels of detail:

- As a full logical code pointer (LCPTR):

`UFC procname+20`	Procedure name lookups return `LCPTR`s.
`UFC pw+4`	Predefined `ENV` variables of type `LCPTR`.
`UFC SYS(2.200)`	Explicit coercion to a `LCPTR` type.

- As a long pointer (LPTR):

  UFC 23.2644 *sid.offset* or *seg.offset*

  The logical file is determined based on the command suffix. For example:

  UFC implies PROG.
  UFCG implies GRP.
  UFCS implies SYS, and so on.

- As a short pointer (SPTR):

  UFC 1024 *offset* only

For NM, the short pointer offset is converted to a long pointer using the function STOLOG, which looks up the SID of the loaded logical file. This is different from the standard short to long pointer conversion, STOL, which is based on the current space registers (SRs).

For CM, the current executing logical segment number and the current executing logical file are used to build a LCPTR.

The search path used for procedure name lookups is based on the command suffix letter:

UFC	Full search path: NM: PROG, GRP, PUB, USER(s), SYS. CM: PROG, GRP, PUB, LGRP, LPUB, SYS.
UFCG	Search GRP, the group library.
UFCP	Search PUB, the account library.
UFCLG	Search LGRP, the logon group library.
UFCLP	Search LPUB, the logon account library.
UFCS	Search SYS, the system library.
UFCU	Search USER, the user library.

For a full description of logical code addresses, refer to the section "Logical Code Addresses" in chapter 2.

fname The file name of the NM USER library. Multiple NM libraries can be bound with the XL= option on a RUN command, for example:

```
:run nmprog; xl=lib1,lib2.testgrp,lib3
```

In this case, it is necessary to specify the desired NM USER library, for example:

```
UFCU lib1 204c
UFCU lib2.testgrp test20+1c0
```

If the file name is not fully qualified, the following defaults are used:

Default account: the account of the program file.
Default group: the group of the program file.

cmabsaddr A full CM absolute code address specifies three necessary items:

- Either the CST or the CSTX.
- The absolute code segment number.
- The CM word offset within the code segment.

Absolute code addresses can be specified in two ways:

- As a long pointer (LPTR):

`UC 2644`	Implicit CST 23.2644
`UCAX 5.3204`	Implicit CSTX 5.3204

- As a full absolute code pointer (ACPTR):

`UCA CST(2.200)`	Explicit CST coercion.
`UCAX CSTX(2.200)`	Explicit CSTX coercion.
`UCAX logtoabs(prog(1.20))`	Explicit absolute conversion.

The search path used for procedure name lookups is based on the command suffix letter:

UCA	GRP, PUB, LGRP, LPUB, SYS
UCAX	PROG

dst.off A data segment address (specified as *dst.offset*) of the data segment to be unfrozen in memory (see the FDA command).

virtaddr The starting virtual address of the page(s) that are to be unfrozen in memory. (Refer to the FVA command.) *Virtaddr* can be a short pointer, a long pointer, or a full logical code pointer.

bytelength This parameter is valid only for nmdebug. It is the desired number of bytes to be unfrozen. Based on the starting virtual address and the specified *bytelength*, the appropriate number of virtual pages are unfrozen. If omitted, four bytes is used as a default. The implementation of this command dictates that the smallest unit that is actually frozen is one page of virtual memory. That is, if you say one byte, the whole page on which that byte resides is made resident.

Examples

```
%cmdebug > ufc sys(12.0)
```

Unfreeze CM logical code segment SYS %12.

```
$nmdebug > ufva 22.104, 1000
```

Unfreeze 1000 bytes starting at virtual address 22.104.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

UNMAP

Closes (unmaps) a file that was opened by the `MAP` command.

Syntax

```
UNMAP index
```

Parameters

index The mapped file index number (displayed with the `MAP` and `MAPLIST` commands).

Examples

```
$nmdebug > mapl
1  DTCDUMP.DUMPUSER.SUPPORT     1000.0  Bytes = 43dc
2  DTCDUMP2.DUMPUSER.SUPPORT    1001.0  Bytes = c84
3  MYFILE.MYGROUP.MYACCT        1005.0  Bytes = 1004

$nmdebug > unmap 2
$nmdebug > unmap mapindex("dtcdump.dumpuser.support")

$nmdebug > mapl
1  DTCDUMP.DUMPUSER.SUPPORT     1000.0  Bytes = 43dc
3  MYFILE.MYGROUP.MYACCT        1005.0  Bytes = 1004
```

Close the file `DTCDUMP2.DUMPUSER.SUPPORT`. Also, close the file `DTCDUMP.DUMPUSER.SUPPORT` (by calling the `MAPINDEX` function that returns the file index number 1).

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

UPD

Updates the windows.

Syntax

```
UPD
```

Parameters

none

Examples

```
%cmdebug > UPD
```

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

USE

System Debug commands can be executed from a file with the `USE` command.

Syntax

```
USE
USE  [filename]  [count]
USENEXT  count
USE [CLOSE][ALL | @]
```

`USE`, entered alone, displays the current open command file(s) and the current line position within the file (current-record/total records).

`USE` *filename* opens the specified file, executes all commands from that file, and then closes the file. An optional *count* parameter is used to read a particular number of lines from the file before returning to interactive user input. If *count* is less than the total number of lines in the file, the file remains open and pending.

`USENEXT count` reads the next *count* lines from the most recently opened file, and once again returns to interactive input.

Up to five command files can be opened at one time; command files are maintained in a stack, and each has its own remaining *count*.

`USE CLOSE` closes (saves) the most recently (still opened) command file. Since files are automatically closed when completed, this is necessary only for partially executed command files.

`USE CLOSE ALL` or `CLOSE @` closes (saves) all (still opened) command files.

Command lines executed from USE files are not displayed, unless the user has explicitly set the environment variable `ECHO_USE`. (Refer to the `ENV ECHO_USE` command.)

Parameters

filename	The file name of the command file that is to be opened and executed. Command files must be ASCII files. If omitted, the status of all open command files is displayed.
count	The number of lines to be executed from the command file. If omitted, all lines in the file are executed, and the file is closed.
`USENEXT` *count*	Executes the next *count* lines from the most recently opened command file.
`USE CLOSE`	Closes the most recently (still opened) command file. The keyword `CLOSE` can be entered in uppercase or lowercase.
`USE CLOSE ALL` `USE CLOSE @`	Closes all (still opened) command files. The keywords `CLOSE` and `ALL` can be entered in uppercase or lowercase.

Examples

```
%cmdebug > use macros
```

Opens the file `macros`, executes all commands from the file, and then closes the file (as is).

```
%cmdebug > use macros 10
```

Opens the file `macros` and executes the first 10 lines from the file, then returns to normal interactive input.

```
%cmdebug > usenext 5
```

Use the next five lines from the current USE file.

```
%cmdebug > use
USE file "macros" OPEN:  15/76
```

Displays the current status of open command files. The file `macros` is opened and positioned at line 15 out of 76 lines.

```
%cmdebug > use close
```

Closes the current open USE file. Note that other nested USE files may still be left open.

Limitations, Restrictions

Command files should be typical unnumbered editor files, ASCII, with a fixed record size less than 256 bytes. Line numbers are not stripped.

There is currently a limit of five nested USE files.

Command lines that are executed from USE files are placed into the command history stack. Long USE files often displace all of the current commands in the stack out of accessible range.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

VAR

Defines a user-defined variable.

Syntax

```
VAR var_name [:var_type] [=] var_value
```

The entire set of currently defined variables can be saved into a binary file for later restoration. (Refer to the STORE and RESTORE commands.)

Parameters

var_name The name of the variable that is being defined. Names must begin with an alphabetic character and are restricted to thirty-two (32) characters, (characters must be alphanumeric, "_", "'", or "$"). Longer names are truncated with a warning. Names are case insensitive.

var_type The type of the variable. The following types are supported:

STR	String
BOOL	Unsigned 16-bit
U16	Unsigned 16-bit
S16	Signed 16-bit
U32	Unsigned 32-bit
S32	Signed 32-bit
S64	Signed 64-bit
SPTR	Short pointer
LPTR	Long pointer
PROG	Program logical address
GRP	Group library logical address
PUB	Account library logical address
LGRP	Logon group library logical address
LPUB	Logon account library logical address

SYS	System library logical address
USER	User library logical address
TRANS	Translated CM code virtual address
EADDR	Extended address
SADDR	Secondary address

If the type specification is omitted, the type is assigned automatically, based on *var_value*.

The optional *var_type* allows the user to explicitly specify the desired internal representation for *var_value* (that is, signed or unsigned, 16 bit or 32 bit) for this particular assignment only. It does *not* establish a fixed type for the lifetime of this variable. A new value of a different type can be assigned to the same variable (name) by a subsequent `VAR` command.

var_value — The new value for the variable, which can be an expression. An optional equal sign "`=`" can be inserted before the variable value.

Examples

```
%cmdebug > var save 302.120
```

Define variable `save` to be the address 302.120. By default, this variable is of type `LPTR` (long pointer) based on the value 302.120.

```
$nmdebug > var count=1c
```

Define variable `count` to be the value 1c.

```
$nmdebug > var s1:str="this is a string"
```

Define variable `s1` to be of type `STR` (string) and assign the value "this is a string".

```
$nmdebug > varlist
var save:lptr    %302.120
var count:u32    $1c
var s1:str       this is a string
```

Display all currently defined user variables.

Limitations, Restrictions

Refer to `ENV VARS`, `ENV VARS_LOC`, and `ENV VARS_LIMIT`. These environment variables determine the maximum number of variables that can be defined.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the

exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

VARD[EL]

Variable delete. Deletes the specified user-defined variable(s).

Syntax

```
VARD[EL] pattern
```

Parameters

pattern The name of the variable(s) to be deleted.

This parameter can be specified with wildcards or with a full regular expression. Refer to appendix A for additional information about pattern matching and regular expressions.

The following wildcards are supported:

`@`	Matches any character(s).
`?`	Matches any alphabetic character.
`#`	Matches any numeric character.

The following are valid name pattern specifications:

`@`	Matches everything; all names.
`pib@`	Matches all names that start with "pib".
`log2##4`	Matches "log2004", "log2754", and so on.

The following regular expressions are equivalent to the patterns with wildcards that are listed above:

```
`.*`
`pib.*`
`log2[0-9][0-9]4`
```

Examples

```
%cmdebug > vardel  count
```

Delete the variable `count`.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

VARL[IST]

Variable list. Lists the value(s) for the specified user-defined variable(s).

Syntax

```
VARL[IST]  [pattern]
```

Variables are always listed in alphabetical order.

Parameters

pattern The name of the variable(s) to be listed.

This parameter can be specified with wildcards or with a full regular expression. Refer to appendix A for additional information about pattern matching and regular expressions.

The following wildcards are supported:

`@`	Matches any character(s).
`?`	Matches any alphabetic character.
`#`	Matches any numeric character.

The following are valid name pattern specifications:

`@`	Matches everything; all names.
`pib@`	Matches all names that start with "pib".
`log2##4`	Matches "log2004", "log2754", and so on.

The following regular expressions are equivalent to the patterns with wildcards that are listed above:

```
'.*'
'pib.*'
'log2[0-9][0-9]4'
```

By default, all user-defined variables are listed.

Examples

```
%cmdebug > varlist
var count : u32  = $1c
var save  : lptr = %302.120
var s1    : str  = this is a string
```

Display all currently defined user variables.

```
%nmdebug > varl s1@
var save  : lptr = %302.120
var s1    : str  = this is a string
```

Display all variables that begin with the letter "s".

Limitations, Restrictions

Variables are not currently listed in sorted alphabetical order.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

W (write)

Writes a list of values, with optional formatting, to output.

Syntax

```
W     valuelist
WL    valuelist
WP    valuelist

WCOL  column
WPAGE
```

`W` (Write), `WL` (Writeln), and `WP` (Prompt) write a list of values, with optional formatting, to output.

`WP` (Prompt) appends the new formatted values to the output buffer, flushes the buffer to output, and maintains the cursor on the same line.

`W` (Write) appends the new formatted values to the output buffer and advances the current buffer position.

`WL` (Writeln) appends the new formatted values to the output buffer, then flushes the buffer to output with a new line. The output buffer is reset.

`WCOL` advances the current output buffer position to the specified column position, blank-filling as necessary if the new position effectively expands the buffer.

`WPAGE` forces all buffered output to be flushed, and a page eject is emitted. The output buffer is reset.

Parameters

valuelist An arbitrary list of values to be written. Values can be separated by blanks or with commas:

```
value1, value2 value3 ...
```

An optional format specification can be appended to each value in the list in order to select specific output base, left or right justification, blank or zero fill, and field width for that value.

```
value1[:fmtspec1] value2[:fmtspec2] ...
```

A format specification is a string list of selected format directives, with individual directives separated by commas or blanks:

```
"directive1,directive2 directive3 ..."
```

W (write)

The following table lists the supported format directives; they can be entered in uppercase or lowercase:

`+`	Current output base (`$`, `#`, or `%` prefix displayed).
`-`	Current output base (no prefix).
`+<`	Current input base (`$`, `#`, or `%` prefix displayed).
`-<`	Current input base (no prefix).
`$`	Hex output base ($ prefix displayed).
`#`	Decimal output base (# prefix displayed).
`%`	Octal output base (% prefix displayed).
`H`	Hex output base (no prefix).
`D`	Decimal output base (no prefix).
`O`	Octal output base (no prefix).
`A`	ASCII base (use "." for nonprintable chars).
`N`	ASCII base (loads actual nonprintable chars).
`L`	Left-justified.
`R`	Right-justified.
`B`	Blank-filled.
`Z`	Zero-filled.
`M`	Minimum field width, based on value.
`F`	Fixed field width, based on the type of value.
`Wn`	User specified field width *n*.
	.
`Cn`	Position the output starting at column *n*.
`T`	Typed (display the type of the value).
`U`	Untyped (do not display the type of the value).
`QS`	Quote single (surround w/ single quotes).
`QD`	Quote double (surround w/ double quotes).
`QO`	Quote original (surround w/ original quote character).
`QN`	Quote none (no quotes).

The `M` directive (minimum field width) selects the minimum possible field width necessary to format all significant digits (or characters in the case of string inputs).

The `F` directive (fixed field width) selects a fixed field width based on type of the value and the selected output base. Fixed field widths are listed in the following table:

	hex($,H)	dec(#,D)	oct(%,0)	ascii(A,N)
`S16,U16`	4	6	6	2
`$32,U32`	8	10	11	4
`S64`	16	20	22	8
`SPTR`	8	10	11	4
`LPTR` Class	8.8	10.10	11.11	8
`EADDR` Class	8.16	10.20	11.22	12
`STR`	field width = length of the string.			

The `Wn` directive (variable field width) allows the user to specify the desired field width. The `W` directive can be specified with an arbitrary expression. If the specified width is less than the minimum necessary width to display the value, the user width is ignored, and the minimum width used instead. All significant digits are always printed. For example:

```
number:"w6", or
number:"w2*3"
```

The number of positions specified (either by `Wn` or `F`) does not include the characters required for the radix indicator (if specified) or sign (if negative). Also, the sign and radix indicator are always positioned just preceding the first (leftmost) character.

Zero versus blank fill applies to leading spaces (for right justification) only. Trailing spaces are always blank filled.

In specifications with quotes, the quotes do not count in the number of positions specified. The string is built such that it appears inside the quotes as it would without the quotes.

The `T` directive (typed) displays the type of the value, preceding the value. The `U` directive (untyped) suppresses the display of the type. Types are displayed in uppercase, with a single trailing blank. The width of the type display string varies, based on the type, and it is independent of any specified width (`M`, `F`, or `Wn`) for the value display.

For values of type `LPTR` (long pointer, *sid.offset*, or *seg.offset*) and `EADDR` (extended address, *sid.offset* or *ldev.offset*), two separate format directives can be specified. Each is separated by a dot, ".", to indicate individual formatting choices for the "*sid*" portion and the "*offset*" portion. This is true for all code pointers (`ACPTR` - absolute code pointers: `CST`, `CSTX`; `LCPTR` - logical code pointers: `PROG, GRP, PUB, LGRP, LPUB, SYS, USER, TRANS`). For example:

```
pc:"+.-, w4.8, r.l, b.z"
```

The following default values are used for omitted format directives. Note that the default format directives depend on the type of value to be formatted:

```
value type                  default format
----------                  --------------

STR, BOOL                   - R B M U
U16,S16,U32,S32,S64         + R B M U
SPTR                        + R Z F U
LPTR                        +.-  R.L  B.Z  M.F  U
 ACPTR    LCPTR             +.-  R.L  B.Z  M.F  T
  CST      PROG             +.-  R.L  B.Z  M.F  T
  CSTX     GRP              +.-  R.L  B.Z  M.F  T
           PUB              +.-  R.L  B.Z  M.F  T
           LGRP             +.-  R.L  B.Z  M.F  T
           LPUB             +.-  R.L  B.Z  M.F  T
           SYS              +.-  R.L  B.Z  M.F  T
           USER             +.-  R.L  B.Z  M.F  T
           TRANS            +.-  R.L  B.Z  M.F  T
EADDR                       +.-  R.L  B.Z  M.F  U
 SADDR                      +.-  R.L  B.Z  M.F  T
```

Note that absolute code pointers, logical code pointers and extended addresses display their types (T) by default. All other types default to untyped (U).

The `Cn` (column *n*) directive moves the current output buffer position to the specified column position prior to the next write into the output buffer. Column numbers start at column 1. For example:

```
number:"c6"
```

Note The `Cn` directive is ignored by the `ASC` function but is honored by the `W`, `WL` and `WP` commands.

Examples

```
$nmdat > var cost 100

$nmdat > w  "the price is "
$nmdat > w  cost
$nmdat > wl " for the goodies."
the price is $100 for the goodies
$nmdat > wl "the price is ", cost, " for the goodies."
the price is $100 for the goodies
```

Two different methods of writing mixed text and formatted numbers.

```
$nmdat > var number:u32=123

$nmdat > wl number
$123
$nmdat > wl number:"-"
123
$nmdat > wl number:"#"
#291
$nmdat > wl number:"d"
291
$nmdat > wl number:"f,r"
     $123
$nmdat > wl number:"r,w6,- z"
$nmdat > wl number:"r,w6,- z t"
U32 000123
```

Several examples of formatting an unsigned 32-bit value.

```
$nmdat > var test='test'

$nmdat > wl test
test
$nmdat > wl test:"t"
STR test
$nmdat > wl test:"+"
$test
```

```
$nmdat > wl test:"w2"
test
$nmdat > wl test:"w8,r"
    test
$nmdat > wl test:"w8, r qd"
"    test"
```

Several examples of formatting a string.

```
$nmdat > var long 2f.42c8

$nmdat > wl long
$2f.42c8
$nmdat > wl long:"t"
LPTR $2f.42c8
$nmdat > wl long:"-.+"
2f.$42c8
$nmdat > wl long:"#.$,m.m"
#47.$42c8
$nmdat > wl long:"r.r f.m z"
      $2f.42c8
$nmdat > wl long:"r.r,w6.6,z.z"
$00002f.0042c8
$nmdat > wl long:"r.r w6.6, z.z, qd"
"$00002f.0042c8"
$nmdat > wl long:"r.r w6.6, b.b, $.$"
    $2f.  $42c8
$nmdat > wl long:"r.l w6.6, b.b, $.$"
$2f    .  $42c8
```

Several examples of formatting a long pointer.

```
$nmdat > wcol 6
      $nmdat > wcol 3
   $nmdat > wcol 6; w 12345; wcol 2; wl 2
 2    $12345

$nmdat > wl '2':'c2' '6':"c6" "4":'c4' "<-- column control":"c8"
 2 4 6 <-- column control

$nmdat > w "123456 <-- column control";wl " ":"c1", " ":"c3", " ":"c5"
 2 4 6 <-- column control
```

These examples demonstrate how the output buffer can be positioned to a specific column number. In the first sequence, the WCOL command is used to specify a new column position. Note that the prompt forces the buffer to be output, and consequently may appear in an unexpected position immediately after a WCOL command.

In the second sequence, the `Cn` column directive is used to specify a column position for each formatted value. The third example demonstrates how portions of the output buffer may be overwritten by new formatted values.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

WHELP

Displays online help messages for the window commands.

Syntax

```
WHELP
```

Parameters

None

Limitations, Restrictions

An overview of the window commands is generated with this command. You may type `HELP` *windowcommand* for specific details on any window command.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

WHILE

While *condition* evaluates to TRUE, executes all commands in *cmdlist*.

Syntax

```
WHILE condition DO cmdlist
```

Parameters

condition A logical expression to be evaluated.

cmdlist A command list (or a single command) executed while condition evaluates to TRUE.

Examples

```
$nmdebug > var n 7
$nmdebug > while n > 0 do {wl n; var n n-1}
7
6
5
4
3
2
1
```

A simple while loop example.

```
$nmdebug > while [pc] >> $10 <> $2000 do ss
```

Single step until the next Pascal/XL statement number.

Limitations, Restrictions

none

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

XL

The `XL` command is a predefined alias for the `PSEUDOMAP` command.

Syntax

```
XL  alias for PSEUDOMAP
```

XLD

Closes files opened with the PSEUDOMAP command.

Syntax

```
XLD   localfile
```

The XLD command removes the specified file previously mapped with the PSEUDOMAP command. The file name given is that of the local disk file, not the loaded file name that was associated with it. File names must be fully qualified.

Related commands: PSEUDOMAP, MAPLIST

Parameters

localfile	The fully qualified name of the file to be unmapped.

Examples

```
$nmdat> xld store.abuild00.official
```

Remove store.abuild00.official from the list of files

Limitations, Restrictions

None.

Caution The output format of all System Debug commands is subject to change without notice. Programs that are developed to postprocess System Debug output should not depend on the exact format (spacing, alignment, number of lines, uppercase or lowercase, or spelling) of any System Debug command output.

XLL

The `XLL` command is a predefined alias for the `MAPLIST` command.

Syntax

```
XLL     alias for MAPLIST
```

5

Symbolic Formatting Symbolic Access

Most of the time spent in the debugging of programs and the analysis of system dumps is in the interpretation of data found in memory images. The symbolic formatter provides a powerful and efficient way of referencing this data symbolically and displaying it using its declared type(s). Regardless of the source language, all data are formatted using a Pascal-style syntax.

Most examples used in this section are based upon the following types:

```
CONST        MINGRADES   = 1;      MAXGRADES   = 10;
             MINSTUDENTS = 1;      MAXSTUDENTS = 5;

TYPE
   GradeRange    = MINGRADES .. MAXGRADES;
   GradesArray   = ARRAY [ GradeRange ] OF integer;

   Class         = ( SENIOR, JUNIOR, SOPHOMORE, FRESHMAN );
   NameStr       = string[8];

   StudentRecord = RECORD
                      Name      : NameStr;
                      Id        : integer;
                      Year      : Class;
                      NumGrades : GradeRange;
                      Grades    : GradesArray;
                   END;

TYPE  Subjects   = (ENGLISH, MATH, HISTORY, HEALTH, PHYSED, SCIENCE);
      SubjectSet = SET of subjects;

TYPE  MStype = (MARRIED, DIVORCED, SINGLE, WIDOWED);
```

```
PersonPtr = ^Person;
Person = RECORD
             Next  : PersonPtr;
             Name  : string[16];
             Sex   : (MALE, FEMALE);
             CASE ms   :  MStype OF
                MARRIED   :  (NumKids : integer);
                DIVORCED  :  (HowLong : integer);
                SINGLE    :  (Looking : boolean);
                WIDOWED   :  ();
         END;
```

The following examples assume the System Debug variable *addr1* contains the virtual address of a data structure corresponding to the type `StudentArray`.

A hexadecimal display of that area of memory would be produced by the following:

```
$nmdebug > dv addr1,10
$ VIRT 7b8.40200010 $ 00000004 42696c6c 00000000 00000000
$ VIRT 7b8.40200020 $ 00000001 00040000 0000002d 00000041
$ VIRT 7b8.40200030 $ 0000004e 00000042 00000000 00000000
$ VIRT 7b8.40200040 $ 00000000 00000000 00000000 00000000

$nmdebug > dv addr1,6,a
$ VIRT 7b8.40200010 A .... Bill .... .... .... ....
```

This leaves to the user the task of matching the displayed data to the declared types. When more complicated data structures are involved, it is easy to see that the process of matching the raw data to the corresponding high-level declarations could become exceedingly cumbersome.

The symbolic formatting facility allows users to display data in terms of the declared structures. In the case of the record `StudentRecord` in the above example, the symbolic formatter produces the following output:

```
$nmdebug > fv addr1 "StudentRecord"

RECORD
   NAME      : 'Bill'
   ID        : 1
   YEAR      : SENIOR
   NUMGRADES : 4
   GRADES    :
      [ 1 ]: 2d
      [ 2 ]: 41
      [ 3 ]: 4e
      [ 4 ]: 42
      [ 5 ]: 0
      [ 6 ]: 0
      [ 7 ]: 0
      [ 8 ]: 0
```

```
      [ 9 ]: 0
      [ a ]: 0
END
```

Just as you can display data symbolically, you can also use symbolic addressing to locate and restrict the data to be displayed. The symbolic access facility allows users to extract simple values from a data structure by name for use in expressions and macros. For example, to test if `year` (year in school) is `SENIOR`, one could write:

```
$nmdebug > VAR year = SYMVAL(addr1, "StudentRecord.Year")
$nmdebug > IF year = "SENIOR" THEN WL "He is a SENIOR!!"
```

This is obviously more lucid than the corresponding bit-extraction sequence:

```
$nmdebug > VAR year = BITX( [addr1+$14], 0, #8 )
$nmdebug > IF ( year = 0 ) THEN WL "He is a SENIOR!!"
```

In summary, the symbolic formatting and access facility allows the user to display and reference data in a more natural way, namely through the use of the symbolic data type names declared at the source level. Furthermore, it frees authors of macros and simple formatted displays from worrying about the allocation of data within a data structure and from tracking changes to these structures as they evolve.

The remaining subsections describe the symbolic formatting and access facility in more detail.

Creating and Accessing Symbol Definitions

Before data structures can be accessed symbolically, their definitions must be made known to System Debug. This subsection describes how the symbolic definitions are generated and how they are subsequently made known to System Debug. The final result is a program file containing symbolic type information. Such files are referred to as symbolic data type files or simply symbolic files.

Generate Symbolic Type Information

The generation of symbolic data type definitions begins at compile time through the use of the `$SYMDEBUG 'xdb'$` option in the Pascal compiler. This option causes symbolic debug records to be emitted into the relocatable object modules contained in the relocatable library produced by the compiler. These symbolic debug records fall into two basic categories: those that define the code being generated and those that define the data type shapes and sizes. System Debug at present uses only the data type definitions.

System Debug does not require that the complete program be compiled with the `$SYMDEBUG$` option; instead, only the types and constants need be compiled. However, even though only types and constants are compiled, the outer block *MUST* have at least one statement (for example, x := 1) in order to generate any debug information, and the types and constants must be declared at the level of the outer block. Also, note that symbolic information is currently not emitted when code optimization is performed. The following example shows a compilation of just a program's types for the purpose of obtaining, in object file form, the symbolic information required to use the symbolic formatter.

```
$SYMDEBUG 'xdb'$

PROGRAM gradtyp;

$include 'tgrades.demo.telesup';       { Include all types/constants }

VAR x : integer;

BEGIN                                  { Outer block must have a stmt }
   x := 1;
END.

:COMMENT *** The above program is in the file OGRADTYP.DEMO.TELESUP
:
:PASXL OGRADTYP,YGRADTYP,$NULL
:
:COMMENT *** The above command generates the file "YGRADTYP"
```

Convert The Relocatable Library into a Program File

The relocatable object module(s) generated by the compiler must now be converted into an executable object module (a program file). This step is performed by using the `LINKEDIT` program.

```
:LINKEDIT.PUB.SYS

HPLinkEditor/XL (HP32650-xx.yy.zz) (c) Hewlett-Packard Co 1986

LinkEd> link from=ygradtyp.demo.telesup;to=gradtyp.demo.telesup
LinkEd> exit

:
```

Preprocess the Program File with PXDB

The program file produced by LINKEDIT must be run through a utility called PXDB. This program preprocesses the symbolic debug information for more efficient access during symbolic debugging.

```
:PXDB.PUB.SYS gradtyp.demo.telesup
Copying gradetyp.demo.telesup ... Done
Procedures: 1
Files: 1
:
```

Prepare the Program File with SYMPREP

System Debug needs to perform additional preprocessing of the object module file after PXDB. Quick data type lookup tables are built and symbols are sorted for fast access. The results of this phase are saved in the program file so it need only be performed once.

Once this step is completed, the file is in a form usable by System Debug. Such a file is called a symbolic data type file. This final task is performed from within DAT or DEBUG by using the SYMPREP command:

```
:DAT

DAT XL A.00.00   Copyright Hewlett-Packard Co. 1987.  All rights reserved.


$1 ($0) $nmdebug > SYMPREP gradtyp
Preprocessing GRADTYP.DEMO.TELESUP
Copying file ...
Building Constant lookup table ...
Sorting ...
Building Type lookup table ...
Sorting ...
Building lookup table header ...
Fixing up SOM directory structure ...
GRADTYP.DEMO.TELESUP preprocessed

$2 ($0) $nmdebug >
```

Open the Symbolic Data Type File with SYMOPEN

The System Debug SYMOPEN command is used to access the symbols in a preprocessed program file (symbolic data type file). The user may optionally assign each symbolic file a symbolic name when it is opened. If no symbolic name is specified, the file name (minus the .GROUP.ACCOUNT) is used as the symbolic name. In the following example, the file gradtyp is opened and assigned the default symbolic name gradtyp.

```
$nmdat > SYMOPEN GRADTYPE
$nmdat > SYMFILES
GRADTYP      GRADTYP.DEMO.TELESUP
$nmdat >
```

In summary the following steps must be performed before a symbolic data type file is ready for use by System Debug:

1. Construct a small program which contains all type declarations to be made available to System Debug. The program must have at least one executable statement, and the type declarations must all appear at the level of the outer block.
2. Compile data types with the `$SYMDEBUG 'xdb'$` option.
3. Run the relocatable library generated by the compiler through the Link Editor.
4. Run the program file generated by the Link Editor through `PXDB`.
5. Prepare the modified program file generated by `PXDB` with System Debug `SYMPREP` command.
6. Open the program file with System Debug `SYMOPEN` command.

The Path Specification

Path specifications are used to qualify data structure references to some desired level of granularity.

Syntax

```
[ symname : ] typename [ selector... ][, variantinfo ]
```

Parameters

symname A symbolic name assigned to a symbolic data type file in the `SYMOPEN` command. This parameter specifies the file in which *typename* is to be found. If omitted, the last symbolic file referenced is used.

typename The name of the data structure to be formatted.

selector ... The selectors used to dereference particular components of the data structure identified by *typename*. Multiple selectors are permitted.

The following selectors, based on Pascal syntax, are recognized:

[*index*]	Array selector specifies a component of an array.
.*field*	Record selector specifies a field within a record.
^	Pointer selector specifies pointer dereferencing.

variantinfo A list of variant tag values to be used when formatting tagless variants, or to override the stored tag field if alternate variants are to be displayed. Multiple tag values are specified as a simple list:

vartagvalue [,...]

FINAL TRIM SIZE : 7.5 in x 9.0 in

The Path Specification

For each variant after the *typename* [*selector*] specification, a *vartagvalue* can be given to specify the desired variant. Multiple tag values may be given, separated by commas, to specify tags for nested variants. The order of the tags should match the order of the variants in the type declaration. If tag value(s) are omitted and the tag is not stored as part of the data structure, data are formatted according to the first declared variant.

The variant descriptor can also be used to override stored tag values for variant records. Normally, the symbolic formatter uses stored tags to select the variants to be formatted. However, if the stored tags are corrupt or the user wishes to have the data interpreted according to different variants, *vartagvalue*s may be used to specify the desired variants.

Variable Substitution

System Debug variables may be used within a path specification. Since the path specification is itself composed of a string, any variable substitution must be performed with string variables. In order for a System Debug variable to be recognized in a path specification, it must be preceded by an exclamation mark. For example:

```
$nmdebug > VAR field "ID"
$nmdebug > FT "StudentRecord.!field"

INTEGER
```

The other area where System Debug variables may be used is in array subscripts. In fact, array subscripts may consist of any valid System Debug expression. Exclamation marks are *not* required to dereference variables in this case.

```
$nmdebug > VAR type  "StudentRecord"
$nmdebug > VAR field "Grades"
$nmdebug > VAR index 5

$nmdebug > FV data "!type.!field[ index - 1 ]"
```

```
42

$nmdebug >
```

Case Sensitivity

System Debug normally upshifts all characters in a path specification before searching for names in a symbol file. This is desirable for languages such as Pascal, which emit upshifted symbols. But for languages such as C, which emit symbols with lower-case characters, this automatic upshifting must be disabled. The environmental variable `SYMPATH_UPSHIFT` controls whether or not pathspec upshifting occurs. If your symbol file contains lower-case symbols, set this environmental variable to FALSE as follows:

```
$nmdebug > ENV SYMPATH_UPSHIFT FALSE
```

The next two sections contain a variety of examples illustrating the use of path specifications.

Using the Symbolic Formatter

This section gives several examples of how to use the symbolic formatting facility.

Formatting Types

Refer to the beginning of this chapter to review the type declarations used in this section.

After the source types are converted into a symbolic data type file, the file is SYMOPENed and given a symbolic name of grades.

```
$nmdebug > SYMOPEN gradtyp.demo grades
```

The symbolic formatter is now able to display type information and format actual data using this symbolic data type file:

```
$nmdebug > FT "grades:StudentRecord"

RECORD
   NAME     : NAMESTR ;
   ID       : INTEGER ;
   YEAR     : CLASS ;
   NUMGRADES: GRADERANGE ;
   GRADES   : GRADESARRAY ;
END
```

Display the structure of StudentRecord. The *symname* part of the path specification is optional. If none is given, the last accessed symbolic file is assumed.

```
$nmdebug > FT "studentrecord" MAP

RECORD
   NAME     : NAMESTR ;   ( 0.0 @ 10.0 )
   ID       : INTEGER ;   ( 10.0 @ 4.0 )
   YEAR     : CLASS ;   ( 14.0 @ 1.0 )
   NUMGRADES: GRADERANGE ;   ( 15.0 @ 1.0 )
   GRADES   : GRADESARRAY ;   ( 18.0 @ 28.0 )
END ;
```

```
RECORD Size: 40 bytes
```

The MAP option of the FT command causes a location map to be printed for components of complex data structures such as records or arrays. The format of the location map is similar to the one generated by the $MAPINFO ON$ option of the Pascal compiler.

```
$nmdebug > FT "studentrecord.grades"

ARRAY [ GRADERANGE  ] OF INTEGER

$nmdebug > FT "graderange"

1 .. 10

$nmdebug > FT "maxgrades"

INTEGER

$nmdebug > FT "class"

( SENIOR, JUNIOR, SOPHOMORE, FRESHMAN )
```

Display various types. Notice that path specification is not limited to a simple type or constant name, but rather it may consist of any composite path specification.

The examples in the following pages include variant records and pointers. The following set of type declarations is used:

```
$nmdebug > ft "PersonPtr"

^ PERSON

$nmdebug > ft "PersonPtr^"

RECORD
   NEXT: PERSONPTR ;
   NAME: STRING[ 10 ];
   SEX : ( MALE, FEMALE );
   CASE MS: MSTYPE OF
      MARRIED : ( NUMKIDS: INTEGER );
```

```
      DIVORCED: ( HOWLONG: INTEGER );
      SINGLE  : ( LOOKING: BOOLEAN );
      WIDOWED : ( );
END

$nmdebug > ft "PersonPtr^.Sex"

( MALE, FEMALE )
```

Notice that you can refer to a type with a pointer dereference. That is, “Show me the type that this pointer points to.”

Formatting Data

The `FV` command allows you to format data at any virtual address using a given data structure:

`format` *at_any_virtual_address as_if_it_were_a_specific_type*

Before proceeding to some examples, we must deal with the question, "How do I find the virtual address of the data structure I want to format?" Most language compilers use the following conventions (as detailed in the *Procedure Calling Conventions Manual* (09740-90015):

- Global data is stored relative to DP (data pointer). DP is an alias for R27.
- Procedure local variables are stored relative to SP (stack pointer). SP in an alias for R30.
- Procedure parameters are stored in the argument registers (ARG0-ARG3) and in the stack relative to PSP (previous stack pointer). PSP is not contained in a register but is a pseudo-register that is computed by System Debug.

A variable map is required to find the location of a variable at any given time. These maps are generated as part of the program listing by the language compilers. Each compiler has a unique compiler option, which must be specified in order for the variable map to be included in the listing. For Pascal, the option is `$TABLES ON$`. For additional details on generating and interpreting this information, refer to the appropriate language reference manual. Each language also has a programmers manual which provides detailed language-specific examples illustrating how to use Debug to debug a program.

Caution If code optimization is done by the compiler, the location of the variables at any given time is indeterminable. Refer to the appropriate language manual for other issues concerning optimized code.

In the following examples, we assume that the System Debug variable `addr1` contains the address of a data structure corresponding to the type `StudentArray`. In addition, located at `dp+8` is a data structure defined by the `person` record. For example,

```
$nmdebug > fv addr1 "StudentRecord"
RECORD
   NAME      : 'Bill'
   ID        : 1
   YEAR      : SENIOR
   NUMGRADES : 4
   GRADES    :
      [ 1 ]: 2d
      [ 2 ]: 41
      [ 3 ]: 4e
      [ 4 ]: 42
      [ 5 ]: 0
      [ 6 ]: 0
      [ 7 ]: 0
      [ 8 ]: 0
      [ 9 ]: 0
      [ a ]: 0
END
```

```
$nmdebug > fv dp+8 "person"
RECORD
   NEXT : 40200024
   NAME : 'Mrs. Smith'
   SEX  : FEMALE
   MS   : MARRIED
      NUMKIDS : 3
END
```

The above examples show complete formatted record structures. Note that for variants with stored tags, the variants formatted are determined by the actual tag values.

When only a small portion of a large data structure needs to be examined, a path specification may be used to specify an item of interest, either simple or composite:

```
$nmdebug > fv addr1 "StudentRecord.Name"
```

```
'Bill'

$nmdebug > fv addr1 "StudentRecord.Year"

SENIOR

$nmdebug > fv dp+8 "Person.sex"

FEMALE
```

The above examples show how any field within a record may be formatted. Note that the address supplied is always the address for the beginning of the record, not the address of the field of interest.

As with field selection, array elements can also be selected. The command

```
$nmdebug > fv addr1 "StudentRecord.Grades[3]"

4e
```

displays only the third element of the field grades within the record `StudentRecord`.

As we saw in the person example above, if a data structure contains a pointer, its value (that is, the address of the pointed-to structure) is displayed. If the target of the pointer is desired, the caret (^) is used to indicate dereferencing. Consider the following examples:

```
$nmdebug > fv dp+8 "person.next"
40200024

$nmdebug > fv dp+8 "person.next^"
RECORD
   NEXT : 40200300
   NAME : 'Mr. Jones'
   SEX  : MALE
   MS   : SINGLE
      LOOKING : TRUE
END

$nmdebug > fv dp+8 "person.next^.next^.next^.next^.name"
'Mrs. Robinson'
```

If you try to dereference a field which contains a nil or invalid pointer, an error message is generated and the formatter stops formatting.

For variant records in which the tag fields are not stored, the variants to be used when formatting them may be specified by including tag field values. If no field is supplied, the first variant of the structure is assumed. The following examples are based on these types:

```
bit8 = 0 .. 255;

CoerceRec = RECORD
   CASE integer OF
        0 : (int   : integer);
        1 : (ch    : PACKED ARRAY [1..4]  OF char);
        2 : (byte  : PACKED ARRAY [1..4]  OF bit8);
        3 : (bool  : PACKED ARRAY [1..32] OF boolean);
END;
```

Consider the following examples assuming that the System Debug variable `addr` contains the address of some data corresponding to a `CoerceRec` data structure:

```
$nmdat > FV addr2 "CoerceRec"

RECORD
      INT : 4a554e4b
END
```

We assume the first variant for the CoerceRec and print out the data as an integer value. We now ask for an explicit variant:

```
$nmdat > FV addr2 "CoerceRec,1"

RECORD
      CH : 'JUNK'
END
```

We may explicitly ask for the data to be formatted in any of the possible variants. In the above example we asked for variant 1 (as characters). Notice that since this is a packed array of char (PAC), the formatter prints the data as a character string. To have PACs printed as arrays, specify the NOPAC option:

```
$nmdat > FV addr2 "CoerceRec,1" NOPAC

RECORD
      CH : [ 1 ]: 'J'
           [ 2 ]: 'U'
           [ 3 ]: 'N'
           [ 4 ]: 'K'
END
```

Also note that packed array of Boolean (PAB) are printed as a string of bits. To have such structures printed as arrays, you can specify the NOPAB options.

```
$nmdat > FV addr2 "CoerceRec,3"

RECORD
      BOOL :
         [ 1 ]: 01001010010101010100111001001011
END


$nmdat > FV addr2 "CoerceRec,3" NOPAB

RECORD
      BOOL :
         [  1 ]: FALSE
         [  2 ]: TRUE
```

```
          [  3 ]: FALSE
                .
                .  <etc for the rest of the array>
                .
          [ 32 ]: TRUE
END
```

Using Symbolic Access

Symbolic access references data through the use of symbolic names declared at the source code level, rather than through addresses and offsets to specific memory locations. This facility allows users to access stored information in a more natural way, leaving the drudgery of translating symbolic names to storage locations up to System Debug.

The chart below summarizes the symbolic functions currently available. These functions allow programmatic access to the information provided by the `FT` and `FV` commands.

Each function takes a path specification as one of its parameters. The form of this parameter is the same as that used by the `FT` and `FV` commands presented on the previous pages.

Each of these functions are presented in detail (including examples) in chapter 8.

`SYMVAL` (*virtaddress, pathspec*)	returns the value of the data structure specified by *pathspec*.
`SYMLEN` (*pathspec*, [*units*])	returns the length of a data structure in bits or bytes.
`SYMADDR` (*pathspec*, [*units*])	returns the bit or byte offset of an element specified by *pathspec*, relative to the start of the path.
`SYMINSET` (*virtaddress, pathspec, element*)	returns a boolean value of TRUE if the set member *element* is in the set specified by *address* and *pathspec*.
`SYMTYPE` (*pathspec*)	Returns the type of a component described by *pathspec*.
`SYMCONST` (*pathspec*)	returns the value of the constant specified by *pathspec*.
Parameters:	
virtaddress	the address of the actual data. **(Required)**
pathspec	a path specification. **(Required)**
units	specifies whether the return value for `SYMLEN` and `SYMADDR` is in bits or bytes. **(Optional)**
element	a set element. **(Required)**

FINAL TRIM SIZE : 7.5 in x 9.0 in

6

System Debug Windows

System Debug offers a powerful and efficient set of screen-oriented "windows," which allow dynamic visual monitoring of the program environment.

