HP OpenVMS Delta/XDelta Debugger Manual

Order Number: BA554-90003

July 2006

This manual describes the OpenVMS DELTA and XDELTA debuggers. OpenVMS DELTA is used to debug programs that run in privileged processor mode at interrupt priority level 0. OpenVMS XDELTA is used to debug system software that runs at an elevated interrupt priority level.

Revision/Update Information:	This manual supersedes the HP OpenVMS Delta/XDelta Debugger Manual, OpenVMS I64 Version 8.2 OpenVMS Alpha Version 7.3 OpenVMS VAX Version 7.3
Software Version:	OpenVMS I64 Version 8.3 OpenVMS Alpha Version 8.3

Hewlett-Packard Company
Palo Alto, California

Printed in the US.

ZK4540

The HP OpenVMS documentation set is available on CD-ROM.

This document was prepared using DECdocument, Version 3.3-1b.

Contents

Preface vii

1 Invoking, Exiting, and Setting Breakpoints

1.1 Overview of the DELTA and XDELTA Debuggers 1–1
1.2 Privileges Required for Running DELTA 1–1
1.3 Guidelines for Using XDELTA 1–2
1.4 Restrictions for XDELTA on OpenVMS I64 Systems 1–2
1.5 Invoking DELTA 1–2
1.6 Exiting from DELTA 1–3
1.7 Invoking XDELTA 1–3
1.8 Requesting an Interrupt 1–5
1.8.1 Requesting Interrupts on VAX 1–6
1.8.2 Requesting Interrupts on Alpha 1–7
1.8.3 Requesting Interrupts on I64 1–7
1.9 Accessing the Initial Breakpoint 1–7
1.10 Proceeding from Initial XDELTA Breakpoints 1–8
1.11 Exiting from XDELTA 1–9

2 DELTA and XDELTA Symbols and Expressions

2.1 Symbols Supplied by DELTA and XDELTA 2–1
2.2 Floating Point Register Support 2–4
2.3 Forming Numeric Expressions 2–4

3 Debugging Programs

3.1 Referencing Addresses 3–1
3.1.1 Referencing Addresses (I64 and Alpha Only) 3–2
3.1.2 Referencing Addresses (VAX Only) 3–4
3.2 Referencing Registers 3–6
3.2.1 Referencing Registers (I64 Only) 3–6
3.2.2 Referencing Registers (Alpha Only) 3–7
3.2.3 Referencing Registers (VAX Only) 3–7
3.3 Interpreting the Error Message 3–8
3.4 Debugging Kernel Mode Code Under Certain Conditions 3–8
3.4.1 Setup Required (I64 and Alpha Only) 3–8
3.4.2 Setup Required (VAX Only) 3–9
3.4.3 Accessing XDELTA 3–9
3.5 Debugging an Installed, Protected, Shareable Image 3–9
3.6 Using XDELTA on Multiprocessor Computers 3–10
3.7 Debugging Code When Single-Stepping Fails (Alpha Only) 3–10
3.8 Debugging Code that Does Not Match the Compiler Listings (I64 and Alpha Only) 3–11

4 DELTA/XDELTA Commands

4.1 Command Usage Summary 4–1
[(left angle bracket)—Set Display Mode 4–3
/ (forward slash)—Open Location and Display Contents in Prevailing Width Mode 4–4
! (exclamation mark)—Open Location and Display Contents in Instruction Mode 4–7
" (double quote)—Open Location and Display Contents in ASCII 4–9
' (single quote)—Deposit ASCII String 4–11
= (equal sign)—Display Value of Expression 4–12
\ string\ —Immediate mode text display command (I64 and Alpha Only) 4–13
ESC (Escape key)—Open Location and Display Previous Location 4–14
EXIT—Exit from DELTA Debugging Session 4–15
LINEFEED (Linefeed key or Ctrl/J)—Close Current Location, Open Next Location 4–16
RETURN (Return or Enter key)— Close Current Location 4–18
TAB (Tab key)—Open Location and Display Indirect Location 4–19
;B—Breakpoint 4–20
;C—Force System to Bugcheck and Crash (I64 and Alpha Only) 4–24
;D—Dump (I64 and Alpha Only) 4–25
;E—Execute Command String 4–27
;G—Go 4–29
;H—Video Terminal Display Command (I64 and Alpha Only) 4–30
;I—List Current Main Image and Its Shareable Images (I64 and Alpha Only) 4–31
;L—List Names and Locations of Loaded Executive Images 4–33
;M—Set All Processes Writable 4–36
;P—Proceed from Breakpoint 4–37
;Q—Validate Queue (I64 and Alpha Only) 4–39
;T—Display Interrupt Stack Frame on XDELTA (I64 Only) 4–40
;W—List Name and Location of a Single Loaded Image (I64 and Alpha Only) 4–43
;X—Load Base Register 4–45
O—Step Instruction over Subroutine 4–48
S—Step Instruction 4–51

A Sample DELTA Debug Session on I64

A.1 Listing File for C Example Program A–1
A.1.1 Source Listing for I64 Debugging Example A–2
A.1.2 Machine Code Listing for I64 Debugging Example A–4
A.2 Example DELTA Debugging Session on I64 . . . . . A–7