The System Debug windows are initially disabled, but can be easily toggled on (`WON`) and off (`WOFF`). Users can continue to use all normal interactive commands while the windows are displayed.

The following windows are provided by System Debug:

- The *register window (R)* displays the current CM register values
- The *general register window (GR)* displays the current NM general register values.
- The *special register window (SR)* displays the current values of a collection of special NM registers (including the space registers).
- The *program window (P)* tracks the program counter in the current mode (NM or CM). Current executing instructions are displayed and breakpoints are flagged. For convenience, the program window for one mode can also be accessed from the other mode with the fully qualified name (CMP or NMP).
- The *frame window (Q)* highlights the most recent CM stack marker. By default, this window displays addresses as unsigned DB-relative values. The user may choose to have addresses displayed relative to DB, Q, S, DL, or the DST base. Addresses may be displayed as signed or unsigned values. For details on these options, see the `QM` command. This window may also be aimed at any valid DST to which the user has access.
- The *stack window (S)* tracks the current CM top of stack. By default, this window displays addresses as unsigned DB relative values. The user may choose to have addresses displayed relative to DB, Q, S, DL the DST base. Addresses may be displayed as signed or unsigned values. For details on

these options, see the `SM` command. This window may also be aimed at any valid DST to which the user has access.

- A *group window (G)* is a special window within which the user can custom-define individual user windows (UW). These user windows (subwindows) can be "aimed" at parameters, variables, data blocks, and so on. Up to three group windows can be defined.
- A *virtual window (V)* displays data at a native mode virtual address. Up to eight virtual windows are available.
- The *memory window (Z)* displays data at a native mode real address.
- The *ldev window (L)* displays the contents of secondary storage at the specified disk address expressed as a logical device (LDEV) and byte offset.
- A *text window (TX)* displays information in a text file. Up to three text windows are available.
- The *command window* provides space for the user to type interactive commands.

Each mode (CM and NM) may have a different set of windows enabled. When one switches from mode to mode, the windows change to reflect the current mode. Note that there is only *one* set of windows; the user may easily specify which windows are enabled in a given mode. This means that virtual window #1 in CM is the same window as virtual window #1 in NM.

Each mode may have any combination of windows displayed together at one time. The only restriction is the number of lines available on the screen. There are 24 lines available for windows. The last two lines are reserved for the command window (where commands are entered and output is displayed). This leaves a maximum of 22 lines for additional windows. Any lines not used by other windows are automatically assigned to the command window. If an attempt is made to expand an existing window, add a new window, or enable an existing window for which there are insufficient free lines on the screen, System Debug will display an error message.

A Typical Screen Display of CM Windows

The following is a typical System Debug screen display with activated CM windows:

```
R % Regs    DB=001000  DBDST=000160  X=000132 STATUS=(mITroc CCG 301)   PIN=061
SDST=000160  DL=177650    Q=000704     S=000710      CMPC=PROG 000000.001667
 CIR=170005  MAPFLAG=1    MAPDST=000000
cmP %    PROG 0.1667          (E) SEG'              CSTX 1               Level   0
001662:     T|2|  PROCESSSTUDENT+%255      031403  3.  EXIT  3
001663:           PROCESSSTUDENT+%256      077777  ..  ADDM  S-%77,I,X
001664:           PROCESSSTUDENT+%257      177777  ..  LRA   S-%77,I,X
001665:      [1]  ?PROCESSSTUDENT          000700  ..  DZRO, NOP
001666:           PROCESSSTUDENT+%261      151605  ..  LDD   Q-5
001667:         > PROCESSSTUDENT+%262      170005  ..  LRA   P+5
001670:           PROCESSSTUDENT+%263      000733  ..  DZRO, INCA
Q %  (DB  mode)                   QDST=000160                        Level   0
000670:  000000   000000   000000   140026   000004   000000   000004   000000
000700:  000002   000132   000253   060301 Q>000010   000000   000000   000000
000710:  000002<S
S %  (DB  mode)                   SDST=000160                        Level   0
000700:  000002   000132   000253   060301 Q>000010   000000   000000   000000
000710:  000002<S
G   Group:1        %
U1  count       DB+5          % 000004      000000      000000      000000
U2  students    DB+2          A   ".."        "Bi"        "ll"        ".."
U3 *currnum     Q-5           % 000002      000132      000253      060301
Commands
Break at: CM    [1] PROG %   0.1665   ?PROCESSSTUDENT
%7 (%61) cmdebug > s 2
%8 (%61) cmdebug >
```

A Typical Screen Display of NM Windows

The following is a typical System Debug screen display with activated NM windows:

```
$   ipsw=0004000f=jthlnxbCvmrQPDI  priv=3  pc=000000f9.00005d24  pin=00000029
  00000000 00000002 00006b1f 81fe0000 r4  c0615c60 00000001 c0000000 00000000
  00000000 00000000 00000000 00000000 r12 00000000 00000000 00000000 00000000
6 00000000 00000000 00000000 40207df4 r20 00000004 00000001 00000001 402080f8
4 00000029 00000005 00000002 40200008 r28 00000002 00000080 40205940 00000005
P$ PROG  f9.5d18  GRADES.DEMO.TELESUP/processstudent.lowsco*+$dc  Level   0,0
005d18:         lowscore+$dc    4.0    4bdc3fa1  LDW      -48(0,30),28
005d1c:    T|2|  lowscore+$e0           e840c000  BV       0(2)
005d20:         lowscore+$e4           37de3fa1  LDO      -48(30),30
005d24:     [1]> processstudent         6bc23fd9  STW      2,-20(0,30)
005d28:         processstudent+$4      6fc30100  STWM     3,128(0,30)
005d2c:         processstudent+$8      6bc43f09  STW      4,-124(0,30)
005d30:         processstudent+$c      6bc53f11  STW      5,-120(0,30)
$ STUDENTS     SID=109     HOME=109.40200010               Values in $
200010:00000004 42696c6c 00000000 00000000 00000001 00040000 0000002d 00000041
200030:0000004e 00000042 00000000 00000000 00000000 00000000 00000000 00000000
$ Virtual      SID=109     HOME=109.40200010               Values in A
200010:  "...."  "Bill"  "...."  "...."  "...."  "...."  "...-"  "...A"
$ NUM          SID=109     HOME=109.40200154               Values in $
200154:00000004 00000000 00000000 00000000 00000000 0000000b a5050000 00000000
mmands
 ($29) nmdebug > vw dp+14c; vl 2;c
eak at: NM    [1] PROG f9.00005d24 processstudent
 ($29) nmdebug >
```

Window Operations

System Debug provides window commands which allow the user to customize individual windows:

- The size (number of lines) of each window can be set individually by the user. This allows the user to give up a few screen lines from one window in order to increase the size of another window. When the size of a particular window is set to 0 lines, then that window is effectively removed from the screen. The command window is the only window that cannot be entirely removed. Banner lines (the first line of the window) are included in the window line count. For example, a virtual window with a length of three lines contains one banner line and two lines of data. (Refer to the `wL` command.)
- Windows can be individually enabled and disabled (`wE` and `wD`) or they be removed (killed). (Refer to the `wK` command.)
- Windows can be scrolled forwards and backwards to display data in the proximity of the current location. (Refer to the `wF` and `wB` commands.)
- Most windows can be jumped to a specified address other than the default current address (which is based on program execution.) (Refer to the `PJ`, `QJ`, `SJ`, `TJ`, `VJ`,and `UJ` commands.)
- Windows can be returned to the "home" position. This is defined as the location displayed in the window when it was created. Some windows (virtual, real, ldev) allow the user to redefine the "home" location of the window. (Refer to the `wH` command.)
- Window values can be displayed in several output bases. Individual windows can be displayed in any selected radix, such as octal, decimal, hex, or ASCII. (Refer to the `wR` command.)
- The Q and S windows display addresses in one of several different modes (either DB, DL, Q, S, or DST). The mode determines how the addresses shown in the left column of the window will be displayed. The default is to display them relative to the current value of the DB register. Addresses may be displayed as signed or unsigned values. (Refer to `wM` command.) In addition, these windows may also be aimed at arbitrary data segments.

- Virtual and user windows can be named or renamed. (Refer to the `VN` and `UN` commands.)
- Virtual, text, and user windows can be used as "current" windows. Performing an operation on a window makes it current. In addition, one may specify explicitly which window to make current. (Refer to the `VC` and `UC` commands.)
- Text and virtual windows can have summary information about their shape and location printed with the "info" (*w*`I`) command.
- Text windows may be scrolled horizontally to view text in files wider than 80 columns. (Refer to the `TXS` command.)

Window Updates

System Debug automatically updates all displayed window values after the completion of every interactive user command list. In addition, when the user single steps (`SS`) the program, or continues (`C`) program execution until the next breakpoint is encountered, System Debug automatically updates the windows.

System Debug knows the current value of each cell in each window on the screen, and is therefore able to efficiently update only those cells that have changed since the last update. Consequently, window updates are very quick and are not distracting to the user. When major changes appear during window updates, these usually reflect a major change in the program environment, such as a procedure call.

Values that have been modified between updates are automatically flagged by System Debug by highlighting them in inverse video. This allows simple visual recognition of cells that are changing. The top of stack area displayed in the frame and stack windows is typically very dynamic.

The user can configure the terminal enhancement used to display these changing values (refer to the `ENV CHANGES` command.) In addition, the user can configure the terminal enhancement used to display the current stack marker (refer to the `ENV MARKER` command.)

Window Real/Virtual Modes

System Debug automatically tracks the translation bits in the processor status word (IPSW). There are two IPSW bits of interest, the C and D bits. These bits indicate if the machine performs "code" and "data" translation, respectively. If the C bit is off, the machine interprets all code addresses as `REAL` addresses rather than virtual addresses. Likewise, if the D bit is off, any data address is interpreted as a `REAL` address rather than a virtual address.

The windows honor this convention by examining the current settings of the bits in the processor status word. This means that any virtual window displays data based on the IPSW D bit. Likewise, the NM program window is affected by the C bit.

The NM program window is flagged as `REAL` when code translation is turned off (for example, the C bit equals 0). Likewise, virtual windows and user windows aimed at virtual address space are flagged as `REAL` when data translation is turned off (for example, the D bit equals 0).

R - The CM Register Window

The CM register window displays the current values of the compatibility mode registers.

```
R % Regs    DB=001000  DBDST=000160  X=000132 STATUS=(mITroc CCG% 301)  PIN=061
 DST=000160  DL=177650     Q=000704     S=000710      CMPC=PROG 000000.001667
 CIR=170005  MAPFLAG=1     MAPDST=000000
```

window banner line

- R % Regs - Abbreviation for the window, the current output display radix, and the name for the window.
- DB, DBDST - The current DB word offset (CM stack base relative) and DBDST data segment number. If DBDST is different from SDST (the stack data segment number), then DB and DBDST are displayed in half-inverse, indicating "split-stack mode."
- X - The current index register.
- STATUS - The current status register. (Refer to the conventions pages for a description of the format of this value.)
- PIN - The process identification number (PIN) for the current process.

window body line(s)

- SDST - The CM stack data segment number.
- DL - The DB relative value of DL.

- Q - The current Q value (stack frame), expressed in CM words, relative to DB.
- S - The current S value (TOS), expressed in CM words, relative to DB.
- CMPC - The current CM program location, expressed as a logical code address. This includes the library (`PROG, GRP, PUB, LGRP, LPUB, SYS`), logical segment number, and program counter in CM words, relative to the base of the current code segment.
- CIR - The current instruction register.
- MAPFLAG - If 0, the current CM segment is logically mapped. If 1, the current CM segment is physically mapped. This is used for CM CST expansion.
- MAPDST - The mapping DST number for CM CST expansion.

Gr - The NM General Registers Window

The NM register window displays the current values of the Native Mode General Registers.

```
GR$   ipsw=0004000f=jthlnxbCvmrQPDI  priv=3  pc=000000f9.00005d24  pin=00000029
r0  00000000 00000002 00006b1f 81fe0000 r4  c0615c60 00000001 c0000000 00000000
r8  00000000 00000000 00000000 00000000 r12 00000000 00000000 00000000 00000000
r16 00000000 00000000 00000000 40207df4 r20 00000004 00000001 00000001 402080f8
r24 00000029 00000005 00000002 40200008 r28 00000002 00000080 40205940 00000005
```

window banner line

- GR$ - Abbreviation for the window and the current output display. This window is always displayed in hexadecimal.

- ipsw - The current processor status word contents. The numeric value as well as the decoded bits are displayed. (Refer to the conventions pages for a description of the format for this value).
- priv - The current privilege level. This is based on the two low-order bits of the PCOF register.
- pc - The current program counter. This is a combination of the PCSF and PCOF registers. The offset part is always displayed word aligned.
- pin - The process identification number (PIN) for the current process.

window body line(s)

- r0 - r31 - The current values of the general registers.

Sr - The NM Special Registers Window

The special register window displays the current values of special NM registers.

```
SR$   isr=0000000a ior=00000000 iir=0000400e eiem=ffffffff rctr=00000000 sar=02
sr0=0000000a 0000000a 000000f8 00000000 sr4=00000101 000000f8 0000000b 0000000a
pcq=00000101.00005d27 00000101.00005d2b tr0=005e5200 00615200     eirr=00000000
pid1=0077(W)  007c(W)  007d(W)  0000(W) iva=00090000 itmr=5d801c34 ccr=80
```

window banner line

- SR$ - Abbreviation for the window and the current output display. This window is always displayed in hexadecimal.
- isr - The interruption space register.
- ior - The interruption offset register.
- iir - The interruption instruction register.
- eiem - The external interrupt enable mask.
- rctr - The recovery counter.
- sar - The shift amount register. (This is a 5 bit register.)

window body line(s)

- sr0 - sr7 - The space registers.
- pcq - The program counter queue.
- tr0 -tr1 - Temporary registers 0 and 1.
- eirr - The external interrupt request register.
- pid1 - pid 4 - The protection ID registers. These are 16-bit registers. (Refer to the conventions pages for a description of the format for this value.
- iva - The interrupt vector address.
- itmr - The interval timer.

- ccr - The coprocessor configuration register. (This is an 8-bit register.)

P (cmP) - The CM Program Window

The CM program window tracks the CM program counter (CMPC), displaying the instructions that are being executed.

```
cmP %     PROG 0.1667          (E) SEG'                CSTX 1                Level  0
001662:    T|2|  PROCESSSTUDENT+%255      031403  3.  EXIT  3
001663:          PROCESSSTUDENT+%256      077777  ..  ADDM  S-%77,I,X
001664:          PROCESSSTUDENT+%257      177777  ..  LRA   S-%77,I,X
001665:     [1]  ?PROCESSSTUDENT          000700  ..  DZRO, NOP
001666:          PROCESSSTUDENT+%261      151605  ..  LDD   Q-5
001667:        > PROCESSSTUDENT+%262      170005  ..  LRA   P+5
001670:          PROCESSSTUDENT+%263      000733  ..  DZRO, INCA
```

window banner line

- cmP % - Abbreviation for the window and the current output display radix for the window.
- PROG 0.1667 - The logical code address for the CM program counter. If the window does not contain the CM program counter, then the value is the logical code address of the first line in the window. In our example, the CM program counter is currently at a program file, logical segment number 0, at an offset of 1667 words. Other possible logical segment types are `GRP`, `PUB`, `LPUB`, `LGRP`, `SYS`.
- (E) - The segment is `(E)` emulated or `(T)` translated.
- SEG' - The segment name for the current segment being displayed.
- CSTX 1 - The `CSTX` (or `CST`) absolute segment number.
- Level 0 - The current stack level. (Refer to the `LEV` command.)

window body line(s)

- offset: - The CM word offset (segment relative) for the instruction line which is being displayed.

- breakpoints - Breakpoints are displayed between the offset and instruction. Refer to the conventions pages for a description of all possible breakpoint notations.

`[1]`	process local breakpoint, index number 1
`T|2|`	process local temporary breakpoint, count not exhausted yet,index number 2.

- > - Flags the current program counter location.
- **procedure-name+offset** - The symbolic procedure name and the CM word offset within the procedure.
- instruction (numeric, ASCII) - The instruction value is displayed formatted in the current output base for the window, and then displayed as two ASCII characters (for literals).
- instruction (disassembly) - The disassembled instruction value.

P (nmP) - The NM Program Window

The NM program window tracks the NM program counter (PC), displaying the instructions that are being executed. The banner line gives information for the *first* address displayed in the program window.

```
nmP$ PROG  f9.5d18  GRADES.DEMO.TELESUP/processstudent.lowsco*+$dc  Level   0,0
00005d18:         lowscore+$dc           4bdc3fa1  LDW       -48(0,30),28
00005d1c:   T|2|  lowscore+$e0           e840c000  BV        0(2)
00005d20:         lowscore+$e4           37de3fa1  LDO       -48(30),30
00005d24:     [1]> processstudent         6bc23fd9  STW       2,-20(0,30)
00005d28:         processstudent+$4      6fc30100  STWM      3,128(0,30)
00005d2c:         processstudent+$8      6bc43f09  STW       4,-124(0,30)
00005d30:         processstudent+$c      6bc53f11  STW       5,-120(0,30)
```

window banner line

- nmP $ - Abbreviation for the window and the current output display radix for the window.
- PROG f9.5d18 - The logical code address for the first line in the window. The program window is aimed at the `PROG`ram file, space: $f9, offset: $5d18.
- GRADES.DEMO.TELESUP/ - The name of the file which contains the displayed code.
- processstudent - The name of the level 1 procedure that appears in the *first* line of the window.
- .lowsco* - The nested procedure that appears in the *first* line of the window. An asterisk is used to flag the fact that the full name of the nested procedure does not fit in the display. (See the `DC` command and the `NMPATH` and `NMPROC` functions for instructions on displaying full procedure names).
- Level 0,0 - The current stack level, interrupt level (refer to the `LEV` command).

window body line(s)

- offset: - The virtual byte offset of the instruction line which is being displayed.
- breakpoints - Breakpoints are displayed between the offset and the instruction. Refer to the Conventions pages for a description of all possible breakpoint notations.

  `[1]` process local breakpoint, index number 1
  `T|2|` process local temporary breakpoint, count not exhausted yet, index number 2.

- > - Flags the current program counter location.
- **procedurename+offset** - The symbolic procedure name and the byte offset within the procedure.
- instruction (numeric) - The instruction value is displayed formatted in the current output base for the window.
- instruction (disassembly) - The disassembled instruction value.

Program Windows for Object Code Translation

A CM code segment (XLSEG11) has been translated by the Object Code Translator (OCT). The CM program window (top) is aimed at the original CM object code. The NM program window (middle) is aimed at the corresponding section of translated code. Fields within the windows that are unique to translated code are described below. Refer to appendix C for a discussion of CM object code translation, node points, and breakpoints in translated CM code.

```
cmP %     SYS  22.5206         (T) XLSEG11           CST 23             Level   0
005206:N   @[1]  ?FOPEN                   170404  ..  LRA    P-4
005207:          FOPEN+%5                 030400  1.  SCAL   0
005210:N    [2]  FOPEN+%6                 000600  ..  ZERO,  NOP
005211:     [3]  FOPEN+%7                 051451  S)  STOR   Q+%51
005212:N         FOPEN+%10                140060  .0  BR     P+%60
005213:          FOPEN+%11                140003  ..  BR     P+3
005214:N    [1]  ?FSOPEN                  170412  ..  LRA    P-%12
nmP$ TRANS 24.6b7bb8  (translated CM Seg SYS %22 XLSEG11)          Level   0,0
006b7bb8:N  @[1]  ?FOPEN                  340c1504  LDO       2690(0),12
006b7bbc:                                 34191510  LDO       2696(0),25
006b7bc0:                                 0c991264  STHS,MA   25,2(0,4)
006b7bc4:                                 d19adff0  EXTRS,>=  12,31,16,26
006b7bc8:                                 e680e792  BLE,N     968(7,20)
006b7bcc:                                 e566204e  BLE,N     53284(4,11)
006b7bd0:N   [2]  FOPEN+%6                0800024c  OR        0,0,12
006b7bd4:N                                646c00a4  STH       12,82(0,3)
006b7bd8:N        FOPEN+%10               e8000232  B,N       $006b7cf8
Commands
%31 (%44)  cmdebug >
```

window banner line

- (T) - The CM segment is currently running in translated mode.

- TRANS 24.6b7bb8 - The NM program window is aimed at translated code. The original CM segment is identified as `SYS %22 XLSEG11`.

window body line(s)

- Node points are denoted by `N`.
- breakpoints - Breakpoints are displayed between the offset and the procedure name. Refer to the conventions pages for a description of all possible breakpoint notations.

  `@[1]` global breakpoint, index number 1
  `[2]` process local breakpoint, index number 2

- **procedurename+offset** - The NM program window shows where each node point is in the original CM object code. The "?" indicates an *entry point* for CM procedure names. Refer to chapter 2, section "Procedure Name Symbols" for details on the conventions used for procedure names.

Q - The CM Stack Frame Window

The frame window tracks Q, the most recent CM stack frame.

```
Q %  (DB  mode)                    QDST=000160                        Level   0
 00670:  000000   000000   000000   140026   000004   000000   000004   000000
 00700:  000002   000132   000253   060301 Q>000010   000000   000000   000000
 00710:  000002<S
```

window banner line

- Q % - Abbreviation for the window and the current output display radix.
- (DB mode) - The address mode for the window. This can be `DB`, `DL`, `Q`, `S`, or `DST`. The address shown at the left side of the window is relative to the indicated base. (Refer to the `QM` command.)

- QDST - QDST is the data segment for the Q window. In most cases, this is the same as the stack DST. This window may be aimed away from the stack, in which case this value indicates the DST being viewed.
- Level 0 - The current stack level. (Refer to the `LEV` command).

window body line(s)

- offset: - The starting CM word offset for the line of displayed values. The values may be unsigned (default) or signed (relative to the address mode base). See the `QM` command for details.
- values - The actual data values are displayed in the current output base of the window.
- Q> - Indicates the location of Q. The stack marker (at Q-3, Q-2, Q-1, Q) is typically underlined. (Refer to the `ENV MARKER` command.)
- <S - Indicates the location of the current top of stack. The TOS value is typically underlined. (Refer to the `ENV MARKER` command.) If the TOS value has changed, the enhancement for the changed value will overwrite the enhancement for the TOS indicator (as in our example).

S - The CM Stack Window

The stack window tracks S, the current top of the CM stack (TOS).

```
S %  (DB  mode)                 SDST=000160                  Level   0
000700:  000002  000132  000253  060301 Q>000010  000000  000000  000000
000710:  000002<S
```

window banner line

- S % - Abbreviation for the window and the current output display radix.

- (DB mode) - The address mode for the window. This can be `DB`, `DL`, `Q`, `S`, or `DST`. The address shown at the left side of the window is relative to the indicated base. (Refer to the `SM` command.)
- SDST - SDST is the data segment for the S window. In most cases, this is the same as the stack dst. This window may be aimed away from the stack, in which case this value indicates the dst being viewed.
- Level 0 - The current stack level. (Refer to the `LEV` command.)

window body line(s)

- offset: - The starting CM word offset for the line of displayed values. The values may be unsigned (default) or signed (relative to the address mode base). See the `SM` command for details.
- values - The actual data values are displayed in the current output base of the window.
- <S - Indicates the location of the current top of stack. The TOS value is typically underlined. (Refer to the `ENV MARKER` command.) If the TOS value has changed, the enhancement for the changed value will overwrite the enhancement for the TOS indicator (as in our example).
- Q> - Indicates the location of Q. The stack marker (at Q-3, Q-2, Q-1, Q) is typically underlined. (Refer to the `ENV MARKER` command.)

G - The Group (of User) Window

The group window is a special window which contains multiple individual user-defined windows.

```
G   Group:1       %
 1  count       DB+5              % 000004       000000       000000       000000
 2  students    DB+2              A    ".."        "Bi"         "ll"         ".."
 3 *currnum       Q-5             % 000002       000132       000253       060301
```

window banner line

- G - Abbreviation for the group window.
- Group:1 - Displays the number of the group window that is currently being displayed. Three separate group windows, numbered from 1 to 3, are available. (Refer to the `WGRP` command).
- % - The current radix used to display addresses. The radix in that the addresses are displayed may be altered. (Refer to the `GR` command.)

window body line(s)

- User-defined window lines appear under the group banner line. Refer to the U (User) window discussion for details about user window lines.

The Command Window

The command window reserves space for the user to enter System Debug commands interactively and for displaying the resulting command output.

```
Commands

reak at: NM    [1] PROG f9.00005d24 processstudent
d ($29) nmdebug >
```

window banner line

- Commands - The name of the commands window.

window body line(s)

- $d ($29) nmdebug > - The System Debug prompt appears in the command window.

U - The User Windows

User-defined windows are custom named pointers.

```
G   Group:1       %
U1  count        DB+5           % 000004     000000     000000     000000
U2  students     DB+2           A   ".."       "Bi"       "ll"       ".."
U3 *currnum       Q-5           % 000002     000132     000253     060301
```

window banner line

- Refer to the G (Group) window discussion for a description of the banner line.

window body line(s)

- U# - The abbreviation for user window, followed by the number of the window. For example, U2 is read "user window number 2."
- * - An asterisk is placed next to the "current" (most recently used) user window. Several window commands are defined to operate on the current window, unless an optional window number is supplied.
- name - The name of the user window; the name is supplied when the window is created.
- address - The address where the user window is located. The address is always displayed based on the current output base of the group window that is displayed in the GW banner. The output base for the group window may be altered (Refer to the `GR` command.)
- %, A - The output display base for the data values in the user windows. The output base for each user window can be individually selected. (Refer to the `UR` command.)
- values - The actual data values are displayed in the current output base for this window.

V - The Virtual Windows

The virtual window displays blocks of Precision Architecture virtual memory.

```
$ STUDENTS    SID=109     HOME=109.40200010              Values in $
200010:00000004 42696c6c 00000000 00000000 00000001 00040000 0000002d 00000041
200030:0000004e 00000042 00000000 00000000 00000000 00000000 00000000 00000000
$ Virtual     SID=109     HOME=109.40200010              Values in A
200010:  "...."  "Bill"  "...."  "...."  "...."  "...."  "...-"  "...A"
$ NUM        SID=109     HOME=109.40200154              Values in $
200154:00000004 00000000 00000000 00000000 00000000 0000000b a5050000 00000000
```

window banner line

- V0, V1, V2 $ - Abbreviation for the virtual window, the virtual window number, and the current output display radix for offsets. At present, up to eight virtual windows may be defined. The current virtual window is indicated by flagging the window abbreviation in half-bright inverse video. In this display, V2 is the current virtual window.
- STUDENTS, Virtual, NUM - The name which was supplied when the window was created (or with the `VN` command). If no name is supplied, the name `"Virtual"` is used.
- SID - The virtual space ID at which the window is aimed.
- HOME - The home address which was originally specified in the `VW` command when the window was defined. Note that a new home address can be specified with the `VH` command.
- Values in $, A - The output display radix for data values. Note that virtual window number 1 has values in ASCII.

window body line(s)

- offset - The starting virtual offset for the line of displayed values.

- values - The actual data values are displayed. Unprintable ASCII data is shown as dots.

Z - The Memory Window

The memory window displays a block of Precision Architecture real memory.

```
$ Memory                                                  Values in $
000000:0004ffff ffff0000 007b434d 434d000f 0000fffc 00030037 0002000a 57697468
000020:20612068 6579204e 656c6c69 0002003c cd02000c 012f000c fffd0063 28660000
000040:0005ffff 534c2e50 55422e53 5953fffa 00070003 00010016 c1028014 05eb001b
```

window banner line

- Z $ Memory - Abbreviation for the window, the current output display radix for real address, and the name for the window.
- Values in $ - The output display base for data values.

window body line(s)

- offset - The real address for the line of displayed values.
- values - The actual data values are displayed.

L - The LDEV Window

The LDEV window displays the contents of secondary storage (data on disk).

```
LDEV $ DISP=1.0                    HOME=1.0                         Values in $
00000000:80004850 45535953 00085be0 10000000 00000008 00000000 00000000 00000000
00000020:00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

window banner line

- LDEV $ - Name of the LDEV window and the current output display radix.
- DISP - The full address of the current position of the LDEV window. (Byte offsets in the window itself contain only the low-order 32 bits.)
- HOME - The home address which was originally specified in the `LW` command when the window was defined. A new home address can be selected with the `LH` command. This address is expressed as a logical device (LDEV) and byte offset (that is, *ldev.offset*) relative to the start of the disk.
- Values in $ - The output display radix for data values.

window body line(s)

- offset - The starting disc offset (in bytes) for the line of displayed values.
- values - The actual data values from secondary storage are displayed.

TX- The Text Windows

The text window displays the contents of ASCII text files.

```
X0$  COL=1         LINE=1e      FNAME=TGRADES.DEMO.TELESUP
 ---------------------------------------------------------------------}
   Globally used TYPES                                                }
 ---------------------------------------------------------------------}

 PE

  GradeRange    = MINGRADES .. MAXGRADES;
  GradesArray   = ARRAY [ GradeRange ] OF integer;

  Class         = ( SENIOR, JUNIOR, SOPHOMORE, FRESHMAN );
X1$  COL=1         LINE=1           FNAME=UPOEM.DEMO.TELESUP
  "Roses are red,"
  "Violets are blue,"
  "Some poems rhyme,"
  "And this one does, too!"

ommands
```

window banner line

- TX0, TX1 - Abbreviation for the window, and the text window number. Currently, up to three text windows may be defined. The current text window is indicated by flagging the window abbreviation in half-bright inverse video. In this example, TX1 is the current text window.
- COL - The column number at which the window is aimed. Text windows may be "shifted" to view data that would otherwise be off the end of the screen.

- LINE - The line number (file record number) at which the window is aimed.
- FNAME - The name of the file at which the text window is aimed.

window body line(s)

- text - The ASCII contents of the text file(s).
- "." - Dots signify lines past the end-of-file count.
- "x" - X's signify an error while reading the data for that line. This could be a protection violation or some other cause (not shown above).

FINAL TRIM SIZE : 7.5 in x 9.0 in

7

System Debug Window Commands

System Debug window commands are most easily understood when they are grouped into two types of commands. The commands in this chapter are ordered as follows:

- General Window Operations:

RED	Redraw the entire screen display.
WDEF	Restore default window sizes.
WGRP	Switch to the specified group of user windows.
WOFF	Turn the windows off.
WON	Turn the windows on.

- Window Operations:

B	Backwards - scroll window backwards.
C	Current - mark window as current window.
D	Disable - disable (turn off) a window.
E	Enable - enable (turn on) a window.
F	Forwards - scroll window forwards.
H	Home - return window to home position.
I	Info - give info about defined windows.
J	Jump - aim window to new address.
K	Kill - remove, deallocate a window.
L	Lines - change window size in lines.
M	Mode - set mode (DB, DL, Q, S, DST) for Q or S.
N	Name - name or rename a user or virtual window.
R	Radix - change window display radix/base.
S	Shift - shift window left or right.
UW*m*	User Window - allocate user window at specified address.
W	Where - aim window to location.

- Window Abbreviations:

CMP	CM program window (from NM).
G	Group window.
GR	NM general registers window.
L	Ldev window.
NMP	NM program window (from CM).
P	Program window (current mode).
Q	CM frame window, Q relative.
R	CM registers window.
S	CM stack window, S relative.
SR	NM special registers window.
TX	Text file window.
U	User-defined window.
V	Virtual address window.
Z	Real memory window.

Put window abbreviations and window operations together to form the desired command. For example:

PB	Program Backward - scroll program window backward.
PF	Program Forward - scroll program window forward.
PL	Program Lines - change the program window size.
VH	Virtual Home - return virtual window to the home position.
VN	Virtual Name - assign a name to a virtual window.
VW	Virtual Where - define a virtual window.
ZR	Z(R)eal Radix - change the radix for the real window.

- Defining User Windows:

Append the desired addressing mode to the `UW`*m* command:

UWA	User window, ABS relative
UWCA	User window, CST relative
UWCAX	User window, CSTX relative
UWD	User window, DST relative
UWDB	User window, DB relative
UWQ	User window, Q relative
UWS	User window, S relative
UWV	User window, Precision Architecture virtual address
UWZ	User window, Precision Architecture real memory address

The Debug window commands are described in detail in the remainder of this chapter. The commands are listed in alphabetical order. Note that all individual window operation commands are constructed by preceding the window operation with the abbreviation for the desired window. To signify this, all window operation commands are listed as *w*`X`, where *w* represents the window abbreviation and `X` represents the command or operation. For example, the window forward command is *w*`F`. The syntax diagram for *w*`F` lists all the window types for which the command is applicable. If a window abbreviation is omitted, then the command does not apply to that window.

RED

Redraws the entire screen display of windows.

Syntax

```
RED
```

Parameters

none

Examples

```
%cmdebug > red
```

Redraws the screen.

Limitations, Restrictions

none

WDEF

Window defaults. Resets the default window sizes.

Syntax

```
WDEF
```

Parameters

none

Examples

```
%cmdebug > wdef
```

Limitations, Restrictions

Virtual and real window sizes default to 0 lines, so that they are effectively killed (VK, ZK) by this command.

WGRP

Changes to the specified group of user-defined windows.

Syntax

```
WGRP   [group_number]
```

Parameters

group_number The number of the group which is to be displayed in the group window. If no value is entered, group 1 is assumed.

Examples

```
%cmdebug > wgrp 2
```

Switch the group window to display group number 2.

Limitations, Restrictions

Current limit: 3 groups of 10 user-defined windows, each numbered from 1 to 10.

WOFF

Windows OFF. Turns off the windows.

Syntax

```
WOFF
```

Parameters

none

Examples

```
%cmdebug > woff
```

Limitations, Restrictions

none

WON

Windows ON. Turns on the windows. If windows are already on, redraws them.

Syntax

```
WON
```

Parameters

none

Examples

```
%cmdebug > won
```

Limitations, Restrictions

none

*w*B

Window back. Scrolls the specified window backwards.

Syntax

```
PB   [amount]                       Program, current mode
CMPB [amount]                       CM program
NMPB [amount]                       NM program

QB   [amount]                       CM frame,  Q relative
SB   [amount]                       CM stack,  S relative

GB   [amount]                       Group window
UB   [amount]  [win_number]         User window

VB   [amount]  [win_number]         Virtual window
ZB   [amount]                       Real memory window
LB   [amount]                       LDEV window
TXB  [amount]  [win_number]         Text window
```

Parameters

amount The number of words or lines to scroll backwards. If omitted, the window is scrolled back the default amount based on the following table:

*w***B**

Cmd	**Units**	**Default**
`PB`	(CM/NM) words*	Previous full screen of instructions
`CMPB`	CM words	Previous full screen of instructions
`NMPB`	NM words	Previous full screen of instructions
`QB`	CM words	Previous full line of data
`SB`	CM words	Previous full line of data
`GB`	User windows	To start of the previous user window
`UB`	(CM/NM) words*	1 line
`VB`	CM words	Previous full screen of data
`ZB`	CM words	Previous full screen of data
`LB`	CM words	Previous full screen of data
`TXB`	Lines	Previous full screen of text

* Based on mode of the window.

win_number The window number for a specific user window (U), virtual window (V), or text window (TX). If *win_number* is omitted, then the current window is used. The current user window is marked by an asterisk, and the current virtual window and text window are marked in inverse video.

Examples

```
%cmdebug > PB 6
```

Scroll the program window (PW) back 6 words.

```
%cmdebug > VB 5 2
```

Scroll virtual window number 2 back by 5 words.

```
%cmdebug > GB 2
```

Scroll the group window (GW) of user windows, back by two user windows.

Limitations, Restrictions

none

*w*C

Window current. Marks the specified window as the current window. Many user window (U), text window (TX), and virtual window (V) commands operate on the current window.

Syntax

```
UC  [win_number]
VC  [win_number]
TXC [win_number]
```

Parameters

win_number The window number for a specific user window (U), text window (TX), or virtual window (V). If *win_number* is omitted, then the current window remains flagged as the current window. The current user window is marked by an asterisk, and the current virtual and text windows are marked in inverse video.

Examples

```
%cmdebug > VC 2
```

Mark virtual window number 2 as the current virtual window.

```
%cmdebug > UC 3
```

Mark user window number 3 as the current user window.

Limitations, Restrictions

none

*w*D

Window disable.

Syntax

```
RD                        CM registers
GRD                       NM general registers
SRD                       NM special registers

PD                        Program, current mode
CMPD                      CM program
NMPD                      NM program
QD                        CM frame,  Q relative
SD                        CM stack,  S relative

GD                        Group window

UD     [win_number]       User  window

VD     [win_number]       Virtual window
ZD                        Real memory window
LD                        LDEV window
TXD    [win_number]       Text window
```

This command causes the window to be removed from the screen temporarily until the window is enabled again (see the `wE` command). Current window attributes (such as size, address, contents, and so on) are retained between disable/enable calls.

Parameters

win_number The window number for a specific user window (U), text window (TX), or virtual window (V). If *win_number* is omitted, then the current window is used. The current user window is marked by an asterisk, and the current virtual and text windows are marked in inverse video.

Examples

```
%cmdebug > PD
```

Disable the (current mode) program window.

```
%cmdebug > UD 3
```

Disable user window number 3.

Limitations, Restrictions

none

*w*E

Window enable.

Syntax

```
RE                        CM registers
GRE                       NM general registers
SRE                       NM special registers

PE                        Program, current mode
CMPE                      CM program
NMPE                      NM program
QE                        CM Frame,  Q relative
SE                        CM Stack,  S relative

GE                        Group window

UE    [win_number]        User window

VE    [win_number]        Virtual window
ZE                        Real memory window
LE                        LDEV window
TXE   [win_number]        Text window
```

This command enables a window that was previously disabled with the *w*D command. The original attributes of the window are retained between disable/enable calls.

Parameters

win_number The window number for a specific user window (U), text window (TX), or virtual window (V). If *win_number* is omitted, then the current window is used. The current user window is marked by an asterisk, and the current virtual and text windows are marked in inverse video.

Examples

```
%cmdebug > NMPE
```

Enable the NM program window. Both the CM and NM program window can appear together.

```
%cmdebug > VE 3
```

Enable virtual window number 3.

Limitations, Restrictions

none

*w*F

Window forward. Scrolls the specified window forward.

Syntax

```
PF   [amount]                         Program current mode
CMPF [amount]                         CM program
NMPF [amount]                         NM program

QF   [amount]                         CM frame,  Q relative
SF   [amount]                         CM stack,  S relative

GF   [amount]                         Group window
UF   [amount]  [win_number]           User window

VF   [amount]  [win_number]           Virtual window
ZF   [amount]                         Real memory window
LF   [amount]                         LDEV window
TXF  [amount]  [win_number]           Text window
```

Parameters

amount The number of words or lines to scroll forward. If *win_number* is omitted, then the window is scrolled forward the default amount based on the following table:

Cmd	Units	Default
PF	(CM/NM) words*	Next full screen of instructions
CMPF	CM words	Next full screen of instructions
NMPF	NM words	Next full screen of instructions
QF	CM words	Next full line of data
SF	CM words	Next full line of data
GF	User windows	To start of the next user window
UF	(CM/NM) words*	1 line
VF	CM words	Next full screen of data
ZF	CM words	Next full screen of data
LF	CM words	Next full screen of data
TXF	CM words	Next full screen of text

* Based on mode of the window.

win_number The window number for a specific user window (U), virtual window (V), or text window (TX). If *win_number* is omitted, then the current window is used. The current user window is marked by an asterisk, and the current virtual and text windows are marked in inverse video.

Examples

```
%cmdebug > PF 6
```

Scroll the (current mode) program window forward six words.

```
%cmdebug > VB 5 2
```

Scroll virtual window number 2 forward by five words.

```
%cmdebug > GF 2
```

Scroll the group window (of user windows) forward by two user windows.

Limitations, Restrictions

none

*w*H

Window home. Returns a window to its original location.

Syntax

```
RH                                   CM registers window
GRH                                  NM general registers window
SRH                                  NM special registers window

PH                                   Program window, current mode
CMPH                                 CM program window
NMPH                                 NM program window
QH                                   CM frame window - Q relative
SH                                   CM stack window - S relative

GH                                   Group window
UH  [win_number]                     User window

VH  [virtaddr]   [win_number]        Virtual window
ZH  [realaddr]                       Real memory window
LH  [ldev.off]                       LDEV window
TXH [win_number]                     Text window
```

This command returns the specified window to its original (home) location. (This is the location specified when the window was created.) This command is useful when a window has been scrolled (F,B) or jumped (J) away from its home location. The virtual (V), real (Z), and LDEV (L) windows may have their home location respecified with this command by supplying a new home location.

Parameters

win_number The window number for a specific user window (U), text window (TX), or virtual window (V). If *win_number* is omitted, then the current window is used. The current user window is marked by an asterisk, and the current virtual and text windows are marked in inverse video.

virtaddr If this parameter is provided, the home address for the virtual window (V) is set to the indicated address. *Virtaddr* can be a short pointer, a long pointer, or a full logical code pointer.

realaddr If this parameter is provided, the home address for the real window (Z) is set to the indicated real address.

ldev.off The disk LDEV and byte offset to which the home address is set.

Examples

```
%cmdebug > PH
```

Home the program window.

```
$nmdebug > VH PSP-40 4
```

Change the home address for virtual window 4 to be the value of PSP-40. Jump the window to the new home address.

```
%cmdebug > UH 3
```

Home user window 3.

Limitations, Restrictions

none

*w*I

Window information. Prints information about the indicated windows. This command is defined for the virtual (V) and text (TX) windows.

Syntax

```
VI  [win_number]
TXI [win_number]
```

Parameters

win_number — The window number for a specific text window (TX) or virtual window (V). If *win_number* is omitted, then information for all of the text or virtual windows is displayed.

The abbreviations used in the output are defined as follows:

`COL`	Column number (1, unless window was "shifted").
`LINE`	Line (record number) where window is aimed.
`REC`	Record size of the file (in bytes).
`EOF`	End of file record number.
`FLIMIT`	File limit (maximum number of records in the file).

The following flags may also appear:

`CCTL`	File has carriage control.
`VAR`	File has variable length records (REC is undefined).
`BIN`	File is binary file.

Examples

```
$nmdebug > vi 2

V2:  HOME= a.00040017    CURR= a.00040017    Lines=3
```

Display information about virtual window number 2.

```
$nmdebug > txi

TX0: TDEBUG.CMDEBUG.OFFICIAL         COL=1            LINE=34c
     REC=50                          EOF=534d       FLIMIT=534d

TX1: LIST.DEBUG.WORK                 COL=a1           LINE=1
     REC=85          CCTL            EOF=1000       FLIMIT=1000
```

Display information about all of the text windows.

Limitations, Restrictions

The format of output may be changed without notice.

*w*J

Window jump. Jumps window to the specified address.

Syntax

PJ	[*logaddr*]	Program file
PJG	[*logaddr*]	Group library
PJP	[*logaddr*]	Account library
PJLG	[*logaddr*]	Logon group library
PJLP	[*logaddr*]	Logon account library
PJS	[*logaddr*]	System library
PJU	[*fname logaddr*]	User library
PJV	[*virtaddr*]	Any virtual address
PJA	[*absaddr*]	Absolute CST
PJAX	[*absaddr*]	Absolute CSTX
CMPJ	[*logaddr*]	Program file
CMPJG	[*logaddr*]	Group library
CMPJP	[*logaddr*]	Account library
CMPJLG	[*logaddr*]	Logon group library
CMPJLP	[*logaddr*]	Logon account library
CMPJS	[*logaddr*]	System library
CMPJA	[*absaddr*]	Absolute CST
CMPJAX	[*absaddr*]	Absolute CSTX
NMPJ	[*logaddr*]	Program file
NMPJG	[*logaddr*]	Group library
NMPJP	[*logaddr*]	Account library
NMPJLG	[*logaddr*]	Logon group library
NMPJLP	[*logaddr*]	Logon account library
NMPJS	[*logaddr*]	System library
NMPJU	[*fname logaddr*]	User library
QJ	[*dst.off*]	CM Frame, Q relative
SJ	[*dst.off*]	CM Stack, S relative
VJ	[*virtaddr*] [*win_number*]	Virtual window
ZJ	[*realaddr*]	Real memory window
LJ	[*Ldev.off*]	LDEV window
TXJ	[*record_number*]	Text window

Parameters

logaddr

PJ, PJG, PJP, PJLG, PJLP, PJS, PJU, and PJV control the current program window, which is based on the current mode (CM or NM).

CMPJ, CMPJG, CMPJP, CMPJLG, CMPJLP, and CMPJS control the CM program window.

NMPJ, NMPJG, NMPJP, NMPJS, NMPJS, and NMPJU control the NM program window.

A full logical code address (LCPTR) specifies three necessary items:

1. The logical code file (PROG, GRP, SYS, and so on).
2. NM: the virtual space ID number (SID).
   CM: the logical segment number.
3. NM: the virtual byte offset within the space.
   CM: the word offset within the code segment.

Logical code addresses can be specified in various levels of detail:

- As a full logical code pointer (LCPTR)

PJ procname+20	Procedure name lookups return LCPTRs.
PJ pw+4	Predefined ENV variables of type LCPTR.
PJ SYS(2.200)	Explicit coercion to a LCPTR type.

- As a long pointer (LPTR)

  PJ 23.2644 *sid.offset* or *seg.offset*

  The logical file is determined based on the command suffix:

  PJ implies PROG
  PJG implies GRP
  PJS implies SYS, and so on.

- As a short pointer (SPTR)

  PJ 1024 *offset* only

For NM, the short pointer offset is converted to a long pointer using the function STOLOG, which looks up the SID of the loaded logical file. This is different from the standard short to long pointer conversion, STOL, which is based on the current space registers (SRs).

For CM, the current executing logical segment number and the current executing logical file are used to build an LCPTR.

The search path used for procedure name lookups is based on the command suffix letter:

PJ	Full search path: NM: PROG, GRP, PUB, USER(s), SYS CM: PROG, GRP, PUB, LGRP, LPUB, SYS
PJG	Search GRP, the group library.
PJP	Search PUB, the account library.
PJLG	Search LGRP, the logon group library.
PJLP	Search LPUB, the logon account library.
PJS	Search SYS, the system library.
PJU	Search USER, the user library.

For a full description of logical code addresses, refer to the section "Logical Code Addresses" in chapter 2.

fname PJU, CMPJU, and NMPJU only. The file name of the NM USER library. Multiple NM libraries can be bound with the XL= option on a RUN command. For example:

```
:RUN NMPROG; XL=LIB1,LIB2.TESTGRP,LIB3
```

In this case it is necessary to specify the desired NM USER library. For example:

```
PJU lib1 204c
PJU lib2.testgrp test20+1c0
```

If the file name is not fully qualified, then the following defaults are used:

Default account: the account of the program file.
Default group: the group of the program file.

virtaddr — The virtual window (V) can be aimed at any Precision Architecture space and offset address. *Virtaddr* can be a short pointer, a long pointer, or a full logical code pointer.

absaddr — `PJA, PJAX, CMPJA, CMPJAX` control the CM program window. A full CM absolute code address specifies three necessary items:

Either the `CST` or the `CSTX`
The absolute code segment number
The CM word offset within the code segment

Absolute code addresses can be specified in two ways:

- As a long pointer (`LPTR`)

`PJA`	`23.2644`	Implicit `CST` 23.2644
`PJAX`	`5.3204`	Implicit `CSTX` 5.3204

- As a full absolute code pointer (`ACPTR`)

`PJA`	`CST(2.200)`	Explicit `CST` coercion
`PJAX`	`CSTX(2.200)`	Explicit `CSTX` coercion
`PJAX`	`logtoabs(prog(1.20))`	Explicit absolute conversion

The search path used for procedure name lookups is based on the command suffix letter:

`PJA`	`GRP, PUB, LGRP, LPUB, SYS`
`PJAX`	`PROG`

dst.off — The stack frame (Q) and top of stack (S) windows can be aimed at any data segment and offset.

ldev.off — The LDEV window can be aimed at a disk *ldev.byte-offset*.

win_number — You may specify which virtual window is the jump window, if there is more than one window.

realaddr — The real memory window (Z) can be aimed at any real address. If no address is given, the address used is the address to which the window previously was pointed (if any).

record_number — The text file record number.

Examples

```
$nmdebug > pj 200
```

Jump to the program file at offset 200. A logical address is expected as the value for this command. Remember that when only an offset is specified as a logical address in the PJ command, the space (SID) for the program is assumed. A STOLOG conversion (with the "prog" selector) will be done to accomplish this.

```
$nmdebug > pj r2
```

Jump to the program file at the offset indicated by register R2. As in the above example, when only an offset is given for a logical address, the space (SID) for the program file is assumed.

```
$nmdebug > pjv r2
```

Jump to the offset indicated by register R2. The space is determined by using the appropriate space register. A STOL conversion is performed to accomplish this.

```
$nmdebug > pjs r2
```

Jump to the system library (NL.PUB.SYS) at the offset indicated by register R2.

```
%cmdebug > pjg 2.200
```

Jump to the group library at logical segment 2 at an offset of 200.

```
$nmdebug > cmpj cmaddr("?fopen")
```

Jump the CM program window to the entry point for the fopen procedure. Note that since we are in native mode, the CMADDR function must be used to look up the address of CM procedures.

```
%cmdebug > nmpj cmtonmnode(?fopen)
```

Jump the NM program window to the nearest translated code node point associated with the CM procedure fopen. Refer to appendix C for a discussion of CM object code translation, node points, and breakpoints in translated CM code.

```
%cmdebug > SJ 12.200
```

Jump the stack window to data segment 12 at an offset of 200.

```
$nmdebug > vw c0.100          /* Create a new virtual window at c0.100
$nmdebug > vj c0.200          /* Jump the window to c0.200
$nmdebug > vj c0.300          /* Jump the window to c0.300
$nmdebug > vj                 /* Jump to previous location (c0.200)
$nmdebug > vh                 /* Jump to home location (c0.100)
```

The end result is to place the current virtual window at 100 (its "home" location).

Limitations, Restrictions

none

*w*K

Window kill.

Syntax

```
RK                          CM registers
GRK                         NM general registers
SRK                         NM special registers

PK                          Program, current mode
CMPK                        CM program
NMPK                        NM program
QK                          CM frame,  Q relative
SK                          CM stack,  S relative

GK                          Group window
UK     [win_number]         User window

VK     [win_number]         Virtual window
ZK                          Real memory window
LK                          LDEV window
TXK    [win_number]         Text window
```

This command removes a window from the screen. It does this by setting the length of a window to zero lines, which effectively makes it disappear. The command permanently deallocates text, user, and virtual windows. (Attempts to set the lines to a value greater than zero for these window results in an error since the window no longer exists.) If the window is a text window, this command closes the file.

Parameters

win_number The window number for a specific user window (U), text window (TX), or virtual window (V). If *win_number* is omitted, then the current window is used. The current user window is marked by an asterisk, and the current virtual and text windows are marked in inverse video.

Examples

```
%cmdebug > PK
```

Kill the (current mode) program window.

```
%cmdebug > PL 6
```

Bring back the program window. Remember, killing a window sets its length to zero.

```
%cmdebug > VK 3
```

Deallocate virtual window number 3. This window cannot be brought back by changing the window length as in the above example. Once a virtual window is killed, it is gone until a new VW command is used to create a new one.

Limitations, Restrictions

none

*w*L

Window lines. Sets the number of lines in a window.

Syntax

```
RL    [numlines]                   CM registers
GRL   [numlines]                   NM general registers
SRL   [numlines]                   NM special registers

PL    [numlines]                   Program, current mode
CMPL  [numlines]                   CM program
NMPL  [numlines]                   NM program
QL    [numlines]                   CM frame,  Q relative
SL    [numlines]                   CM stack,  S relative

GL    [numlines]                   Group window
UL    [numlines]  [win_number]     User window

VL    [numlines]  [win_number]     Virtual window
ZL    [numlines]                   Real memory window
LL    [numlines]                   LDEV window
TXL   [numlines]  [win_number]     Text window
```

Parameters

numlines Set the window size to this number of lines. If no value is given, the default is the initial size for the specified window.

win_number The window number for a specific user window (U), text window (TX), or virtual window (V). If *win_number* is omitted, then the current window is used. The current user window is marked by an asterisk, and the current virtual and text windows are marked in inverse video.

Examples

```
%cmdebug > pl 7
```

Set the (current mode) program window to 7 lines.

```
%cmdebug > gl 0; vl 5
```

Turn off the group window and set the current virtual window to 5 lines.

Limitations, Restrictions

none

*w*M

Window mode. Changes the mode for the Q or S window.

Syntax

```
QM  [addressmode]  [signed]
SM  [addressmode]  [signed]
```

Parameters

addressmode	This parameter specifies the mode in which addresses are to be displayed. If no value is specified, DB is the default. The following values are allowed:

`DB`	Display address as DB-relative values (initial mode).
`DL`	Display address as DL-relative values.
`DST`	Display address as DST-base-relative values.
`Q`	Display address as Q-relative values.
`S`	Display address as S-relative values.

If the window is jumped to a data segment other than the stack data segment (`SDST`), only DST mode is allowed.

Addresses entered with the `QJ` and `SJ` commands are interpreted based on the mode of the respective window.

signed — This parameter indicates if addresses are to be displayed as signed or unsigned values. If no value is specified, `UNSIGNED` is the default.

The following values are allowed:

`UNSIGNED`	Display address as unsigned values (initial setting).
`SIGNED`	Display address as signed values (+/- present in address).

Examples

```
$nmdebug > qm dst
```

Set the `Q` window to display addresses as DST-relative (stack-base relative) values.

```
$nmdebug > sm ,signed
```

Set the `S` window to have addresses displayed as signed values.

Limitations, Restrictions

none

*w*N

Renames a virtual window or a user-defined window.

Syntax

```
UN   [name] [win_number]    User window
VN   [name] [win_number]    Virtual window
```

Parameters

name The name for this user window. Names are restricted to eight alphanumeric characters.

If the name is omitted, the following default names are used:

Window	Default Name
USER (U)	<user>
VIRTUAL (V)	Virtual

win_number The window number for a specific user window (U) or virtual window (V). If *win_number* is omitted, then the current window is used. The current user window is marked by an asterisk, and the current virtual window is marked in inverse video.

Examples

```
%cmdebug > un datablk
```

Rename the current user window to "datablk."

```
%cmdebug > vn parms 4
```

Rename virtual window number four to "parms."

Limitations, Restrictions

none

*w*R

Sets the radix (output base) for the specified window.

Syntax

```
RR    base                    CM registers

PR    base                    Program, current mode
CMPR  base                    CM program
NMPR  base                    NM program
QR    base                    CM frame,  Q relative
SR    base                    CM stack,  S relative

GR    base                    Group window
UR    base [win_number]       User window

VR    base [win_number]       Virtual window
ZR    base                    Real memory window
LR    base                    Ldev window
```

Parameters

base The desired representation mode for output values:

`%` or `OCTAL`	Octal representation
`#` or `DECIMAL`	Decimal representation
`$` or `HEXADECIMAL`	Hexadecimal representation
`ASCII`	ASCII representation

This parameter can be abbreviated to as little as a single character.

win_number The window number for a specific user window (U) or virtual window (V). If *win_number* is omitted, then the current window is used. The current user window is marked by an asterisk, and the current virtual window is marked in inverse video.

Examples

```
%cmdebug > qr a
```

Display the values in the stack frame window in ASCII.

```
%cmdebug > ur d 3
```

Display user window number 3 in decimal.

Limitations, Restrictions

The R, GR, SR, and CMP windows cannot be set to an ASCII base.

The radix for the NMP, SR, and GR windows cannot be altered from its initial hexadecimal value.

*w*S

Window shift. Shifts a window to the left or right. This command is defined for text windows (TX).

Syntax

```
TXS  [ amount ] [win_number]
```

Parameters

amount This is the number of columns to shift the window. A positive value shifts the window right (view data past the right end of the screen). A negative value shifts the window left (view data past the left end of the screen). If no value is given, the window is shifted to column 1.

win_number The window number for a specific text window (TX). If *win_number* is omitted, then the current window is used.

Examples

```
$nmdebug > TXS #20
```

Shift the window 20 columns to the right.

```
$nmdebug > TXS -9999
```

Shift the window to the left. Any column number less than 1 is automatically converted to column 1.

Limitations, Restrictions

none

UW*m*

Allocates a named user window at the specified address. The command name specifies which type of window to define. User windows are displayed within the group window.

Syntax

```
UWA    offset      [name]    Absolute memory relative (ABS)
UWDB   offset      [name]    DB relative
UWS    offset      [name]    S relative
UWQ    offset      [name]    Q relative

UWD    dst.off     [name]    Data segment and offset

UWCA   cmabsaddr   [name]    Code (CST) segment and offset
UWCAX  cmabsaddr   [name]    Code (CSTX) segment and offset

UWV    virtaddr    [name]    Virtual address
UWZ    realaddr    [name]    Real address
```

Parameters

offset — UWA, UWDB, UWQ, UWS only. The CM word offset which specifies the relative starting location.

dst.off — UWD only. The data segment and offset where to aim the window.

cmabsaddr — UWCA, UWCAX only. A full CM absolute code address. This code address specifies three necessary items:

Either the CST or the CSTX
The absolute code segment number
The CM word offset within the code segment

Absolute code addresses can be specified in two ways:

- As a long pointer (`LPTR`):

`UWCA`	`23.2644`	Implicit `CST` 23.2644
`UWCAX`	`5.3204`	Implicit `CSTX` 5.3204

- As a full absolute code pointer (`ACPTR`):

`UWCA`	`CST(2.200)`	Explicit `CST` coercion
`UWCAX`	`CSTX(2.200)`	Explicit `CSTX` coercion
`UWCAX`	`logtoabs(prog(1.20))`	Explicit absolute conversion

The search path used for procedure name lookups is based on the command suffix letter:

UWCA	GRP, PUB, LGRP, LPUB, SYS
UWCAX	PROG

virtaddr — UWV only. A Precision Architecture virtual address. *Virtaddr* can be a short pointer, a long pointer, or a full logical code pointer.

realaddr — UWZ only. A Precision Architecture real memory address.

name — The name for this user window. Names are restricted to eight alphanumeric characters. If *name* is omitted, the window is named "user".

Examples

```
%cmdebug > UWQ-30 parms
```

Create a user window at Q-30 and name it "parms".

```
%cmdebug > UWDB+112, globvar
```

Create a user window at DB+112 and name it "globvar".

```
$nmdebug > UWV SP-30, count
```

Create a user window at SP-30 (stack pointer - 30) and name it "count".

Limitations, Restrictions

Current limit: 10 user-defined windows per group.

*w*W

Defines (enables) new windows.

Syntax

```
VW  virtaddr  [name]        Virtual window
ZW  realaddr                Real Memory
LW  Ldev.off                LDEV (Secondary Storage) window
TXW filename                Text window

UWm                         User window (see UWm command)
```

The `VW` and `TXW` commands allocate the next available virtual (V) or text (TX) window. The window is aimed at the specified address (V) or file (TX). Finally, the window is marked as the "current window."

The `LW` and `ZW` commands aim/enable the real memory window (ZW) and the LDEV window (LW) respectively. There is only one of each of these windows.

By default these windows are created with an initial length of three lines (one banner line and two data lines). The size of the windows may be changed once they are created (Refer to the *w*`L` command.)

Parameters

virtaddr The virtual window can be aimed at any Precision Architecture space and offset address. *Virtaddr* can be a short pointer, a long pointer, or a full logical code pointer.

name This is the name with which to label the virtual window being defined. If no name is specified, "`Virtual`" is used as a default.

realaddr The real memory window can be aimed at any real address.

Ldev.off The LDEV window can be aimed at any valid disk LDEV number at a specified byte offset.

*w***W**

filename The file name to which the text window is aimed.

Examples

```
%cmdebug > VW a.c0000000 SYSGLOB
```

Allocate a new virtual window and aim it at a.c0000000. Label the window with the name SYSGLOB.

```
%cmdebug > ZW 1800
```

Aim the real memory window to physical address 1800.

```
$nmdebug > TXW TGRADES.DEMO.TELESUP
```

Create and aim a text window at the file TGRADES.DEMO.TELESUP.

Limitations, Restrictions

A total of seven virtual windows and three text windows are available. There is only one LDEV and one real window.

8

System Debug Standard Functions

This chapter presents the full formal declaration for each of the standard functions which are defined in System Debug.

All functions are callable from both DAT and Debug. All functions can be called from both Native Mode (NM) and Compatibility Mode (CM). Some functions, however, deal specifically with NM or CM attributes. Input parameters are always interpreted based on the current mode, so care must be exercised when specifying procedure names and numeric literals.

Functions are logically divided into groups and can be listed with the `FUNCL[IST]` command, filtered by the group name.

The following table lists all functions, sorted by group name. For each function, the name, type, and a brief description is presented.

COERCION Functions

Name	Type	Description
ASCC	: STR	Coerces an expression to ASCII
BOOL	: BOOL	Coerces an expression to Boolean
CST	: CST	Coerces an expression to CST ACPTR
CSTX	: CSTX	Coerces an expression to CSTX ACPTR
EADDR	: EADDR	Coerces an expression to extended address.
GRP	: GRP	Coerces an expression to GRP LCPTR
LGRP	: LGRP	Coerces an expression to LGRP LCPTR
LPTR	: LPTR	Coerces an expression to long pointer.
LPUB	: LPUB	Coerces an expression to LPUB LCPTR
PUB	: PUB	Coerces an expression to PUB LCPTR
S16	: S16	Coerces an expression to signed 16-bit INT
S32	: S32	Coerces an expression to signed 32-bit INT
S64	: S64	Coerces an expression to signed 64-bit INT
SADDR	: SADDR	Coerces an expression to secondary address.
SPTR	: SPTR	Coerces an expression to short pointer
SYS	: SYS	Coerces an expression to SYS LCPTR
TRANS	: TRANS	Coerces an expression to TRANS LCPTR
USER	: USER	Coerces an expression to USER LCPTR
U16	: U16	Coerces an expression to unsigned 16-bit INT
U32	: U32	Coerces an expression to unsigned 32-bit INT

UTILITY Functions

Name	Type	Description
ASC	: STR	Converts an expression to an ASCII string
BIN	: INT	Converts an ASCII string to binary value
BITD	: ANY	Bit deposit
BITX	: ANY	Bit extract
BOUND	: STR	Tests for current definition of an operand
CISETVAR	: BOOL	Sets a new value for a CI variable
CIVAR	: ANY	Returns the current value of a CI variable
ERRMSG	: STR	Returns an error message string
MACBODY	: STR	Returns the macro body of a specified macro
TYPEOF	: STR	Returns the type of an expression
MAPINDEX	: U16	Returns the index number of a mapped file
MAPSIZE	: U32	Returns the size of a mapped file
MAPVA	: LPTR	Returns the virtual address of a mapped file

ADDRESS Functions

Name	Type	Description
ABSTOLOG	: LCPTR	CM absolute address to logical code address
BTOW	: U16	Converts a CM byte offset to a word offset
CMNODE	: LCPTR	CM address of closest CM node point
CMTONMNODE	: TRANS	NM address of closest CM node point
CMVA	: LPTR	Converts CM code address to a virtual address
DSTVA	: LPTR	Converts CM dst.off to virtual address
HASH	: S32	Hashes a virtual address
LOGTOABS	: ACPTR	CM logical code address to absolute address
LTOLOG	: LCPTR	Long pointer to logical code address
LTOS	: SPTR	Long pointer to short pointer
NMNODE	: TRANS	Address of closest NM node point
	NM	
NMTOCMNODE	: LCPTR	CM address of closest NM node point
OFF	: U32	Extracts offset part of a virtual address
PHYSTOLOG	: LCPTR	CM physical segment/map bit to logical
RTOV	: LPTR	real to virtual
SID	: U32	Extracts the SID (space) part of a long pointer
STOL	: LPTR	Short pointer to long pointer
STOLOG	: LCPTR	Short pointer to logical code address
VTOR	: U32	Virtual to real
VTOS	: SADDR	Virtual to secondary store address

PROCESS Functions

Name	Type	Description
CMG	: SPTR	Short pointer address of CMGLOBALS record
CMSTACKBASE	: LPTR	Virtual address of the CM stack base
CMSTACKDST	: U16	Data segment number of the CM stack
CMSTACKLIMIT	: LPTR	Virtual address of the CM stack limit
NMSTACKBASE	: LPTR	Virtual address of the NM stack base
NMSTACKLIMIT	: LPTR	Virtual address of the NM stack limit
PCB	: SPTR	Address of process control block
PCBX	: SPTR	Address of process control block extension
PIB	: SPTR	Address of process information block
PIBX	: SPTR	Address process information block extension
PSTATE	: STR	Returns the process state for specified PIN
TCB	: U32	Real address of the task control block
VAINFO	: ANY	Returns virtual object information

PROCEDURE Functions

Name	Type	Description
CMADDR	: LCPTR	Logical address of a CM procedure name
CMBPADDR	: LCPTR	Logical address of a CM breakpoint index
CMBPINDEX	: S16	Index number of a CM breakpoint address
CMBPINSTR	: S16CM	Instruction at a CM breakpoint address
CMENTRY	: LCPTR	Logical entry address of a CM procedure
CMPROC	: STR	Returns the name of a CM procedure
CMPROCLEN	: U16	Returns the length of CM procedure
CMSEG	: STR	Returns the CM segment name at logical address
CMSTART	: LCPTR	Logical start address of CM procedure
NMADDR	: LCPTR	Logical address of NM procedure name
NMBPADDR	: LCPTR	Logical address of NM breakpoint index
NMBPINDEX	: S16	Index number of a NM breakpoint address
NMBPINSTR	: S32NM	Instruction at a NM breakpoint address
NMCALL	: S32NM	Dynamically invokes the specified NM routine
NMENTRY	: LCPTR	Logical entry address of NM procedure
NMFILE	: STR	Name of file containing NM logical address
NMMOD	: STR	Name of NM module at NM logical address
NMPATH	: STR	Returns the full code path of a NM procedure
NMPROC	: STR	Name of NM procedure at NM logical address

STRING Functions

Name	Type	Description
STR	: STR	Extracts a substring from a string
STRAPP	: STR	String append
STRDEL	: STR	String delete
STRDOWN	: STR	Downshifts a string
STREXTRACT	: STR	Extracts a string at a virtual address
STRINPUT	: STR	Prompts for and reads string input
STRINS	: STR	String insert
STRLEN	: U16	Returns the current length of a string
STRLTRIM	: STR	Removes leading blanks from a string
STRMAX	: U16	Returns the maximum length of a string
STRPOS	: U16	Locates a substring within a string
STRRPT	: STR	String repeat
STRRTRIM	: STR	Removes trailing blanks from a string
STRUP	: STR	Upshifts a string
STRWRITE	: STR	Builds a string from a value list

SYMBOLIC Functions

Name	Type	Description
SYMADDR	: U32	Returns the offset within a type to the specified symbolic field
SYMCONST	: ANY	Returns the value of a declared constant
SYMINSET	: BOOL	Tests for set inclusion
SYMLEN	: U32	Returns the length of the field based on a symbolic path
SYMTYPE	: STR	Returns the symbolic type based on a symbolic path
SYMVAL	: ANY	Returns the value found at a virtual address based on a symbolic path

The formal declaration of functions are presented with the following format:

function_name : *function_return_type* (*function_ parameters*)

The function parameters are presented as follows:

parm_name : *parm_type* [=*default_parm_value*]

func abstolog

Converts an CM absolute code address (ACPTR) to a CM logical code (LCPTR) address.

Syntax

```
abstolog (cmabsaddr)
```

Formal Declaration

```
abstolog:lcptr (cmabsaddr:acptr)
```

Parameters

cmabsaddr The CM absolute code address which is to be converted to a CM logical code address.

Cmabsaddr must be a full CM absolute code address (ACPTR). For Example:

CST(2.102)	CST segment 2 offset 102
CSTX(1.330)	CSTX segment 1 offset 330
LOGTOABS(cmpc)	Explicit absolute conversion

Examples

```
%cmdebug > wl cmpc
PROG %0.1273
%cmdebug > wl logtoabs(cmpc)
CSTX %1.1273

%cmdebug > wl abstolog(cstx(1.1273))
PROG %0.1273
```

func abstolog

Absolute CM address `CSTX 1.1273` is converted into logical address `PROG %0.1273`.

```
%cmdebug > wl abstolog(cst(43.304))
SYS %32.304
```

Absolute CM address `CST 43.304` is converted into logical address `SYS %32.304`.

```
%cmdebug > wl abstolog(cst(103.4274))
GRP %4.4274
```

Absolute CM address `CST 103.4274` is converted into group library logical address `GRP 4.4274`.

Limitations, Restrictions

none

func asc

Evaluates an expression and converts the result to an ASCII string.

Syntax

```
asc (value [formatspec])
```

Formal Declaration

```
asc:str (value:any [formatspec:str = ''])
```

Parameters

value The expression to be formatted.

formatspec An optional format specification string can be specified in order to select specific output base, left or right justification, blank or zero fill, and field width.

A format specification string is a list of selected format directives, optionally separated by blanks or commas in order to avoid ambiguity.

"*directive1 directive2, directive3 directive4 ...* "

The following table lists the supported format directives which can be entered in upper- or lower-case:

`+`	Current output base (`$`, `#`, or `%` prefix displayed)
`-`	Current output base (no prefix)
`+<`	Current input base (`$`, `#`, or `%` prefix displayed)
`-<`	Current input base (no prefix)
`$`	Hex output base (`$` prefix displayed)
`#`	Decimal output base (`#` prefix displayed)
`%`	Octal output base (`%` prefix displayed)

`H`	Hex output base (no prefix)
`D`	Decimal output base (no prefix)
`O`	Octal output base (no prefix)
`A`	ASCII base (use "." for non-printable chars)
`N`	ASCII base (loads actual non-printable chars)
`L`	Left justified
`R`	Right justified
`B`	Blank filled
`Z`	Zero filled
`M`	Minimum field width, based on value
`F`	Fixed field width, based on the type of value
`Wn`	User specified field width *n*
`T`	Typed (display the type of the value)
`U`	Untyped (do not display the type of the value)
`QS`	Quote single (surround w/ single quotes)
`QD`	Quote double (surround w/ double quotes)
`QO`	Quote original (surround w/ original quote character)
`QN`	Quote none (no quotes)

The `M` directive (minimum field width) selects the minimum possible field width necessary to format all significant digits (or characters in the case of string inputs).

The `F` directive (fixed field width) selects a fixed field width based on the type of the value and the selected output base. Fixed field widths are listed in the following table:

Types	hex($,H)	dec(#,D)	oct(%,O)	ascii(A,N)
`S16,U16`	4	6	6	2
`S32,U32`	8	10	11	4
`S64`	16	20	22	8
`SPTR`	8	10	11	4
`LPTR` Class	8.8	10.10	11.11	8
`EADDR` Class	8.16	10.20	11.22	12
`STR`	field width = length of the string			

The `Wn` directive (variable field width) allows the user to specify the desired field width. The `W` directive can be specified with an arbitrary expression. If the specified width is less than the minimum necessary width to display the value, then the user width is ignored, and the minimum width is used instead. All significant digits are always printed. For example:

```
number:"w6"
number:"w2*3"
```

The number of positions specified (either by `Wn` or `F`) does not include the characters required for the radix indicator (if specified) or sign (if negative). Also, the sign and radix indicator is always positioned just preceding the first (leftmost) character.

Zero versus blank fill applies to leading spaces (for right justification) only. Trailing spaces are always blank filled.

In specifications with quotes, the quotes do not count in the number of positions specified. The string is built such that it appears inside the quotes as it would without the quotes.

The `T` directive (typed) displays the type of the value, preceding the value. The `U` directive (untyped) suppresses the display of the type. Types are displayed in uppercase, with a single trailing blank. The width of the type display string varies, based on the type, and it is independent of any specified width (`M`, `F`, or `Wn`) for the value display.

For values of type `LPTR` (long pointer, *sid.offset*, or *seg.offset*) and `EADDR` (extended address, *sid.offset* or *ldev.offset*), two separate format directives can be specified. Each is separated by a dot, ".", to indicate individual formatting choices for the "*sid*" portion and the "*offset*" portion. This is true for all code pointers (`ACPTR` - Absolute Code pointers: `CST, CSTX; LCPTR` - Logical Code Pointers: `PROG, GRP, PUB, LGRP, LPUB, SYS, USER, TRANS`). For example:

```
pc:"+.-, w4.8, r.l, b.z"
```

The following default values are used for omitted format directives. Note that the default format directives depend on the type of value to be formatted:

```
value type                default format
----------                --------------

STR, BOOL                 - R B M U
U16,S16,U32,S32,S64       + R B M U
SPTR                      + R Z F U
LPTR                      +.-  R.L  B.Z  M.F  U
 ACPTR    LCPTR           +.-  R.L  B.Z  M.F  T
  CST      PROG           +.-  R.L  B.Z  M.F  T
  CSTX     GRP            +.-  R.L  B.Z  M.F  T
           PUB            +.-  R.L  B.Z  M.F  T
           LGRP           +.-  R.L  B.Z  M.F  T
           LPUB           +.-  R.L  B.Z  M.F  T
           SYS            +.-  R.L  B.Z  M.F  T
           USER           +.-  R.L  B.Z  M.F  T
           TRANS          +.-  R.L  B.Z  M.F  T
EADDR                     +.-  R.L  B.Z  M.F  U
 SADDR                    +.-  R.L  B.Z  M.F  T
```

Note that absolute code pointers, logical code pointers and extended addresses display their types (`T`) by default. All other types default to (`U`) untyped.

The `Cn` (column n) directive moves the current output buffer position to the specified column position prior to the next write into the output buffer. Column numbers start at column 1. For example:

```
number:"c6"
```

Note: The `Cn` directive is ignored by the `ASC` function but is honored by the `W, WL` and `WP` commands.

Examples

```
$nmdat > var number u32(123)
$nmdat > wl asc(number)
$123
$nmdat > wl asc(number,"-")
123
$nmdat > wl asc(number,"t")
U32 $123
$nmdat > wl asc(number "#")
#291
$nmdat > wl asc(number, 'd')
291
$nmdat > wl asc(number  'fr')
     $123
$nmdat > wl asc(number, "r,w6,-,z")
000123
```

Several examples of formatting an unsigned 32-bit value.

```
$nmdat > var s1="test"
$nmdat > wl asc(s1)
test
$nmdat > wl asc(s1, "QS")
'test'
$nmdat > wl asc(s1  "QO")
"test"
$nmdat > wl asc(s1  "t")
STR test
$nmdat > wl asc(s1  "w2")
test
$nmdat > wl asc(s1, "w2*4,r")
    test
$nmdat > var curwidth 8
$nmdat > wl asc(s1  'wcurwidth, r QD')
"    test"
```

Several examples of formatting a string.

```
$nmdat > var long 2f.42c8

$nmdat > wl asc(long)
$2f.000042c8
$nmdat > wl asc(long, "t")
LPTR $2f.000042c8
$nmdat > wl asc(long, "-.+")
2f.$000042c8
$nmdat > wl asc(long, "#.$ m.m")
#47.$42c8
$nmdat > wl asc(long, "r.r, f.m z")
0000002f.42c8
$nmdat > wl asc(long, "r.r w6.6 z.z")
00002f.0042c8
$nmdat > wl asc(long, 'r.r  w6.2*3  z.z  qd')
"00002f.0042c8"
$nmdat > wl asc(long, 'r.r,w(2*3).(4+2),b.b,$.$')
    $2f.  $42c8

$nmdat > var width 6.6
$nmdat > wl asc(long, 'r.l Wwidth, b.b, $.$')
$2f    .  $42c8
```

Several examples of formatting a long pointer.

Limitations, Restrictions

none

func ascc

Coerces an expression into a string value.

Syntax

```
ascc (value)
```

Formal Declaration

`ascc:str (`*value*`:any)`

Parameters

value An expression to be coerced. Its type can be anything except `BOOL`.

This function takes the internal bit pattern for *value* and treats it as a sequence of ASCII characters. The function value returned is a string made up of these characters, the length of which is determined by the natural size of value according to the following table:

Lengths of Coerced Strings

Parameter Type	String Length
`U16, S16`	2
`U32, S32, SPTR`	4
`S64, LONG` class	8
`EADDR, SADDR`	12
`STR`	Parameter string length

Examples

```
$nmdebug > = ascc(%100+%1)
'A'
$nmdebug > wl strlen (ascc(%100+%1))
$2
```

The expression `%100+%1` is evaluated and coerced into a string value. Since the parameter type is effectively `U16`, the string contains two characters, a NULL (0) followed by a capital “A”.

```
$nmdebug > var bell strdel(ascc(7),1,1)
$nmdebug > wl bell
<beep>
```

This example builds a single-character string and assigns the result to the variable named `bell`. The `STRDEL` function is used to delete the leading NULL character, which is returned in the two-character string returned by the function `ASCC`.

Limitations, Restrictions

none

func bin

Converts a string expression to return a binary value.

Syntax

```
bin (strexp)
```

Formal Declaration

```
bin:any (strexp:str)
```

Parameters

strexp A string expression to be converted from ASCII into binary.

Examples

```
%cmdebug > wl bin("1+2")
%3
```

The contents of the string "1+2" are evaluated as an expression, and the result (3) is converted into a binary value.

Limitations, Restrictions

If the string parameter *strexp* contains an expression that, when evaluated, results in a string, the resulting string is returned. It is *not* converted into a binary value. For example:

```
$nmdat > wl bin ('"A"+"B"')
AB
$nmdat > wl typeof(bin('"A"+"B"'))
STR
```

func bitd

Bit deposit. Deposits a value into a specified range of bits.

Syntax

```
bitd (value position length target)
```

Formal Declaration

bitd:any (*value*:any *position*:s16 *length*:u16 *target*:any)

Parameters

value The value to deposit into the target. Its type is restricted to the INT and PTR classes.

position This parameter specifies the starting bit position (positive value) or the ending bit position (negative value) of the deposit. Regardless of the size of the target, bit positions are always numbered from left to right. The leftmost bit of the target is bit 0.

length The number of bits to deposit. This value may not exceed 64.

target The expression in which to deposit the specified bit pattern. Its type is restricted to the INT and PTR classes.

This function is sensitive to the type of the *target* parameter. As examples, if a S32 or U32 value is passed, the format of the word (start/end positions) is as follows:

```
                     1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------------------------------------------------------+
```

```
|                                                               |
+---------------------------------------------------------------+
```

If a S16 or U16 value is passed, the format of the word (start/end positions) is as follows:

```
                        1 1 1 1 1 1
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
 +-------------------------------+
 |                               |
 +-------------------------------+
```

Examples

For our example, we use a 32-bit word containing the bit pattern for the hex value 4015381f:

```
                      1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
 +---------------------------------------------------------------+
 |0 1 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 1 1 1 0 0 0 0 0 0 1 1 1 1 1|
 +---------------------------------------------------------------+
```

```
$nmdebug > var xx:u32 4015381f
$nmdebug > wl bitd(0,#30,2,xx)
$4015381c
```

Deposit the value 0 into the last two bits of XX.

```
$nmdebug > wl bitd(3,-#1,2,xx)
$c015381f
```

Deposit the value 3 (11) into XX, ENDING at bit position 1.

```
$nmdebug > wl bitd(2d,-#9,6,xx)
$4b55381f
```

Deposit the value 2d (101101) into XX, ending at bit position 9 with a length of 6 (start position would be 4).

Limitations, Restrictions

The value to be deposited is truncated as necessary on the left to fit within the field width of *length*.

If an extended address *target* is passed, the deposit location must fall entirely within the 64-bit offset part. Since `EADDR` types have a total of 96 bits, the valid bit positions are 32 through 95.

func bitx

Bit extract. Extracts a range of bits from an expression.

Syntax

```
bitx (source position length)
```

Formal Declaration

`bitx:any (`*source*`:any` *position*`:s16` *length*`:u16)`

Parameters

source The value from which to extract a range of bits. Its type is restricted to the `INT` and `PTR` classes.

position This parameter specifies the starting bit position (positive value), or the ending bit position (negative value) of the extraction. Regardless of the size of the *source* value, bit positions are always numbered from left to right. The leftmost bit of the *source* is bit 0.

length The number of bits to extract. This value may not exceed 64.

This function is sensitive to the type of the *source* parameter. If a `S32` or `U32` value is passed, the format of the word (start/end positions) is as follows:

```
                     1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------------------------------------------------------+
|                                                               |
+---------------------------------------------------------------+
```

If a `S16` or `U16` value is passed, the format of the word (start/end positions) is as follows:

```
                     1 1 1 1 1 1
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-------------------------------+
|                               |
+-------------------------------+
```

Examples

This is a 32-bit word containing the bit pattern for the hex value 4015381c:

```
                     1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------------------------------------------------------+
|0 1 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 1 1 1 0 0 0 0 0 0 1 1 1 0 0|
+---------------------------------------------------------------+
```

```
$nmdebug > var xx:u32 4015381c
$nmdebug > wl bitx(xx,#10,5)
$a
```

Extract five bits starting at position 10 (this yields the bit pattern 01010).

```
$nmdebug > wl bitx(xx,-#14,5)
$a
```

Extract five bits ending at position 14 (this yields the bit pattern 01010). This is the same field of bits as in the previous example.

Limitations, Restrictions

If an extended address *source* is passed, the extraction location must fall entirely within the 64-bit offset part. Since EADDR types have a total of 96 bits, the valid bit positions are 32 through 95.

func bool

Coerces an expression into a Boolean value.

Syntax

```
bool (value)
```

Formal Declaration

bool:bool (*value*:any)

Parameters

value	An expression to be coerced. Its type can be anything except `STR`. The coercion will evaluate to `FALSE` if the value of the expression is 0; otherwise, the value of the coercion will be `TRUE`.

Examples

```
$nmdebug > wl bool(0)
FALSE

$nmdebug > wl bool(1)
TRUE

$nmdebug > wl bool(123)
TRUE

$nmdebug > wl bool(a.c00023c4)
TRUE

$nmdebug > wl bool(0.0)
```

```
FALSE
```

Limitations, Restrictions

none

func bound

Checks for an existing definition of an operand and returns its definition type.

Syntax

```
bound (operand)
```

The BOUND function uses the name in *operand* to check for an existing definition for that name. The type of the definition is returned in a string. The following table lists all possible types:

NUMBER	A valid numeric expression (in current input base)
ENV	A predefined environment variable
VAR	A user defined variable
FUNC	A predefined function
MACRO	A user defined macro
PROCEDURE	A valid procedure name (in current mode)
ALIAS	An alias definition
COMMAND	A command name
WINDOW_COMMAND	A window command name
UNDEFINED	No definition is currently bound

The table is searched in order from top to bottom. The first type which matches is returned. Additional matches may be possible but are not tested.

Formal Declaration

```
bound:str (operand:str)
```

Parameters

operand A string expression naming the *operand* for which the definition type is returned.

Examples

```
$nmdebug > if bound('list') <> 'VAR' then var list slowbuildlist('ALL')
```

BOUND is often used to determine if a particular variable has been defined. In this example, which might typically be found in a macro, BOUND is used to test for the prior definition of the variable named "list". If the variable has not yet been defined, then it is created and assigned the return value from the macro named slowbuildlist.

```
$nmdebug > wl bound('123')
NUMBER
$nmdebug > wl bound('add')
NUMBER
```

123 and ADD are both numbers (in the current input base).

```
$nmdebug > wl bound('s')
ENV
```

S is an environment variable (the CM S register). Note that S is also a command name (Single Step), but only the first match is returned.

```
$nmdebug > wl bound('BOUND')
FUNC
```

BOUND is a function (in fact, the one this page is describing).

```
$nmdebug > wl bound('slowbuildlist')
MACRO
```

SLOWBUILDLIST is a user defined macro.

```
$nmdebug > wl bound('12w')
UNDEFINED
```

12w is undefined. No existing definition for 12w could be located.

Limitations, Restrictions

none

func btow

Byte to word. Converts a CM DB-relative byte address to a CM DB-relative word address.

Syntax

```
btow (byteaddress [splitstack])
```

Formal Declaration

```
btow:I16 (byteaddress:I16 [splitstack:bool=FALSE])
```

Parameters

byteaddress The CM DB-relative byte address which is to be converted into a CM DB-relative word address.

splitstack If *splitstack* is `FALSE`, then *byteaddress* is assumed to be within the current process's CM stack. The byte address is logically shifted right by one bit. If the result is greater than the current S location, then `%100000` is added. This effectively turns on the sign bit. By default, *splitstack* is `FALSE`.

If *splitstack* is `TRUE`, then *byteaddress* is assumed to be a data segment (DST) relative offset. The byte address is logically shifted right by one bit. No special test for the current location of S is performed.

Examples

```
%cmdebug > dr
DBDST=%204   DB=%1000    X=%0      STATUS=%100030=(Mitroc CCG 030) PIN=%40
SDST=%204    DL=%177650  Q=%726    S=%41767   CMPC=SYS %27.253
CIR=%041601  MAPFLAG=%1  MAPDST=%0
```

func btow

```
%cmdebug > wl btow (100002)
%40001

%cmdebug > wl btow (177776)
%177777
```

These examples assume the current CM registers which are displayed above. Note the large stack usage above DB.