B Sample DELTA Debug Session on Alpha

B.1 Listing File for C Example Program B–1
B.1.1 Source Listing for Alpha Debugging Example B–2
B.1.2 Machine Code Listing for Alpha Debugging Example B–4
B.2 Example DELTA Debugging Session on Alpha B–8
B.2.1 DELTA Debugging Session Example on Alpha - Part 1 . . . . . B–8
B.2.2 DELTA Debugging Session Example on Alpha - Part 2 . . . . . B–9
B.2.3 DELTA Debugging Session Example on Alpha - Part 3 . . . . . B–11

C Sample DELTA Debug Session on VAX

C.1 Listing Files for Example Program C–1
C.1.1 Source Listing for VAX Debugging Example C–2
C.1.2 Map File for VAX Debugging Example C–3
C.2 Example DELTA Debugging Session on VAX C–4
C.2.1 DELTA Debugging Session Example on VAX - Part 1 . . . . . C–4
C.2.2 DELTA Debugging Session Example on VAX - Part 2 . . . . . C–5
C.2.3 DELTA Debugging Session Example on VAX - Part 3 . . . . . C–6
C.2.4 DELTA Debugging Session Example on VAX - Part 4 . . . . . C–7

Index

Examples

A–1 Listing File for LOG: C Source Code A–2
A–2 Listing File for LOG: Machine Code _MAIN Procedure A–4
A–3 Listing File for LOG: Machine Code MAIN Procedure A–5
A–4 Listing File for LOG: Machine Code PRINT_LINE Procedure A–6
A–5 .MAP File for the Sample Program A–7
A–6 DELTA Debugging Session on I64 - Part 1 . . . . . A–8
A–7 DELTA Debugging Session on I64 - Part 2 . . . . . A–9
B–1 Listing File for LOG: C Source Code B–2
B–2 Listing File for LOG: Machine Code B–4
B–3 .MAP File for the Sample Program B–8
B–4 DELTA Debugging Session on Alpha- Part 1 . . . . . B–8
B–5 DELTA Debugging Session on Alpha - Part 2 . . . . . B–9
B–6 DELTA Debugging Session Example on Alpha - Part 3 . . . . . B–11
C–1 Program for Getting LOGINTIMs C–2
C–2 LOGINTIM Program .Map File C–3
C–3 DELTA Debugging Session Example on VAX - Part 1 . . . . . C–4
C–4 DELTA Debugging Session Example on VAX - Part 2 . . . . . C–5
C–5 DELTA Debugging Session Example on VAX - Part 3 . . . . . C–6
C–6 DELTA Debugging Session Example on VAX - Part 4 . . . . . C–7

Tables

1–1	Boot Command Qualifier Values	1–4
2–1	DELTA/XDELTA Symbols for OpenVMS I64 systems	2–1
2–2	DELTA/XDELTA Symbols for OpenVMS Alpha systems	2–2
2–3	DELTA/XDELTA Symbols for OpenVMS VAX systems	2–3
2–4	Floating Point Register Support by Platform	2–4
2–5	Arithmetic Operators	2–5
3–1	Intel Itanium Registers and their Associated Symbols	3–6
4–1	DELTA/XDELTA Command Summary (All platforms)	4–1
4–2	DELTA/XDELTA Command Summary (I64 and Alpha Only)	4–2
4–3	DELTA/XDELTA Command Summary (I64 Only)	4–2

Preface

Intended Audience

This manual is written for programmers who debug system code for device drivers and other images that execute in privileged processor-access modes or at an elevated interrupt priority level (IPL).

Document Structure

This manual consists of the following chapters and appendixes:

- Chapter 1 provides an overview and descriptions for the DELTA and XDELTA Debuggers and breakpoints.
- Chapter 2 describes the DELTA and XDELTA symbols.
- Chapter 3 describes how to debug programs.
- Chapter 4 describes the DELTA and XDELTA commands.
- Appendix A describes an OpenVMS I64 debugging session using DELTA.
- Appendix B describes an OpenVMS Alpha debugging session using DELTA.
- Appendix C describes an OpenVMS VAX debugging session using DELTA.

Related Documents

This manual refers to several documents that contain the primary descriptions of topics discussed in this manual. The following table lists the topics and those documents.

Topic	Document
Accessing OpenVMS VAX through a lower priority interrupt level	HP OpenVMS System Manager's Manual
Boot command qualifiers for Volume Shadowing	HP Volume Shadowing for OpenVMS
Device name parameters	HP OpenVMS System Manager's Manual
IPRs for OpenVMS Alpha PALcode opcodes for OpenVMS Alpha	Alpha Architecture Reference Manual
Intel® Itanium® hardware architecture and environment	Intel® IA-64 Architecture Software Developer's Manual Volume 1: IA-64 Application Architecture Intel® IA-64 Architecture Software Developer's Manual Volume 2: IA-64 System Architecture

For additional information about HP OpenVMS products and services, visit the following World Wide Web address:

http://www.hp.com/go/openvms

Reader's Comments

HP welcomes your comments on this manual. Please send comments to either of the following addresses:

Internet	**openvmsdoc@hp.com**
Postal Mail	Hewlett-Packard Company OSSG Documentation Group, ZKO3-4/U08 110 Spit Brook Rd. Nashua, NH 03062-2698

How to Order Additional Documentation

For information about how to order additional documentation, visit the following World Wide Web address:

http://www.hp.com/go/openvms/doc/order

Conventions

The following conventions are used in this manual:

I64	Abbreviation representing "HP OpenVMS for Integrity servers".
Ctrl/x	A sequence such as Ctrl/x indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
PF1 x	A sequence such as PF1 x indicates that you must first press and release the key labeled PF1 and then press and release another key or a pointing device button.
Return	In examples, a key name enclosed in a box indicates that you press a key on the keyboard. (In text, a key name is not enclosed in a box.) In the HTML version of this document, this convention appears as brackets, rather than a box.
. . .	A horizontal ellipsis in examples indicates one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
( )	In command format descriptions, parentheses indicate that you must enclose choices in parentheses if you specify more than one.