```
%cmdebug > dr
DBDST=%204   DB=%70000   X=%0      STATUS=%100030=(Mitroc CCG 030) PIN=%40
SDST=%204    DL=%110650  Q=%726    S=%1204    CMPC=SYS %27.253
CIR=%041601  MAPFLAG=%1  MAPDST=%0

%cmdebug > wl btow (177776)
%177777

%cmdebug > wl btow (100002)
%140001

%cmdebug > wl btow (40002)
%120001
```

These examples assume the current CM registers displayed above. Note the huge DL area.

Limitations, Restrictions

none

func cisetvar

Sets a new value for the specified CI (MPE XL Command Interpreter) variable.

Syntax

```
cisetvar (civarname newvalue)
```

This function is implemented by calling the `HPCIPUTVAR` intrinsic. String variables are stored as strings. They are not interpreted numerically.

Formal Declaration

```
cisetvar:bool (civarname:str newvalue:any)
```

Parameters

civarname The name of the CI variable to be assigned a new value.

newvalue The new value to be assigned to the specified CI variable.

Examples

```
$nmdebug > wl cisetvar ("testvar", #123);
TRUE
```

Assign the value decimal 123 to the CI variable named `testvar`. The result, `TRUE`, implies that the assignment was successful.

```
$nmdebug > wl civar ("testvar"):"d"
123
$nmdebug > :showvar testvar
TESTVAR = 123
```

Confirm that the value was set by retrieving the value using the `CIVAR` function and by executing a CI command to display the variable's value.

Limitations, Restrictions

none

func civar

Returns the current value of a CI (MPE XL Command Interpreter) variable.

Syntax

```
civar (civarname [stropt])
```

This function is implemented by calling the `HPCIGETVAR` intrinsic.

Formal Declaration

```
civar:any (civarname:str [stropt:str="NOEV"])
```

Parameters

civarname The name of the CI variable.

stropt A string that determines whether the CI should attempt to evaluate the named variable.

`EVALUATE`	Evaluate the CI variable
`NOEVALUATE`	Do not evaluate the CI variable (Default)

This string parameter can be abbreviated.

Examples

```
$nmdebug > wl civar ("hpgroup");
DEMO

$nmdebug > wl civar ("hpaccount");
TELESUP
```

Display the current value of the CI variables named `HPGROUP` and `HPACCOUNT`.

```
$nmdebug > wl civar( "hpusercapf" )
SM,AM,AL,GL,DI,OP,CU,UV,LG,PS,NA,NM,CS,ND,SF,BA,IA,PM,MR,DS,PH
```

Display the current value of the CI variable `HPUSERCAPF`.

func civar

```
$nmdat > :showvar one
ONE = !TWO
$nmdat > :showvar two
TWO = 2

$nmdat > wl civar("one")
!TWO
$nmdat > wl civar("one" "EVAL")
2
```

Two CI variables have already been defined. Variable `one` references variable `two` which is assigned the value of 2.

The first use of the function `CIVAR` defaults to `NOEVALUATE`, and as a result the value of `one` is returned as `!TWO`.

In the second use of the function `CIVAR`, the *stropt* is explicitly specified as `EVALUATE`, and so the MPE XL CI evaluates the value of `one`, which indirectly references the variable `two`, and the final result of 2 is returned.

Limitations, Restrictions

none

func cmaddr

Converts a CM procedure name (or primary/secondary entry point) to a CM logical code address.

Syntax

```
cmaddr (procname [lib])
```

The `CMADDR` function is especially useful for locating CM procedures when the current mode is NM, since procedure name lookups are based on the current mode. `CMADDR` explicitly requests a CM procedure name lookup.

Compatibility Mode code may be emulated, or translated into NM. This function always returns addresses based on emulated CM object code.

Another function (`CMTONMNODE`) can be used to locate the nearest corresponding NM node point address if the CM object code has been translated into NM.

Refer to Appendix C for discussion of CM Object Code Translation, node points, and breakpoints in translated CM mode.

Formal Declaration

```
cmaddr:lcptr (procname:str [lib:str=''])
```

Parameters

procname The CM procedure name to be located and converted to a CM logical code address. Primary and secondary entry points can be located by preceding the procedure name with a question mark.

lib An optional string which indicates where the search for the named procedure should begin. By default, the program and then all currently loaded libraries will be searched.

PROG	Search the program file
GRP	Search the group library
PUB	Search the account library
LGRP	Search the logon group library
LPUB	Search the logon account library
SYS	Search the system library

Examples

```
$nmdebug > wl cmaddr( "my'lib'proc" "pub")
PUB $2.124
```

Look up the start address of `my'lib'proc` in the CM group library.

```
$nmdebug > wl cmaddr( "?fopen" ):"%.o"
SYS %22.5000
```

Look up the entry point address of `fopen` and display the address in octal.

Limitations, Restrictions

none

func cmbpaddr

Returns the address corresponding to the indicated CM breakpoint index.

Syntax

```
cmbpaddr (bpindex [pin])
```

This function accepts an index for an existing CM breakpoint and returns the address where the breakpoint is located. The default action is to look for breakpoints set by the current PIN. Breakpoint addresses for other pins (including the global PIN) may be retrieved by utilizing the optional *pin* parameter.

Formal Declaration

```
cmbpaddr:lcptr (bpindex:u16 [pin:s16=0])
```

Parameters

bpindex	The breakpoint index to look for.
pin	Look for breakpoints set by this PIN. Default is the caller's PIN (a pin of 0 implies this). To specify system (global) breakpoints, use a -1 (or 32762) as the PIN.

Examples

```
%cmdebug > bl
CM      [1] PROG % 2.3401    TEST'SCREEN+%26
CM      [2] PROG % 0.347     TEST'FILES+%0
CM     @[1] SYS  % 161.5274  FOPEN+%0
```

First, list the existing breakpoints.

```
%cmdebug > wl cmbpaddr(1)
PROG %2.3401

%cmdebug > wl cmbpaddr(1, -1)
SYS %161.5274
```

Now use the function to return the address associated with process local breakpoint number one and then with system breakpoint number one.

Limitations, Restrictions

none

func cmbpindex

Returns the CM breakpoint index associated with the indicated CM code address.

Syntax

```
cmbpindex (cmaddr [pin])
```

This function accepts the address (either logical or absolute) of an existing CM breakpoint and returns the logical index number associated with that breakpoint. The default action is to look for breakpoints set by the current PIN. Breakpoint indices for other PINs (including the global PIN) may be retrieved by utilizing the optional *pin* parameter.

Formal Declaration

`cmbpindex:u16 (`*cmaddr*`:cptr` [*pin*:s16=0])

Parameters

cmaddr
: Look for this address in the CM breakpoint table. Both logical and absolute code addresses are supported.

pin
: Look for breakpoints set by this PIN. Default is the caller's PIN (a *pin* of 0 implies this). To specify system (global) breakpoints, use a -1 (or 32762) as the PIN.

Examples

```
%cmdebug > bl
CM      [1] PROG % 2.3401    TEST'SCREEN+%26
CM      [2] PROG % 0.347     TEST'FILES+%0
CM     @[1] SYS  % 161.5274  FOPEN+%0
```

First, list the existing breakpoints.

```
%cmdebug > wl cmbpindex(TEST'FILES)
%2
```

Go find the CM breakpoint index associated with the address `TEST'FILES`.

func cmbpindex

```
%cmdebug > wl cmbpindex(FOPEN)
No breakpoint exists in the breakpoint tables with that address.  (error
#1080)

Error evaluating a predefined function.  (error #4240)
  function is"cmbpindex"
    wl cmbpindex(FOPEN)
```

Now, go find the breakpoint index for the breakpoint at `FOPEN`. In this example we get an error. This is because we did not specify a PIN and thus searched only for process local breakpoints. We do not have a process local breakpoint at `FOPEN`.

```
%cmdebug > wl cmbpindex(FOPEN, -1)
%1
```

Go find the breakpoint index for the breakpoint at `FOPEN`. This time we specify a -1 to tell the function to search the list of system breakpoints.

Limitations, Restrictions

none

func cmbpinstr

Returns the original CM instruction at a specified CM code address where a CM breakpoint has been set.

Syntax

```
cmbpinstr (cmaddr [pin])
```

This function accepts the address (either logical or absolute) of an existing CM breakpoint and returns the instruction associated with that breakpoint. The default action is to look for breakpoints set by the current PIN. Breakpoint indices for other PINs (including the global pin) may be retrieved by utilizing the optional *pin* parameter.

Formal Declaration

`cmbpinstr:s16 (`*cmaddr*`:cptr [`*pin*`:s16=0])`

Parameters

cmaddr Look for this address in the CM breakpoint table. Both logical and absolute code addresses are supported.

pin Look for breakpoints set by this PIN. Default is the caller's PIN (a *pin* of 0 implies this). To specify system (global) breakpoints, use a -1 (or 32762) as the PIN.

Examples

```
%cmdebug > dc FOPEN,1
%005274:  FOPEN+%0                      004300  ..   STAX, NOP
```

Display code at the address of `FOPEN` so we can see what the current instruction at that address is.

func cmbpinstr

```
%cmdebug > b FOPEN
added: CM      [1] SYS  % 161.5274  FOPEN+%0

%cmdebug > dc FOPEN,1
%005274:  FOPEN+%0                    003600  <.   BRKP
```

Now set a breakpoint at `FOPEN` and display the code there. The old instruction has been replaced with a breakpoint instruction.

```
%cmdebug > wl cmbpinstr(FOPEN)
%4300
```

Use the function to look up the actual instruction. The instruction that is stored in the system breakpoint table is returned by the function.

Limitations, Restrictions

none

func cmentry

Returns the CM (primary) entry point address of the CM procedure containing the specified CM logical code address.

Syntax

```
cmentry (cmlogaddr)
```

Entry point addresses correspond to the `ENTRY` column in the `PMAP` generated by the Segmenter. See the CM program example below.

Formal Declaration

`cmentry:lcptr (`*cmlogaddr*`:lcptr)`

Parameters

cmlogaddr A CM logical code address. The entry point of the surrounding level one CM procedure is returned as a CM logical code address.

Cmlogaddr must be a full CM logical code address (`LCPTR`). For example:

`CMPC`	Current CM program counter
`CMPW+4`	Top of CM program window + 4
`PROG(2.102)`	Program file logical seg 2 offset 102
`fopen+102`	CM procedure `fopen` + %102 (assumes CM mode)
`cmaddr('fopen')+%102`	CM procedure `fopen` + %102 (NM or CM mode)

FINAL TRIM SIZE : 7.5 in x 9.0 in

func cmentry

Examples

Assume that the following single segment CM program has been compiled, linked with the PMAP and FPMAP options, and is now being executed:

```
PROGRAM test (input,output);

PROCEDURE one;
begin {one}
  writeln('ONE');
end;  {one}

PROCEDURE two;

    PROCEDURE three;
    begin {three}
      writeln('THREE');
    end;  {three}

begin {two}
  writeln('TWO');
  three;
end;  {two}

begin {main body}     { Outer block is named "ob'" by the compiler }
  one;
  two;
end.  {main body}

PROGRAM FILE PTEST.DEMO.TELESUP

SEG'             0
   NAME            STT  CODE ENTRY SEG
   OB'               1     0    13
   TERMINATE'        5               ?
   P'RESET           6               ?
   P'REWRITE         7               ?
   P'CLOSEIO        10               ?
```

```
   P'INITHEAP'3000  11               ?
   TWO               2    71   123
   P'WRITELN        12               ?
   P'WRITESTR       13               ?
   ONE               3   142   155
   SEGMENT LENGTH        210

PRIMARY DB          2    INITIAL STACK   10240    CAPABILITY        600
SECONDARY DB      430    INITIAL DL          0    TOTAL CODE        210
TOTAL DB          432    MAXIMUM DATA        ?    TOTAL RECORDS      11
ELAPSED TIME   00:00:01.365                  PROCESSOR TIME   00:00.740

END OF PREPARE
```

func cmentry

```
%cmdebug > wl ob'
PROG %0.0
%cmdebug > wl cmstart(ob')
PROG %0.0
```

Two methods of displaying the start address of the procedure `ob'`.

```
%cmdebug > wl ?ob'
PROG %0.13
%cmdebug > wl cmentry(ob')
PROG %0.13
```

Two methods of displaying the entry address of the procedure `ob'`.

```
%cmdebug > wl cmstart(one)
PROG %0.142

%cmdebug > wl cmentry(one)
PROG %0.155

%cmdebug > wl cmstart(two)
PROG %0.71

%cmdebug > wl cmentry(two)
PROG %0.123
```

Limitations, Restrictions

The names and addresses of nested CM procedures, such as procedure `three`, are *not* available within the CM `FPMAP` records. Addresses that fall within nested procedures (`three`) are returned as offsets relative to the parent procedure (`two`).

func cmg

Returns the virtual address (SPTR) of a process's CMGLOBALS record.

Syntax

```
cmg (pin)
```

Formal Declaration

```
cmg:sptr (pin:u16)
```

Parameters

pin The process identification number (PIN) for which the address of the CMGLOBALS record is to be returned.

Examples

```
$nmdebug > wl cmg($8)
$c4680000
```

Limitations, Restrictions

If the PIN does not exist, the function result is undefined and an error status is set.

func cmnode

Returns the address of the closest CM node point corresponding to the specified CM logical code address.

Syntax

```
cmnode (cmlogaddr [node])
```

Refer to appendix C for a discussion of CM Object Code Translation (OCT), node points, and breakpoints in translated CM code.

Formal Declaration

```
cmnode:lcptr (cmlogaddr:lcptr [node:str="PREV"])
```

Parameters

cmlogaddr — The CM logical code address within a translated code segment for which the closest CM node point is desired.

Cmlogaddr must be a full CM logical code address (`LCPTR`). For example:

`CMPC`	Current CM program counter
`CMPW+4`	Top of CM program window + 4
`PROG(2.102)`	Program file logical seg 2 offset 102
`fopen+102`	CM procedure `fopen` + %102 (assumes CM mode)
`cmaddr('fopen')+%102`	CM procedure `fopen` + %102 (NM or CM mode)

node — The desired node point, either `PREV` (closest previous node) or `NEXT` (closest next node). If unspecified, then `PREV` is assumed.

Examples

```
%cmdebug > wl cmnode(sys(2.226))
SYS %2.224
```

Print the CM address of the closest CM previous (by default) node point.

```
%cnmdebug > wl cmnode(sys(2.226), "next")
SYS %2.232
```

Print the CM address of the closest CM next node point.

Limitations, Restrictions

none

func cmproc

Returns the CM procedure name and offset corresponding to a CM logical code address.

Syntax

```
cmproc (cmlogaddr)
```

The string returned by `CMPROC` can be either of the two following formats :

?*entrypoint_name*
or
procedure_name + *base offset*

Detailed descriptions of each of the above return strings follow:

entrypoint_name The name of the CM entry point (primary/secondary).

procedure_name The name of the CM procedure.

base The output radix used to represent *offset*, which depends on the current output base.

```
%    Octal
$    Hexadecimal
#    Decimal
```

offset If the offset is nonzero, then it is returned, appended to the procedure name. The offset is formatted based on the current fill, justification, and output base values.

Formal Declaration

`cmproc:str (`*cmlogaddr*`:lcptr)`

Parameters

cmlogaddr The CM logical code address for which the CM symbolic procedure name/offset is to be returned.

Cmlogaddr must be a full CM logical code address (`LCPTR`). For example:

`CMPC`	Current CM program counter
`CMPW+4`	Top of CM program window + 4
`PROG(2.102)`	Program file logical seg 2 offset 102
`fopen+102`	CM procedure `fopen` + %102 (assumes CM mode)
`cmaddr('fopen')+%102`	CM procedure `fopen` + %102 (NM or CM mode)

Examples

Assume that the following single-segment CM program has been compiled, linked with the `PMAP and FPMAP` options, and is now being executed:

```
PROGRAM test (input,output);

PROCEDURE one;
begin {one}
  writeln('ONE');
end;  {one}

PROCEDURE two;

    PROCEDURE three;
    begin {three}
      writeln('THREE');
```

```
      end;  {three}

  begin {two}
    writeln('TWO');
    three;
  end;  {two}

  begin {main body}     { Outer block is named "ob'" by the compiler }
    one;
    two;
  end.  {main body}


PROGRAM FILE PTEST.DEMO.TELESUP

SEG'               0
   NAME            STT  CODE ENTRY SEG
   OB'               1     0    13
   TERMINATE'        5               ?
   P'RESET           6               ?
   P'REWRITE         7               ?
   P'CLOSEIO        10               ?
   P'INITHEAP'3000  11               ?
   TWO               2    71   123
   P'WRITELN        12               ?
   P'WRITESTR       13               ?
   ONE               3   142   155
   SEGMENT LENGTH        210

PRIMARY DB          2    INITIAL STACK   10240    CAPABILITY        600
SECONDARY DB      430    INITIAL DL          0    TOTAL CODE        210
TOTAL DB          432    MAXIMUM DATA        ?    TOTAL RECORDS      11
ELAPSED TIME   00:00:01.365                  PROCESSOR TIME   00:00.740

END OF PREPARE
       %cmdebug > wl cmproc(prog(0.142))
       ONE+%0
```

```
%cmdebug > wl cmproc(prog(0.155))
?ONE

%cmdebug > wl cmproc(prog(0.147))
ONE+%5

%cmdebug > wl cmproc(prog(0.66))
OB'+%66

%cmdebug > wl cmproc(prog(0.101))
TWO+%10



%cmdebug > wl cmproc(sys(22.5000))
?FOPEN

%cmdebug > wl cmproc(sys(22.5035))
FOPEN+%41

%cmdebug > wl cmproc(sys(22.5036))
?MUSTOPEN

%cmdebug > wl cmproc(sys(22.5037))
FOPEN+%43
```

The primary entry point `?FOPEN`, and the secondary entry point `?MUSTOPEN` are located, along with two other offsets within system SL procedure `FOPEN`.

Limitations, Restrictions

The names and addresses of nested CM procedures, such as procedure `three`, are not available within the CM `FPMAP` records. Addresses which fall within nested procedures (`three`) are returned as offsets relative to the parent procedure (`two`).

func cmproclen

Returns the length of the CM procedure which contains the specified CM logical code address.

Syntax

```
cmproclen (cmlogaddr)
```

The procedure length (from procedure start to procedure end) is returned in CM (16-bit) words.

Formal Declaration

`cmproclen:u16 (`*cmlogaddr*`:lcptr)`

Parameters

cmlogaddr The CM logical code address of a procedure whose length is desired.

Cmlogaddr must be a full CM logical code address (`LCPTR`). For example:

`CMPC`	Current CM program counter
`CMPW+4`	Top of CM program window + 4
`PROG(2.102)`	Program file logical seg 2 offset 102
`fopen+102`	CM procedure `fopen` + %102 (assumes CM mode)
`cmaddr('fopen')+%102`	CM procedure `fopen` + %102 (NM or CM mode)

Examples

```
%cmdebug > wl cmproclen(cmpc)
%843
```

Print the length of the current CM procedure located at the CM program counter CMPC.

```
%cmdebug > wl cmproclen(fopen)
%1642
```

Print the length of the CM procedure fopen.

func cmproclen

Assume that the following single segment CM program has been compiled, linked with the PMAP and FPMAP options, and is now being executed:

```
PROGRAM test (input,output);

PROCEDURE one;
begin {one}
  writeln('ONE');
end;  {one}

PROCEDURE two;

    PROCEDURE three;
    begin {three}
      writeln('THREE');
    end;  {three}

begin {two}
  writeln('TWO');
  three;
end;  {two}

begin {main body}     { Outer block is named "ob'" by the compiler }
  one;
  two;
end.  {main body}
```

```
PROGRAM FILE PTEST.DEMO.TELESUP

SEG'              0
   NAME            STT  CODE ENTRY SEG
   OB'               1     0    13
   TERMINATE'        5               ?
   P'RESET           6               ?
   P'REWRITE         7               ?
   P'CLOSEIO        10               ?
```

```
   P'INITHEAP'3000  11              ?
   TWO               2    71   123
   P'WRITELN        12              ?
   P'WRITESTR       13              ?
   ONE               3   142   155
   SEGMENT LENGTH        210

PRIMARY DB          2    INITIAL STACK   10240    CAPABILITY        600
SECONDARY DB      430    INITIAL DL          0    TOTAL CODE        210
TOTAL DB          432    MAXIMUM DATA        ?    TOTAL RECORDS      11
ELAPSED TIME   00:00:01.365                  PROCESSOR TIME   00:00.740

END OF PREPARE
```

func cmproclen

```
%cmdebug > wl cmstart(ob')
PROG %0.0
%cmdebug > wl cmstart(two)
PROG %0.71
%cmdebug > wl cmstart(one)
PROG %0.142

%cmdebug > wl cmproclen(ob')
%71
%cmdebug > wl cmstart(two) - cmstart(ob')
%71

%cmdebug > wl cmproclen(two)
%51
%cmdebug > wl cmstart(one)-cmstart(two)
%51

%cmdebug > wl cmproclen(one)
%30
```

Limitations, Restrictions

The names and addresses of nested CM procedures, such as procedure `three`, are not available within the CM `FPMAP` records. Addresses that fall within nested procedures (`three`) are returned as offsets relative to the parent procedure (`two`).

func cmseg

Returns the CM segment name for the specified CM logical code address.

Syntax

```
cmseg (cmlogaddr)
```

Formal Declaration

cmseg:str (*cmlogaddr*:lcptr)

Parameters

cmlogaddr The CM logical code address for which the segment name is desired.

Cmlogaddr must be a full CM logical code address (LCPTR). For example:

CMPC	Current CM program counter
CMPW+4	Top of CM program window + 4
PROG(2.102)	Program file logical seg 2 offset 102
fopen+102	CM procedure fopen + %102 (assumes CM mode)
cmaddr('fopen')+%102	CM procedure fopen + %102 (NM or CM mode)

Note that the offset portion of the LCPTR address is required, but ignored.

func cmseg

Examples

```
$cmdebug > wl cmseg(prog(0.0))
SEG'

$cmdebug > wl cmseg(fopen)
XLSEG11
```

Limitations, Restrictions

none

func cmstackbase

Returns the starting virtual address of a process's compatibility mode stack.

Syntax

```
cmstackbase (pin)
```

Formal Declaration

```
cmstackbase:lptr (pin:u16)
```

Parameters

pin The process identification number (PIN) for which the starting virtual address of the CM stack is to be returned.

Examples

```
$nmdebug > wl cmstackbase(%10)
$2c4.40011cb0
```

Display the virtual address of the CM stack base for PIN %10.

```
$nmdat > wl "CM stack size = ", cmstacklimit(pin) - cmstackbase(pin) + 1
CM stack size = $4350
```

Calculate and display the CM stack length (in bytes) for the current PIN.

Limitations, Restrictions

If the PIN does not exist, the function result is undefined and an error status is set.

func cmstackdst

Returns the DST number for a process's compatibility mode stack.

Syntax

```
cmstackdst (pin)
```

Formal Declaration

cmstackdst:u16 (*pin*:u16)

Parameters

pin The process identification number (PIN) for which the DST number of the CM stack is to be returned.

Examples

```
$nmdebug > wl cmstackdst(8)
$4f
```

Limitations, Restrictions

If the PIN does not exist, the function result is undefined and an error status is set.

func cmstacklimit

Returns the virtual address for the limit of a process's compatibility mode stack.

Syntax

```
cmstacklimit (pin)
```

The virtual address of the last usable byte in the CM stack is returned.

Formal Declaration

```
cmstacklimit:lptr (pin:u16)
```

Parameters

pin — The process identification number (PIN) for which the virtual address of the CM stack limit is to be returned.

Examples

```
$nmdebug > wl cmstacklimit(%10)
$2c4.40015fff
```

Display the virtual address of the CM stack limit for pin %10.

```
$nmdat > wl "CM stack size = ", cmstacklimit(pin) - cmstackbase(PIN) +1
CM stack size = $4350
```

Calculate and display the CM stack length (in bytes) for the current PIN.

Limitations, Restrictions

If the PIN does not exist, the function result is undefined and an error status is set.

func cmstart

Returns the starting point of the procedure containing the indicated CM logical code address.

Syntax

```
cmstart (cmlogaddr)
```

Start addresses correspond to the `CODE` column in the `PMAP` generated by the Segmenter. Refer to the CM program example below.

Formal Declaration

`cmstart:lcptr (`*cmlogaddr*`:lcptr)`

Parameters

cmlogaddr A CM logical code pointer address for which the starting address of the containing level one procedure is to be returned.

Cmlogaddr must be a full CM logical code address (`LCPTR`). For example:

`CMPC`	Current CM program counter
`CMPW+4`	Top of CM program window + 4
`PROG(2.102)`	Program file logical seg 2 offset 102
`fopen+102`	CM procedure `fopen` + %102 (assumes CM mode)
`cmaddr('fopen')+%102`	CM procedure `fopen` + %102 (NM or CM mode)

Examples

Assume that the following single segment CM program has been compiled, linked with the PMAP and FPMAP options, and is now being executed:

```
PROGRAM test (input,output);

PROCEDURE one;
begin {one}
  writeln('ONE');
end;  {one}

PROCEDURE two;

    PROCEDURE three;
    begin {three}
      writeln('THREE');
    end;  {three}

begin {two}
  writeln('TWO');
  three;
end;  {two}

begin {main body}     { Outer block is named "ob'" by the compiler }
  one;
  two;
end.  {main body}
```

```
PROGRAM FILE PTEST.DEMO.TELESUP

SEG'                0
   NAME             STT  CODE ENTRY SEG
   OB'                1     0    13
   TERMINATE'         5               ?
   P'RESET            6               ?
   P'REWRITE          7               ?
   P'CLOSEIO         10               ?
```

```
   P'INITHEAP'3000  11               ?
   TWO               2    71   123
   P'WRITELN        12               ?
   P'WRITESTR       13               ?
   ONE               3   142   155
   SEGMENT LENGTH        210

PRIMARY DB           2    INITIAL STACK   10240    CAPABILITY        600
SECONDARY DB       430    INITIAL DL          0    TOTAL CODE        210
TOTAL DB           432    MAXIMUM DATA        ?    TOTAL RECORDS      11
ELAPSED TIME   00:00:01.365                     PROCESSOR TIME   00:00.740

END OF PREPARE
```

func cmstart

```
%cmdebug > wl ob'
PROG %0.0
%cmdebug > wl cmstart(ob')
PROG %0.0
```

Two methods of displaying the start address of the procedure ob'.

```
%cmdebug > wl ?ob'
PROG %0.13
%cmdebug > wl cmentry(ob')
PROG %0.13
```

Two methods of displaying the entry address of the procedure ob'.

```
%cmdebug > wl cmstart(one)
PROG %0.142

%cmdebug > wl cmentry(one)
PROG %0.155

%cmdebug > wl cmentry(one+10)
PROG %0.155

%cmdebug > wl cmstart(two)
PROG %0.71

%cmdebug > wl cmstart(two+5)
PROG %0.71

%cmdebug > wl cmentry(two)
PROG %0.123
```

Limitations, Restrictions

The names and addresses of nested CM procedures, such as procedure three, are not available within the CM FPMAP records. Addresses that fall within nested procedures (three) are returned as offsets relative to the parent procedure (two).

func cmtonmnode

Returns the address of the closest NM node point corresponding to the specified CM logical code address.

Syntax

```
cmtonmnode (cmlogaddr [node])
```

Refer to Appendix C for a discussion of CM Object Code Translation (OCT) node points, and breakpoints in translated CM code.

Formal Declaration

```
cmtonmnode:trans (cmlogaddr:lcptr [node:str=PREV])
```

Parameters

cmlogaddr The CM logical code address of translated code for which the closest NM node point is desired.

Cmlogaddr must be a full CM logical code address (`LCPTR`). For example:

`CMPC`	Current CM program counter
`CMPW+4`	Top of CM program window + 4
`PROG(2.102)`	Program file logical seg 2 offset 102
`fopen+102`	CM procedure `fopen` + %102 (assumes CM mode)
`cmaddr('fopen')+%102`	CM procedure `fopen` + %102 (NM or CM mode)

node The desired node point, either `PREV` (closest previous node) or `NEXT` (closest next node). If unspecified, then `PREV` is assumed.

Examples

```
$nmdebug > wl cmtonmnode(sys(2.%226))
TRANS $21.24024
```

Print the NM address of the closest CM previous (by default) node point.

```
$nmdebug > wl cmtonmnode(sys(2.%226), "next")
TRANS $21.2404c
```

Print the NM address of the closest CM next node point.

Limitations, Restrictions

none

func cmva

Returns the virtual address of a specified CM code address.

Syntax

```
cmva (cmaddr [pin])
```

Compatibility mode code may be emulated or translated into NM. This function always returns addresses based on emulated CM object code.

Another function (`CMTONMNODE`) can be used to locate the nearest corresponding NM node point address if the CM object code has been translated into NM.

Refer to appendix C for a discussion of CM object code translation, node points, and breakpoints in translated CM code. See the `T(ranslate)` commands in Chapter 4 for additional information.

Formal Declaration

```
cmva:lptr (cmaddr:cptr [pin:u16 = 0])
```

Parameters

cmaddr — A CM code address to be converted to a virtual address. Both logical and absolute code addresses are supported.

pin — The process identification number (PIN) to which the code segment belongs. If *pin* is not specified, it defaults to 0, which is defined to be the current PIN.

Examples

```
$nmdebug > wl cmva(cmpc)
$26.0000124c
```

Convert the current CM logical address pointer, for the current PIN, to a NM virtual address and display the result.

```
$nmdebug > wl cmva(SYS(%23.%250,$24))
$3f.00000250
```

Convert CM logical address `SYS %23.%250`, for the process associated with PIN $24, to a NM virtual address and display the result.

```
$nmdebug > wl cmva(CST(3.0))
$21.000034c4
```

Convert absolute CM address `CST` 3.0, for the current PIN, to a NM virtual address and display the result.

Limitations, Restrictions

none

func cst

Coerces an expression into a `CST` absolute code pointer (`ACPTR`).

Syntax

```
cst (value)
```

CM program segments are loaded into the CSTX. CM library segments are loaded into the CST.

During the evaluation of the parameter to the `CST` function, the following CM search path is used for procedure name lookups:

```
GRP, PUB, LGRP, LPUB, SYS
```

Formal Declaration

```
cst:cst (value:any)
```

Parameters

value An expression to be coerced. All types are valid.

Derivation of the CST Bit Pattern

Parameter Type	Action
BOOL	0.1 if TRUE, 0.0 if FALSE.
U16 S16	Set the high-order 32 bits (SID or segment part) to zero. Right justify the original 16-bit value in the low-order 32 bits (offset part) with zero fill.
U32 S32 SPTR	Set the high-order 32 bits (SID or segment part) to zero. Transfer the original bit pattern into the low-order 32 bits (offset part) unchanged.
LPTR SYS PROG USER GRP TRANS PUB CST LGRP CSTX LPUB	Transfer both parts of the address unchanged.
STR	Transfer the ASCII bit pattern for the last eight characters in the string. Strings shorter than eight characters are treated as if they were extended on the left with nulls.

Examples

```
%cmdebug > wl cst(12.304)
CST %12.304
```

Coerce the simple long pointer into a `CST` absolute code pointer.

```
%cmdebug > wl sort
PROG %4.3302

%cmdebug > wl grp (sort)
GRP %2.1364

%cmdebug > wl cst (sort)
CST %73.1364
```

Print the address of the procedure named `sort`. The first lookup uses the standard procedure name lookup search path and finds the procedure `sort` in the program file. The second lookup restricts the search path to the group library, and another `sort` procedure is located. The third lookup restricts the search path to all of the currently loaded libraries, and the second procedure is located again (within the group library).

```
%cmdebug > wl cst(sys(24.630))
CST %24.630
```

The coercion simply changes the associated absolute file. Note that no complicated conversion or range checking is performed.

Limitations, Restrictions

none

func cstx

Coerces an expression into a `CSTX` absolute code pointer (`ACPTR`).

Syntax

```
cstx (value)
```

CM program segments are loaded into the CSTX. CM library segments are loaded into the CST.

During the evaluation of the parameter to the `CSTX` function, the CM search path is limited to the program file (`PROG`).

Formal Declaration

```
cstx:cstx (value:any)
```

Parameters

value An expression to be coerced. All types are valid.

Derivation of the CSTX Bit Pattern

Parameter Type	Action
BOOL	0.1 if TRUE, 0.0 if FALSE.
U16 S16	Set the high-order 32 bits (SID or segment part) to zero. Right justify the original 16-bit value in the low-order 32 bits (offset part) with zero fill.
U32 S32 SPTR	Set the high-order 32 bits (SID or segment part) to zero. Transfer the original bit pattern into the low-order 32 bits (offset part) unchanged.
LPTR SYS PROG USER GRP TRANS PUB CST LGRP CSTX LPUB	Transfer both parts of the address unchanged.
EADDR SADDR	Transfer both parts of the address, truncating the 32 high-order bits of the offset.
STR	Transfer the ASCII bit pattern for the last eight characters in the string. Strings shorter than eight characters are treated as if they were extended on the left with nulls.

Examples

```
%cmdebug > wl cstx(12.304)
CSTX %12.304
```

Coerce the simple long pointer into a CSTX absolute code pointer.

```
%cmdebug > wl cstx( sort )
CSTX %4.3302
```

Print the address of the procedure named sort. Note that the search path used for procedure name lookups is restricted to the program file (PROG).

```
%cmdebug > wl cstx(sys(24.630))
CSTX %24.630
```

The coercion simply changes the associated absolute file. Note that no complicated conversion or range checking is performed.

Limitations, Restrictions

none

func dstva

Converts a CM data segment address to a virtual address.

Syntax

```
dsvta (dstoff)
```

Formal Declaration

```
dstva:lptr (dstoff:lptr)
```

Parameters

dstoff The CM data segment address which is to be converted to a virtual address. This is specified as *dst.offset*.

Examples

```
$nmdebug > = dstva(%20.0)
$38.00000000
```

Convert the data segment address `%20.0` to a virtual address and display the result.

Limitations, Restrictions

none

func eaddr

Coerces an expression into an extended address.

Syntax

```
eaddr (value)
```

Formal Declaration

`eaddr:eaddr (`*value*`:any)`

Parameters

value	An expression to be coerced. All types are valid.

Derivation of the EADDR Bit Pattern

Parameter Type	Action
BOOL	0.1 if TRUE, 0.0 if FALSE.
U16 U32 SPTR	Set the SID part to zero. Right justify the original value in the low-order 64 bits of the offset part with zero fill.
S16 S32 S64	Set the SID part to zero. Right justify the original value in the low-order 64 bits of the offset part with sign extension.
LONG Class	Transfer the SID part unchanged. Right justify the original offset part in the low-order 64 bits of the offset part with zero fill.
EADDR SADDR	Transfer both parts of the address unchanged.
STR	Transfer the ASCII bit pattern for the last twelve characters in the string. Strings shorter than twelve characters are treated as if they were extended on the left with nulls.

Examples

```
$nmdat > wl eaddr( 1 )
$0.1

$nmdat > wl eaddr( ffff )
$0.ffff

$nmdat > wl eaddr( 1234abcd )
$0.1234abcd

$nmdat > wl eaddr( -1 )
$0.ffffffffffffff

$nmdat > wl eaddr( 1234.5678 )
$1234.5678
```

func eaddr

```
$nmdat > wl eaddr( true )
$0.1

$nmdat > wl eaddr( prog(1.2) )
$1.2
```

Limitations, Restrictions

none

func errmsg

Returns an error message string, based on error number and an optional subsystem number.

Syntax

```
errmsg (errnum [subsys])
```

Formal Declaration

```
errmsg:str (errnum:s16 [subsys:u16=$a9])
```

Parameters

errnum The error number, typically negative for errors, positive for warnings.

subsys The subsystem number. By default, the Debug subsystem number ($a9) is used.

Examples

```
$nmdebug > wl errmsg (-#1055)
Expected a string for a pattern name  (error #1105)
```

Display the System Debug error message string for error number 1105.

```
$nmdebug > wl errmsg (-#52, #10)
NONEXISTENT PERMANENT FILE  (FSERR #52)
```

Display the error message string for error number -#52, for subsys #10.

```
$nmdat > wl errmsg(-#37,#36)
External error   -  subsys: #36  info: #37
```

func errmsg

If the error message is not found in the system message catalog, this form of message is returned.

Limitations, Restrictions

none

func grp

Coerces an expression into a `GRP` logical code pointer (`LCPTR`).

Syntax

```
grp (value)
```

During the evaluation of the parameter to this function, the search path used for procedure name lookups is limited to the group library file (`GRP`).

Formal Declaration

```
grp:grp (value:any)
```

Parameters

value An expression to be coerced. All types are valid.

func grp

Derivation of the GRP Bit Pattern

Parameter Type	Action
BOOL	0.1 if TRUE, 0.0 if FALSE.
U16 U32 SPTR	Set the SID part to zero. Right justify the original value in the low-order 32 bits of the offset part with zero fill.
S16 S32 S64	Set the SID part to zero. Right justify the original value in the low-order 32 bits of the offset part with sign extension.
LONG Class	Transfer both parts of the address unchanged.
EADDR SADDR	Transfer the SID part unchanged. Transfer the low-order 32 bits of the offset part.
STR	Transfer the ASCII bit pattern for the last eight characters in the string. Strings shorter than eight characters are treated as if they were extended on the left with nulls.

Examples

```
%cmdebug > wl grp( 12.304 )
GRP %12.304
```

Coerce the simple long pointer into a GRP logical code pointer.

```
%cmdebug > wl grp( sort )
GRP %2.1364
```

Print the address of the procedure named sort. Note that the search path used for procedure name lookups is restricted to the group library (GRP).

```
%cmdebug > wl grp( sys(24.630) )
GRP %24.630
```

The coercion simply changes the associated logical file. Note that no complicated conversion or range checking is performed.

```
$nmdat > wl grp( 1 )
GRP $0.1
```

```
$nmdat > wl grp( ffff )
GRP $0.ffff

$nmdat > wl grp( 1234abcd )
GRP $0.1234abcd

$nmdat > wl grp( -1 )
GRP $0.ffffffff

$nmdat > wl grp( 1234.5678 )
GRP $1234.5678

$nmdat > wl grp( true )
GRP $0.1

$nmdat > wl grp( "ABCDEFG" )
GRP $414243.44454647
```

Limitations, Restrictions

none

func hash

Hashes a virtual address into a hash table (real) offset.

Syntax

```
hash (virtaddr)
```

The hash value can be added to the Hash table base real address (TR1) to determine the real offset to the first PDIR entry.

Formal Declaration

hash:s32 (*virtaddr*:ptr)

Parameters

virtaddr The virtual address that is to be hashed.

Virtaddr can be a short pointer, a long pointer, or a full logical code pointer.

Examples

```
nmdat > wl pc
SYS $a.d87f8

nmdat > wl hash(pc)
$103c4

nmdat > dz tr1+hash(pc)
REAL $103c4    $ 00001b00

nmdat > dz tr0+1b00,4
```

```
REAL $0061dd00 $ 80000000 0000000a 000d8000 82800000
```

Hash the virtual address for PC ($a.d87f8) to get real address $103c4. Add the hash value ($103c4) to the base of the Hash table (TR1) to get the offset of the first PDIR entry ($1b00). Add this offset to the base of the PDIR table (TR0), and display the four-word PDIR entry.

Limitations, Restrictions

none

func lgrp

Coerces an expression into a `LGRP` logical code pointer (`LCPTR`).

Syntax

```
lgrp (value)
```

During the evaluation of the parameter to this function, the search path used for procedure name lookups is limited to the logon group library file (`LGRP`).

Formal Declaration

```
lgrp:lgrp (value:any)
```

Parameters

value	An expression to be coerced. All types are valid.

Derivation of the LGRP Bit Pattern

Parameter Type	Action
BOOL	0.1 if TRUE, 0.0 if FALSE.
U16 U32 SPTR	Set the SID part to zero. Right justify the original value in the low-order 32 bits of the offset part with zero fill.
S16 S32 S64	Set the SID part to zero. Right justify the original value in the low-order 32 bits of the offset part with sign extension.
LONG Class	Transfer both parts of the address unchanged.
EADDR SADDR	Transfer the SID part unchanged. Transfer the low-order 32 bits of the offset part.
STR	Transfer the ASCII bit pattern for the last eight characters in the string. Strings shorter than eight characters are treated as if they were extended on the left with nulls.

Examples

```
%cmdebug > wl lgrp(12.304)
LGRP %12.304
```

Coerce the simple long pointer into a LGRP logical code pointer.

```
%cmdebug > wl lgrp( sort )
LGRP %0.6412
```

Print the address of the procedure named sort. Note that the search path used for procedure name lookups is restricted to the logon group library (LGRP).

```
%cmdebug > wl lgrp(sys(24.630))
LGRP %24.630
```

The coercion simply changes the associated logical file. The pointer's bit pattern remains unchanged.

```
$nmdat > wl lgrp( 1 )
```

func lgrp

```
LGRP $0.1

$nmdat > wl lgrp( ffff )
LGRP $0.ffff

$nmdat > wl lgrp( 1234abcd )
LGRP $0.1234abcd

$nmdat > wl lgrp( -1 )
LGRP $0.ffffffff

$nmdat > wl lgrp( 1234.5678 )
LGRP $1234.5678

$nmdat > wl lgrp( true )
LGRP $0.1

$nmdat > wl lgrp( "ABCDEFG" )
LGRP $414243.44454647

$nmdat > wl lgrp( prog(1.2) )
LGRP $1.2
```

Limitations, Restrictions

none

func logtoabs

Logical to absolute. Converts a CM logical code address (`LCPTR`) into a CM absolute code address (`ACPTR`).

Syntax

`logtoabs (`*cmlogaddr*`)`

Formal Declaration

`logtoabs:acptr (`*cmlogaddr*`:lcptr)`

Parameters

cmlogaddr The CM logical code address to be converted into an absolute code pointer.

Cmlogaddr must be a full CM logical code address (`LCPTR`). For example:

`CMPC`	Current CM program counter
`CMPW+4`	Top of CM program window + 4
`PROG(2.102)`	Program file logical seg 2 offset 102
`fopen+102`	CM procedure `fopen` + %102 (assumes CM mode)
`cmaddr('fopen')+%102`	CM procedure `fopen` + %102 (NM or CM mode)

Examples

```
%cmdebug > wl logtoabs(prog(0.1273))
CSTX %1.1273
```

Logical CM address `PROG 0.1273` is converted into absolute address `CSTX 1.1273`.

```
%cmdebug > wl logtoabs(sys(32.304))
CST %43.304
```

Logical CM address `SYS 32.304` is converted into absolute address `CST 43.304`.

```
%cmdebug > wl logtoabs(grp(4.4274))
CST %103.4274
```

Logical group library address `GRP` 4.4274 is converted into absolute address `CST 103.4274`.

Limitations, Restrictions

none

func lptr

Coerces an expression into a long pointer.

Syntax

```
lptr (value)
```

Formal Declaration

```
lptr:lptr (value:any)
```

Parameters

value An expression to be coerced. All types are valid.

Derivation of the LPTR Bit Pattern

Parameter Type	Action
BOOL	0.1 if TRUE, 0.0 if FALSE.
U16 U32 SPTR	Set the SID part to zero. Right justify the original value in the low-order 32 bits of the offset part with zero fill.
S16 S32 S64	Set the SID part to zero. Right justify the original value in the low-order 32 bits of the offset part with sign extension.
LONG Class	Transfer both parts of the address unchanged.
EADDR SADDR	Transfer the SID part unchanged. Transfer the low-order 32 bits of the offset part.
STR	Transfer the ASCII bit pattern for the last eight characters in the string. Strings shorter than eight characters are treated as if they were extended on the left with nulls.

Examples

```
$nmdat > wl lptr( 1 )
$0.1

$nmdat > wl lptr( ffff )
$0.ffff

$nmdat > wl lptr( 1234abcd )
$0.1234abcd

$nmdat > wl lptr( -1 )
$0.ffffffff

$nmdat > wl lptr( 1234.5678 )
$1234.5678

$nmdat > wl lptr( true )
```

```
$0.1

$nmdat > wl lptr( "ABCDEFG" )
$414243.44454647

$nmdat > wl lptr( prog(1.2) )
$1.2
```

Limitations, Restrictions

none

func lpub

Coerces an expression into a LPUB logical code pointer (LCPTR).

Syntax

```
lpub (value)
```

During the evaluation of the parameter to this function, the search path used for procedure name lookups is restricted to the logon account library file (LPUB).

Formal Declaration

`lpub:lpub (`*value*`:any)`

Parameters

value	An expression to be coerced. All types are valid.

Derivation of the LPUB Bit Pattern

Parameter Type	Action
BOOL	0.1 if TRUE, 0.0 if FALSE.
U16 U32 SPTR	Set the SID part to zero. Right justify the original value in the low-order 32 bits of the offset part with zero fill.
S16 S32 S64	Set the SID part to zero. Right justify the original value in the low-order 32 bits of the offset part with sign extension.
LONG Class	Transfer both parts of the address unchanged.
EADDR SADDR	Transfer the SID part unchanged. Transfer the low-order 32 bits of the offset part.
STR	Transfer the ASCII bit pattern for the last eight characters in the string. Strings shorter than eight characters are treated as if they were extended on the left with nulls.

Examples

```
%cmdebug > wl lpub(12.304)
LPUB %12.304
```

Coerce the simple long pointer `12.304` into a `LPUB` logical code pointer.

```
%cmdebug > wl lpub( sort )
LPUB %2.6632
```

Print the address of the procedure named `sort`. Note that the search path used for procedure name lookups is restricted to the logon account library (`LPUB`).

```
%cmdebug > wl lpub(sys(24.630))
LPUB %24.630
```

The coercion simply changes the associated logical file. The pointer's bit pattern remains unchanged.

```
$nmdat > wl lpub( 1 )
```

func lpub

```
LPUB $0.1

$nmdat > wl lpub( ffff )
LPUB $0.ffff

$nmdat > wl lpub( 1234abcd )
LPUB $0.1234abcd

$nmdat > wl lpub( -1 )
LPUB $0.ffffffff

$nmdat > wl lpub( 1234.5678 )
LPUB $1234.5678

$nmdat > wl lpub( true )
LPUB $0.1

$nmdat > wl lpub( "ABCDEFG" )
LPUB $414243.44454647

$nmdat > wl lpub( prog(1.2) )
LPUB $1.2
```

Limitations, Restrictions

none

func ltolog

Long to logical. Converts a long pointer into a NM logical code pointer (LCPTR).

Syntax

```
ltolog (longptr)
```

The SID of the long pointer (input parameter) is compared with the SID of each of the loaded NM executable libraries for a match. If a SID match is found, then the appropriate logical code pointer is returned.

If the SID does not match any of the loaded NM files, then the long pointer is tested to see if it points to a NM section of translated CM code produced by the Object Code Translator (OCT). If the long pointer is found to be translated code, then a special TRANS logical code pointer is returned.

Refer to appendix C for a discussion of CM object code translation, node points, and breakpoints in translated CM code.

If both of the previous tests fail, then a special unknown type (UNKN) is returned.

Formal Declaration

```
ltolog:lcptr (longptr:lptr)
```

Parameters

longptr The long pointer to be converted into a NM logical code pointer.

Examples

```
$nmdebug > wl ltolog (a.2034c)
SYS $a.2034
```

The SID $a matches the SID for the system library (SYS) NL.PUB.SYS. The long pointer is converted into the logical code pointer SYS a.2034.

```
$nmdebug > wl ltolog (3c.3208)
PROG $3c.3208
```

The SID $3c matches the SID of the program file.

```
$nmdebug > wl ltolog (20.10264)
TRANS $20.10264
```

The SID $20 does not match any of the loaded NM files. A final test is applied, in case the virtual address is in translated CM code. In this example, the address does point to a NM section of translated CM object code (translated by the Object Code Translator).

```
$nmdebug > wl ltolog (123.45678)
UNKN $123.45678
```

The SID $123 does not match any of the loaded NM files and does not point to translated code. The special unknown logical code pointer is returned.

Limitations, Restrictions

none

func ltos

Long to short. Converts a virtual address to a short pointer.

Syntax

```
ltos (virtaddr)
```

The `LTOS` function converts a virtual address to a short pointer.

If the parameter *virtaddr* is already a short pointer, it is simply returned.

If the parameter *virtaddr* is a long pointer, or a full logical code address, a special additional test is performed to ensure that the offset portion can be returned as the short pointer value. The SID (space) portion must match the current value of the associated space register. This ensures that the returned short pointer value can be successfully converted back into the long pointer argument.

Formal Declaration

```
ltos:sptr (virtaddr:ptr)
```

Parameters

virtaddr	The virtual address to be converted to a short pointer. *Virtaddr* can be a short pointer, a long pointer, or a full logical code pointer.

Examples

```
$nmdebug > wl pc
PROG $3c.12004
$nmdebug > wl ltos(pc)
$12004

$nmdebug > var save 42.40151025
$nmdebug > wl ltos(save)
$40151025


$nmdat > dr sr4
SR4=$a
$nmdat > wl ltos(22.200)
SID in LPTR for LTOS conversion does not match corresponding space reg.
Error evaluating a predefined function.  (error #4240)
  function is"ltos"
```

In this example SR4 contains `$a`. The function `LTOS` detects that the SID portion of the long pointer (`$22`) does not match the value of the associated space register (SR4=`$a`), and the conversion fails.

Limitations, Restrictions

none

func macbody

Returns a string that is the macro body for the specified macro name.

Syntax

```
macbody (macroname)
```

Formal Declaration

```
macbody:str (macroname:str)
```

Parameters

macroname The name of the macro whose body is to be returned.

Examples

```
$nmdebug > wl macbody("showtime")
wl time
```

Display the macro body for the macro command named `showtime`.

```
$nmdebug > wl macbody("min")
if p1 <= p2 then return p1 else return p2
```

Display the macro body for the macro function named `min`.

Limitations, Restrictions

none

func mapindex

Returns the map index number of the specified file name which has been previously mapped into virtual space with the `MAP` command.

Syntax

```
mapindex (filename)
```

Formal Declaration

pindex:u16 (*filename*:str)

Parameters

filename The name of the previously mapped file whose index number is to be returned.

Examples

```
$nmdebug > maplist
1  DTCDUMP.DUMPUSER.SUPPORT     1000.0  Bytes = 43dc
2  DTCDUMP2.DUMPUSER.SUPPORT    1001.0  Bytes = c84
3  MYFILE.MYGROUP.MYACCT        1005.0  Bytes = 1004

$nmdebug > wl mapindex("DTCDUMP")
$1
```

Limitations, Restrictions

none

func mapsize

Returns the size in bytes of the specified mapped file.

Syntax

```
mapsize (filename)
```

Formal Declaration

mapsize:u32 (*filename*:str)

Parameters

filename The name of the previously mapped file whose size is to be returned.

Examples

```
$nmdebug > maplist
1  DTCDUMP.DUMPUSER.SUPPORT      1000.0  Bytes = 43dc
2  DTCDUMP2.DUMPUSER.SUPPORT     1001.0  Bytes = c84
3  MYFILE.MYGROUP.MYACCT         1005.0  Bytes = 1004

$nmdebug > = mapsize("DTCDUMP2.DUMPUSER")
c84
```

Limitations, Restrictions

none

func mapva

Returns the virtual address of the specified mapped file.

Syntax

```
mapva (filename)
```

Formal Declaration

mapva:lptr (*filename*:str)

Parameters

filename The name of the mapped file whose virtual address is to be returned.

Examples

```
$nmdebug > maplist
1  DTCDUMP.DUMPUSER.SUPPORT      1000.0  Bytes = 43dc
2  DTCDUMP2.DUMPUSER.SUPPORT     1001.0  Bytes = c84
3  MYFILE.MYGROUP.MYACCT         1005.0  Bytes = 1004

$nmdebug > = mapva("DTCDUMP")
1000.0
```

Limitations, Restrictions

none

func nmaddr

Returns the virtual address of the specified NM procedure/data path.

Syntax

```
nmaddr (path [lookupid])
```

The values returned by this function are the values as found in the symbol table that is searched. This function does not perform any form of symbol location fixups. The address returned for most data symbols must be relocated relative to DP to be useful.

Formal Declaration

`nmaddr:long (`*path*`:str [`*lookupid*`:str="PROCEDURE"])`

Parameters

path The path specification for the NM procedure or data specified in the form:

*file_name***/***module_name*:*procedure***/***dataname*

or, for nested procedures:

*file_name***/***module_name*:*parent_procedure*.*procedure*

lookupid A keyword indicating where to look for the code path specification given above. Refer to the "Procedure Name Symbols" section in chapter 2 for additional details. Valid keywords and their meanings are as follows:

Keyword	Meaning
UNIVERSAL	Search exported procedures in the SOM symbols.

`LOCAL`	Search nonexported procedures in the SOM symbols.
`NESTED`	Search nested procedures in the SOM symbols.
`PROCEDURES`	Search local or exported procedures in the SOM symbols.
`ALLPROC`	Search local/exported/nested procedures in the SOM symbols.
`EXPORTSTUB`	Search export stubs in the SOM symbols.
`DATAANY`	Search exported and local data SOM symbols.
`DATAUNIV`	Search exported data SOM symbols.
`DATALOCAL`	Search local data SOM symbols.
`LSTPROC`	Search exported level 1 procedures in the LST.
`LSTEXPORTSTUB`	Search export stubs in the LST.
`ANY`	Search for any type of symbol in the SOM symbols.

If a keyword is not given, the default PROCEDURES is used. In all cases, if the path contains a procedure name that appears as a nested procedure (for example: *name.name*), the function assumes the caller meant to use the NESTED keyword.

The keyword may be abbreviated. The table of keywords (above) is searched from top to bottom. Thus DATA is resolved as DATAANY.

Note Searching the SOM symbols is noticeably slower than searching the LST symbols.

Examples

```
$nmdebug > wl processstudent
PROG $4d5.5d24

$nmdebug > wl nmaddr("processstudent")
PROG $4d5.5d24
```

Write the address for the processstudent procedure. The expression evaluator can locate the procedure since it is an exported universal procedure. The procedure may also be located by using the NMADDR function. The default *lookupid* PROCEDURES is used.

```
$nmdebug > wl processstudent.highscore
Expected a number, variable, function, or procedure (error #3720)
  undefined operand is: "processstudent"
  wl processstudent.highscore
```

The above example attempts to locate the nested procedure highscore. The expression evaluator fails. This is due to the fact that a dot "." is used to separate parts of a long pointer by the expression evaluator. The correct method of locating a nested procedure is demonstrated in the following example.

```
$nmdebug > wl nmaddr("processstudent.highscore")
PROG $4d5.5b50
```

func nmaddr

The NMADDR function parses the dot in the nested procedure name and finds it's location.

```
$nmdebug > wl nmaddr("highscore")
Couldn't translate path to an address.  (error #1612)
Error evaluating a predefined function.  (error #4240)
  function is"nmaddr"
  wl nmaddr("highscore")

$nmdebug > wl nmaddr("highscore" "nested")
PROG $4d5.5b50
```

In the above example an error occurs because the default *lookupid* of `PROCEDURES` is used. Since `highscore` is a nested procedure, `NMADDR` fails to locate it. When the `NESTED` *lookupid* parameter is specified, the search succeeds.

```
$nmdebug > wl nmaddr("input" "data")
PROG $4d5.400003a8
```

The `NMADDR` function is also able to look up data symbols. The above example locates the address for the symbol `input`. The value returned is the value found in the SOM symbol table. This function does not perform data symbol location fixups. Only those data symbols placed into the SOM symbol table by the language compilers are locatable. Most language compilers *do not* place the program's variables into this data structure.

```
$nmdebug > wl average
GRP $4d8.15c88

$nmdebug > wl nmaddr("average")
GRP $4d8.15c88
```

The above example locates the address for the `average` procedure. Note that this procedure resides in the group library.

```
$nmdebug > wl nmaddr('p heap:P NEW HEAP')
USER $10d.12f3dc
```

The above example prints out the address of one of the Pascal library routines. Notice the module qualifier.

func nmaddr

```
$nmdebug > wl FOPEN
SYS $a.3f8140

$nmdebug > wl nmaddr("FOPEN")
SYS $a.3f8140

$nmdebug > wl nmaddr("nl.pub.sys/FOPEN")
SYS $a.3f8140

$nmdebug > wl nmaddr("FOPEN" "LST")
SYS $a.3f8140

$nmdebug > wl ?FOPEN
SYS $a.3f80e4

$nmdebug > wl nmaddr("FOPEN" "EXPORTSTUB")
SYS $a.3f80e4
```

The last set of examples show various methods of locating the entry point and export stub for the FOPEN intrinsic. Notice that the question mark is not used in the NMADDR function when referring to stubs.

Limitations, Restrictions

Only addresses corresponding to the process's loaded file set (program file and libraries) succeed.

System Debug displays stubs by preceding the symbol name with a question mark. For example, the export stub for FOPEN would appear as ?FOPEN. This form is not honored by this function (see the last example above).

The addresses for data symbols are not relocated.

func nmbpaddr

Returns the address corresponding to the indicated NM breakpoint index.

Syntax

```
%nmbpaddr (bpindex [pin])
```

This function accepts an index for an existing NM breakpoint and returns the address where the breakpoint is located. The default action is to look for breakpoints set by the current PIN. Breakpoint addresses for other PINs (including the global PIN) may be retrieved by using the optional *pin* parameter.

Formal Declaration

```
nmbpaddr:lptr (bpindex:u32 [pin:s16=0])
```

Parameters

bpindex The index of the breakpoint whose address is to be returned.

pin Look for breakpoints set by this PIN. Default is the caller's PIN (a *pin* of 0 implies this). To specify system (global) breakpoints, use a -1 (or 32762) as the PIN.

Examples

```
$nmdebug > bl
NM      [1] PROG $ c3.56d80   test_screen+$ab3
NM      [2] PROG $ c3.4cf18   test_file^
NM     @[1] SYS  $ a.004b9130 FOPEN
```

First, list the existing breakpoints.

func nmbpaddr

```
$nmdebug > wl nmbpaddr(1)
PROG $c3.56d80

$nmdebug > l nmbpaddr(1, -1)
SYS $a.4b9130
```

Now use the function to return the address associated with process local breakpoint number one and then with system breakpoint number one.

Limitations, Restrictions

none

func nmbpindex

Returns the NM breakpoint index for the NM breakpoint that has been set at the specified NM code address.

Syntax

```
nmbpindex (virtaddr [pin])
```

This function accepts the address of an existing NM breakpoint and returns the logical index number associated with that breakpoint. The default action is to look for breakpoints set by the current PIN. Breakpoint indices for other PINs (including the global PIN) may be retrieved by using the optional *pin* parameter.

Formal Declaration

```
nmbpindex:u32 (virtaddr:ptr [pin:s16=0])
```

Parameters

virtaddr The address of an NM breakpoint whose index is to be returned.

Virtaddr can be a short or long pointer.

pin Look for breakpoints set by this PIN. Default is the caller's PIN (a *pin* of 0 implies this). To specify system (global) breakpoints, use a -1 (or 32762) as the PIN.

Examples

```
$nmdebug > bl
NM      [1] PROG $ c3.56d80   test_screen+$ab3
NM      [2] PROG $ c3.4cf18   test_files
NM     @[1] SYS  $ a.004b9130 FOPEN
```

First, list the existing breakpoints.

```
$nmdebug > wl nmbpindex(test_files)
$2
```

Find the NM breakpoint index associated with the address `test_files`.

```
$nmdebug > wl nmbpindex(FOPEN)
No breakpoint exists in the breakpoint tables with that address.
(error #1080)
Error evaluating a predefined function.  (error #4240)
  function is"nmbpindex"
    wl nmbpindex(FOPEN)
```

Now, go find the breakpoint index for the breakpoint at `FOPEN` . In this example we get an error. This is because we did not specify *pin* and thus searched only for process local breakpoints. We do not have a process local breakpoint at `FOPEN` .

```
$nmdebug > wl nmbpindex(FOPEN, -1)
$1
```

Find the breakpoint index for the breakpoint at `FOPEN` . This time we specify a -1 to tell the function to search the list of system breakpoints.

Limitations, Restrictions

none

func nmbpinstr

Returns the original NM instruction at a specified NM code address where a NM breakpoint has been set.

Syntax

```
nmbpinstr (virtaddr [pin])
```

This function accepts the address of an existing NM breakpoint and returns the instruction associated with that breakpoint. The default action is to look for breakpoints set by the current PIN. Breakpoint indices for other PINs (including the global PIN) may be retrieved by using the optional *pin* parameter.

Formal Declaration

```
nmbpinstr:s32 (virtaddr:ptr [pin:s16=0])
```

Parameters

virtaddr The address of an NM breakpoint at which the stored instruction is to be returned.

Virtaddr can be a short pointer, a long pointer, or a full logical code pointer.

pin Look for breakpoints set by this PIN. Default is the caller's PIN (a *pin* of 0 implies this). To specify system (global) breakpoints, use a -1 (or 32762) as the PIN.

Examples

```
$nmdebug > dc FOPEN,1
SYS $a.4b9130
004b9130  FOPEN  6bc23fd9  STW      2,-20(0,30)
```

Display code at the address of `FOPEN` so we can see what the current instruction is at that address.

```
$nmdebug > b FOPEN
added: NM    [1] SYS  $a.004b9130  FOPEN

$nmdebug > dc FOPEN,1
SYS $a.4b9130
004b9130  FOPEN  0000400e  BREAK    (nmdebug bp)
```

Now set a breakpoint at `FOPEN` and display the code there. The old instruction has been replaced with a breakpoint instruction.

func nmbpinstr

```
$nmdebug > wl nmbpinstr(FOPEN)
$6bc23fd09
```

Use the function to look up the actual instruction. The instruction that is stored in the system breakpoint table is returned by the function.

Limitations, Restrictions

none

func nmcall

Dynamically calls a procedure/function, passing up to four parameters.

Syntax

```
nmcall (path) [parm1] [parm2] [parm3] [parm4]
```

This function is used to perform a dynamic procedure call. It is implemented by calling the `HPGETPROCPLABEL` intrinsic to ensure the desired routine is loaded, and then uses the `FCALL` routine in the Pascal/XL compiler to invoke the routine. The called code is invoked at the same privilege level as the routine that invoked Debug (for example, the privilege level contained in the `PRIV` environment variable). DAT invokes the routine from privilege level 2. This function is not available from SAT. Four parameters are *always* passed to the indicated routine. These values are placed in the argument registers (arg0..arg3). It is up to the called code to correctly define its parameter list and interpret the parameters appropriately.

If you are not familiar with the procedure calling conventions as used by the language compilers, please refer to the *Procedure Calling Conventions Reference Manual* (09740-64003).

The value returned by the called routine (if any) in the function return register (R28), is used as the result of the `NMCALL` function. Because this register contains only a 32-bit value, code that returns data larger than 32 bits should not be invoked. If the called routine does not return a value, whatever value that happens to be in R28 is used as the value of this function (for example, the function is undefined).

func nmcall

Formal Declaration

nmcall:s32 (*path*:str [*parm1*:sptr=0][*parm2*:sptr=0] [*parm3*:sptr=0] [*parm4*:sptr=0]

Parameters

path The code path specification for the NM procedure/function to be called. The format of this parameter is:

file_name/*procname*

The *file_name* part specifies the library to be searched for *procname*. The *file_name* part is optional. If it is not provided, the current list of loaded files for the process (see the `LOADINFO` command) will be searched. Refer to the `HPGETPROCPLABEL` intrinsic for additional details, assumptions, and restrictions involving searching libraries.

Note Unlike the other forms of procedure `PATH` specifications (for example, the `NMADDR` function), module names and nested procedures are not supported by this function.

---

parm1,2,3,4 These parameters are used to pass values to the routine being called. They are passed in arg0 (r26), arg1 (r25), arg2 (r24), and arg3 (r23). Each may contain *any* value up to 32 bits in length. The called code must know how to interpret these values. If the called routine has fewer parameters, the zeros passed in the remaining argument registers are harmless. If the called routine has additional parameters, their values are undefined. Be *sure* you understand the procedure calling conventions and the parameter type alignment restrictions imposed by the various language compilers before trying to pass complicated parameters.

Examples

```
$nmdat > wl nmcall("nl.pub.sys/CLOCK")
$d1f3709

$ nmdat > wl nmcall("CLOCK")
$d1f3b00
```

Call the `CLOCK` intrinsic which is in the system library. Since that library is part of every process's loaded file list, the library name is optional.

Limitations, Restrictions

This function is not supported in SAT.

Debug only is affected by the following restrictions. Currently, you must have privileged mode (PM) to call this function. Furthermore, only code that has been running at privilege level 0, 1, or 2 (see the `PRIV` environment variable) is able to use this function. This is due to security problems that would occur due to the internal implementation of the function.

Caution Because the called code runs on the stack above the debugger, it is possible for the called code to write into the stack space where the debugger currently exists. It is conceivable that a process abort or even system abort could result when returning from the called code due to modification of the debugger's portion of the stack.

func nmentry

Returns the entry point of the NM procedure containing the indicated address.

Syntax

```
nmentry (virtaddr)
```

Formal Declaration

```
nmentry:lptr (virtaddr:ptr)
```

Parameters

virtaddr	The virtual address for which the entry point of the surrounding (level one) NM procedure is to be returned. *Virtaddr* can be a short pointer, a long pointer, or a full logical code pointer.

Examples

```
$nmdebug > wl average
GRP $4d8.15c88

$nmdebug > wl nmentry( average+20 )
GRP $4d8.15c88
```

Print the address for the procedure `average`. Given any offset within the procedure, the `NMENTRY` function returns the address of the procedure's entry point.

```
$nmdebug > wl nmaddr("processstudent.highscore")
PROG $4d5.5b50

$nmdebug > wl nmentry ( nmaddr( "highscore" "nested") + 40 )
```

```
PROG $4d5.5b50
```

Print the address for the nested procedure `highscore`. Given any offset within the nested procedure, the `NMENTRY` function will return the address of the nested procedure's entry point.

Limitations, Restrictions

none

func nmfile

Returns the file name corresponding to the indicated NM (code) address.

Syntax

```
nmfile (virtaddr [length])
```

Formal Declaration

```
nmfile:str (virtaddr:ptr [length:u16=$20])
```

Parameters

virtaddr The virtual address (of NM code) for which the file name is to be returned.

Virtaddr can be a short pointer, a long pointer, or a full logical code pointer.

length The maximum length of the file name string to be returned. If the name does not fully fit into the space specified, it is truncated and followed by an asterisk (*) to indicate the truncation.

Examples

```
$nmdebug >  loadinfo
nm  PROG   GRADES.DEMO.TELESUP          SID=$4d5
    parm=0  info=""
nm  GRP    XL.DEMO.TELESUP              SID=$4d8
nm  USER   XL.PUB.SYS                   SID=$10d
nm  SYS    NL.PUB.SYS                   SID=$a
cm  SYS    SL.PUB.SYS
```

Show the files loaded by the current process.

```
$nmdebug > wl nmfile( average )
XL.DEMO.TELESUP

$nmdebug > wl nmfile ( FOPEN )
NL.PUB.SYS

$nmdebug > wl nmfile ( P_NEW_HEAP )
XL.PUB.SYS

$nmdebug > wl nmfile( processstudent )
GRADES.DEMO.TELESUP

$nmdebug > wl nmfile( processstudent 7 )
GRADES*
```

The above examples show how the `NMFILE` function, given various addresses (all specified as symbolic procedure names), returns the name of the loaded file that contains each address.

Limitations, Restrictions

Only addresses corresponding to the process's loaded file set (program file and libraries) succeed.

func nmmod

Returns the NM module name corresponding to the indicated address.

Syntax

```
nmmod (virtaddr [length])
```

Formal Declaration

```
nmmod:str (virtaddr:ptr [length:u16=$20])
```

Parameters

virtaddr — The virtual address for which the symbolic module name is to be returned.

Virtaddr can be a short pointer, a long pointer, or a full logical code pointer.

length — The maximum length of the module name string to be returned. If the name does not fully fit into the space specified, it will be truncated and followed by an asterisk (*) to indicate the truncation.

If the indicated address is not contained in a named module, an empty string is returned.

Examples

```
$nmdebug > wl nmpath( P_NEW_HEAP )
XL.PUB.SYS/p_heap:P_NEW_HEAP

$nmdebug > wl nmmod ( P_NEW_HEAP )
p_heap
```

This example shows a Pascal library routine called P_NEW_HEAP which is contained in the module named p_heap.

Limitations, Restrictions

none

func nmnode

Returns the NM logical code address (TRANS) of the closest NM node point corresponding to the specified NM address.

Syntax

```
nmnode (virtaddr [node])
```

Refer to appendix C for a discussion of CM object code translation, node points, and breakpoints in translated CM code.

Formal Declaration

nmnode:trans (*virtaddr*:ptr [*node*:str="PREV"])

Parameters

virtaddr The NM address of translated code for which the closest NM node point is to be returned.

virtaddr can be a short pointer, a long pointer, or a full logical code pointer.

node The desired node point, either PREV (closest previous node) or NEXT (closest next node). The default is PREV.

Examples

```
$nmdebug > wl nmnode(21.24030)
TRANS $21.24024
```

Print the NM address of the closest previous (by default) NM node point.

```
$nmdebug > wl nmnode(21.24030,"next")
TRANS $21.2404c
```

Print the NM address of the next NM node point.

Limitations, Restrictions

none

func nmpath

Returns the full NM code path name corresponding to the indicated address.

Syntax

```
nmpath (virtaddr [length])
```

The string returned by `NMPATH` is one of the following two formats:

file_name/module_name:*parent_procname.procname*
or
file_name/module_name:*procname*

Detailed descriptions of each of the above return strings follow:

file_name — The name of the file containing the procedure.

module_name — The name of the module containing the procedure.

parent_procname — The name of the level one procedure containing the nested procedure at the specified address.

procname — The name of the procedure.

Formal Declaration

```
nmpath:str (virtaddr:ptr [length:u16=$50])
```

Parameters

virtaddr — The address for which the symbolic procedure path name is to be returned.

Virtaddr can be a short pointer, a long pointer, or a full logical code pointer.

length The maximum length of the path name string to be returned. If the path name does not fully fit into the space specified, it is truncated and terminated with an asterisk (*) to indicate the truncation.

Examples

```
$nmdebug > wl nmpath( processstudent )
GRADES.DEMO.TELESUP/processstudent

$nmdebug > wl nmpath( processstudent+30 )
GRADES.DEMO.TELESUP/processstudent+$30

$nmdebug > wl nmpath( processstudent+30, #30 )
GRADES.DEMO.TELESUP/processst*
```

The above examples show how NMPATH is used to print out the full path for the procedure processstudent. Notice in the last example that a maximum length of 30 characters is specified, so the full path is truncated and terminated with an asterisk.

```
$nmdebug > wl nmpath ( average )
XL.DEMO.TELESUP/average
$nmdebug > wl nmpath( P_NEW_HEAP )
XL.PUB.SYS/p_heap:P_NEW_HEAP

$nmdebug > wl nmpath( FOPEN )
NL.PUB.SYS/FOPEN

$nmdebug > wl nmpath (nmaddr( "highscore" "nested") + 40 ) )
GRADES.DEMO.TELESUP/processstudent.highscore+$40

$nmdebug > wl nmpath ( nmentry ( nmaddr( "highscore" "nested") + 40 ) )
GRADES.DEMO.TELESUP/processstudent.highscore
```

The above examples show how NMPATH is used to print out path names for routines in various libraries and how it may combined with other functions.

Limitations, Restrictions

none

func nmproc

Returns the NM procedure name and offset corresponding to the specified virtual address.

Syntax

```
nmproc (virtaddr [length])
```

The string returned by NMPROC is one of the following two formats:

parent_procname.procedure_name+base offset
or
procedure_name+base offset

Detailed descriptions of each of the above return strings follow:

parent_procname — The name of the level one procedure containing the nested procedure at the specified address.

procedure_name — The name of the procedure. If the name is longer than *length* characters, it is truncated with an asterisk (*).

base — The output base used to represent *offset*.

$	Hexadecimal
%	Octal
#	Decimal

offset — If the offset is nonzero, then it is returned, appended to the procedure name. The offset is formatted based on the current fill, justification, and output base values.

func nmproc

Formal Declaration

`nmproc:str (`*virtaddr*`:ptr [`*length*`:u16=$40])`

Parameters

virtaddr The address for which the symbolic procedure name/offset is to be returned.

Virtaddr can be a short pointer, a long pointer, or a full logical code pointer.

length The maximum length of the procedure name and offset string to be returned. If the name does not fully fit into the space specified, the procedure name is truncated and is followed by an asterisk (*) to indicate the truncation.

Examples

```
$nmdebug > wl FOPEN
SYS $a.3f8140

$nmdebug > wl nmproc( a.3f8140 )
OPEN

$nmdebug > wl FOPEN+40
SYS $a.3f8180

$nmdebug > wl nmproc( a.3f8180 )
FOPEN+$40

$nmdebug > wl nmproc( pc )
PROGRAM+4c
```

Limitations, Restrictions

none

func nmstackbase

Returns the virtual address of the start of the process's NM stack.

Syntax

```
nmstackbase (pin)
```

Formal Declaration

```
nmstackbase:lptr (pin:u16)
```

Parameters

pin The process identification number (PIN) for which the starting virtual address of the NM stack is to be returned.

Examples

```
$nmdebug > wl nmstackbase(8)
$5e4.4020ea00
```

Display the virtual address of the NM stack base for PIN 8.

```
$nmdat > wl "NM stack size = ", nmstacklimit(pin) - nmstackbase(pin)
NM stack size = $60000
```

Calculate and display the NM stack length (in bytes) for the current PIN.

Limitations, Restrictions

If the PIN does not exist, the function result is undefined and an error status is set.

func nmstacklimit

Returns the virtual address of the limit of a process's NM stack.

Syntax

```
nmstacklimit (pin)
```

Formal Declaration

```
nmstacklimit:lptr (pin:u16)
```

Parameters

pin The process identification number (PIN) for which the virtual address of the NM stack limit is to be returned.

Examples

```
$nmdebug > wl nmstacklimit (8)
$5e4.4026ea00
```

Display the virtual address of the NM stack limit for PIN 8.

```
$nmdat > wl "NM stack size = ", nmstacklimit(pin) - nmstackbase(pin)
NM stack size = $60000
```

Calculate and display the NM stack length (in bytes) for the current PIN.

Limitations, Restrictions

If the PIN does not exist, the function result is undefined and an error status is set.

func nmtocmnode

Returns the CM logical code address of the closest CM node point corresponding to the specified NM address.

Syntax

```
nmtocmnode (virtaddr [node])
```

Refer to appendix C for a discussion of CM object code translation, node points, and breakpoints in translated CM code.

Formal Declaration

```
nmtocmnode:lcptr (virtaddr:lptr [node:str="PREV"])
```

Parameters

virtaddr The virtual address of NM translated code for which the closest CM node point is to be returned.

Virtaddr can be a short pointer, a long pointer, or a full logical code pointer.

node The desired node point, either `PREV` (closest previous node) or `NEXT` (closest next node). If unspecified, then `PREV` is assumed.

Examples

```
$nmdebug > wl nmtocmnode(21.24030):"%"
SYS %12.224
```

Print the CM address of the closest NM previous (by default) node point.

```
$nmdebug > wl nmtocmnode(21.24030, "next"):"%"
SYS %12.232
```

func nmtocmnode

Print the CM address of the closest NM next node point.

Limitations, Restrictions

none

func off

Returns the offset portion of a virtual or extended address.

Syntax

```
off (virtaddr)
```

Formal Declaration

```
off:u32 (virtaddr:ptr)
```

Parameters

virtaddr The virtual address whose offset portion is to be returned.

Virtaddr can be a short pointer, a long pointer, or an extended address.

Examples

```
$nmdebug > wl pc
PROG $2e.213403

$nmdebug > wl off(pc)
$213403

$nmdebug > wl off(a.1234)
$1234
```

Limitations, Restrictions

none

func pcb

Returns the virtual address (SPTR) of a process's process control block (PCB).

Syntax

```
pcb (pin)
```

Formal Declaration

```
pcb:sptr (pin:u16)
```

Parameters

pin The process identification number (PIN) for which the address of the PCB is to be returned. Note that this is a CM data structure.

Examples

```
$nmdebug > wl pcb(8)
$80001750
```

Limitations, Restrictions

If the PIN does not exist, the function result is undefined and an error status is set.

func pcbx

Returns the virtual address (SPTR) of a process's process control block extension (PCBX).

Syntax

```
pcbx (pin)
```

Formal Declaration

`pcbx:sptr (`*pin*`:u16)`

Parameters

pin — The process identification number (PIN) for which the address of the PCBX is to be returned. Note that this is a CM data structure.

Examples

```
$nmdebug > wl pcbx(8)
$40010db0
```

Limitations, Restrictions

If the PIN does not exist, the function result is undefined and an error status is set.

func phystolog

Converts a CM physical segment number and mapping bit to a CM logical code address.

Syntax

```
phystolog (physsegnum [mappingbit])
```

This function is typically used to manually examine CM stack markers, and CM external plabels.

The offset part of the returned CM logical code address is always set to zero.

Formal Declaration

```
phystolog:lcptr (physsegnum:u16 [mappingbit:bool=FALSE])
```

Parameters

physsegnum The CM physical segment number to be converted to a CM logical address.

mappingbit A Boolean that implies that the segment is physically mapped (`TRUE = 1`) or logically mapped (`FALSE = 0`). By default, *mappingbit* is `FALSE`.

Examples

```
%cmdebug > wl phystolog( 303 )
PROG %2.0
```

Physical segment number %303 is converted into logical code segment `PROG 2`.

```
%cmdebug > wl phystolog( 122 )
GRP %2.0
```

Physical segment number %122 is converted into logical code segment `GRP %2`.

Limitations, Restrictions

none

func pib

Returns the virtual address (`SPTR`) of a process's process information block (PIB).

Syntax

```
pib (pin)
```

Formal Declaration

```
pib:sptr (pin:u16)
```

Parameters

pin The process identification number (PIN) for which the address of the PIB is to be returned.

Examples

```
$nmdebug > wl pib(8)
$c3583a20
```

Limitations, Restrictions

If the PIN does not exist, the function result is undefined and an error status is set.

func pibx

Returns the virtual address (`SPTR`) of a process's process information block extension (PIBX).

Syntax

```
pibx (pin)
```

Formal Declaration

```
pibx:sptr (pin:u16)
```

Parameters

pin The process identification number (PIN) for which the address of the PIBX is to be returned.

Examples

```
$nmdebug > wl pibx(8)
$c4680000
```

Limitations, Restrictions

If the PIN does not exist, the function result is undefined and an error status is set.

func prog

Coerce an expression into a `PROG` logical code pointer (`LCPTR`).

Syntax

```
prog (value)
```

During the evaluation of the parameter to this function, the search path used for procedure name lookups is restricted to the program file (`PROG`).

Formal Declaration

```
prog:prog (value:any)
```

Parameters

value An expression to be coerced. All types are valid.

Derivation of PROG LGRP Bit Pattern

Parameter Type	Action
BOOL	0.1 if TRUE, 0.0 if FALSE.
U16 U32 SPTR	Set the SID part to zero. Right justify the original value in the low-order 32 bits of the offset part with zero fill.
S16 S32 S64	Set the SID part to zero. Right justify the original value in the low-order 32 bits of the offset part with sign extension.
LONG Class	Transfer both parts of the address unchanged.
EADDR SADDR	Transfer the SID part unchanged. Transfer the low-order 32 bits of the offset part.
STR	Transfer the ASCII bit pattern for the last eight characters in the string. Strings shorter than eight characters are treated as if they were extended on the left with nulls.

Examples

```
%cmdebug > wl prog(12.304)
PROG %12.304
```

Coerce the simple long pointer into a PROG logical code pointer.

```
%cmdebug > wl prog( sort )
PROG %2.346
```

Print the address of the procedure named sort. Note that the search path used for procedure name lookups is restricted to the program file (PROG).

```
%cmdebug > wl prog(pub(24.630))
PROG %24.630
```

The coercion simply changes the associated logical file. The pointer's bit pattern remains unchanged.

```
$nmdat > wl prog( 1 )
PROG $0.1
```

func prog

```
$nmdat > wl prog( ffff )
PROG $0.ffff

$nmdat > wl prog( 1234abcd )
PROG $0.1234abcd

$nmdat > wl prog( -1 )
PROG $0.ffffffff

$nmdat > wl prog( 1234.5678 )
PROG $1234.5678

$nmdat > wl prog( true )
PROG $0.1

$nmdat > wl prog( "ABCDEFG" )
PROG $414243.44454647

$nmdat > wl prog( grp(1.2) )
PROG $1.2
```

Limitations, Restrictions

none

func pstate

Returns the process state for the specified PIN as a string.

Syntax

```
pstate (pin)
```

The following table lists all possible returned process state strings:

```
UNBORN
INITIATE
ALIVE
DYING
DEAD
UNKNOWN
```

Note that the process state string is always returned in capital letters.

Formal Declaration

```
pstate:str (pin:u16)
```

Parameters

pin The process identification number (PIN) of the process whose process state is to be returned.

Examples

```
$nmdebug > wl pstate(8)
INITIATE

$nmdebug > wl pstate(f)
DYING
```

func pstate

```
$nmdebug > if pstate(16) = "ALIVE" then formatprocess(16)
```

Limitations, Restrictions

none

func pub

Coerces an expression into a PUB logical code pointer (LCPTR).

Syntax

```
pub (value)
```

During the evaluation of the parameter to this function, the search path used for procedure name lookups is limited to the account library file (PUB).

Formal Declaration

pub:pub (*value*:any)

Parameters

value	An expression to be coerced. All types are valid.

Derivation of the PUB Bit Pattern

Parameter Type	Action
BOOL	0.1 if TRUE, 0.0 if FALSE.
U16 U32 SPTR	Set the SID part to zero. Right justify the original value in the low-order 32 bits of the offset part with zero fill.
S16 S32 S64	Set the SID part to zero. Right justify the original value in the low-order 32 bits of the offset part with sign extension.
LONG Class	Transfer both parts of the address unchanged.
EADDR SADDR	Transfer the SID part unchanged. Transfer the low-order 32 bits of the offset part.
STR	Transfer the ASCII bit pattern for the last eight characters in the string. Strings shorter than eight characters are treated as if they were extended on the left with nulls.

Examples

```
%cmdebug > wl pub(12.304)
PUB %12.304
```

Coerce the simple long pointer into a PUB logical code pointer.

```
%cmdebug > wl pub( sort )
PUB %3.2632
```

Print the address of the procedure named sort. Note that the search path used for procedure name lookups is restricted to the account library (PUB).

```
%cmdebug > wl pub(sys(24.630))
PUB %24.630
```

The coercion simply changes the associated logical file. The pointer's bit pattern remains unchanged.

```
$nmdat > wl pub( 1 )
PUB $0.1
```

```
$nmdat > wl pub( ffff )
PUB $0.ffff

$nmdat > wl pub( 1234abcd )
PUB $0.1234abcd

$nmdat > wl pub( -1 )
PUB $0.ffffffff

$nmdat > wl pub( 1234.5678 )
PUB $1234.5678

$nmdat > wl pub( true )
PUB $0.1

$nmdat > wl pub( "ABCDEFG" )
PUB $414243.44454647

$nmdat > wl pub( prog(1.2) )
PUB $1.2
```

Limitations, Restrictions

none

func rtov

Real to virtual. Converts a real address to a virtual address.

Syntax

```
rtov (realaddr)
```

Formal Declaration

```
rtov:lptr (realaddr:u32)
```

Parameters

realaddr The real address to be converted to a virtual address.

Examples

```
$nmdebug > wl pc
PROG $741.5934
```

Display the current logical code address (LCPTR) of the NM program counter.

```
$nmdebug > wl vtor(pc)
$1827934
```

Translate the logical code address (LCPTR) into the corresponding real address.