[ ]	In command format descriptions, brackets indicate optional choices. You can choose one or more items or no items. Do not type the brackets on the command line. However, you must include the brackets in the syntax for OpenVMS directory specifications and for a substring specification in an assignment statement.
|	In command format descriptions, vertical bars separate choices within brackets or braces. Within brackets, the choices are optional; within braces, at least one choice is required. Do not type the vertical bars on the command line.
{}	In command format descriptions, braces indicate required choices; you must choose at least one of the items listed. Do not type the braces on the command line.
bold text	This typeface represents the introduction of a new term. It also represents the name of an argument, an attribute, or a reason.
italic text	Italic text indicates important information, complete titles of manuals, or variables. Variables include information that varies in system output (Internal error number), in command lines (/PRODUCER=name), and in command parameters in text (where dd represents the predefined code for the device type).
UPPERCASE TEXT	Uppercase text indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.
`Monospace text`	Monospace type indicates code examples and interactive screen displays. In the C programming language, monospace type in text identifies the following elements: keywords, the names of independently compiled external functions and files, syntax summaries, and references to variables or identifiers introduced in an example.
-	A hyphen at the end of a command format description, command line, or code line indicates that the command or statement continues on the following line.
numbers	All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

1

Invoking, Exiting, and Setting Breakpoints

This chapter presents an overview of the DELTA and XDELTA debuggers, and provides the following information:

- Privileges required for running DELTA
- Guidelines for using XDELTA
- Invoking and terminating DELTA and XDELTA debugging sessions on OpenVMS systems
- Booting XDELTA, requesting interrupts, and accessing initial breakpoints on OpenVMS systems

1.1 Overview of the DELTA and XDELTA Debuggers

The DELTA and XDELTA debuggers are used to monitor the execution of user programs and the OpenVMS operating system. They use the same commands and the same expressions, but they differ in how they operate. DELTA operates as an exception handler in a process context. XDELTA is invoked directly from the hardware SCB vector in a system context.

Use DELTA to debug process-context programs that execute at interrupt priority level (IPL) 0 in any processor mode. You cannot use DELTA to debug code that executes at an elevated IPL. To debug with DELTA, invoke it from within your process by specifying it as the debugger (as opposed to the symbolic debugger).

Use XDELTA to debug system software executing in any processor mode or at any IPL level. Because XDELTA is not process specific, it is not invoked from a process. To debug with XDELTA, you must boot the processor with commands to include XDELTA in memory. XDELTA's existence terminates when you reboot the processor without XDELTA.

1.2 Privileges Required for Running DELTA

No privileges are required to run DELTA to debug a program that runs in user mode. To debug a program that runs in other processor-access modes, the process in which you run the program must have the necessary privileges.

To use the ;M command, your process must have change-mode-to-kernel (CMKRNL) privilege. The ;M command sets all processes writable.

To use the ;L command (List All Loaded Executive Modules), you must have change-mode-to-executive (CMEXEC) privilege.

1.3 Guidelines for Using XDELTA

Because XDELTA is not process specific, privileges are not required.

When using XDELTA, you must use the console terminal. You should run XDELTA only on a standalone system because all breakpoints are handled at IPL 31.

You cannot redirect output from XDELTA. To determine if your system maintains a log file, check your hardware manual. You can produce a log of console sessions by connecting the console serial port of the system that will boot with XDELTA to the serial port of a LAT server. Then, from another system, use the command SET HOST/LAT/LOG to that LAT port.

1.4 Restrictions for XDELTA on OpenVMS I64 Systems

The following Intel® Itanium® hardware registers are not supported by XDELTA on OpenVMS I64 systems:

- CPUID
- Debug Data Break Registers
- Debug Instruction Break Registers
- Region Registers
- Protection Key Registers
- Instruction Translation Registers
- Data Translation Registers
- Device Interrupt Control Register

1.5 Invoking DELTA

To invoke DELTA, perform the following steps after assembling (or compiling) and linking your program:

1. Define DELTA as the default debugger instead of the symbolic debugger with the following command:

   ```
   $ DEFINE LIB$DEBUG SYS$LIBRARY:DELTA
   ```

2. Use the following RUN command to execute your program:

   ```
   $ RUN/DEBUG MYPROG
   ```

When DELTA begins execution, it displays its name and the first executable instruction in the program with which it is linked. It displays the address of that instruction, a separator—an exclamation point (!) on I64 and Alpha, and a slash (/) on VAX—and the instruction and its operands.

On I64, the name and starting address are displayed as follows:

```
hp OpenVMS Industry Standard 64 DELTA Debugger

Brk 0 at address

address!       instruction    operands
```

On Alpha, the name and starting address are displayed as follows:

```
OpenVMS Alpha DELTA Debugger

Brk 0 at address

address!       instruction    operands
```

On VAX, the name, current version number, and address are displayed as follows:

```
DELTA Version 5.5
address/instruction operands
```

DELTA is then ready for your commands.

You can redirect output from a DELTA debugging session by assigning DBG$DELTA to the I/O device.