```
$nmdebug > wl rtov(1827934)
$741.5934
```

Convert the real address back into a virtual address (LPTR).

Limitations, Restrictions

none

func s16

Coerces an expression into a signed 16-bit value.

Syntax

```
s16 (value)
```

Formal Declaration

```
s16:s16 (value:any)
```

Parameters

value An expression to be coerced. All types are valid.

Derivation of the S16 Bit Pattern

Parameter Type	Action
BOOL	1 if TRUE, 0 if FALSE.
U16 S16	Transfer the original bit pattern unchanged.
U32 S32 S64 SPTR	Transfer the low-order 16 bits.
LONG Class EADDR SADDR	Transfer the low-order 16 bits of the offset part.
STR	Transfer the ASCII bit pattern for the last two characters in the string. Strings shorter than two characters are treated as if they were extended on the left with nulls.

func s16

Examples

```
$nmdat > wl s16( 1 )
$1

$nmdat > wl s16( ffff )
$ffff

$nmdat > wl s16( ffff ):"#"
#-1

$nmdat > wl s16( 1234abcd )
$abcd

$nmdat > wl s16( -1 )
$ffff

$nmdat > wl s16( 1234.5678 )
$5678

$nmdat > wl s16( true )
$1

$nmdat > wl s16( "ABCDEFG" )
$4647

$nmdat > wl s16( prog(1.2) )
$2
```

Limitations, Restrictions

none

func s32

Coerces an expression into a signed 32-bit value.

Syntax

```
s32 (value)
```

Formal Declaration

```
s32:s32 (value:any)
```

Parameters

value An expression to be coerced. All types are valid.

func s32

Derivation of the S32 Bit Pattern

Parameter Type	Action
BOOL	1 if TRUE, 0 if FALSE.
U16	Right justify the original 16-bit value in 32 bits with zero fill.
S16	Right justify the original 16-bit value in 32 bits with sign extension.
U32 S32 SPTR	Transfer the original bit pattern unchanged.
S64	Transfer the low-order 32 bits.
LONG Class EADDR SADDR	Transfer the low-order 32 bits of the offset part.
STR	Transfer the ASCII bit pattern for the last four characters in the string. Strings shorter than four characters are treated as if they were extended on the left with nulls.

Examples

```
$nmdat > wl s32( 1 )
$1

$nmdat > wl s32( ffff )
$ffff

$nmdat > wl s32( ffff ):"#"
#65535

$nmdat > wl s32( 1234abcd )
$1234abcd

$nmdat > wl s32( -1 )
$ffffffff
```

```
$nmdat > wl s32( ffffffff ):"#"
$#-1

$nmdat > wl s32( 1234.5678 )
$5678

$nmdat > wl s32( true )
$1

$nmdat > wl s32( "ABCDEFG" )
$44454647

$nmdat > wl s32( prog(1.2) )
$2
```

Limitations, Restrictions

none

func s64

Coerces an expression into a signed 64-bit value.

Syntax

```
s64 (value)
```

Formal Declaration

s64:s64 (*value*:any)

Parameters

value An arbitrary expression to be coerced.

Derivation of the S64 Bit Pattern

Parameter Type	Action
BOOL	1 if TRUE, 0 if FALSE.
U16 U32 SPTR	Right justify the original value in 64 bits with zero fill.
S16 S32 S64	Right justify the original value in 64 bits with sign extension.
LONG Class	Transfer the concatenation of the SID and offset parts.
EADDR SADDR	Transfer the offset part unchanged.
STR	Transfer the ASCII bit pattern for the last eight characters in the string. Strings shorter than eight characters are treated as if they were extended on the left with nulls.

Examples

```
$nmdebug > wl s64(1.2):"ZF"
$0000000100000002
```

The long pointer value (1.2) is coerced into a signed 64-bit value and displayed zero-filled ("Z") in a fixed field width ("F") format.

Limitations, Restrictions

none

func saddr

Coerces an expression into a secondary address.

Syntax

```
saddr (value)
```

Formal Declaration

saddr:saddr (*value*:any)

Parameters

value An expression to be coerced. All types are valid.

Derivation of the EADDR Bit Pattern

Parameter Type	Action
`BOOL`	0.1 if `TRUE`, 0.0 if `FALSE`.

`U16`
`U32`
`SPTR`
	Set the SID (LDEV) part to zero. Right justify the original value in the low-order 64 bits of the offset part with zero fill.

`S16`
`S32`
`S64`
	Set the SID (LDEV) part to zero. Right justify the original value in the low-order 64 bits of the offset part with sign extension.

`LONG` Class	Transfer the SID part unchanged. Right justify the original offset part in the low-order 64 bits of the offset part with zero fill.
`EADDR` `SADDR`	Transfer both parts of the address unchanged.
`STR`	Transfer the ASCII bit pattern for the last twelve characters in the string. Strings shorter than twelve characters are treated as if they were extended on the left with nulls.

Examples

```
$nmdat > wl saddr( 1 )
SADDR $0.1

$nmdat > wl saddr( ffff )
SADDR $0.ffff

$nmdat > wl saddr( 1234abcd )
SADDR $0.1234abcd

$nmdat > wl saddr( -1 )
SADDR $0.ffffffffffffffff

$nmdat > wl saddr( 1234.5678 )
SADDR $1234.5678
```

```
$nmdat > wl saddr( true )
SADDR $0.1

$nmdat > wl saddr( prog(1.2) )
SADDR $1.2
```

Limitations, Restrictions

none

func sid

Returns the space ID (SID) portion of a virtual or extended address.

Syntax

```
sid (virtaddr)
```

The SID function returns the space ID portion of a virtual address.

If the parameter *virtaddr* is a short pointer (`SPTR`) it is internally converted to a long pointer by the `STOL` function, and the resulting SID portion is returned.

If the parameter *virtaddr* is a long pointer or an extended address, the SID portion is simply extracted and returned.

Formal Declaration

```
sid:u32 (virtaddr:ptr)
```

Parameters

virtaddr — The virtual address from which the space ID (SID) portion is returned.

Virtaddr can be a short pointer, a long pointer, or an extended address.

Examples

```
$nmdebug > wl pc
PROG $2e.213403

$nmdebug > wl sid(pc)
$2e
```

func sid

```
$nmdebug > wl sid(213403)
$2e

$nmdebug > wl sid(a.1234)
$a
```

Limitations, Restrictions

none

func sptr

Coerces an expression into a short pointer.

Syntax

```
sptr (value)
```

Formal Declaration

```
sptr:sptr (value:any)
```

Parameters

value An expression to be coerced. All types are valid.

Derivation of the SPTR Bit Pattern

Parameter Type	Action
BOOL	1 if TRUE, 0 if FALSE.
U16 S16	Right justify the original 16-bit value in 32 bits with zero fill.
U32 S32 SPTR	Transfer the original bit pattern unchanged.
LONG Class	Transfer the low-order 32 bits of the address (offset part) unchanged. The segment number or SID part of the address is discarded.
EADDR SADDR	Transfer the low-order 32 bits of the address (offset part). All other parts of the address are discarded.
STR	Transfer the ASCII bit pattern for the last four characters in the string. Strings shorter than four characters are treated as if they were extended on the left with nulls.

Examples

```
$nmdat > wl sptr( 1 )
$1

$nmdat > wl sptr( ffff )
$ffff

$nmdat > wl sptr( 1234abcd )
$1234abcd

$nmdat > wl sptr( -1 )
$ffffffff

$nmdat > wl sptr( 1234.5678 )
$5678
```

```
$nmdat > wl sptr( true )
$1

$nmdat > wl sptr( "ABCDEFG" )
$44454647

$nmdat > wl sptr( prog(1.2) )
$2
```

Limitations, Restrictions

none

func stol

Short to long. Converts a virtual address to a long pointer.

Syntax

```
stol (virtaddr)
```

If the parameter *virtaddr* is a short pointer (`SPTR`), then it is converted based on the space registers for the current PIN.

If the parameter *virtaddr* is a already a long pointer (`LPTR`) or a code pointer (`ACPTR` or `LCPTR`), then the long pointer (portion) is simply returned.

Formal Declaration

```
stol:lptr (virtaddr:ptr)
```

Parameters

virtaddr The virtual address to be converted to a long pointer.

Virtaddr can be either a short or long pointer.

Examples

```
$nmdebug > dr sr4; dr sr5
sr4=$41
sr5=$53

$nmdebug > wl sp
$40163088

$nmdebug > wl stol(sp)
$53.40163088
```

```
$nmdebug > wl stol(1cbb8c)
$41.1cbb8c

$nmdebug > wl stol(15f.1cbb8c)
$15f.1cbb8c
```

Limitations, Restrictions

none

func stolog

Short to logical. Converts a NM short pointer (`SPTR`) to a NM logical code address (`LCPTR`).

Syntax

```
stolog (shortptr [logsel] [userfname])
```

Based on a logical file selector, *logsel*, the SID of a loaded NM executable library is used to build a logical code pointer.

This conversion is very different from the `STOL` conversion, which uses the current space registers SR4 - SR7 to determine the SID.

Formal Declaration

```
stolog:lcptr (shortptr:sptr [logsel:str="PROG"] [userfname:str])
```

Parameters

shortptr The short pointer to be converted into a logical code pointer.

logsel A string which selects a particular logical file. The SID portion of the resulting logical pointer are based on the SID of the specified logical file selector. Valid selector strings are:

`'PROG'`	Program file
`'GRP'`	Group library
`'PUB'`	Account library
`'SYS'`	System library
`'USER'`	User library

By default, the selector `'PROG'` will be used.

userfname The file name of a user library file. Since multiple NM user libraries can be in use simultaneously, the *userfname* parameter is required when the logical file selector *logsel* is `'USER'`.

If *userfname* is not fully qualified, the program file's group and account are used to fully qualify the file name.

Examples

```
$nmdebug > wl stolog(104c)
PROG $42.104c
```

By default, the logical selector 'PROG' is used to convert short pointer 104c to the logical code pointer PROG 42.104c.

```
$nmdebug > wl stolog(20b34, 'sys')
SYS $a.20b34
```

The logical selector 'SYS' is used to look up the SID for NL.PUB.SYS, and the resulting logical code pointer is SYS a.20b34.

```
$nmdebug > wl stolog(1c68, 'user')
Missing required user library filename for USER logical selector.
```

When the logical selector 'USER' is specified, the parameter *userfname* is required to specify which user library file, since several may be loaded simultaneously.

```
$nmdebug > wl stolog(1c68, 'user', 'LIB3')
USER $3c.1c68
```

The SID for user library is determined to be $3c. The short pointer is converted into logical code pointer USER 3c.1c68.

Limitations, Restrictions

none

func str

Returns a substring of a source string.

Syntax

```
str (source position length)
```

Formal Declaration

str:str (*source*:str *position*:u16 *length*:u16)

Parameters

source — The string from which to extract the substring.

position — The index of the first character to extract. String indices are 1-based. (That is, indices are 1, 2, 3, ... rather than 0, 1, 2, ...)

length — The number of characters to extract. If this value is larger than the actual number of characters in the string, the string is returned from the starting position to the end without an error indication.

Examples

```
$nmdebug > = str("I am sincere.", 6, 3)
"sin"
```

Starting at position 6, extract the next three characters.

```
$nmdebug > = str("Hello mom! I don't know how long this is", 7, 1000)
"mom! I don't know how long this is"
```

Extract the remainder of the string starting at position 7.

func str

Limitations, Restrictions

none

func strapp

String append. Returns the result of concatenating two strings.

Syntax

```
strapp (source tail)
```

Formal Declaration

```
strapp:str (source:str tail:str)
```

Parameters

source	The string to which *tail* is appended.
tail	The string to append to the tail of *source*.

Examples

```
$nmdebug > var stuff "Cream"
$nmdebug > wl strapp("Ice ", stuff)
Ice Cream
```

Append the string contained in the variable `stuff` to the string “`Ice`”.

```
$nmdebug > = strapp("Hello, ", strapp("How", " Are You?") )
"Hello, How Are You?"
```

Print the result of concatenating the string literals.

Limitations, Restrictions

If the resultant string is larger than the maximum supported string length (see the `STRMAX` function), it is truncated.

func strdel

String delete. Returns a string with a substring deleted from the source string.

Syntax

```
strdel (source position length)
```

Formal Declaration

strdel:str (*source*:str *position*:u16 *length*:u16)

Parameters

source The string from which to delete the substring.

position The index of the starting character to delete. String indices are 1-based. (That is, indices are 1, 2, 3, . . . rather than 0, 1, 2,. . . .)

length The number of characters to delete. If this value is larger than the actual number of characters in the string, the string is deleted from the starting position to the end without an error indication.

Examples

```
$nmdebug > = strdel("This is NOT fun", 9, 4)
"This is fun"
```

Starting at position 9, delete the next four characters.

```
$nmdebug > wl strdel("Fishy, fishy, in the brook.", 13, 1000)
Fishy, fishy
```

Delete characters from position 13 to the end of the string.

Limitations, Restrictions

none

func strdown

String downshift. Returns a string that is the result of downshifting all alphabetic characters in the source string.

Syntax

```
strdown (source)
```

Formal Declaration

strdown:str (*source*:str)

Parameters

source The string for which to downshift all alphabetic characters.

Examples

```
$nmdebug > var list '"CHRIS" "WICKY" "PAT" "HOFMANN" "HELMUT"'
$nmdebug > foreach j list wl strdown (j)
chris
wicky
pat
hofmann
helmut
```

Downshift and print each name in the string variable `list`.

```
$nmdebug > if strdown(strinput("continue? ")) = "n" then abort
```

Prompt the user to continue and, if the response is `N` or `n`, then abort.

Limitations, Restrictions

none

func strextract

String extract. Returns a string (extracted) from the specified virtual address.

Syntax

```
strextract (virtaddr [length])
```

Formal Declaration

```
strextract:str (virtaddr:ptr [length:u16=$4])
```

Parameters

virtaddr The virtual address of the start of the string.

Virtaddr can be a short pointer, a long pointer, or a full logical code pointer.

length The number of characters to retrieve starting at *virtaddr*. If this parameter is not specified, the string returned will be four characters long. If the value given in *length* is greater than the maximum string size, the string returned is truncated to the maximum size.

Examples

```
$nmdebug > dv r28, 4, a
VIRT $12f.4000d638   ASCII  EXCL USIV E VI OLAT
$nmdebug > wl strextract (r28, 9)
EXCLUSIVE
```

Register R28 is used as the virtual address at which a nine-character string is extracted.

```
$nmdebug > var tblname strextract(b0002c40)
```

The variable `tblname` is assigned a four-character string which is extracted from the virtual address defined by the short pointer (`b0002c40`).

Limitations, Restrictions

If length is greater than the maximum supported string length (see the `STRMAX` function), only up to `STRMAX` characters are returned.

func strinput

Prompts on the input device for user input and returns the user input line as a string.

Syntax

```
strinput (prompt)
```

Formal Declaration

```
strinput:str (prompt:str)
```

Parameters

prompt The prompt string to be displayed.

Examples

```
$nmdebug > wl strinput("input a number>")
input a number > 1234
1234
```

Prompt the user for a number and write it back.

```
$nmdebug > var n bin(strinput("input a number>"))
input a number > 1+3
```

Prompt the user for a number, convert the input string to a number, and assign it to the variable named `n`.

Limitations, Restrictions

If `STRINPUT` is issued in a job (for example, through the `HPDEBUG` intrinsic command string), an error is displayed, and Debug returns to the caller.

func strins

String insert. Returns a string after inserting another string into the source string.

Syntax

```
strins (insert source position)
```

Formal Declaration

strins:str (*insert*:str *source*:str *position*:u16)

Parameters

insert	The string to be inserted into *source*.
source	The *source* string into which *insert* is to be inserted.
position	The position where *insert* is to be inserted in *source*. String indices are 1-based. (That is, indices are 1, 2, 3, ... rather than 0, 1, 2, ...) If *position* is greater than the string length of *source*, *insert* is appended to *source*.

Examples

```
$nmdebug > var name "Smith, "
$nmdebug > wl strins(name, "Dear Ms. How are You?", 10)
Dear Ms. Smith, How are You?
```

Insert the string variable `NAME` into a literal string at position 10.

```
$nmdebug > wl strins(" NOW!", "Go Home", 100):"qo"
"Go Home NOW!"
```

func strins

Insert "`NOW!`" into the source at position 100. Since the source is only seven characters long, "`NOW!`" is appended at the end of the source string.

Limitations, Restrictions

If the resultant string is larger than the maximum supported string length (see the `STRMAX` function), it is truncated.

func strlen

String length. Returns the current size of a string.

Syntax

```
strlen (source)
```

Formal Declaration

```
strlen:u32 (source:str)
```

Parameters

source Any string literal or variable.

Examples

```
$nmdebug > wl strlen("")
$0
```

Print the length (number of characters) in the empty string.

```
$nmdebug > var company "Hewlett-Packard Co."
$nmdebug > = strlen(company),d
#19
```

Limitations, Restrictions

none

func strltrim

String left trim. Deletes leading blanks from the source string.

Syntax

```
strltrim (source)
```

Formal Declaration

```
strltrim:str (source:str)
```

Parameters

source The string from which all leading blanks are to be deleted.

Examples

```
$nmdebug > wl strltrim("  A string with extra blanks.  "):"qo"
"A string with extra blanks.  "

%cmdebug > = strltrim(strrtrim("   ABCD    "))
"ABCD"
```

Delete both leading and trailing blanks.

Limitations, Restrictions

none

func strmax

String maximum. Returns the (constant) maximum size of a string.

Syntax

```
strmax (source)
```

Formal Declaration

```
strmax:u32 (source:str)
```

Parameters

source Any string literal or variable. The result of this function is a constant. All strings have the same maximum length.

Examples

```
$nmdebug > wl strmax("date"):"#"
#2048

$cmdat > = strmax(""),d
#2048
```

Limitations, Restrictions

The maximum number of characters in a string currently is 2048.

func strpos

String position. Returns the index of the first occurrence of one string in another.

Syntax

`strpos (`*source searchstring* `[`*position*`])`

If *searchstring* is not found in *source* then zero (0) is returned.

Formal Declaration

`strpos:u32 (`*source*`:str` *searchstring*`:str [`*position*`:u32=1])`

Parameters

source The string in which *searchstring* is to be found.

searchstring The string to be found in *source*. It may be either a single- or double-quoted string literal, or a back-quoted regular expression.

position The character position in *source* where the search is to begin. If this parameter is not specified, the search starts at the first character. If this value is greater than the size of the source string, a zero result is returned.

Examples

```
$nmdebug > var source "Oh where oh where has my little dog gone"
$nmdebug > var searchstring "where"
$nmdebug > var first = strpos(source, searchstring)
$nmdebug > wl first
```

```
$4
```

Look for the string "`where`" in the source string and print the position where it was found.

func strpos

```
$nmdebug > first = first + strlen(searchstring)
$nmdebug > var second = strpos(source, searchstring, first)
$nmdebug > wl second
$d
```

Look for the next occurrence of "`where`" in the source string and print the position where it was found.

```
$nmdebug > second = second + strlen(searchstring)
$nmdebug > var third = strpos(source, searchstring, second)
$nmdebug > wl third
#0
```

Look for another occurrence of "`where`" in the source string. Since the search string is not found, the value of zero (0) is returned.

Limitations, Restrictions

none

func strrpt

String repeat. Returns a string composed of repeated occurrences of a source string.

Syntax

```
strrpt (source count)
```

Formal Declaration

strrpt:str (*source*:str *count*:u32)

Parameters

source The *source* string to repeat.

count The number of times to repeat *source*.

Examples

```
$nmdebug > var digits:str "0123456789"
$nmdebug > wl strrpt(digits, 7)
0123456789012345678901234567890123456789012345678901234567890123456789
```

Print out the string of digits "0 .. 9" repeated seven times.

Limitations, Restrictions

If the resultant string is larger than the maximum supported string length (see the STRMAX function), it is truncated at the maximum length.

func strrtrim

String right trim. Deletes trailing blanks from the source string.

Syntax

```
strrtrim (source)
```

Formal Declaration

```
strrtrim:str (source:str)
```

Parameters

source The string from which all trailing blanks are to be deleted.

Examples

```
$nmdebug > wl strrtrim("  A string with extra blanks.  "):"qo"
"  A string with extra blanks."

%cmdebug > = strltrim(strrtrim("   ABCD    "))
"ABCD"
```

Delete both leading and trailing blanks.

Limitations, Restrictions

none

func strup

String upshift. Returns a string which is the result of upshifting all alphabetic characters in the source string.

Syntax

```
strup (source)
```

Formal Declaration

```
strup:str (source:str)
```

Parameters

source The string whose alphabetic characters are to be upshifted.

Examples

```
$nmdebug > var cows "brindle and bessie.  jenny and boss."
$nmdebug > wl strup(cows)
BRINDLE AND BESSIE.  JENNY AND BOSS.
```

Upshift the string variable and display the results.

```
$nmdebug > if strup(strinput("continue? ")) = "N" then abort
```

Prompt the user to continue and if the response is `N` or `n` then abort.

Limitations, Restrictions

none

func strwrite

Returns a string which is the result of formatting one or more expressions in a manner equivalent to that of the W (WRITE) command.

Syntax

```
strwrite (valuelist)
```

Formal Declaration

`strwrite:str (`*valuelist*`:str)`

Parameters

valuelist A list of expressions, in the form of a single string, to be formatted. The expressions can be separated by blanks or commas:

value1, value2 value3 ...

An optional format specification can be appended to each expression, introduced with a required colon, in order to select one of the following: a specific output base, left or right justification, blank or zero fill, and a field width for the value.

value1[:*fmtspec1*] *value2*[:*fmtspec2*] ...

A format specification string is a list of selected format directives, with each directive separated by blanks, commas or nothing at all:

"*directive1 directive2*, *directive3directive4* ..."

The following table lists the supported format directives that can be entered in upper- or lower-case:

+ Current output base ($, #, or % prefix displayed)

`-`	Current output base (no prefix)
`+<`	Current input base (`$`, `#`, or `%` prefix displayed)
`-<`	Current input base (no prefix)
`$`	Hex output base (`$` prefix displayed)
`#`	Decimal output base (`#` prefix displayed)
`%`	Octal output base (`%` prefix displayed)
`H`	Hex output base (no prefix)
`D`	Decimal output base (no prefix)
`O`	Octal output base (no prefix)
`A`	ASCII base (use "." for non-printable chars)
`N`	ASCII base (loads actual non-printable chars)

`L`	Left justified
`R`	Right justified
`B`	Blank filled
`Z`	Zero filled
`M`	Minimum field width, based on value
`F`	Fixed field width, based on the type of value
`Wn`	User specified field width *n*
`T`	Typed (display the type of the value)
`U`	Untyped (do not display the type of the value)
`QS`	Quote single (surround w/ single quotes)
`QD`	Quote double (surround w/ double quotes)
`QO`	Quote original (surround w/ original quote character)
`QN`	Quote none (no quotes)

The `M` directive (minimum field width) selects the minimum possible field width necessary to format all significant digits (or characters in the case of string inputs).

The `F` directive (fixed field width) selects a fixed field width based on type of the value and the selected output base. Fixed field widths are listed in the following table:

Types	hex($,H)	dec(#,D)	oct(%,O)	ascii(A,N)
`S16,U16`	4	6	6	2
`S32,U32`	8	10	11	4
`S64`	16	20	22	8
`SPTR`	8	10	11	4
`LPTR` Class	8.8	10.10	11.11	8
`EADDR` Class	8.16	10.20	11.22	12
`STR`	field width = length of the string			

The `Wn` directive (variable field width) allows the user to specify the desired field width. The `W` directive can be specified with an arbitrary expression. If the specified width is less than the minimum necessary width to display the value, then the user width is ignored, and the minimum width used instead. All significant digits are always printed. For example:

```
number:"w6"
```

or

```
number:"w2*3"
```

The number of positions specified (either by `Wn` or `F`) does not include the characters required for the radix indicator (if specified) or sign (if negative). Also, the sign and radix indicator will always be positioned just preceding the first (leftmost) character.

Zero versus blank fill applies to leading spaces (for right justification) Trailing spaces are always blank filled.

In specifications with quotes, the quotes do not count in the number of positions specified. The string is built such that it appears inside the quotes as it would without the quotes.

The `T` directive (typed) displays the type of the value, preceding the value.

The `U` directive (untyped) suppresses the display of the type. Types are displayed in upper case, with a single trailing blank. The width of the type display string varies, based on the type, and it is independent of any specified width (`M`, `F`, or `Wn`) for the value display.

For values of type `LPTR` (long pointer, *sid.offset*, or *seg.offset*) and `EADDR` (extended address, *sid.offset* or *ldev.offset*), two separate format directives can be specified. Each is separated by a dot, ".", to indicate individual formatting choices for the "*sid*" portion and the "*offset*" portion. This is true for all code pointers (`ACPTR` - absolute code pointers: `CST, CSTX`; `LCPTR` - Logical Code Pointers: `PROG, GRP, PUB, LGRP, LPUB, SYS, User, TRANS`). For example:

```
pc:"+.-, w4.8, r.l, b.z"
```

The following default values are used for omitted format directives. Note that the default format directives depend on the type of value to be formatted:

```
value type                default format
----------                --------------
STR, BOOL                 - R B M U
U16,S16,U32,S32,S64       + R B M U
SPTR                      + R Z F U
```

```
LPTR                          +.-  R.L  B.Z  M.F  U
 ACPTR    LCPTR               +.-  R.L  B.Z  M.F  T
  CST      PROG               +.-  R.L  B.Z  M.F  T
  CSTX     GRP                +.-  R.L  B.Z  M.F  T
           PUB                +.-  R.L  B.Z  M.F  T
           LGRP               +.-  R.L  B.Z  M.F  T
           LPUB               +.-  R.L  B.Z  M.F  T
           SYS                +.-  R.L  B.Z  M.F  T
           USER               +.-  R.L  B.Z  M.F  T
           TRANS              +.-  R.L  B.Z  M.F  T
EADDR                         +.-  R.L  B.Z  M.F  U
 SADDR                        +.-  R.L  B.Z  M.F  T
```

Note that absolute code pointers, logical code pointers and secondary addresses display their types (`T`) by default. All other types default to (`U`) untyped.

The `C`*n* (Column *n*) directive moves the current output buffer position to the specified column position prior to the next write into the output buffer. Column numbers start at column 1. For example:

```
number:"c6"
```

Note The `C`*n* directive is ignored by the `ASC` function but is honored by the `W`, `WL` and `WP` commands.

Examples

```
$nmdat > var save = strwrite('1 2 3 "-->" 4:"z w4 r z"  5')
$nmdat > wl save
$1$2$3-->0004$5
```

The string variable `save` is used to store the function return value. `STRWRITE` is equivalent to the `W`(`WRITE`) command, but the formatted output is returned in a string.

Note the single quotes which surround the value list. These turn the value list into a string. Double quotes are then used to form individual string values and format specifications.

func strwrite

STRWRITE is similar to the ASC function. The major difference is that ASC accepts a single expression with an optional format specification:

```
wl ASC(1+2, "w4")
```

while STRWRITE accepts a list of expressions, each with optional formatting:

```
var title = strwrite('"Current Pin:"  pin:"w4", " PC:",  pc')
```

Limitations, Restrictions

none

func symaddr

Returns the bit- or byte-relative offset of a component specified through the path specification, relative to the outer structure.

Syntax

```
symaddr (pathspec [units])
```

Formal Declaration

```
symaddr:u32 (pathspec:str [units:u16=8])
```

Parameters

pathspec A path specification, as described in chapter 5, "Symbolic Formatting/Symbolic Access."

units Specifies the units (that is, bit width) in which the result is given. 1 means bits, 8 means bytes, 32 means words. The default is bytes.

Symbolic offsets are rounded down to the nearest whole unit.

Examples

```
$nmdebug > symopen gradtyp.demo
```

Opens the symbolic data type file `gradtyp.demo`. It is assumed that the Debug variable *addr* contains the address of a `StudentRecord` data structure in virtual memory. The following code fragment is from this file:

```
CONST     MINGRADES   = 1;     MAXGRADES   = 10;
          MINSTUDENTS = 1;     MAXSTUDENTS = 5;

TYPE
```

```
GradeRange    = MINGRADES .. MAXGRADES;
GradesArray   = ARRAY [ GradeRange ] OF integer;

Class         = ( SENIOR, JUNIOR, SOPHOMORE, FRESHMAN );
NameStr       = string[8];

StudentRecord = RECORD
                  Name      : NameStr;
                  Id        : Integer;
                  Year      : Class;
                  NumGrades : GradeRange;
                  Grades    : GradesArray;
                END;
```

```
$nmdebug > wl SYMADDR("StudentRecord.Name")
$0
```

Print the byte offset of the name field for StudentRecord. Since it is the first item in the record, its offset is zero.

```
$nmdebug > wl SYMADDR("StudentRecord.NumGrades"  1)
$a8
```

Print the bit offset of the NumGrades field for StudentRecord.

```
$nmdebug > wl SYMADDR("StudentRecord.Grades[4]" #32)
$9
```

Print the word offset of the fourth element of the grades field for StudentRecord.

Limitations, Restrictions

none

func symconst

Returns the value of a declared constant.

Syntax

```
symconst (pathspec)
```

Formal Declaration

```
symconst:any (pathspec:str)
```

Parameters

pathspec A path specification, as described in chapter 5, "Symbolic Formatting/ Symbolic Access."

Examples

```
$nmdebug > symopen gradtyp.demo
```

Opens the symbolic data type file `gradtyp.demo`. It is assumed that the Debug variable *addr* contains the address of a `StudentRecord` data structure in virtual memory. The following code fragment is from this file:

```
CONST     MINGRADES   = 1;     MAXGRADES   = 10;
          MINSTUDENTS = 1;     MAXSTUDENTS = 5;

TYPE
   GradeRange    = MINGRADES .. MAXGRADES;
   GradesArray   = ARRAY [ GradeRange ] OF integer;

   Class         = ( SENIOR, JUNIOR, SOPHOMORE, FRESHMAN );
   NameStr       = string[8];

   StudentRecord = RECORD
```

func symconst

```
  Name      : NameStr;
  Id        : Integer;
  Year      : Class;
  NumGrades : GradeRange;
  Grades    : GradesArray;
END;
```

```
$nmdebug > wl "Max Number of students = "  SYMCONST("MAXSTUDENTS")
Max Number of students = $5
```

Returns the value of the constant `MaxStudents`.

Limitations, Restrictions

none

func syminset

Returns a Boolean value of TRUE if the set member specified by the member parameter is in the set specified by the virtual address and the path specification.

Syntax

```
syminset (virtaddr pathspec member)
```

Formal Declaration

```
syminset:bool (virtaddr:ptr pathspec:str member:str)
```

Parameters

virtaddr — The virtual address of the start of the set.

Virtaddr can be a short pointer, a long pointer, or a full logical code pointer.

pathspec — The path specification as described in chapter 5, "Symbolic Formatting/Symbolic Access."

member — The string value of the member to test for.

Examples

The following examples assume the following types exist. We also assume that a variable of type `SubjectSet` is located at the virtual address SP-34.

```
VAR  myset : SubjectSet;

   BEGIN
      myset := [ HISTORY, HEALTH, PHYSED ];
   END;
```

```
$nmdat > wl syminset(sp-34, 'subjectset', 'math')
FALSE

$nmdat > wl syminset(sp-34, 'subjectset', 'physed')
TRUE
```

In the example above, the symbolic file name is not specified. The last symbolic file accessed is, therefore, used by default.

Limitations, Restrictions

none

func symlen

Returns the length of a data structure in bits or bytes.

Syntax

```
symlen (pathspec [units])
```

Formal Declaration

```
symlen:u32 (pathspec:str [units:u32=$8])
```

Parameters

pathspec A path specification, as described in chapter 5, "Symbolic Formatting/Symbolic Access."

units Specifies the units (that is, bit width) in which the result is given. 1 means bits, 8 means bytes, 32 means words. The default is bytes.

The symbolic length is rounded up to the nearest whole unit.

Examples

```
$nmdebug > symopen gradtyp.demo
```

Opens the symbolic data type file `gradtyp.demo`. It is assumed that the Debug variable *addr* contains the address of a `StudentRecord` data structure in virtual memory. The following code fragment is from this file:

```
CONST     MINGRADES   = 1;     MAXGRADES   = 10;
          MINSTUDENTS = 1;     MAXSTUDENTS = 5;

TYPE
   GradeRange    = MINGRADES .. MAXGRADES;
```

```
GradesArray   = ARRAY [ GradeRange ] OF integer;

Class         = ( SENIOR, JUNIOR, SOPHOMORE, FRESHMAN );
NameStr       = string[8];

StudentRecord = RECORD
                  Name      : NameStr;
                  Id        : Integer;
                  Year      : Class;
                  NumGrades : GradeRange;
                  Grades    : GradesArray;
                END;
```

func symlen

```
$nmdebug > wl SYMLEN("StudentRecord")
$40
```

Returns the size of a complete `StudentRecord` in bytes.

```
$nmdebug > wl SYMLEN("StudentRecord"  1)
$200
```

Returns the size of a complete `StudentRecord` in bits.

```
$nmdebug > wl SYMLEN("StudentRecord.Grades" #32)
$a
```

Returns the size of `grades` field in a `StudentRecord` in words.

Limitations, Restrictions

none

func symtype

Returns the type of a component described by the path specification.

Syntax

```
symtype  (pathspec)
```

Formal Declaration

`symtype:int (`*pathspec*`:str)`

Parameters

pathspec The path specification as described in chapter 5, "Symbolic Formatting/Symbolic Access." The last element of the path *must* correspond to a user-defined type with a name. Elements of type `integer`, `array`, or `subrange` result in an error. Any value returned by this function may be used successfully in the `FT` command.

Examples

```
$nmdebug > symopen gradtyp.demo
```

Opens the symbolic data type file `gradtyp.demo`. It is assumed that the Debug variable *addr* contains the address of a `StudentRecord` data structure in virtual memory. The following code fragment is from this file:

```
CONST     MINGRADES   = 1;     MAXGRADES   = 10;
          MINSTUDENTS = 1;     MAXSTUDENTS = 5;

TYPE
   GradeRange    = MINGRADES .. MAXGRADES;
   GradesArray   = ARRAY [ GradeRange ] OF integer;
```

func symtype

```
Class         = ( SENIOR, JUNIOR, SOPHOMORE, FRESHMAN );
NameStr       = string[8];

StudentRecord = RECORD
                  Name      : NameStr;
                  Id        : Integer;
                  Year      : Class;
                  NumGrades : GradeRange;
                  Grades    : GradesArray;
                END;
```

```
$nmdebug > wl symtype("StudentRecord.NumGrades")
GRADERANGE
```

Print out the type name of the `NumGrades` field of a `StudentRecord`.

Limitations, Restrictions

None.

func symval

Returns the value of a simple data type specified by a virtual address and a path.

Syntax

```
symval (virtaddr pathspec)
```

Formal Declaration

symval:any (*virtaddr*:ptr *pathspec*:str)

Parameters

virtaddr The virtual address of the data structure.

Virtaddr can be a short pointer, a long pointer, or a full logical code pointer.

pathspec A path specification, as described in chapter 5, "Symbolic Formatting/Symbolic Access."

Examples

```
$nmdebug > symopen gradtyp.demo
```

Opens the symbolic data type file `gradtyp.demo`. It is assumed that the Debug variable *addr* contains the address of a `StudentRecord` data structure in virtual memory. The following code fragment is from this file:

```
CONST     MINGRADES   = 1;     MAXGRADES   = 10;
          MINSTUDENTS = 1;     MAXSTUDENTS = 5;

TYPE
   GradeRange    = MINGRADES .. MAXGRADES;
```

```
GradesArray   = ARRAY [ GradeRange ] OF integer;

Class         = ( SENIOR, JUNIOR, SOPHOMORE, FRESHMAN );
NameStr       = string[8];

StudentRecord = RECORD
                  Name      : NameStr;
                  Id        : Integer;
                  Year      : Class;
                  NumGrades : GradeRange;
                  Grades    : GradesArray;
                END;
```

func symval

```
$nmdebug > wl symval(addr "StudentRecord.Name")
Bill

$nmdebug > wl symval(addr, "StudentRecord.Year")
SENIOR

$nmdebug > IF symval(addr "StudentRecord.Year") = "SENIOR" THEN wl "GRAD!"
GRAD!
```

Refer to the section "Using the Symbolic Formatter" in chapter 5 for more examples including pointers, arrays, and variant/invariant record structures.

Limitations, Restrictions

The path specification used by the `SYMVAL` function must evaluate to a simple type or a string. In particular, `SYMVAL` does not return an array, a record, or a set data structure.

func sys

Coerces an expression into a `SYS` logical code pointer (`LCPTR`).

Syntax

```
sys (value)
```

During the evaluation of the parameter to this function, the search path used for procedure name lookups is limited to the system library file (`SYS`).

Formal Declaration

```
sys:sys (value:any)
```

Parameters

value	An expression to be coerced. All types are valid.

Derivation of the SYS Bit Pattern

Parameter Type	Action
BOOL	0.1 if TRUE, 0.0 if FALSE.
U16 U32 SPTR	Set the SID part to zero. Right justify the original value in the low-order 32 bits of the offset part with zero fill.
S16 S32 S64	Set the SID part to zero. Right justify the original value in the low-order 32 bits of the offset part with sign extension.
LONG Class	Transfer both parts of the address unchanged.
EADDR SADDR	Transfer the SID part unchanged. Transfer the low-order 32 bits of the offset part.
STR	Transfer the ASCII bit pattern for the last eight characters in the string. Strings shorter than eight characters are treated as if they were extended on the left with nulls.

Examples

```
%cmdebug > wl sys(12.304)
SYS %12.304
```

Coerce the simple long pointer into a SYS logical code pointer.

```
%cmdebug > wl sys(pub(24.630))
SYS %24.630
```

The coercion simply changes the associated logical file. Note that no complicated conversion or range checking is performed.

```
$nmdat > wl sys( 1 )
SYS $0.1

$nmdat > wl sys( ffff )
SYS $0.ffff

$nmdat > wl sys( 1234abcd )
```

```
SYS $0.1234abcd

$nmdat > wl sys( -1 )
SYS $0.ffffffff

$nmdat > wl sys( 1234.5678 )
SYS $1234.5678

$nmdat > wl sys( true )
SYS $0.1

$nmdat > wl sys( "ABCDEFG" )
SYS $414243.44454647

$nmdat > wl sys( prog(1.2) )
SYS $1.2
```

Limitations, Restrictions

none

func tcb

Returns the real address of a process' TCB (task control block).

Syntax

```
tcb (pin)
```

Formal Declaration

```
tcb:u32 (pin:u16)
```

Parameters

pin	The process identification number (PIN) for which the real address of the TCB is to be returned.

Examples

```
$nmdebug > wl tcb(8)
$8b5480
```

Display the real address of the task control block for process 8.

```
$nmdebug > dz tcb(8),4
REAL $008b5480   $ 40200000 40260000 000000000 00000000
```

Display real memory for four words at the real address of the task control block.

```
$nmdebug > dv 0.tcb(8),4
VIRT $0.8b5480   $ 40200000 40260000 000000000 00000000
```

The real address can also be used as virtual address by using the space ID (SID) of zero (0), and the real address as the virtual offset.

Limitations, Restrictions

none

func trans

Coerces an expression into a `TRANS` logical code pointer (`LCPTR`).

Syntax

```
trans (value)
```

Formal Declaration

`trans:trans (`*value*`:any)`

Parameters

value An expression to be coerced. All types are acceptable.

Derivation of the TRANS Bit Pattern

Parameter Type	Action
BOOL	0.1 if TRUE, 0.0 if FALSE.
U16 U32 SPTR	Set the SID part to zero. Right justify the original value in the low-order 32 bits of the offset part with zero fill.
S16 S32 S64	Set the SID part to zero. Right justify the original value in the low-order 32 bits of the offset part with sign extension.
LONG Class	Transfer both parts of the address unchanged.
EADDR SADDR	Transfer the SID part unchanged. Transfer the low-order 32 bits of the offset part.
STR	Transfer the ASCII bit pattern for the last eight characters in the string. Strings shorter than eight characters are treated as if they were extended on the left with nulls.

Examples

```
%cmdebug > wl trans(12.304)
TRANS %12.304
```

Coerce the simple long pointer into a TRANS logical code pointer.

```
%cmdebug > wl trans(sys(24.630))
TRANS %24.630
```

The coercion simply changes the type. Note that no complicated conversion or range checking is performed.

Limitations, Restrictions

none

func typeof

Returns the type of an evaluated expression as a string.

Syntax

```
typeof (expr)
```

Formal Declaration

typeof:str (*expr*:any)

Parameters

expr Any expression for which the resultant type is desired.

Examples

```
$nmdebug > wl typeof(1+2+3)
U16

$nmdebug > wl typeof(#65535)
U16

$nmdebug > wl typeof(#65535+1)
U32

$nmdebug > wl typeof (-1)
S16

$nmdebug > wl typeof ($1ffff)
S32

$nmdebug > wl typeof(true)
```

```
BOOL

$nmdebug > wl typeof("Nellie of Meadow Farm")
STR

$nmdebug > wl typeof(typeof(123))
STR

$nmdebug > wl typeof(pc)
SYS
```

func typeof

```
$nmdebug > wl typeof(cmpc)
GRP

$nmdebug > wl typeof(cmtonmnode(cmpc))
TRANS

$nmdebug > wl typeof(a.c00024c8)
LPTR

$nmdebug > wl typeof(pib(pin))
SPTR
```

Limitations, Restrictions

none

func u16

Coerces an expression into an unsigned 16-bit value.

Syntax

```
u16 (value)
```

Formal Declaration

```
u16:u16 (value:any)
```

Parameters

value An expression to be coerced. All types are valid.

Derivation of the U16 Bit Pattern

Parameter Type	Action
`BOOL`	1 if `TRUE`, 0 if `FALSE`.
`U16` `S16`	Transfer the original bit pattern unchanged.

`U32`
`S32`
`S64`
`SPTR`
	Transfer the low-order 16 bits.

`LONG` Class
`EADDR`
`SADDR`
	Transfer the low-order 16 bits of the offset part.

`STR`	Transfer the ASCII bit pattern for the last two characters in the string. Strings shorter than two characters are treated as if they were extended on the left with nulls.

Examples

```
$nmdat > wl u16( 1 )
$1

$nmdat > wl u16( ffff )
$ffff

$nmdat > wl u16( ffff ):"#"
$65535

$nmdat > wl u16( 1234abcd )
$abcd

$nmdat > wl u16( -1 )
$ffff

$nmdat > wl u16( ffffffff ):"#"
#65535

$nmdat > wl u16( 1234.5678 )
$5678

$nmdat > wl u16( true )
$1

$nmdat > wl u16( "ABCDEFG" )
$4647

$nmdat > wl u16( prog(1.2) )
$2
```

Limitations, Restrictions

none

func u32

Coerces an expression into an unsigned 32-bit value.

Syntax

```
u32 (value)
```

Formal Declaration

```
u32:u32 (value:any)
```

Parameters

value An expression to be coerced. All types are valid.

Derivation of the U32 Bit Pattern

Parameter Type	Action
BOOL	1 if TRUE, 0 if FALSE.
U16 S16	Right justify the original 16-bit value in 32 bits with zero fill.
U32 S32 SPTR	Transfer the original bit pattern unchanged.
S64	Transfer the low-order 32 bits.
LONG Class EADDR SADDR	Transfer the low-order 32 bits of the offset part.
STR	Transfer the ASCII bit pattern for the last four characters in the string. Strings shorter than four characters are treated as if they were extended on the left with nulls.

Examples

```
$nmdat > wl u32( 1 )
$1

$nmdat > wl u32( ffff )
$ffff

$nmdat > wl u32( ffff ):"#"
#65535

$nmdat > wl u32( 1234abcd )
$1234abcd

$nmdat > wl u32( -1 )
$ffff

$nmdat > wl u32( ffffffff ):"#"
```

```
#4294967295

$nmdat > wl u32( 1234.5678 )
$5678

$nmdat > wl u32( true )
$1

$nmdat > wl u32( "ABCDEFG" )
$44454647

$nmdat > wl u32( prog(1.2) )
$2
```

Limitations, Restrictions

none

func user

Coerces an expression into a USER library logical code pointer (LCPTR).

Syntax

```
user ([library] value)
```

Formal Declaration

```
user:user ([library:str=''] value:any)
```

Parameters

library — If this value is provided, System Debug restricts procedure name searches to the indicated executable library. This restriction remains in effect until the function's parameters have been completely evaluated. The program file's group and account are used to fully qualify the library file name if needed. The library must have been loaded by the process. If this parameter is omitted, procedure name searches begin at the first user library as specified in the LIBLIST= option of the RUN command (if any). Strings longer than valid file names are truncated to the maximum file name string length.

value — An expression to be coerced. All types are valid.

Derivation of the USER Bit Pattern

Parameter Type	Action
BOOL	0.1 if TRUE, 0.0 if FALSE.
U16 U32 SPTR	Set the SID part to zero. Right justify the original value in the low-order 32 bits of the offset part with zero fill.
S16 S32 S64	Set the SID part to zero. Right justify the original value in the low-order 32 bits of the offset part with sign extension.
LONG Class	Transfer both parts of the address unchanged.
EADDR SADDR	Transfer the SID part unchanged. Transfer the low-order 32 bits of the offset part.
STR	Transfer the ASCII bit pattern for the last eight characters in the string. Strings shorter than eight characters are treated as if they were extended on the left with nulls.

Examples

```
$nmdebug > wl user(,1c.304c)
USER $1c.304c
```

Coerce the simple long pointer into a USER logical code pointer.

```
$nmdebug > wl user(,sys(24.630))
USER $24.630
```

The coercion simply changes the associated logical file. Note that no complicated conversion or range checking is performed.

```
$nmdebug > wl user("mylib.test" myproc )
USER $3f.4c04
```

We asked for the address of the procedure myproc. By providing a library name, we restricted the search for the procedure to the executable library named mylib.test.

func user

Limitations, Restrictions

none

func vainfo

Returns selected information for the specified virtual address.

Syntax

```
vainfo (virtaddr selector)
```

Formal Declaration

vainfo:any (*virtaddr*:ptr *selector*:str)

Parameters

virtaddr The virtual address of the object for which the information is desired.

Virtaddr can be a short pointer, a long pointer, or a full logical code pointer.

selector Selects the process information which is to be returned:

Selector	DEBUG	DAT	SAT
ACCESS_RIGHTS	Yes	No	No
ACCESS_RIGHTS_FMT	Yes	No	No
BASE_VA	Yes	Yes	Yes
BYTES_TO_END	Yes	Yes	Yes
CURRENT_SEC_SPACE	Yes	Yes	Yes
CURRENT_SIZE	Yes	Yes	Yes
DFLT_ACCESS_RIGHTS	Yes	No	No
DFLT_ACCESS_RIGHTS_FMT	Yes	No	No
DIS_EXP_ID	Yes	No	No
ENDING_VBA	No	Yes	Yes
HELP	Yes	Yes	Yes

MAX_SEC_SPACE	Yes	Yes	Yes
MAX_SIZE	Yes	Yes	Yes
OBJECT_CLASS	Yes	Yes	Yes
OPTIONS	Yes	Yes	Yes
PAGES_IN_MEM	Yes	No	No
PDIR_HASH	No	Yes	Yes
PID	Yes	Yes	Yes
VS_OD_PTR	No	Yes	Yes
VPN_CACHE_ENTRY_PTR	No	Yes	Yes
VS_BTREE_HASH	No	Yes	Yes
VS_VPN_CACHE_HASH	No	Yes	Yes

Examples

```
$nmdat > var pibva pib(1)
$nmdat > wl vainfo (pibva, "vs_od_ptr")
$a.c1002ec0
$nmdat > dv c1002ec0,58/4
$ VIRT a.c1002ec0 $ 00000001 08010000 7ffd7ffd 7ffd0000
$ VIRT a.c1002ed0 $ 00000000 0000000a c3580000 c35f4806
$ VIRT a.c1002ee0 $ 00074807 50000000 032a0000 00000056
$ VIRT a.c1002ef0 $ 00000000 00000000 00000000 00000000
$ VIRT a.c1002f00 $ 00000000 00000000 00000000 02000000
$ VIRT a.c1002f10 $ 00000000 ffff0000
```

Define a variable `pibva` to be the address of the PIB (process information block) for PIN 1. Get the address of its `vs_od_ptr`, then display its `vs_od_ptr` in hex.

```
$nmdat > wl vainfo(pibva base_va'')
$a.c3580000
$nmdat > wl vainfo(pibva "ending_vba")
$c35f4806
$nmdat > wl vainfo(pibva "current_size")
$74807
$nmdat > wl vainfo(pibva "object_class")
$56
$nmdat > wl vainfo(pibva "vs_btree_hash")
$0
```

```
$nmdat > wl vainfo(pibva "vs_vpn_cache_hash")
$0
$nmdat > wl vainfo(pibva "pdir_hash")
$0
```

Shows more of the object information for the PIB for PIN 1.

Limitations, Restrictions

none

func vtor

Virtual to real. Converts a virtual address to a real address.

Syntax

```
vtor (virtaddr)
```

In Debug, if the virtual address is not resident, it is brought into memory.

In DAT, if the virtual address is not resident, an error is generated.

Formal Declaration

```
vtor:u32 (virtaddr:ptr)
```

Parameters

virtaddr The virtual address to be converted to a real address.

Virtaddr can be either a short or long pointer.

Examples

```
$nmdebug > wl pc
PROG $741.5934
```

Display the current logical code address (LCPTR) of the NM program counter.

```
$nmdebug > wl vtor(pc)
$1827934
```

Translate the logical code address (LCPTR) into the corresponding real address.

```
$nmdebug > wl rtov(1827934)
$741.5934
```

Converts the real address back into a virtual address (LPTR).

Limitations, Restrictions

none

func vtos

Virtual to secondary. Converts a virtual address to a secondary storage address.

Syntax

```
vtos (virtaddr)
```

The function VTOS returns a secondary storage address as an SADDR, whose SID part is the secondary storage LDEV number and whose offset part is the disk byte address.

Formal Declaration

vtos:saddr (*virtaddr*:ptr)

Parameters

virtaddr The virtual address to be converted to a secondary storage address.

Virtaddr can be either a short or long pointer.

Examples

```
$nmdebug > wl vtos(b.40040200)
SADDR $14.e0200
```

Convert the virtual address b.40040200 to a secondary storage address and display the result. The secondary storage address is LDEV $14 at byte offset $e0200.

Limitations, Restrictions

none

FINAL TRIM SIZE : 7.5 in x 9.0 in

9

Dump Analysis Tool (DAT)

The Dump Analysis Tool (DAT) is a program you can use interactively to analyze MPE XL system events such as process hangs, operating system failures, or hardware failures. DAT is used primarily by Hewlett-Packard support and lab personnel.

How DAT Works

As input the DAT program accepts a snapshot dump generated by the DUMP utility. For output, DAT reads the dump tape into one or more disk files, called the dump file set.

`GETDUMP` is the DAT command that reads the DUMP utility tape into the dump file set so that the information can be analyzed interactively.

DAT commands allow the user to display data in the main memory dump as well as secondary store data provided by DUMP. The `OPENDUMP` command opens a dump for analysis; `PURGEDUMP` deletes a dump.

Physical, secondary, and virtual addressing modes are supported. Physical and secondary addressing can be performed regardless of the accuracy of the dump contents. However, virtual addressing requires that certain data structures involved in the address translation process not be corrupt. Most System Debug symbolic formatting commands and functions may be used to symbolically format data within a dump.

Operating DAT

Follow these steps to use DAT:

1. Take a snapshot dump of the system that failed, using the DUMP utility. Refer to *System Startup, Configuration, and Shutdown Reference Manual* (32650-90042) for information about making a DUMP tape.
2. Invoke the DAT utility; the command interpreter prompt (usually a colon) is replaced by the DAT program prompt:

   ```
   :DAT
   $nmdat>
   ```

   OR:

   ```
   :RUN DAT.DAT.TELESUP
   $nmdat>
   ```

3. Create the dump. A request will appear on the system console to mount the dump tape. The following example creates the dump EXAMP.

   ```
   $nmdat>GETDUMP examp

   Please mount dump volume #1.
   ```

4. Mount the dump tape when prompted by the message on the system console. Press RETURN. As the dump is being loaded, DAT will display a series of messages about the dump indicating `GETDUMP` progress:

```
Tape created by SOFTDUMP 99999X A.00.00
MPE-XL B.05.09 dumped on SAT, OCT 20, 1990,  1:44 AM

     Dump Tape Contents
     ------------------

   PIM00       4.0 Kbytes
   MEMDUMP    32.0 Mbytes
   VM001      59.5 Mbytes
```

```
This dump will require approximately 32.1 Mbytes (#131387 sectors) of disc
space.

Please stand by for disc space allocation.

                             0        100%
Loading tape file PIM00   :  +....+....+
Loading tape file MEMDUMP :  +....+....+
Loading tape file VM001   :  +....+....+

Please stand by while dump pages are posted to disk.

Dump disc file space reduced by 59% due to LZ data compression.
$nmdat>
```

5. Open the dump. The following example opens the dump EXAMP.

```
$nmdat>OPENDUMP examp

Dump Title: System failure during performance testing.
Last PIN  : 7  On ICS stack -- Dispatcher running

$nmdat>
```

6. Analyze the dump, using the commands and DAT macros described later in this chapter. If the dump file set was opened successfully, you can display the machine registers, any data locations (using physical, secondary and virtual addressing modes), and the basic tables used in the virtual address translation process.

7. When finished with a dump file set, you can exit the utility or open another file set. All dump file sets remain in the system until you explicitly purge them with the PURGEDUMP command.

```
$nmdat> PURGEDUMP examp
$nmdat> EXIT
:
```

Note When you use the `EXIT` command in DAT, the DAT program terminates immediately.

Using the info= String

DAT automatically executes any commands specified within the `info=` string on a `RUN DAT` command. These commands are executed *before* any commands found in the optional `DATINIT` file(s).

```
run dat; info='{cmd1, cmd2, cmd3}'
```

Automatic `DATINIT` Files

DAT supports the automatic execution of commands with special initialization files named `DATINIT`, if any exist. These files must be standard USE files (see the `USE` command).

DAT first tests for an initialization file (`DATINIT`) in the same group and account as the DAT program file that is being executed. Secondly, DAT looks for an initialization file in the logon group and account (if different from the program file's group and account).

Based on the existence of these special files, it is possible to execute initialization command files from the program's group and account, from the user's group and account, or from both.

The following initialization sequence is possible for DAT:

1. `run dat; info="{cmdlist}"` *INFO string command list*

2. `DATINIT.ProgGrp.ProgAcnt` *program file group/account*

3. `DATINIT.UserGrp.UserAcnt` *user's group/account*

To *prevent* use of the `DATINIT` files, use the following `RUN` command with `info=` string:

```
run dat;info="use close; use close"
```

Since the `info=` string has precedence over the `DATINIT` files, the `use close` commands are the first commands that DAT executes. In this case, any open `DATINIT` files are closed before any commands are read from them.

Operating Restrictions

The following limitations exist in DAT:

- The only symbols that are accessible in CM are the SL.PUB.SYS symbols. This is because SL.PUB.SYS is the only CM library/program file that is dumped by the DUMP utility.
- Typically, only NL.PUB.SYS symbols are accessible in NM. This is because NL.PUB.SYS is treated as a special file by the DUMP utility. The complete NL is dumped along with a pre-built symbol table which enables DAT to quickly map back and forth between addresses and symbol names. Additional executable libraries may also be accessible, *if* they have been marked to be dumped.
- NM stack traces will only trace procedures in NL.PUB.SYS. An exception to this is when the unwind descriptors for the code which called NL.PUB.SYS are memory-resident.
- For the standard functions `nmaddr` and `nmfile`, only addresses contained in the system library are valid.
- You *cannot* use the following DEBUG commands in DAT:

- `B` (set a breakpoint)
- `BD`
- `BL`
- `C` (continue)
- `DATAB`
- `DATABD`
- `DATABL`
- `F` (freeze)
- `M` (modify)
- `S,SS`
- `TRAP`
- `U` (unfreeze)

The following is a summary of DAT commands.

`CLOSEDUMP` closes a dump file set

`DEBUG` gives access to restricted debugging mode

`DPIB`	displays data from PIB for a block
`DPTREE`	prints the process tree
`DUMPINFO`	displays dump file set information
`GETDUMP`	reads in dump tape, creates dump file set
`INIT`*xx*	initializes DAT registers from specified location
`OPENDUMP`	opens a dump file set
`PURGEDUMP`	deletes a dump file set

The DAT Macros

The commands provided by DAT presuppose a solid background in MPE XL internals. To help reduce the need for every dump analysis engineer to possess detailed knowledge of MPE XL, a group of dump analysis macros have been developed to assist field and lab support personnel in the task of dump analysis.

This group of macros (*MPEXL OS DAT MACROS, HP30357 A*) is referred to as "The DAT Macros." An external specification document and quick reference guide is available from HP support organizations. The DAT program, supported macros, (MOS), and symbolic data type files (SYMOS, VAMOS)are distributed in the `TELESUP` account.

How to Get Started with the DAT Macros

Using the DAT macro package is the simplest way to analyze a dump. Additional documentation is required to make use of the macros. Contact your Response Center for further information.

To use this package, log on to the `TELESUP` account in the `USER` group. The `TELESUP` account is where the DAT program, the macro files, and the symbolic data type files are located. The first step is to start the DAT program and invoke the DAT Macros startup macro. Entering "macstart" loads Macros and symbols.

Examples

Some examples of DAT macros follow. Please note that these macros are dynamic. They *will* change and be improved. The output from these examples may differ from what future macros produce.

```
:DAT

DAT XL A.00.00   Copyright Hewlett-Packard Co. 1987.  All rights reserved.

$e ($0) nmdat > macstart

Welcome to the DAT Macro facility.

Enter the dump file set name to process: d7850.dumps

Dump Title: System abort 1019 subsys 101 System Halt 7, $03FB
Last PIN  : 77

MPE XL HP31900a.21.19   USER VERSION: X.13.20

(UNWIND   - Unwinding Out Of Lockup Loop)
(UWLOCKUP - HALT $7,$3fb = #7,#1019)

OS Symbol file SYMOS.OSA20.TELESUP is now open.

Next line maps VAMOS.OSA20.TELESUP
1  VAMOS.OSA20.TELESUP   10000.0 Bytes = 1bd0
WARNING!  OS Build ID Timestamps in System Globals and SYMOS do NOT match.
   OS Build ID Timestamp in System Globals   = 1989050816
   OS Build ID Timestamp in SYMOS File       = 1989040717

OS Macros restored from file MOS.OSA20.TELESUP.
OS DAT MACROS HP30357 A.00.27   Copyright Hewlett-Packard Co.  1987
```

At this point, the dump has been opened and all of the DAT macros have been loaded.

This example displays the basic state of the machine at the time it was dumped.

```
1e ($77) nmdat > machine_state
NWIND   - Unwinding Out Of Lockup Loop)
WLOCKUP - HALT $7,$3fb = #7,#1019)

3000 Series 930 With Processor Revision 0.

STEM ABORT #1019 FROM SUBSYSTEM #101 (Memory Manager)
e MEMORY MANAGER was unable to access the I/O notification port.

E/XL VERSION:  A21.19          CPU: PROCESS_RUNNING

STEM CONSOLE AT LDEV #20

RRENT REGISTERS:

 =00000000 c0000000 002d5838 c0000000 R4 =00000002 4027637c 00000001 40276310
 =40276370 20000000 ffffffff 00000001 R12=00000001 00000b3a fffffd88 00000000
6=0000000a ffffffff 00000000 809766bc R20=00000001 00000e00 ffffffff 00000000
4=00000000 00000000 03fb0065 c0202008 R28=00000001 40276370 40276600 002d5838

SW=0004ff0b=jthlnxbCvmrQpDI  PRIV=0   SAR=0002  PCQF=a.196eb8  a.196ebc

T=0000000a 000002e4 0000000a 00000000 SR4=0000000a 000002e4 0000000b 0000000a
0=00814200 00844200 00000000 40276600 TR4=c0000000 00002058 0000002e 00000000
D1=0280=0140(W)  PID2=07de=03ef(W)    PID3=0000=0000(W)  PID4=0000=0000(W)

TR=00000000 ISR=0000000a IOR=00000000 IIR=00020005 IVA=00169800 ITMR=c931977a
EM=ffffffff EIRR=80000000 CCR=0080

NWIND   - Unwinding Out Of Lockup Loop)
WLOCKUP - HALT $7,$3fb=#7,#1019)
```

The following example shows the dispatcher's state and queues:

```
$11f ($77) nmdat > process_dispatcher

                 Processes on the Dispatch Queue
                 -------------------------------

            ===== DISPATCHER INFORMATION FOR A PROCESS =====

S
y
s
p
r
o
c PIN #   State       Wait Event    Pri   Class  Blocked Reason
- -----   -----       ----------    ---   -----  --------------
  $77     EXECUTING   Not Waiting   $1aff  DS    NOT_BLOCKED
  $2d     READY       Not Waiting   $1aff  DS    MEM_MGR_PREFETCH
  $6f     READY       Not Waiting   $1aff  DS    MEM_MGR_PREFETCH
  $72     READY       Not Waiting   $1aff  DS    MEM_MGR_PREFETCH
  $40     READY       Not Waiting   $1aff  DS    MEM_MGR_PREFETCH
  $39     READY       Not Waiting   $1aff  DS    NM_CODE_PAGE_FAULT
  $47     READY       Not Waiting   $1aff  DS    USER_TO_DEBUG_MSG
  $8B     READY       Not Waiting   $1aff  DS    NOT_BLOCKED

AS  BASEPRI= $70ff  LIMPRI= $4e7f
BS  BASEPRI= $4dff  LIMPRI= $34ff
CS  BASEPRI= $33ff  LIMPRI= $1bff  MINQUANTUM= $186a00  MAXQUANTUM= $f42400
DS  BASEPRI= $1aff  LIMPRI= $8ff
ES  BASEPRI= $7ff   LIMPRI= $17f

Processor State  : PROCESS_RUNNING
Disp Disable PIN : $7ffd     Disp Disable Count : $0
Active PIN       : $77       Active Pri         : $1aff
Pending PIN      : $7ffd     Pending Pri        : $0

Total of #8 processes
```

The following example shows all the configured devices on the system. This macro was terminated with a Control Y before it reached normal completion.

```
$121 ($77) nmdat >  config_device_ldev

LDEV#   TYPE              LDM Port  LDM PDA     DM Port   DM PDA
----     ----              --------  -------     -------   ------
1       IO-DISC           ffffffca  b.80429b00  ffffffcb  b.80140240
2       IO-DISC           ffffffa2  b.8042b180  ffffffa3  b.801409c0
3       IO-DISC           ffffffa0  b.8042c800  ffffffa1  b.80141140
4       IO-DISC           ffffff9e  b.8042de80  ffffff9f  b.801418c0
5       IO_TERMINAL       fffffec6  b.80446e80  0         0.0
6       IO_PRINTER        ffffff88  b.8043a900  ffffff89  0.0
7       IO_TAPE           ffffff91  b.80436580  ffffff92  b.80fe8780
8       IO_TAPE           ffffff93  b.80434f00  ffffff94  b.80fe8140
9       IO_TERMINAL       fffffec5  b.80447dc0  0         0.0
10      IO_TAPE           ffffff8f  b.80437c00  ffffff90  b.80fe8dc0
11      IO_TERMINAL       fffffec4  b.80448d00  0          0.0
12      IO_TERMINAL       fffffec3  b.80449c40  0          0.0
13      IO_TERMINAL       fffffec2  b.8044ab80  0          0.0
14      IO_DISC           ffffff9c  b.8042f500  ffffff9d  b.80142040
15      IO_DISC           ffffff9a  b.80430b80  ffffff9b  b.801427c0
16      IO_DISC           ffffff98  b.80432200  ffffff99  b.80142f40
17      IO_DISC           ffffff96  b.80433880  ffffff97  b.801436c0
18      IO_TERMINAL       fffffec1  b.8044bac0  0         0.0
19      IO_SERIAL_PRINTER ffffff8d  b.80439280  ffffff8e  a.c0c38140
20      IO_TERMINAL       ffffffcd  b.80428480  ffffffce  b.80080240
21      IO_TERMINAL       fffffec0  b.8044ca00  0         0.0
22      IO_TERMINAL       fffffebf  b.8044d940  0         0.0
23      IO_TERMINAL       fffffebe  b.8044e880  0         0.0
24      IO_TERMINAL       fffffebd  b.8044f7c0  0         0.0
100     IO_TERMINAL       ffffff50  b.8043bf80  ffffff51  a.cc810240
101     IO_TERMINAL       ffffff4b  b.8043c5c0  ffffff4c  a.cc810cc0
102     IO_TERMINAL       ffffff46  b.8043cc00  ffffff47  a.cc811740
103     IO_TERMINAL       ffffff41  b.8043d240  ffffff42  a.cc8121c0
104     IO_TERMINAL       ffffff3c  b.8043d880  ffffff3d  a.cc812c40
105     IO_TERMINAL       ffffff37  b.8043dec0  ffffff38  a.cc8136c0
108     IO_TERMINAL       ffffff32  b.8043e500  ffffff33  a.cc814140
109     IO_TERMINAL       ffffff2d  b.8043eb40  ffffff2e  a.cc814bc0
110     IO_TERMINAL       ffffff28  b.8043f180  ffffff29  a.cc815640
Control-Y encountered
```

The following example shows all of the jobs and sessions on the system.

```
                                                                    JSMAIN
JOBNUM   STATE  IPRI  JIN   JLIST  INTRODUCED  JOB NAME                PIN

#S20     EXEC   8     108   108    135  15:47 DAVE,MANAGER.SYS,PUB         $23
#s17     EXEC   8      20    20    135  14:37 DAVE,MANAAGER.SYS,PUB        $20
#J7      EXEC   8      10S   12    135  13:43 PEGASUS,SMGR.TEST,PEGASUS    $21
#J147    EXEC   8      10S   12    135  16:19 TPXRI16J,MGR.FVSTEST,TP      $4c
#J10     EXEC   8      10S   12    135  13:43 PEGASUS,SMGR.TEST,PEGASUS    $35
#J34     EXEC   8      10S   12    135  13:48 PEGASUS,SMGR.TEST,PEGASUS    $42
#J22     EXEC   8      10S   12    135  13:46 PEGASUS,SMGR.TEST,PEGASUS    $27
#J52     EXEC   8      10S   12    135  13:52 PEGASUS,SMGR.TEST,PEGASUS    $67
#J28     EXEC   8      10S   12    135  13:47 PEGASUS,SMGR.TEST,PEGASUS    $48
#J31     EXEC   8      10S   12    135  13:47 PEGASUS,SMGR.TEST,PEGASUS    $4e
#J37     EXEC   8      10S   12    135  13:49 PEGASUS,SMGR.TEST,PEGASUS    $34
#J40     EXEC   8      10S   12    135  13:49 PEGASUS,SMGR.TEST,PEGASUS    $53
#J43     EXEC   8      10S   12    135  13:50 PEGASUS,SMGR.TEST,PEGASUS    $4d
#J154    EXEC   8      10S   12    135  16:19 PHCRP13J,MGR.FVSTEST,PH      $61
#J155    EXEC   8      10S   12    135  16:20 CICAL20J,MGR.FVSTEST,CI      $8c
#J61     EXEC   8      10S   12    135  13:54 PEGASUS,SMGR.TEST,PEGASUS    $65
#J55     EXEC   8      10S   12    135  13:53 PEGASUS,SMGR.TEST,PEGASUS    $6c
#58      EXEC   8      10S   12    135  13:54 PEGASUS,SMGR.TEST,PEGASUS    $5c
#J157    EXEC   8      10S   12    135  16:20 ACALG12J,MGR.FVSTEST,AC      $44
#S8      EXEC   8     122    122    35  13:55 MGR.FVSTEST,PUB              $6d

20 JOBS:
    0 INITIALIZING;    0 INTRODUCED
    0 WAIT
   20 EXEC;   INCL     3 SESSIONS
    0 SCHEDULED;       0 SUSPENDED
    0 TERMINATING;     0 ERROR STATE
JOBFENCE= 7;  JLIMIT= 60;  SLIMIT= 60
```

The above examples give a hint of the power and convenience of using the DAT macros package for dump analysis. There are many more macros; they format an operating system table, print process information, display resource allocation, help find deadlocks, and so on.

10

Standalone Analysis Tool (SAT)

The Standalone Analysis Tool (SAT) aids support and lab personnel in analyzing MPE XL system events such as process hangs, operating system failures, and hardware failures.

How SAT Works

SAT is implemented as a standalone image. You can boot it from ISL. This means you can analyze system failures as soon as they occur without taking a dump.

Being a bootable utility, SAT runs in the area of memory saved by MMSAVE during the boot from the primary boot path. SAT directly accesses main memory, the memory save area on LDEV 1 and virtual storage on the system disks. Like DAT, SAT requires that the data structures involved in virtual address translation be intact in order to support virtual addressing.

SAT lets you analyze a failure quickly without going through the dump process. Then, if you do decide to make a dump tape, exit to ISL and invoke the DUMP utility. The main memory contents and the data on disk are not altered by SAT.

Operating SAT

Follow these steps to use SAT:

1. First, be sure the system has failed.
2. Use the `TC` command to restart the failed or hung system through the access port. This preserves memory.

   Do *not* use the `RS` command—it erases memory!

Note If SAT is not present on disk and must be booted from tape, ISL *must* first be booted from disk so that the MMSAVE utility has a chance to save main memory to disk. If this step is skipped, SAT is loaded into memory, overlaying the state of the machine.

The following example shows what a user might see entering `TC` to transfer control, then `CO` to return to console mode.

```
CM>TC
Transfer Control
CM>CO
Return to Console mode
Processor Dependent Code (PDC) Revision 3

Console Path = 8.1.0.0.0.0.0
Primary boot Path = 8.0.0.0.0.0.0
Alternate boot path = 8.2.3.0.0.0.0

Autoboot from primary path enabled.
To override, press any key within 10 seconds.

10 seconds expired -- proceeding with autoboot.

Booting from primary boot path = 8.0.0.0.0.0.0

Console IO Dependent Code (IODC) revision   3
Boot    IO Dependent Code (IODC) revision   3

Soft Booted.

MMSAVE Version 9.60
DUMPAREA found, save main memory to disk

ISL loaded

ISL Revision 2634 August, 1986
```

3. Invoke SAT from the ISL interface.The following output is a sample SAT session:

```
ISL> SAT
MPE/XL launch facility
Initialize_genesis - Version : <<870204.1552>>
TUE, MAY 16, 1989, 3:35:13 PM (y/n)?  y
[TMUX_DAM] 19 7 8 2
Initialize memory manager completed.
SAT/XL A.00.13  Copyright Hewlett-Packard Co. 1987. All rights reserved.

Locating LIF file: DUMPAREA
LIF file: DUMPAREA   Ldev: 1  Sector: 477744  Length: 65536
Configuring disk drives
Configuring Path 8.0.1 as Ldev 2
Configuring complete
Initialize system related information

Hardware Model: Series 930

Last CPU PIM:

PC = a.ad8ac

General Registers
-----------------
R 0/00000000 fd3c336b 00160d20 c7400380 c7400380 c7400380 00007ffd 40000000
R 8/00000002 c7400380 c7400380 c7400380 c7400380 00000001 80000000 00000007
R16/00000000 0000000e 00000003 00678000 8118a000 00000014 c6809880 00000000
R24/00000000 00000000 0004007b c0200008 fba8b500 0000000e 8118a6e0 00d84200

Space Registers
---------------
S 0/0000000a 0000010d 00000000 00000000 0000000a 0000000a 0000000b 0000000a

Control Registers
-----------------
C 0/00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
C 8/00000102 00000000 00000080 00000002 00000000 00000000 0008d000 ffffffff
C16/fd3c3e64 0000000a 000ad8a8 b7e07000 0000000a 00000000 0004ff0a 00000000
C24/005e4200 00634200 c0000000 001efb98 ffffffff 000888d0 fc8a711d 00007ffd

Current CPU:  0   Original CUP:  0   Monarch CPU:  0   MP array at: 720000

Main memory: 27fffff
Hash table: 634200.40000  Pdir table: 5e4200.50000
RGLOB: 678000  ICS: 8a9000  TCB_BASE: 8d1000  TCB: 8d6900
```

4. Analyze the failure. Most of the System Debug commands are available to you; restrictions are listed below. If you want to make a dump tape, return control to ISL with the `EXIT` command, then invoke the DUMP utility.

Operating Restrictions

The following limitations exist in SAT:

- The symbolic access functions are not available.
- The only symbols that are accessible in CM are the SL.PUB.SYS symbols.
- No operation that involves the file system, such as use files, list, or log files is allowed, since the file system is not available in a standalone environment.
- Some commands and functions are different in SAT:
  - The `EXIT` and `C[ONTINUE]` commands return control to ISL. The `EXIT` command has two additional parameters, *ISL_Command* and *ABORT*. An example follows in "SAT Commands" in this chapter.
  - The `FPMAP` command is automatic and is executed at boot time. When the most recent process is executing in REAL mode, it may be necessary to switch to another PIN and issue the FPMAP comand explicitly. Since only SL.PUB.SYS CM symbols are accessible, no parameters are need with `FPMAP`.
  - For standard functions **`nmaddr`** and **`nmfile`**, only addresses contained in the system library will succeed.
  - For standard function **`strmax`**, SAT strings are limited to 1024 characters.
- The following System Debug commands *cannot* be used in SAT:

`:`	Call the MPE XL command interpreter.
`ABORT`	Abort the process.
`B`	All forms of the Break command.
`BD`	Breakpoint Delete.
`BL`	Breakpoint List.

CLOSEDUMP	Close a dump file.
C[ONTINUE]	Continue.
DATAB	Data Breakpoint.
DATABD	Data Breakpoint Delete.
DATABL	Data Breakpoint List.
DEBUG	Enter the debugger.
DUMPINFO	Display dump file information.
F	All forms of the `Freeze` command.
FINDPROC	Dynamically load NL library procedure.
FT	Format type.
FV	Format virtual.
GETDUMP	Read in a dump tape to create a dump file.
KILL	Kill a process.
LIST	Create list files.
LOADINFO	Display currently loaded program/libraries.
LOADPROC	Dynamically load CM library procedure.
LOG	Create log files.
M	Most forms of the `Modify` command. (MSEC, MV, MZ, *are* supported).
MAP	Map a file into virtual memory.
MAPL	List mapped files.
MODD	Delete temporary dump modification(s) in DAT.
MODL	List temporary dump modification(s) in DAT.
NMCALL	Dynamically invoke the specified routine.
OPENDUMP	Open a dump file.
PAUSE	Sleep for a bit.
PSEUDOMAP	Maps in a local copy of a code file to a virtual address.
PURGEDUMP	Purge a dump file.
REGLIST	List registers to a file.
RESTORE	Restore macros/variables from a binary file.
S[S]	Single Step.
STORE	Store macros/variables to a binary file.

`SYMOPEN`	Symbolic type files cannot be accessed in SAT
`TERM`	Terminal Semaphore control.
`TRAP`	Arm/Disarm/List Traps.
`TX@`	All text window commands.
`UF`	All forms of the `UnFreeze` command.
`UNMAP`	Unmap a mapped file.
`USE`	Read command from a file.
`XLD`	Remove an alternate file of procedure names.

SAT Functions and Commands

Some functions are different in SAT. Three `MODIFY` commands are enabled for SAT, and the `FPMAP` and `EXIT` commands are changed. SAT is a standalone environment, so the file system is not available. This means that no operation which involves the file system, such as USE files, List or Log files is allowed.

For standard functions `nmaddr` and `nmfile`, only addresses contained in the system library succeed. For standard function `strmax`, strings are limited to 1024 characters.

There are no additional commands for SAT, but three DEBUG `MODIFY` commands have been enabled for it so that repairs may be made to the machine state, system tables or other data structures. These commands are summarized below. For more information, see the `M` (`MODIFY`) command description in Chapter 4.

`MV`	modifies a virtual address
`MZ`	modifies a real address
`MSEC`	modifies addresses in secondary (disk drive) storage

Note Take care when using these commands; modifications can be permanent, such as disk changes.

The FPMAP command is automatic and is executed at boot time. When the most recent process is executing in REAL mode, it may be necessary to switch to another PIN and issue the FPMAP command explicitly. Only SL.PUB.SYS CM symbols are accessible, so FPMAP alone (no parameters) is sufficient.

The exit and c[ontinue] commands return control to ISL. However, the exit command has two additional parameters, as shown in the following syntax example:

EXIT [*ISL_Command*] [,*ABORT*]

Parameters:

ISL_Command	Allows you to directly pass a command to ISL. For example, enter the following to tell ISL to load the START PME: exit start.
ABORT	This option tells ISL to abort the AUTOBOOT sequence if it is enabled.

A

Patterns and Regular Expressions

Several System Debug commands apply the concept of pattern matching. Commands such as `CMDLIST`, `ENVLIST`, `FUNCLIST`, `MACLIST`, `PROCLIST`, `SYMLIST`, and `VARLIST` support pattern matching in order to select which commands, functions, macro names, procedure names, symbol names, or variables are to be displayed.

In a similar manner, the `FILTER` environment variable is used to selectively filter all System Debug output, displaying only those lines that match the pattern or regular expression.

Regular expressions are used to find or match some specified text within a large amount of surrounding text. A typical example is to find all lines in a file that contain the word "computer."

A regular expression can be a single character, like the letter "c" or a more elaborate construct built up from simple things like the string "computer".

Literal Expressions (Match Exactly These Characters)

Any literal character, such as "c", is a regular expression and matches that same character in the text being scanned. Regular expressions may be concatenated: a regular expression followed by another regular expression forms a new regular expression that matches anything matched by the first followed immediately by anything matched by the second. A sequence of literal characters is an example of concatenated expressions. For example, "c0000000" or "computer" is a pattern that matches any occurrence of that sequence of characters in the line it is being compared against.

A regular expression is said to match part of a text line if the text line contains an occurrence of the regular expression. For example, the pattern "aa" matches the line “aabc” once at position 1, and the line “aabcaabc” in two places, and the line “aaaaaa” in five (overlapping) places. Matching is done on a line-by-line basis; no regular expression can match across a line boundary.

Metacharacters

In order to express more general patterns than just literals, some specific characters have been defined. For example, the character “.” as a regular expression matches any single character. The regular expression “a.b” matches “a+b”, “aZb”, and similar strings.

The “.” and other reserved characters are called metacharacters. The special meaning of any metacharacter can be turned off by preceding it with the escape character "\". Thus, "\." matches the literal period character and "\\" matches the literal backslash.

Two positional metacharacters exist. “^” matches the beginning of a line: “^HP” is a regular expression that matches “HP” only if it occurs as the first two characters of the line. Similarly, “$” matches the end of a line: “HP$” matches “HP” only if it is the last thing on a line. Of course, these can work together: “^HP$” matches a line that contains only “HP”.

Character Classes (Match Any One of the Following Characters)

The metacharacter “[” signals that the characters following, up to the next “]”, form a character class, that is, a regular expression that matches any single character from the bracketed list. The character class “[aA]” matches “a” or “A”. A dash “-” is used to signify a range of characters in the ASCII collating sequence. For example, “[a-zA-Z]” matches any alphabetic character, while “[0-9]” matches any numeric character. If the first character in a character class is an “^”, then any character not in the class constitutes a match; for example, [^a] matches any character except an “a”.

Expression Closure (Match Zero or More of the Previous Expressions)

Any regular expression that matches a single character (that is, everything but "^" and "$") can be followed by the character "*" to make a regular expression that matches zero or more successive occurrences of the single character pattern. The resulting expression is called a *closure*. For example, "x*" matches zero or more x's; "xx*" matches one or more "x's"; "[a-z]*" matches any string of zero or more lowercase letters. If there is a choice of the number of characters to be matched, the longest possible string is used even when a match with the null string is equally valid. "[a-zA-Z]*" matches an entire word (which may be a null string); "[a-zA-Z][a-zA-Z]*" matches at least an entire word (one or more letters but not a null string); and ".*" matches a whole line (which may be a null string). Any ambiguity in deciding which part of a line matches an expression is resolved by choosing the match beginning with the leftmost character, then choosing the longest possible match at the point. So "[a-zA-Z][a-zA-Z0-9_]*" matches the leftmost Pascal identifier on a line, "(.*)" matches anything between parentheses (not necessarily balanced), and "..*" matches an entire line of one or more characters but not a null string.

Technical Summary

The following list summarizes the expressions discussed above:

`c`	Literal character
`.`	Any character except newline
`^`	Beginning of line
`$`	End of line (null string before newline)
`[xyz]`	Character class (any one of these characters)
`[^xyz]`	Negated character class (all but these characters)
`*`	Closure (zero or more instances of previous pattern)
`\c`	Escaped literal character (for example, `\^`, `\[`, `\*`)

Any special meaning of metacharacters in a regular expression is lost when 1) escaped, 2) inside `[ ... ]`, or 3) for the following characters:

`^`	When not at the beginning of an expression
`$`	When not at end of an expression
`*`	When beginning an expression

A character class consists of zero or more of the following elements, surrounded by `[` and `]`:

`c`	Literal characters, including `[`
`a-b`	Range of characters (digits, lowercase or uppercase)
`^`	Negated character class if at beginning
`\c`	Escaped character (for example, `\^ \- \\ \]`)

Special meaning of characters in a character class is lost when 1) escaped or 2) for the following characters:

`^`	When not at beginning of a character class
`-`	When at beginning or end of a character class

An escape sequence consists of the character `\` followed by a single character:

`\t`	`tab`
`\\`	`\`
`\c`	`c`

System Debug expects regular expressions to be enclosed in back quotes "`".

System Debug commands support MPE XL style wildcard patterns. These are converted into regular expressions for evaluation.

`@`	Matches any character (same as '`.*`')
`?`	Matches any alphabetic character (same as '`[a-zA-Z]`')
`#`	Matches a numeric character (same as '`[0-9]`')

FINAL TRIM SIZE : 7.5 in x 9.0 in

B

Expression Diagrams

The following diagrams depict valid expressions for DAT/Debug:

```
expression :=        ---+-->     simpexpr       ---+-->
                        |                          |
                        +<--   simpexpr relop    --+


simpexpr :=

                 [+-] --+-->      term          --+-->
                        |                         |
                        +<----     + -       -----+   add, subtract
                        |                         |
                        +<----     OR        -----+   logical OR


term :=

                   --+-->      factor         --+-->
                     |                          |
                     +<----      * /       -----+   multiply, divide
                     |                          |
                     +<-------  MOD      -------+   modulus
                     |                          |
                     +<-------  AND      -------+   logical AND


factor :=

                   --+-->    addrvalue     -----+-->
                     |                          |
                     +<----     <<         -----+   left shift bits
                     +<----     >>         -----+   right shift bits
```

```
 |                         |
 +<-------- BAND ---------+   bit AND
 +<-------- BOR  ---------+   bit OR
```

addrvalue :=

```
--+-->        value          --->
  |
  +-->     value.value       --->  a.cooo2c40
                                    pc, cmpc, pw
```

value :=

```
--+-->   numeric-literal     ---+-->   224
  |                             |
  +-->    string-literal     -->+      "AB", 'ab', `ab`
  |                             |
  +-->      variable         -->+      sdst
  |                             |
  +-->  [ indirect_addr ]    -->+      contents of
  |                             |
  +-->    ( simpexpr )       -->+      ( 25/3 + 1 )
  |                             |
  +-->   NOT expression      -->+      NOT (n < 6 )
  +-->  BNOT expression      -->+      BNOT $FFOF
```

numeric-literal :=

```
        123 | %123 | #123 | $123  default, oct, dec, hex
```

string-literal :=

```
        "ABCD"  |  'ABCD'  |  `abcd`
```

relop :=

```
        <  <=  =  >  >=  <>
```

indirect-addr :=

```
        CST     seg.offset
        CSTX    seg.offset
        DST     seg.offset
        ABS     [offset]
        DB      [offset]
        S       [offset]
        Q       [offset]
        P       [offset]
        REAL    offset

        [VIRT]  offset
        [VIRT]  sid.offset
        [VIRT]  nmlogaddr

        CMLOG   cmlogaddr
```

`SEC` $ldev.offset$

C

Emulated/Translated CM Code

Compatibility mode code segments are executed in *emulation mode*, unless they have been translated by the Object Code Translator (OCT).

Emulation of an instruction can be described in the following way:

1. Fetch the instruction at the current program counter (CMPC).
2. Emulate that instruction with NM precision architecture instructions.
3. Update the program counter to point at the next instruction.

Note that multiple NM Precision Architecture instructions must be executed during the emulation of every single CM instruction. Besides the obvious cost of fetching and emulating the instruction, there is usually additional, less obvious overhead, such as indirection and indexing, and updating STATUS register bits (that is, condition code, carry).

```
              CM Object Code

              CM Instructions

              +--------------+
              | PROC+%0      |
              | PROC+%1      |
              | PROC+%2      |
              | PROC+%3      |
        P >   | PROC+%4      |  ----> Fetch    PUSH S-2,X
              | PROC+%5      |
              | PROC+%6      |
              | PROC+%7      |
              | PROC+%10     |
              | PROC+%11     |
```

```
| PROC+%12    |
| PROC+%13    |
| PROC+%14    |
| PROC+%15    |
| PROC+%16    |
+-------------+
```

Debugging Emulated CM Code

Debugging emulated CM code is relatively straightforward. Since each CM instruction is fetched and emulated, it is necessary to know only where you wish to set a breakpoint.

For emulated CM code you can break at any instruction:

```
$ cmdebug > B PROC+%6
$ cmdebug > B PROC+%10
$ cmdebug > B PROC+%15
```

The debugger places a special BRKP instruction at the specified addresses. When an emulated breakpoint is encountered, the emulator traps it into Debug before the original instruction is emulated. The environment variable *entry_mode* is set to "cm", and the user enters CMDebug.