Note

The image activator on OpenVMS Alpha systems automatically activates SYS$SHARE:SYS$SSISHR.EXE when an image is debugged using the RUN/DEBUG command or is linked using the /DEBUG qualifier. The presence of this image should not alter your program's correctness, but if your program is sensitive to virtual address layout or if for some reason SYS$SHARE:SYS$SSISHR.EXE is not installed properly on your system, you may want to bypass its automatic activation.

To keep the image activator from activating SYS$SHARE:SYS$SSISHR.EXE for you, define the logical name SSI$AUTO_ACTIVATE to be "OFF" before running the program to be debugged with DELTA.

1.6 Exiting from DELTA

To exit from DELTA, type EXIT and press the Return key. When you are in user mode, you exit DELTA and your process remains. When you are in a privileged access mode, your process can be deleted.

1.7 Invoking XDELTA

To invoke XDELTA, perform the following steps:

1. Boot the system using a console command or a command procedure that includes XDELTA.
2. On VAX, an initial XDELTA breakpoint is taken so that you can set additional breakpoints or examine and change locations in memory. XDELTA displays the following breakpoint message:

   ```
   1 BRK at address
    address/instruction
   ```

Note

Never clear breakpoint 1 from any code being debugged in XDELTA. If you accidentally clear breakpoint 1 and no other breakpoints are set, you cannot use XDELTA until you reboot again with XDELTA.

On I64 and Alpha, two initial XDELTA breakpoints are taken so that you can set additional breakpoints or examine and change locations in memory. XDELTA displays the following message for the first breakpoint:

```
BRK 0 at address
 address!instruction
```

3. On all processors, proceed from the initial breakpoint, using the following command:

```
;P  Return
```

On VAX, the procedure for booting the system with XDELTA differs, depending on the model of your system. Each procedure uses commands that include XDELTA in memory and cause the execution of a breakpoint in OpenVMS initialization routines. Execution of the breakpoint instruction transfers program control to a fault handler located in XDELTA.

Some boot procedures require the use of the /R5 qualifier with the boot command. The /R5 qualifier enters a value for a flag that controls the way XDELTA is loaded. The flag is a 32-bit hexadecimal integer loaded into R5 as input to VMB.EXE, the primary boot program. For a description of the valid values for this flag, see Table 1–1.

Note

When you deposit a boot command qualifier value in R5, make sure that any other values you would normally deposit are included. For example, if you were depositing the number of the system root directory from which you were booting and an XDELTA value, R5 would contain both values.

For directions for booting XDELTA on VAX, see the OpenVMS VAX supplement specific to your computer.

On Alpha, the procedure for booting all Alpha systems with XDELTA is the same. For one example of how to boot XDELTA, use the boot command as follows:

```
>>> BOOT -FLAG 0,6
```

On I64, the procedure for booting with XDELTA is the same. For an example of how to boot XDELTA, use the boot command as follows:

```
fs0:\efi\vms\> vms_loader -fl 0,6
```

On I64 and Alpha, the flag for specifying boot qualifiers is a 64-bit integer that is passed directly as input to the primary boot program; IPB.EXE on I64 and APB.EXE on Alpha. For a description of the valid values for this flag, see Table 1–1.

Table 1–1 Boot Command Qualifier Values

Value	Description
0	Normal, nonstop boot (default)
1	Stop in SYSBOOT

(continued on next page)

Table 1–1 (Cont.) Boot Command Qualifier Values

Value	Description
2	Include XDELTA, but do not take the initial breakpoint
3	Stop in SYSBOOT, include XDELTA, but do not take the initial breakpoint
6	Include XDELTA, and take the initial breakpoint
7	Include XDELTA, stop in SYSBOOT, and take the initial breakpoint at system initialization

1.8 Requesting an Interrupt

If you set the boot control flag to 6, XDELTA will stop at an initial breakpoint during the system boot process. You can then set other breakpoints or examine locations in memory.

Your program can also call the routine INI$BRK, which in turn executes the first XDELTA breakpoint. For the breakpoint procedure, see Section 1.9.

Once loaded into memory, XDELTA can also be invoked at any time from the console by requesting a software interrupt. For example, you might need to use a software interrupt to enter XDELTA if your program is in an infinite loop or no INI$BRK call had been made.

On VAX, INI$BRK is defined as XDELTA's breakpoint 1.

Note

On VAX, never clear breakpoint 1 from any code being debugged in XDELTA. If you accidentally clear breakpoint 1 and no other breakpoints are set, you cannot use XDELTA again until you reboot with XDELTA.

On I64 and Alpha, INI$BRK is defined as XDELTA's breakpoint 0. It is not possible to clear breakpoint 0 from any code being debugged in XDELTA.

1.8.1 Requesting Interrupts on VAX

On VAX 8530, 8550, 8600, 8650, 8810 (8700), 8820, 8820-N (8800), 8830, 8840, VAX-11/780, or VAX-11/785 computers, enter the following commands at the console terminal to request the interrupt:

```
$ Ctrl/P
>>> HALT
>>> D/I 14 E
>>> C
```

For a VAX 9000 computer, enter the following commands at the console terminal to request the interrupt:

```
$ Ctrl/P
>>> HALT/CPU=ALL
>>> D/I 14 E
>>> C/CPU=ALL
```

For a VAX 6000 series, 8200, 8250, 8300, 8350, VAX-11/730, or a VAX-11/750 computer, enter the following commands:

```
$ Ctrl/P
>>> D/I 14 E
>>> C
```

For a VAXstation 3520 or 3540 computer, perform the following steps:

1. Press and release the Halt button on the CPU control panel. When you release the Halt button, make sure it is popped out or the system will remain halted. You can also press the Break key (if enabled) on the console terminal.
2. Enter the following commands:

   ```
   >>> D/I 14 E
   >>> C/ALL
   ```

For a VAXft-3000, VAXft-410, VAXft-610, or VAXft-612 computer, enter the following commands at the console terminal to request the interrupt:

```
$ Break or F5
>>> HALT
>>> D/I 14 E
>>> CONT
>>> PIO
```

For a VAX 7000 or VAX 10000 series computer, enter the following commands at the console terminal to request the interrupt. If you are operating in secure mode, first set the keyswitch to ENABLE before entering these commands.

```
$ Ctrl/P
>>> D IPR:14 E
>>> CONT
```

For a VAXstation 2000, MicroVAX 2000, MicroVAX 3300/3400 series, MicroVAX or VAXstation 3500/3600 series, MicroVAX 3800/3900 series, VAX 4000 series, or MicroVAX II computer, perform the following steps:

1. Press and release the Halt button on the CPU control panel. When you release the Halt button, make sure it is popped out or the system will remain halted. You can also press the Break key (if enabled) on the console terminal.
2. Enter the following commands:

```
>>> D/I 14 E
>>> C
```

For an alternative method of accessing OpenVMS through a lower priority interrupt, see the HP OpenVMS System Manager's Manual.

1.8.2 Requesting Interrupts on Alpha

On Alpha systems, perform the following steps to request an interrupt:

1. Halt the processor with the following command:

```
^P
```

2. Request an IPL 14 software interrupt with the following command:

```
>>> DEP SIRR E
```

   This command deposits a 14_{10} into the software interrupt request register.

3. Reactivate the processor by issuing the CONTINUE command as follows:

```
>>> CONT
```

The process should enter XDELTA as soon as IPL drops to 14.

The following message is displayed:

```
Brk 0 at address
address! instruction
```

At this point, the exception frame is on the stack. The saved PC/PS in the exception frame tells you where you were in the program when you requested the interrupt.

1.8.3 Requesting Interrupts on I64

To request an interrupt on I64, type CTRL/P on the console terminal. Note that XDELTA must have been loaded previously.

When you press CTRL/P, the system is halted at the current PC and at the current IPL. The system must be executing below IPL 8. When the system reaches this state, execution is suspended at the PC that was executing at the time of the interrupt.

1.9 Accessing the Initial Breakpoint

When debugging a program, you can set a breakpoint in the code so that XDELTA gains control of program execution.

To set a breakpoint, place a call to the system routine INI$BRK in the source code.

On systems that are booted with XDELTA, the INI$BRK routine executes a breakpoint instruction. On systems that are not booted with XDELTA, INI$BRK is effectively a NOP instruction.

You can use the INI$BRK routine as a debugging tool, placing calls to this routine in any part of the source code you want to debug.

On VAX, the instruction following the breakpoint is RSB. After the break is taken, the return address (the address in the program to which control returns when you proceed from the breakpoint) is on the top of the stack.

The following command calls the INI$BRK system routine to reach the breakpoint:

```
JSB G^INI$BRK
```

On Alpha, the instruction following the breakpoint is JSR R31,(R26). After the break is taken, the return address (the address in the program to which control returns when you proceed from the breakpoint) is in R26.

On I64, simply step until you reach a br.ret instruction.

The following C routine calls the INI$BRK system routine to reach the breakpoint:

```
extern void ini$brk(void);

main()
{
 ini$brk();
}
```

1.10 Proceeding from Initial XDELTA Breakpoints

On VAX, when XDELTA reaches one of its breakpoints, it displays the following message:

```
1 BRK AT nnnnnnnn
address/instruction operands
```

On I64 and Alpha, when XDELTA reaches one of its breakpoints, it displays the following message:

```
BRK 1 AT nnnnnnnn
address!instruction operands
```

On multiprocessor computers, the XDELTA breakpoint is taken on the processor upon which the XDELTA software interrupt was requested, which is generally the primary processor.

At this point, XDELTA is waiting for input. If you want to proceed with program execution, enter the ;P command. If you want to do step-by-step program execution, enter the S command. If you know where you have set breakpoints, examine them using the ;B command. You can also set additional breakpoints or modify existing ones.

If you entered the ;P command to proceed with program execution and the system halts with a fatal bugcheck, the system prints the bugcheck information on the console terminal. Bugcheck information consists of the following:

- Type of bugcheck
- Contents of the registers
- A dump of one or more stacks
- A list of loaded executive images

The contents of the program counter (PC) and the stack indicate where the failure was detected. Then, if the system parameter BUGREBOOT was set to 0, XDELTA issues a prompt. You can examine the system's state further by entering XDELTA commands.

1.11 Exiting from XDELTA

XDELTA remains in memory with the operating system until you reboot without it.

2

DELTA and XDELTA Symbols and Expressions

This chapter describes how to form the symbolic expressions used as arguments to many DELTA and XDELTA commands.

2.1 Symbols Supplied by DELTA and XDELTA

DELTA and XDELTA define symbols that are useful in forming expressions and referring to registers.

- Table 2–1 shows the symbols that pertain to OpenVMS I64 systems.
- Table 2–2 shows symbols that pertain to OpenVMS Alpha systems.
- Table 2–3 shows symbols that pertain to OpenVMS VAX systems.