```
        CM Object Code

        CM Instructions

        +-------------+
        | PROC+%0     |
        | PROC+%1     |
        | PROC+%2     |
        | PROC+%3     |
P  >    | PROC+%4     |
        | PROC+%5     |
    [1] | PROC+%6     |    <- Breakpoints are set in the object code
        | PROC+%7     |       at the specified addresses
    [2] | PROC+%10    |
        | PROC+%11    |
        | PROC+%12    |
        | PROC+%13    |
        | PROC+%14    |
    [3] | PROC+%15    |
        | PROC+%16    |
        +-------------+
```

Object Code Translation

The Object Code Translator (OCT) can be used to analyze CM object code and to translate the CM object code instructions into NM precision architecture instructions. Please refer to *MPE V to MPE XL: Getting Started* (30367-90002).

Translated object code executes significantly faster than the original CM code can be emulated.

The object code translator looks at small object code instruction sequences and translates these individual "sections" of code into a corresponding NM section of code.

```
Original CM
Object Code
                    Translated
                       Into
CM Instructions                         NM Instructions
|             |                         |             |
+-------------+      --------->         +-------------+
|   Orig      |                         |             |
|   CM        |                         | Translated  |
|   Section   |                         |      NM     |
+-------------+      ----+              |   Section   |
                         |              |             |
                         |              |             |
                         +---->         +-------------+
                                        |             |
```

Each CM object code instruction may expand to several NM instructions during translation, but the total translated section requires fewer NM instructions than would be used to emulate the original object code.

The CM emulator updates CM registers (such as STATUS) during the emulation of every single instruction. The OCT may recognize that the STATUS register is not accessed by a sequence of object code, and so ignore updating the STATUS register until later, when it is actually referenced. Performance is improved because unnecessary emulator cycles are saved.

It is important to understand, however, that during the execution of the resulting NM section of code, the actual MITROC bit values in the CM STATUS register may be undefined or incorrect in the middle of the section.

Only at the beginning of each section is the CM state known to be correct. These "safe" boundaries, between sections, are called *node points*.

Node Points in Translated Code

The following diagram shows adjacent sections of CM object code that have been translated into new sections of NM code. The first instruction of each section is marked as a node point. Each CM node point has a corresponding NM node point.

```
          Original CM                          Translated
          Object Code                          Object Code
                              Translated
                                 Into
          CM Instructions                      NM Instructions

          +-------------+      --------->      +-------------+
CM node 1 | Section 1   |                      | Section 1   | NM node 1
          |             |                      |             |
          | ----------- |      ----+           |             |
CM node 2 | Section 2   |          |           |             |
          |             |          +---->      | ----------- |
          |             |                      | Section 2   | NM node 2
          |             |                      |             |
          | ----------- |      ----+           |             |
CM node 3 | Section 3   |          |           |             |
          |             |          |           |             |
          |             |          +---->      | ----------- |
          |             |                      | Section 3   | NM node 3
          |             |                      |             |
          | ----------- |      ----+           |             |
CM node 4 | Section 4   |          |           |             |
          |             |          +---->      | ----------- |
          |             |                      | Section 4   | NM node 4
          |             |                      |             |
          +-------------+      ----+           |             |
                                   |           |             |
                                   |           |             |
                                   |           |             |
                                   |           |             |
                                   |           |             |
```

```
+---->        +------------+
```

Executing a Translated Section

The following diagram indicates that the NM program counter (pc >) is located at the start (node point) of a NM translated section of code.

When all of the instructions in this section are executed, (that is, when pc advances to the next node point at `PROC+%6`), then the state of the machine is exactly the same as if the four original CM object code instructions had been executed (`PROC+%2 through PROC+%6`).

```
             Original CM                             Translated
             Object Code                             Object Code
                                   Translated
                                      Into
             CM Instructions                         NM Instructions

                                                     +-------------+
                                                     | PROC+%0     | N
                                                     |             |
             | ----------- |      ----+              |             |
           N | PROC+%2     |          |              |             |
             | PROC+%3     |          +----          | ----------- |
             | PROC+%4     |                    pc > | PROC+%2     | N
             | PROC+%5     |                         |             |
             | ----------- |      ----+              | Translated  |
                                      |              | Section     |
                                      |              |             |
                                      +----          | ----------- |
                                                     | PROC+%6     | N
                                                     |             |
                                                     |             |
                                                     |             |
                                                     | ----------- |
                                                     | PROC+%13    | N
                                                     |             |
                                                     |             |
                                                     |             |
                                                     |             |
```

```
                                                  |             |
                                                  |             |
                                                  |             |
                                                  +-------------+
```

Note that if, for example, only half of the NM translated section has been executed, it is not equivalent to emulating the first half of the original CM object code instructions.

Note There may not be any correspondence between the relative position and sizes of emulated versus translated code sections.

The Node Functions

Four special functions (`CMNODE`, `CMTONMNODE`, `NMNODE`, `NMTOCMNODE`) are provided to locate the nearest "previous" and "next" nodes for translated code.

The following diagram shows CM object code loaded at %12.0 with its corresponding NM translated code loaded at $1c.34b0. Node points are flagged with an "N".

```
            Original CM                                     Translated
            Object Code                                     Object Code

            CM Instructions                                 NM Instructions
Seg.Off                                     Sid.Off
            +-------------+                                 +-------------+
%12.0     N | PROC+%0     |                 $1c.34b0     N  | PROC+%0     |
%12.1       | PROC+%1     |                 $1c.34b4        |             |
%12.2     N | PROC+%2     |  ---+           $1c.34b8        |             |
%12.3       | PROC+%3     |     |           $1c.34bc        |             |
%12.4       | PROC+%4     |     +-->        $1c.34c0     N  | PROC+%2     |
%12.5       | PROC+%5     |                 $1c.34c4        |             |
%12.7     N | PROC+%6     |  <--+           $1c.34c8        |             |
%12.10      | PROC+%7     |     |           $1c.34cc        |             |
%12.11      | PROC+%10    |     |           $1c.34d0        |             |
%12.12      | PROC+%11    |     +---        $1c.34d4     N  | PROC+%6     |
%12.13      | PROC+%12    |                 $1c.34d8        |             |
%12.14    N | PROC+%13    |                 $1c.34dc        |             |
%12.15      | PROC+%14    |                 $1c.34e0        |             |
%12.16      | PROC+%15    |                 $1c.34e4     N  | PROC+%13    |
%12.17      | PROC+%16    |                 $1c.34e8        |             |
            +-------------+                 $1c.34ec        |             |
                                            $1c.34f0        |             |
                                            $1c.34f4        |             |
                                            $1c.34f8        |             |
                                            $1c.34fc        |             |
                                            $1c.3500        +-------------+
```

```
CMNODE(%12.4)              = %12.2        NMNODE($1c.34dc)             = $1c.34d4
CMNODE(%12.4,"prev")       = %12.2        NMNODE($1c.34dc,"prev")      = $1c.34d4
CMNODE(%12.4,"next")       = %12.7        NMNODE($1c.34dc,"next")      = $1c.34e4

CMTONMNODE(%12.4)          = $1c.34c0     NMTOCMNODE($1c.34dc)          = %12.7
CMTONMNODE(%12.4,"prev")   = $1c.34c0     NMTOCMNODE($1c.34dc,"prev")   = %12.7
CMTONMNODE(%12.4,"next")   = $1c.34d4     NMTOCMNODE($1c.34dc,"next")   = %12.14
```

CM Breakpoints in Translated Code

The following discussion assumes that the current Debug mode is CM (prompt is: `%cmdebug >`).

When a CM breakpoint is set at a CM address of a segment that has been translated, Debug actually sets two breakpoints simultaneously:

1. A CM breakpoint at the specified CM address in the emulated object code, in case the code runs emulated.
2. An NM breakpoint at CMTONMNODE (CM address), that is, at the closest corresponding previous node in the NM translated code.

For example, with the following command, the two breakpoints marked as [1] are set simultaneously:

```
%cmdebug > B 12.4

              Original CM                                    Translated
              Object Code                                    Object Code

              CM Instructions                                NM Instructions
Seg.Off                                  Sid.Off
              +-------------+                                +-------------+
%12.0       N | PROC+%0     |            $1c.34b0      N | PROC+%0     |
%12.1         | PROC+%1     |            $1c.34b4        |             |
%12.2       N | PROC+%2     |            $1c.34b8        |             |
%12.3         | PROC+%3     |            $1c.34bc        |             |
%12.4    [1]  | PROC+%4     |            $1c.34c0  [1] N | PROC+%2     |
%12.5         | PROC+%5     |            $1c.34c4        |             |
%12.7       N | PROC+%6     |            $1c.34c8        |             |
%12.10        | PROC+%7     |            $1c.34cc        |             |
%12.11   [2]  | PROC+%10    |            $1c.34d0        |             |
%12.12        | PROC+%11    |            $1c.34d4  [2] N | PROC+%6     |
%12.13   [3]  | PROC+%12    |            $1c.34d8        |             |
%12.14      N | PROC+%13    |            $1c.34dc        |             |
%12.15        | PROC+%14    |            $1c.34e0        |             |
%12.16        | PROC+%15    |            $1c.34e4      N | PROC+%13    |
              +-------------+                                +-------------+
```

Note that multiple CM address breakpoints may map to the same NM previous node breakpoint. For example:

```
%cmdebug > B PROC+10
```

brkpt # 2 maps to NM $1c.34d4

```
%cmdebug > BPROC+12
```

brkpt # 3 maps to NM $1c.34d4 also

Only one NM breakpoint is needed at $1c.34d4.

NM Breakpoints in Translated Code

The following discussion assumes that the current Debug mode is NM (prompt is: `$nmdebug >`).

NM breakpoints can be set at every instruction within translated code even if the instruction is not at a node point.

This allows careful inspection of the actual sections of NM translated code.

Note Portions of the CM state may be undefined or incorrect when a NM breakpoint is encountered between node points.

For example, the following commands set two breakpoints. The first is at a node point, and the second is not at a node point:

```
$nmdebug > B $1c.34d4
$nmdebug > B $1c.34ec
```

```
                 Translated
                 Object Code

                 NM Instructions
Sid.Off
                 +-------------+
1c.34b0       N  | PROC+%0     |
1c.34b4          |             |
1c.34b8          |             |
1c.34bc          |             |
1c.34c0       N  | PROC+%2     |
1c.34c4          |             |
1c.34c8          |             |
1c.34cc          |             |
1c.34d0          |             |
1c.34d4   [1] N  | PROC+%6     |
1c.34d8          |             |
1c.34dc          |             |
1c.34e0          |             |
```

```
1c.34e4       N | PROC+%13     |
1c.34e8         |              |
1c.34ec  [2]    |              |
1c.34f0         |              |
                +--------------+
```

The single step command (`S`) can be used to step through individual NM Instructions within translated code.

Examples: CM Breakpoints in Translated Code

The following examples show CM breakpoints being set in a segment that has been translated, and is executing translated:

```
%cmdebug > bs ?LSEARCH
added: CM    [1] SYS    12.20251  LSEARCH+%0
       NM    [1] TRAN 21.00530994 XLSEG3:LSEARCH+%0

%cmdebug > bs ?LSEARCH+3
added: CM    [2] SYS    12.20254  LSEARCH+%2
       NM    [2] TRAN 21.0053099c XLSEG3:LSEARCH+%1

%cmdebug > bs 12.20256
added: CM    [3] SYS    12.20256  LSEARCH+%5
       NM    [3] TRAN 21.005309ac XLSEG3:LSEARCH+%4

%cmdebug > bs 12.20260
added: CM    [4] SYS    12.20260  LSEARCH+%7
       NM    [3] TRAN 21.005309ac XLSEG3:LSEARCH+%4

%cmdebug > bl
CM    [1] SYS    12.20251  LSEARCH+%0         XLSEG3       (CST 13)
       Corresponding NM bp = 1
CM    [2] SYS    12.20254  LSEARCH+%2         XLSEG3       (CST 13)
       Corresponding NM bp = 2
CM    [3] SYS    12.20256  LSEARCH+%5         XLSEG3       (CST 13)
       Corresponding NM bp = 3
CM    [4] SYS    12.20260  LSEARCH+%7         XLSEG3       (CST 13)
       Corresponding NM bp = 3
```

Examples showing breakpoints in translated code.

Examples: Program Windows for Translated Code

The following window commands allow inspection of the breakpoints that were just set on the previous page:

```
%cmdebug > rd;qd;sd                   /* clear some room for NM
%cmdebug > nmpe                       /* enable the NM program window
%cmdebug > cmpj ?LSEARCH              /* jump CM to ?LSEARCH
%cmdebug > nmpj cmtonmnode(?LSEARCH)  /* jump NM to nearest node


cmP % SYS    12.20251  (T) XLSEG3            CST 13                Level  0
020251:N    [1]   LSEARCH+%0            035001  :.  ADDS  1
020252:N          LSEARCH+%1            041604  C.  LOAD  Q-4
020253:     [2]   LSEARCH+%2            022007  $.  CMPI  7
020254:           LSEARCH+%3            141535  .]  BNE   P+%35
020255:N          LSEARCH+%4            000600  ..  ZERO, NOP
020256:     [3]   LSEARCH+%5            040020  @.  LOAD  P+%20
020257:           LSEARCH+%6            004300  ..  STAX, NOP
020260:     [4]   LSEARCH+%7            020320   .  PLDA
020261:           LSEARCH+%10           031063  23  PCAL EXCHANGEDB
nmP $ TRANS 21.530994  (Translated CM Seg SYS %12 XLSEG3)          Level  0,0
00530994:N   [1]  LSEARCH+%0           b4840004  ADDI     2,4,4
00530998:                              64800000  STH      0,0(0,4)
0053099c:N   [2]  LSEARCH+%1           446c3ff1  LDH      -8(0,3),12
005309a0:                              3407000e  LDO      7(0),7
005309a4:                              d1861ff0  EXTRS    12,31,16,6
005309a8:                              88e621fa  COMBF,=,N6,7,$00530aac
005309ac:N   [3]  LSEARCH+%4           0800024c  OR       0,0,12
005309b0:                              340d052c  LDO      662(0),13
005309b4:                              d1a91ff0  EXTRS    13,31,16,9
```

FINAL TRIM SIZE : 7.5 in x 9.0 in

D

Reserved Variables/Functions

Table D-1 lists the reserved names for the predefined environment variables (env) and functions (func).

Table D-1. Predefined Environment Variables and Functions

Name	Type	Description
abstolog	func : lcptr	CM absolute address to logical address
arg0..arg3	env : u32	argument registers
asc	func : str	converts an expression to an ASCII string
ascc	func : str	coerces an expression to an ASCII string
autoignore	env : bool	ignores errors on every command
autorepeat	env : bool	repeat last command with carriage return
bin	func : u32	converts an ASCII string to a number
bitd	func : u32	bit deposit
bitx	func : u32	bit extract
bool	func : bool	coerces an expression to BOOL type
bound	func : str	tests for current definition of an operand
btow	func : s16	converts a CM byte offset to a word offset
ccode	env : str	condition code
ccr	env : u32	coprocessor configuration register
changes	env : str	video enhancements for changed window values
checkpstate	env : bool	controls process state verification
cir	env : u16	current instruction register
cisetvar	func : bool	sets a new value for a CI variable
civar	func : any	returns current value of a CI variable
cmaddr	func : lcptr	logical address of a specified CM procedure
cmbpaddr	func : lcptr	logical address of a CM breakpoint index
cmbpindex	func : u16	index number of CM breakpoint at address
cmbpinstr	func : s16	CM instruction at CM breakpoint address
cmdlinesubs	env : bool	enables/disables command line substitutions
cmdnum	env : u32	current command number
cmentry	func : lptr	entry address of CM procedure
cmg	func : sptr	short pointer address of CMGLOBALS record

Table D-1.
Predefined Environment Variables and Functions (continued)

Name	Type	Description
`cmnode`	`func : lptr`	closest CM node point
`cmpc`	`env : lcptr`	full CM program counter logical address
`cmpw`	`env : lcptr`	current CM program window logical address
`cmproc`	`func : str`	returns the name of CM procedure
`cmproclen`	`func : u16`	returns the length of CM procedure
`cmseg`	`func : str`	returns the name of CM segment
`cmstackbase`	`func : lptr`	virtual address of the CM stack base
`cmstackdst`	`func : u16`	data segment number of the CM stack
`cmstacklimit`	`func : lptr`	virtual address of the CM stack limit
`cmstart`	`func : lptr`	start address of CM procedure
`cmtonmnode`	`func : trans`	closest NM node to a CM logical address
`cmva`	`func : lptr`	converts CM code address to virtual address
`cm_inbase`	`env : str`	current CM input base
`cm_outbase`	`env : str`	current CM output base
`column`	`env : u16`	current output column position
`console_debug`	`env : u16`	use system console for I/O
`cpu`	`env : u16`	cpu number of the current processor
`cr0, cr8..cr31`	`env : u32`	control registers
`cst`	`func : cst`	coerces an expression to CST type
`cstbase`	`env : lptr`	virtual address of the CM Code Segment Table
`ccstx`	`func : cstx`	coerces an expression to CSTX type
`cst_expansion`	`env : bool`	CM CST Expansion is supported on MPE XL
`date`	`env : str`	current date
`db`	`env : u16`	CM DB register
`dbdst`	`env : u16`	CM DB data segment number
`disp`	`env : bool`	dispatcher is running
`dl`	`env : u16`	CM DL register

Table D-1.
Predefined Environment Variables and Functions (continued)

Name	Type	Description
`dp`	`env : sptr`	data pointer (alias for R27)
`dstbase`	`env : lptr`	virtual address of the CM Data Segment Table
`dstva`	`func : lptr`	converts CM dst.off to virtual address
`dumpalloc_lz`	`env : u16`	sets disk preallocation for LZ compression
`dumpalloc_rle`	`env : u16`	sets disk preallocation for RLE compression
`dump_comp_algo`	`env : str`	returns compression algo for current dump
`eaddr`	`func : eaddr`	coerces an expression to EADDR type
`echo_cmds`	`env : bool`	echo commands before execution
`echo_subs`	`env : bool`	echo command line substitutions
`echo_use`	`env : bool`	echo use file commands before execution
`eiem`	`env : u32`	external interrupt enable mask
`eirr`	`env : u32`	external interrupt request register
`entry_mode`	`env : str`	mode at entry ("cm" or "nm")
`errmsg`	`func : str`	error message string for error number/subsys
`error`	`env : s32`	most recent error number
`exec_mode`	`env : str`	process execution mode from TCB ("cm" or "nm")
`escapecode`	`env : u32`	last escapecode value
`false`	`env : bool`	the constant FALSE
`fill`	`env : str`	fill character for data display
`filter`	`env : str`	filter pattern for output
`fp0..fp15`	`env : lptr`	floating point registers
`fpe1..fpe7`	`env : s32`	floating point exception registers
`fpstatus`	`env : u32`	floating point status register
`getdump_comp_algo`	`env : str`	sets compression algo for next GETDUMP
`grp`	`func : grp`	coerces an expression to a GRP LCPTR type

Table D-1.
Predefined Environment Variables and Functions (continued)

Name	Type	Description
hash	addr : ptr	hash a virtual address
hexupshift	env : bool	upshifts all HEX output to upper case
icsnest	env : u16	number of nested pending ICS interrupts
icsva	env : lptr	interrupt control stack virtual address
iir	env : u32	interrupt instruction register
inbase	env : str	current input base
ior	env : u32	interrupt offset register
ipsw	env : u32	interrupt processor status word
isr	env : u32	interrupt space register
itmr	env : u32	interval timer
iva	env : u32	interrupt vector address
job_debug	env : u16	enables/disables job debugging
justify	env : str	controls justification for data display
lastpin	env : u16	pin number of process at entry
lgrp	func : lgrp	coerces an expression to a LGRP type
list_input	env : u16	echo user input to list file
list_pagelen	env : u16	page length (in lines) of list file
list_pagenum	env : u16	current page number of list file
list_paging	env : bool	enables/disables paging of list file
list_title	env : str	title for each page of list file
list_width	env : u16	width (in characters) of list file
logtoabs	func : acptr	CM logical address to absolute address
lookup_id	env : str	NM procedure name lookup mechanism
lptr	func : lptr	coerces an expression to LPTR type
lpub	func : lpub	coerces an expression to LPUB type
ltolog	func : lcptr	converts long pointer to logical code pointer
ltos	func : sptr	converts long pointer to short pointer
lw	env : saddr	current LW address in form ldev.offset

Table D-1.
Predefined Environment Variables and Functions (continued)

Name	Type	Description
`macbody`	`func : str`	returns macro body string
`macros`	`env : u16`	the number of macros that can be defined
`macros_limit`	`env : u16`	absolute maximum limit for "macros" (above)
`macro_depth`	`env : u16`	current nested call level for macros
`mapdst`	`env : s16`	current CST Expansion mapping dst number
`mapflag`	`env : s16`	CM segment is logically or physically mapped
`mapindex`	`func : u32`	index number of a MAPPED file
`mapsize`	`func : u32`	size in bytes of a MAPPED file
`mapva`	`func : lptr`	virtual address of a MAPPED file
`markers`	`env : str`	video enhancement for windowed stack markers
`mode`	`env : str`	current mode ("cm" or "nm")
`monarchcpu`	`env : u16`	cpu number of the monarch processor
`mpexl_table_va`	`env : lptr`	address of the table for the MPEXL command
`multi_line_errs`	`env : u16`	controls quantity of lines to display forerrors in a multiple line command
`nmaddr`	`func : ptr`	address of a NM procedure or global data
`nmbpaddr`	`func : lptr`	address of a NM breakpoint index
`nmbpindex`	`func : u32`	index number of NM breakpoint at address
`nmbpinstr`	`func : s32`	NM instruction at NM breakpoint address
`nmcall`	`func : s32`	dynamically invokes the specified routine
`nmentry`	`func : lptr`	entry address of NM procedure
`nmfile`	`func : str`	name of file containing mapped vaddr
`nmmod`	`func : str`	name of NM module
`nmnode`	`func : trans`	closest NM node
`nmpath`	`func : str`	code path for a virtual address
`nmproc`	`func : str`	name of NM procedure
`nmpw`	`env : lcptr`	current NM program window logical address
`nmstackbase`	`func : lptr`	virtual address of the NM stack base
`nmstacklimit`	`func : lptr`	virtual address of the NM stack limit

Table D-1.
Predefined Environment Variables and Functions (continued)

Name	Type	Description
nmtocmnode	func : lptr	closest CM node to NM translated code
nm_inbase	env : str	NM input base
nm_outbase	env : str	NM output base
nonlocalvars	env : bool	enables/disables access to variables which are not local during macro execution
off	func : u32	extract OFFset part of a long pointer
outbase	env : str	current output base
pc	env : lptr	NM program counter (sid.off)
pcb	func : sptr	process control block
pcbx	func : sptr	process control block extension
pcob	env : sptr	program counter offset back (off)
pcof	env : sptr	program counter offset front (off)
pcqb	env : lptr	program counter queue back (sid.off)
pcqf	env : lptr	program counter queue front (sid.off)
pcsb	env : u32	program counter space back (sid)
pcsf	env : u32	program counter space front (sid)
phystolog	func : lcptr	CM physical seg/map bit to logical code ptr
pib	func : sptr	process info block
pibx	func : sptr	process info block ext.
pid1..pid4	env : u32	protection ID registers
pin	env : u16	current PIN number
priv	env : u16	current privilege level (based on PC)
priv_user	env : u16	user has PM (privileged mode) capability
prog	func : prog	coerces an expression to PROG type
progname	env : str	either "dat" or "debug"
prompt	env : str	current user prompt
pseudovirtread	misc: bool	last access came from pseudomapped file
psp	env : u32	previous stack pointer

Table D-1.
Predefined Environment Variables and Functions (continued)

Name	Type	Description
pstate	func : str	process state
pstmt	env : u16	enables/disables the display of statement numbers in NM program window
psw	env : u32	an alias for “ipsw”
pub	func : pub	coerces an expression to PUB type
pw	env : lptr	current program window logical address
pwo	env : sptr	current program window (offset part)
pws	env : u32	current program window (SID/seg part)
q	env : u16	CM Q register
quiet_modify	env : bool	skip display of current values for modifies
r0 .. r31	env : u32	general registers r0, r1, r2, .. r31
rctr	env : u32	recovery counter
ret0 .. ret1	env : u32	return registers 0 and 1
rp	env : sptr	return pointer
rtov	func : lptr	real to virtual
s	env : u16	CM S register
s16	func : s16	coerces an expression to S16 type
s32	func : s32	coerces an expression to S32 type
s64	func : s64	coerces an expression to S64 type
saddr	func : saddr	coerces an expression to SADDR type
sar	env : u32	shift amount register
sdst	env : u16	CM stack data segment number
sid	func : u32	extracts SID part of a long pointer
sl	env : sptr	static link register
sp	env : sptr	stack pointer register
sptr	func : sptr	coerces an expression to SPTR type
sr0 .. sr7	env : u32	space registers sr0, sr1, sr2, . . . sr7
status	env : u16	CM STATUS register

Table D-1.
Predefined Environment Variables and Functions (continued)

Name	Type	Description
`stol`	`func : lptr`	converts a short pointer to long pointer
`stolog`	`func : lcptr`	converts short pointer to logical code pointer
`str`	`func : str`	extracts a sub-string from a string
`strapp`	`func : str`	string append
`strdel`	`func : str`	string delete
`strdown`	`func : str`	downshifts a string
`strextract`	`func : str`	returns a string from memory
`strinput`	`func : str`	prompts for a string input
`strins`	`func : str`	string insert
`strlen`	`func : u32`	returns the current length of a string
`strltrim`	`func : str`	removes leading blanks from a string
`strmax`	`func : u32`	maximum length of a string (constant)
`strpos`	`func : u32`	position of a substring within a string
`strrpt`	`func : str`	string repeat
`strrtrim`	`func : str`	removes trailing blanks from a string
`strup`	`func : str`	upshifts a string
`strwrite`	`func : str`	string write (ala Pascal strwrite)
`symaddr`	`func : u32`	returns the offset to a symbol in a structure
`symconst`	`func : any`	returns the value of a symbolic constant
`syminset`	`func : bool`	test for membership of a symbol in a set
`symlen`	`func : u32`	returns the length of a symbolic data structure
`sympath_upshift`	`env : bool`	controls upshifting of path specs
`symtype`	`func : str`	returns the symbolic type of a specified path
`symval`	`func : any`	returns the value at a virtual address based on a specified symbolic path
`sys`	`func : sys`	coerces an expression to a SYS LCPTR type
`tcb`	`func : u32`	task control block
`term_keeplock`	`env : bool`	retain the terminal locking semaphore

FINAL TRIM SIZE : 7.5 in x 9.0 in

Table D-1.
Predefined Environment Variables and Functions (continued)

Name	Type	Description
`term_ldev`	`env : u16`	the ldev used for I/O
`term_locking`	`env : bool`	enables_disables terminal process queueing
`term_loud`	`env : bool`	enables/disables output echoing to screen
`term_paging`	`env : bool`	enables/disables =terminal screen paging
`term_width`	`env : u16`	width (in characters) of terminal output
`time`	`env : str`	current time of day
`tr0 .. tr7`	`env : u32`	temp registers tr0, tr1, tr2, ..tr7
`trace_func`	`env : u16`	trace function entry, exit and parameters
`trans`	`func : trans`	coerces an expression to a TRANS LCPTR type
`true`	`env : bool`	the constant TRUE
`typeof`	`func : str`	returns type of an expression
`u16`	`func : u16`	coerces an expression to U16 type
`u32`	`func : u32`	coerces an expression to U32 type
`unwind`	`env  : u16`	automatic unwinding enabled
`user`	`func : user`	coerces an expression to a USER LCPTR type
`vainfo`	`func : any`	information about a virtual object
`vars`	`env : u16`	number of variables that can be defined
`vars_limit`	`env : u16`	absolute sum limit of "vars" and "vars_loc"
`vars_loc`	`env : u16`	number of local variables that can be defined
`vars_table`	`env : u16`	current sum of "vars" and "vars_loc"
`version`	`env : str`	version ID for DAT/DEBUG
`vtor`	`func : u32`	virtual to real
`vtos`	`func : lptr`	virtual to secondary storage address
`vw`	`env : lptr`	current virtual window address (lptr)
`vwo`	`env : sptr`	current virtual window address (offset part)
`vws`	`env : u32`	current virtual window space
`win_length`	`env : u32`	number of lines on display terminal
`win_width`	`env : u32`	number of columns on display terminal
`x`	`env : u16`	CM X register (Index Register)
`zw`	`env : u32`	current real memory window address

E

Command Summary

```
Standard Commands
        Window Commands

:                   access to the command interpreter
=                   calculator, expression evaluation
ABORT               terminate dat/debug session
ALIAS               define a user alias
ALIASD[EL]          delete a command alias
ALIASINIT           restore the pre-defined aliases
ALIASL[IST]         list current command alias
B                   set breakpoint
BA                  set breakpoint at an absolute CST address
BAX                 set breakpoint at an absolute CSTX address
BD                  delete breakpoint(s)
BG                  set breakpoint in group library
BL                  list breakpoint(s)
BLG                 set breakpoint in logon group library
BLP                 set breakpoint in logon account library
BP                  set breakpoint in account library
BS                  set breakpoint in system library
BU                  set breakpoint in any NM (user) library
BV                  set breakpoint at a virtual (code) address
C[ONTINUE]          continue program execution
CLOSEDUMP           close a dump file set
CM                  enter Compatibility Mode  (cmdat/cmdebug)
CMDL[IST]           list commands

CMG                 display cmglobals for a process
        CMPB        scroll the CM program window backwards
        CMPD        disable the CM program window
```

CMPE	enable the CM program window
CMPF	scroll the CM program window forwards
CMPH	home the CM program window
CMPJ	jump the CM program window
CMPJA	jump the CM program window to a CST segement
CMPJAX	jump the CM program window to a CSTX segement
CMPJG	jump the CM program window to the group library
CMPJLG	jump the CM program window to the logon group library
CMPJLP	jump the CM program window to the logon account library
CMPJP	jump the CM program window to the account library
CMPJS	jump the CM program window to the system library
CMPK	kill the CM program window
CMPL	change the size of the CM program window
CMPR	change the radix of the CM program window

DA	display absolute memory relative
DATAB	set a data breakpoint
DATABD	delete a data breakpoint
DATABL	list data breakpoints
DC	display code
DCA	display code in a CST segement
DCAX	display code in a CSTX segement
DCG	display code in the group library
DCLG	display code in the logon group library
DCLP	display code in the logon account library
DCP	display code in the account library
DCS	display code in the system library
DCU	display code in any (user) NM library
DD	display data segment
DDB	display CM DB-relative
DELETEALIAS	predefined alias for ALIASD
DELETEB	predefined alias for BD
DELETEERR	predefined alias for ERRD
DELETEMAC	predefined alias for MACD
DELETEVAR	predefined alias for VARD
DEMO	select terminal ldevs for DAT/DEBUG demonstrations
DIS	disassemble code
DO	redo a command from history

DPIB	display a process's information block
DPTREE	display the process tree
DQ	display CM Q-relative
DR	display registers
DS	display CM S-relative
DSEC	display secondary storage relative
DUMPINFO	display information about the open dump
DV	display virtual memory
DZ	display real memory
E[XIT]	exit (predefined alias for C[ONTINUE])
ENV	set an environmental variable value
ENVL[IST]	display environmental variable values
ERR	push an error string onto the error stack
ERRD[EL]	reset the error stack
ERRL[IST]	list the contents of the error stack
FC	freeze code
FCA	freeze code in a CST segement
FCAX	freeze code in a CSTX segement
FCG	freeze code in the group library
FCLG	freeze code in the logon group library
FCLP	freeze code in the logon account library
FCP	freeze code in the account library
FCS	freeze code in the system library
FCU	freeze code in any (user) NM library
FDA	freeze a data segment into memory
FINDPROC	dynamically load a procedure from a NM library
FOREACH	execute a command(list) FOREACH value in a valuelist
FPMAP	Re-initializes CM symbolic procedure names
FT	format a type declaration
FUNCL[IST]	list all the DEBUG/DAT functions
FV	format virtual as a type
FVA	freeze virtual address (range) in memory
GB	scroll group window back
GD	disable the group window
GE	enable the group window
GETDUMP	read a dump tape into disc files

```
            GF          scroll group window forward
            GH          home the group window
            GK          kill the group window
            GL          change the size of the group window
            GR          change the radix for the group window
            GRD         disable the NM general registers window
            GRE         enable the NM general registers window
            GRK         kill the NM general registers window
            GRL         change the size of the NM general registers window
    H[ELP]              print help
    HIST[ORY]           print history of command stack
    IF                  IF <condition> THEN {cmdlist} ELSE {cmdlist}
    IGNORE              ignore error test after the following command
    INITCM              initialize CM registers from any address
    INITNM              initialize NM registers from any address
    KILL                kill the indicated PIN
            LB          scroll the Ldev window back
            LD          disable the Ldev window
            LE          enable the Ldev window

 LEV                set environment to stack level
         LF         scroll the Ldev window forward
         LH         home the Ldev window
 LIST               controls the recording of input and output to a listfile
 LISTREDO           predefined alias for HIST[ORY]
         LJ         jump the Ldev window
         LK         kill the Ldev window
         LL         change the size of the window program
 LOADINFO           give info on loaded NM and CM program/libraries
 LOADPROC           dynamically load a procedure from a CM library
 LOC                declare a local variable
 LOCL[IST]          list the local variables
 LOG                controls the recording of input to a logfile
         LR         change the radix of the Ldev window
         LW         allocate a new virtual window

MA                 modify absolute
MAC[RO]            define a macro
```

MACD[EL]	delete macro definition(s)
MACECHO	enable echoing of each line of macro(s)
MACL[IST]	list the macro definition(s)
MACREF	reset macro reference counts
MACTRACE	enable tracing for macro(s)
MAP	open and map a file into virtual space
MAPL[IST]	list files opened by the MAP command
MC	modify code
MCA	modify code in a CST segement
MCAX	modify code in a CSTX segement
MCG	modify code in the group library
MCLG	modify code in the logon group library
MCLP	modify code in the logon account library
MCP	modify code in the account library
MCS	modify code in the system library
MCU	modify code in any (user) NM library
MD	modify CM data segment
MDB	modify CM DB-relative
MODD	delete temporary dump modification(s) in DAT
MODL	list temporary dump modification(s) in DAT
MPEXL	display version info about MPEXL files in the OS SOM in NL
MPSW	modify the PSW
MQ	modify CM Q-relative
MR	modify registers
MS	modify CM S-relative
MSEC	modify secondary store
MV	modify virtual memory
MZ	modify real memory

NM		enter Native Mode (nmdat/nmdebug)
	NMPB	scroll the NM program window backwards
	NMPD	disable the NM program window
	NMPE	enable the NM program window
	NMPF	scroll the NM program window forwards
	NMPH	home the NM program window
	NMPJ	jump the NM program window
	NMPJG	jump the NM program window to the group library
	NMPJP	jump the NM program window to the account library

```
               NMPJS     jump the NM program window to the system library
               NMPJU     jump the NM program window to any (user) NM library
               NMPK      kill the NM program window
               NMPL      change the size of the CM program window
               NMPR      change the radix of the CM program window

OPENDUMP             open dump disc files for analysis
PAUSE                pause (sleep) for <n> seconds
        PB           scroll the program window backwards
        PD           disable the program window
        PE           enable the program window
        PF           scroll the program window forwards
        PH           home the program window
PIN                  switch context to a specified process
        PJ           jump the current program window
        PJA          jump the current program window to a CST segement
        PJAX         jump the current program window to a CSTX segement
        PJG          jump the current program window to the group library
        PJLG         jump the current program window to the logon group library
        PJLP         jump the current program window to the logon account library
        PJP          jump the current program window to the account library
        PJS          jump the current program window to the system library
        PJU          jump the current program window to any (user) NM library
        PJV          jump the current program window to a virtual address
        PK           kill the program window
        PL           change the size of the program window
        PR           change the radix of the program window
PROCLIST             list NM procedures/dat symbols in a NM executable file
PSEUDOMAP            fill in virtual memory holes from mapped file
PURGEDUMP            delete all disc files in a dump set
        QB           scroll CM frame window back
        QD           disable the CM frame window
        QE           enable the CM frame window
        QF           scroll CM frame window forward
        QH           home the CM frame window
        QJ           jump the CM frame window
        QK           kill the CM frame window
        QL           change the size of the CM frame window
```

```
QR          change the radix of the CM frame window
RD          disable the CM register window
RE          enable the CM register window
RED         redraw the screen

REDO               redo a command after (optionally) editing it
REGLIST            writes NM register values to a file in USE format
RESTORE            restore macros or variables from a file
RET[URN]           return an optional value from a macro
        RH         home the CM register window
        RK         kill the CM register window
        RL         change the size of the CM register window
        RR         change the radix of the CM register window

S[S]             single step, same as SS
        SB       scroll CM stack window back
        SD       disable the CM stack window
        SE       enable the CM stack window
SET              set user configurable options
SETALIAS         predefined alias for ALIAS
SETENV           predefined alias for ENV
SETERR           predefined alias for ERR
SETLOC           predefined alias for LOC
SETMAC           predefined alias for MAC
SETVAR           predefined alias for VAR
        SF       scroll stack window forward
        SH       home the stack window
SHOWALIAS        predefined alias for ALIASL
SHOWB            predefined alias for BL
SHOWCMD          predefined alias for CMDL
SHOWDATAB        predefined alias for DATABL
SHOWENV          predefined alias for ENVL
SHOWERR          predefined alias for ERRL
SHOWFUNC         predefined alias for FUNCL
SHOWLOC          predefined alias for LOCL
SHOWMAC          predefined alias for MACL
SHOWSET          predefined alias for SET (no parms)
SHOWSYM          predefined alias for SYML
```

```
SHOWVAR            predefined alias for VARL
        SJ         jump the CM stack window to a new location
        SK         kill the CM stack window
        SL         change the size of the CM stack window
STORE              store macros or variables to a file
        SR         change the radix of the CM stack window
        SRE        enable the NM special registers window
        SRD        disable the NM special registers window
        SRH        home the NM special registers window
        SRK        kill the NM special registers window
        SRL        change the size of the NM special registers window
SYMCLOSE           close a symbolic data file
SYMF[ILES]         list the currently opened symbolic files
SYMINFO            display info about opened symbolic files
SYML[IST]          display symbolic file information
SYMOPEN            open a symbolic file with data types in debug records
SYMPREP            preprocesses a symbolic data file with SYMDEBUG information

 TA                 translate CM ABS-relative address to virtual
 TC                 translate CM program file code address to virtual
 TCA                translate CM CST code address to virtual
 TCAX               translate CM CSTX code address to virtual
 TCG                translate CM group library code address to virtual
 TCLG               translate CM logon group library code address to virtual
 TCLP               translate CM logon account library code address to virtual
 TCP                translate CM account library code address to virtual
 TCS                translate CM system library code address to virtual
 TD                 translate CM data segment to virtual
 TDB                translate CM DB-relative address to virtual
 TERM               control terminal semephore ownership
 TQ                 translate CM Q-relative address to virtual
 TR[ACE]            stack trace
 TRAP               arm/disarm/list various catchable traps

TS                 translate CM S-relative address to virtual
        TXB        scroll text window backward
        TXC        mark the text window as current
        TXD        disable the text window
```

	TXE	enable the text window
	TXF	scroll text window forward
	TXH	home the text window
	TXI	information about the text window
	TXJ	jump the text window
	TXK	kill the text window
	TXL	change the size of the text window
	TXS	shift text window to left or right
	TXW	allocate a new text window
	UB	scroll user window backward
	UC	mark the user window as current
UFC		un-freeze code in the program file
UFCA		un-freeze code in a CST segement
UFCAX		un-freeze code in a CSTX segement
UFCG		un-freeze code in the group library
UFCLG		un-freeze code in the logon group library
UFCLP		un-freeze code in the logon account library
UFCP		un-freeze code in the account library
UFCS		un-freeze code in the system library
UFCU		un-freeze code in any (user) NM library
UFDA		un-freeze a data segment in memory
UFVA		unfreeze a virtual address (range)
	UD	disable a user window
	UE	enable a user window
	UF	scroll user window forward
	UH	home the user window
	UK	kill a user window
	UL	change the size of a user window
	UN	rename a user window
UNMAP		close file opened by MAP command
UNWIND		restore processor to known state
UPD		update windows
	UR	change the radix of a user window
USE		execute commands from a file
USENEXT		execute a specified number of lines from a command file
	UWA	define a user window absolute relative
	UWCA	define a user window CST segment relative
	UWCAX	define a user window CSTX segment relative

```
UWD     define a user window data segment relative
UWDB    define a user window CM DB-relative
UWL     define a user window LDEV relative
UWS     define a user window CM S-relative
UWQ     define a user window CM Q-relative
UWV     define a user window Precision Architecture virtual address
UWZ     define a user window Precision Architecture real address
```

```
VAR             define/list a user variable
VARD[EL]        delete a user variable
VARL[IST]       list user variables
        VB      scroll virtual window backward
        VC      mark virtual window as current
        VD      disable the virtual window
        VE      enable the virtual window
        VF      scroll virtual window forward
        VH      home the virtual window
        VJ      jump the virtual window to a new location
        VI      information about indicated or all windows
        VK      kill the virtual window
        VL      change the size of the virtual window
        VN      rename the virtual window
        VR      change the radix of the virtual window
        VW      allocate a new virtual window
W               write formatted value list
WCOL            set output position to column
        WDEF    set default window sizes
        WGRP    select a group of windows
WHELP           window help
WHILE           WHILE <condition> DO
WL              write line formatted value list
        WOFF    turn windows off
        WON     turn windows on
WP              write prompt
WPAGE           write page eject
```

```
XL          open a program/library file to access symbol information.
XLD         close a file previously opened via the XL command
XLL         list files opened via the XL command
```

ZB	scroll real memory window backward
ZD	disable real memory window
ZE	enable real memory window
ZF	scroll real memory window forward
ZH	home the real memory window
ZJ	jump the real memory window
ZK	kill the real memory window
ZL	change the size of the real memory window
ZR	change the radix of the real memory window
ZW	aim the real memory window

FINAL TRIM SIZE : 7.5 in x 9.0 in

Index

FINAL TRIM SIZE : 7.5 in x 9.0 in

B

C

D

F

G

H

M

N

O

Q

R

T

U

V

W

X

Z

FINAL TRIM SIZE : 7.5 in x 9.0 in