Table 2–1 DELTA/XDELTA Symbols for OpenVMS I64 systems

Symbol	Description
.	The address of the current location. The value of this symbol is set by the Open Location and Display Contents (/), Open Location and Display Instruction (!), and the Open Location and Display Indirect (TAB) commands.
ARn	Application register n where n can range from 0 to 127 (decimal). Also see the P(ipr) symbol description.
BRn	Branch register n where n can range from 0 to 7.
CRn	Control register n where n can range from 0 to 127 (decimal). See also the P(ipr) symbol description.
FPn	Floating point register n, where n can range from 0 to 127 (decimal).
FPSR	The floating point status register.
G	^XFFFFFFFF80000000, the prefix for system space addresses.
H	^X7FFE0000, the prefix for addresses in the control region (P1 space). H2E, for example, is equivalent to ^X7FFE002E.

(continued on next page)

Table 2–1 (Cont.) DELTA/XDELTA Symbols for OpenVMS I64 systems

Symbol	Description
P(ipr)	The OpenVMS I64 software implementation of an Alpha internal processor register whose name is ipr. See the Alpha Architecture Reference Manual for the names and descriptions of these processor registers. Not all Alpha internal processor registers are implemented on OpenVMS I64. This syntax is also used to refer to Intel Itanium application and control registers using meaningful names, where ipr is the name of the Intel Itanium register. For example, you can refer to Intel Itanium register CR20 using either of the following: P(IFA) P(CR.IFA) See the Intel® IA-64 Architecture Software Developer's Manual, Volume 2: IA-64 System Architecture manual for the names of the application and control registers.
PC	The OpenVMS I64 software implementation of a program counter register, formed by the union of the IP (instruction bundle pointer) and the slot offset (PSR.ri).
pid:Rn	General register n in the process specified by process ID pid.
PS	The processor status register.
Pn	Predicate register n where n can range from 0 to 63 (decimal).
Q	The last value displayed. The value of Q is set by every command that causes DELTA or XDELTA to display the contents of memory or the value of an expression.
Rn	General register n where n can range from 0 to 127 (decimal).
Xn	Base register n, where n can range from 0 to 15 (decimal). These registers are used for storing values, most often the base addresses of data structures in memory. For XDELTA only, X14 and X15 contain the addresses of two command strings that XDELTA stores in memory. See the Execute Command String (;E) command for more information. For XDELTA only, registers X4 and X5 contain specific addresses. X4 contains the address of the location that contains the PCB address of the current process on the current processor. The address that X4 contains is that of the per-CPU database for the current processor. X5 contains SCH$GL_PCBVEC, the symbolic address of the start of the PCB vector, and the list of PCB slots.

Table 2–2 DELTA/XDELTA Symbols for OpenVMS Alpha systems

Symbol	Description
.	The address of the current location. The value of this symbol is set by the Open Location and Display Contents (/), Open Location and Display Instruction (!), and the Open Location and Display Indirect (TAB) commands.
FPn	Floating point register n, where n can range from 0 to 31 (decimal).
FPCR	The floating point control register.
G	^XFFFFFFFF80000000, the prefix for system space addresses.

(continued on next page)

Table 2–2 (Cont.) DELTA/XDELTA Symbols for OpenVMS Alpha systems

Symbol	Description
H	^X7FFE0000, the prefix for addresses in the control region (P1 space). H2E, for example, is equivalent to ^X7FFE002E.
PC	The program counter register.
pid:PC	The program counter in the process specified by process ID pid.
PS	The processor status register.
Q	The last value displayed. The value of Q is set by every command that causes DELTA or XDELTA to display the contents of memory or the value of an expression.
pid:Rn	General register n in the process specified by process ID pid.
Rn	General register n, where n can range from 0 to 31 (decimal).
Xn	Base register n, where n can range from 0 to 15 (decimal). These registers are used for storing values, most often the base addresses of data structures in memory. For XDELTA only, X14 and X15 contain the addresses of two command strings that XDELTA stores in memory. See the Execute Command String (;E) command for more information. For XDELTA only, registers X4 and X5 contain specific addresses. X4 contains the address of the location that contains the PCB address of the current process on the current processor. The address that X4 contains is that of the per-CPU database for the current processor. X5 contains SCH$GL_PCBVEC, the symbolic address of the start of the PCB vector, and the list of PCB slots.

Table 2–3 DELTA/XDELTA Symbols for OpenVMS VAX systems

Symbol	Description
.	The address of the current location. The value of this symbol is set by the Open Location and Display Contents (/), Open Location and Display Instruction (!), and the Open Location and Display Indirect (TAB) commands.
G	^X80000000, the prefix for system space addresses. G2E, for example, is equivalent to ^X8000002E.
H	^X7FFE0000, the prefix for addresses in the control region (P1 space). H2E, for example, is equivalent to ^X7FFE002E.
Pn	The internal processor register at processor address n, where n can range from 0 to 3F (hexadecimal). See the VAX Architecture Reference Manual for a description of these processor registers.
Q	The last value displayed. The value of Q is set by every command that causes DELTA or XDELTA to display the contents of memory or the value of an expression.
Rn	General register n, where n can range from 0 to F (hexadecimal). RF+4 is the processor status longword (PSL), RE is the stack pointer, and RF is the program counter (PC).

(continued on next page)

Table 2–3 (Cont.) DELTA/XDELTA Symbols for OpenVMS VAX systems

Symbol	Description
Xn	Base register n, where n can range from 0 to F (hexadecimal). These registers are used for storing values, most often the base addresses of data structures in memory. For XDELTA only, XE and XF contain the addresses of two command strings that XDELTA stores in memory. See the Execute Command String (;E) command for more information. For XDELTA only, registers X4 and X5 contain specific addresses. X4 contains the address of the location that contains the PCB address of the current process on the current processor. The address that X4 contains is that of the per-CPU database for the current processor. X5 contains SCH$GL_PCBVEC, the symbolic address of the start of the PCB vector, and the list of PCB slots.

2.2 Floating Point Register Support

On OpenVMS Alpha, floating point registers can be accessed from DELTA and from XDELTA but only if floating point arithmetic is enabled in the current process. On OpenVMS I64, floating point registers FP6 through FP11 are always available. The other floating point registers are available if floating point arithmetic is enabled in the current process.

DELTA runs in the context of a process. On OpenVMS Alpha, access to floating-point registers is enabled as soon as the first floating point instruction in the code being examined is executed. Access is disabled as soon as that image completes execution. On OpenVMS I64, floating-point registers are always available to DELTA.

Table Table 2–4 shows these relationships:

Table 2–4 Floating Point Register Support by Platform

	Alpha	I64
XDELTA	No access	FP6—FP11
DELTA	FPn if FP access is enabled	Always available

When the system enters XDELTA, it may not be obvious which process is the current process. If the current process happens to have floating point enabled (because a floating point instruction has executed and the image containing the floating point instruction is still executing), then you can access the floating point registers. Otherwise, you cannot. XDELTA checks the FEN (floating point enable) IPR (internal processor register) to see whether it needs to provide access to floating point registers.

2.3 Forming Numeric Expressions

Expressions are combinations of numbers, symbols that have numeric values, and arithmetic operators.

On all platforms, DELTA and XDELTA store and display all numbers in hexadecimal. They also interpret all numbers as hexadecimal.

Expressions are formed using regular (infix) notation. Both DELTA and XDELTA ignore operators that trail the expression. The following is a typical expression (in hexadecimal):

```
G4A32+24
```

DELTA and XDELTA evaluate expressions from left to right. No operator takes precedence over any other.

DELTA and XDELTA recognize five binary arithmetic operators, one of which also acts as a unary operator. They are listed in Table 2–5.

Table 2–5 Arithmetic Operators

Operator	Action
+ or SPACE	Addition
-	Subtraction when used as a binary operator, or negation when used as a unary operator
*	Multiplication
%	Division
@	Arithmetic shift

The following example shows the arguments required by the arithmetic-shift operator:

n@j

In this example, n is the number to be shifted, and j is the number of bits to shift it. If j is positive, n is shifted to the left; if j is negative, n is shifted to the right. Argument j must be less than 20_{16} and greater than -20_{16}. Bits shifted beyond the limit of the longword are lost; therefore, the result must fit into a longword.

Note

Do not enter unnecessary spaces, as DELTA/XDELTA treats the space as an additional operator.

3

Debugging Programs

When you use DELTA or XDELTA, there are no prompts, few symbols, and one error message. You move through program code by referring directly to address locations. This chapter provides directions for the following actions:

- Referencing addresses
- Referencing registers, the PSL or PS, and the stack
- Interpreting the error message
- Debugging kernel mode code under certain conditions
- Debugging an installed, protected, shareable image
- Using XDELTA on multiprocessor computers
- Debugging code when single-stepping fails (Alpha only)
- Debugging code that does not match the compiler listings (I64 and Alpha only)

For examples of DELTA debugging sessions on various OpenVMS platforms, see Appendix A for I64, Appendix B for Alpha, and Appendix C for VAX.

3.1 Referencing Addresses

When using DELTA or XDELTA to debug programs, you move through the code by referring to addresses. To help you identify address locations within your program, use a list file and a map file. The list file (.LIS) lists each instruction and its offset value from the base address of the program section. The full map file (.MAP) lists the base addresses for each section of your program. To determine the base address of a device driver program, see the OpenVMS VAX Device Support Manual[1].

Once you have the base addresses of the program sections, locate the instruction in the list file where you want to start the debugging work. Add the offset from the list program to the base address from the map file. Remember that all calculations of address locations are done in hexadecimal. You can use DELTA/XDELTA to do the calculations for you with the =command.

To make address referencing easier, you can use offsets to a base address. Then you do not have to calculate all address locations. First, place the base address into a base register. Then move to a location using the offset to the base address stored in the register.

Whenever DELTA/XDELTA displays an address, it will display a relative address if the offset falls within the permitted range (see the ;X command in Chapter 4).

[1] This manual has been archived but is available on the OpenVMS Docmentation CD-ROM.

3.1.1 Referencing Addresses (I64 and Alpha Only)

On I64 and Alpha, to reference addresses during a DELTA debug session, use the following OpenVMS Alpha example as a guide. The example uses a simple C program (HELLO.C). You can also use the same commands in an XDELTA debug session.

```
#include <stdio.h>

main()
{
 printf("Hello world\n");
}
```

The following procedure generates information to assist you with the address referencing:

1. Use the /LIST and /MACHINE_CODE qualifiers to compile the program and generate the list file containing the Alpha machine instructions.

   To generate the list file for the previous example, use the following command:

   ```
   $ cc/list/machine_code hello
   ```

   The compiler will generate the following Alpha code in the machine code portion of the listing file:

   ```
      .PSECT $CODE, OCTA, PIC, CON, REL, LCL, SHR,-
       EXE, NORD, NOWRT
   0000    main::                                                  ; 000335
   0000            LDA     SP, -32(SP)             ; SP, -32(SP)
   0004            LDA     R16, 48(R27)            ; R16, 48(R27)      ; 000337
   0008            STQ     R27, (SP)               ; R27, (SP)         ; 000335
   000C            MOV     1, R25                  ; 1, R25            ; 000337
   0010            STQ     R26, 8(SP)              ; R26, 8(SP)        ; 000335
   0014            STQ     FP, 16(SP)              ; FP, 16(SP)
   0018            LDQ     R26, 32(R27)            ; R26, 32(R27)      ; 000337
   001C            MOV     SP, FP                  ; SP, FP            ; 000335
   0020            LDQ     R27, 40(R27)            ; R27, 40(R27)      ; 000337
   0024            JSR     R26, DECC$GPRINTF       ; R26, R26
   0028            MOV     FP, SP                  ; FP, SP            ; 000338
   002C            LDQ     R28, 8(FP)              ; R28, 8(FP)
   0030            LDQ     FP, 16(FP)              ; FP, 16(FP)
   0034            MOV     1, R0                   ; 1, R0
   0038            LDA     SP, 32(SP)              ; SP, 32(SP)
   003C            RET     R28                     ; R28
   ```

   Notice the statement numbers on the far right of some of the lines. These numbers correspond to the source line statement numbers from the listing file as shown next:

   ```
   335 main()
   336 {
   337     printf("Hello world\n");
   338 }
   ```

2. Use the /MAP qualifier with the link command to generate the full map file (.MAP file). To produce a debuggable image, make sure that either /DEBUG or /TRACEBACK (the default) is also specified with the link command.

   To generate the map file for the example program, use the following command:

   ```
   $ LINK/MAP/FULL HELLO
   ```

3. See the Program Section Synopsis of the map file. Locate the code section that you want to debug and its base address.

For the example program, the map file is HELLO.MAP. A portion of the Program Section Synopsis is shown below. The $CODE section of the program has a base address of 20000.

```
  +--------------------------+
  ! Program Section Synopsis !
  +--------------------------+

Psect Name      Module Name       Base     End          Length
----------      -----------       ----     ---          ------
$LINKAGE                        00010000 0001007F 00000080 (       128.)
                HELLO           00010000 0001007F 00000080 (       128.)
$CODE                           00020000 000200BB 000000BC (       188.)
                HELLO           00020000 000200BB 000000BC (       188.)
```

4. See the list file for the location where you want to start debugging. First find the source line statement number. Next find that statement number in the machine code listing portion of the list file. This is the specific instruction where you want to start debugging.

   For the example program, source statement 337 is the following:

   ```
   printf("Hello world\n");
   ```

   Search the machine code listing for statement 337. The first occurrence is the instruction at offset 4 from the start of "main::" and the base of the $CODE PSECT.

5. Enable DELTA using the following commands:

   ```
   $ DEFINE LIB$DEBUG SYS$LIBRARY:DELTA
   $ RUN/DEBUG HELLO
   ```

6. If you want to store the base address in a base register, use the ;X command to load the base register.

   For the example program, use the following DELTA/XDELTA command to store the base address of 20000 in base register 0.

   ```
   20000,0;X
   ```

7. Now you can move to specific address locations.

   For example, if you want to place a breakpoint at offset 4, you would calculate the address as 20000 (base address) plus 4 (offset), or 20004, and specify the ;B command as follows:

   ```
   20004;B
   ```

   Alternatively, if you stored the base address in the base register, you could use the address expression X0+4 (or "X0 4", where the + sign is implied) to set the breakpoint as follows:

   ```
   X0+4;B
   ```

Reverse this technique to find an instruction displayed by DELTA/XDELTA in the .LIS file, as follows:

1. Note the address of the instruction you want to locate in the .LIS file.

   For example, DELTA/XDELTA displays the following instruction at address 20020:

   ```
   20020!  LDQ             R27,#X0028(R27)
   ```

   The following steps allow you to find this instruction in the .LIS file.

2. See the .MAP file, and identify the psect and module where the address of the instruction is located. Check the base address value and the end address value of each psect and module. When the instruction address is between the base and end address values, record the psect and module names.

   In the example, the instruction address is located in the HELLO module ($CODE PSECT). The address, 20020, is between the base address 20000 and the end address 200BB.

3. Subtract the base address from the instruction address. Remember that all calculations are in hexadecimal and that you can use the DELTA/XDELTA = command to do the calculations. The result is the offset.

   For example, subtract the base address of 20000 from the instruction address 20020. The offset is 20.

4. See the .LIS file. Look up the module and then find the correct psect. Look for the offset value you calculated in the previous step.

   In the example, there are two psects and one module but only one $CODE psect. Look up the instruction at offset 20, and you will find the following in the .LIS file:

```
0020        LDQ R27, 40(R27)            ; R27, 40(R27)      ; 000337
```

3.1.2 Referencing Addresses (VAX Only)

On VAX, to reference addresses during a DELTA debug session, use the following example as a guide. The example uses a simple VAX MACRO program (EXAMPLE.MAR). You can also use the same commands in an XDELTA debugging session.

```
0000    1   .title  example
0000    2
0000    3   .entry  start   ^M<r3,r4>
0002    4           clrl    r3
0004    5           movl    #5,r4
0007    6   10$:    addl    r4,r3
000A    7           sobgtr  r4,10$
000D    8           ret
000E    9
000E   10   .end    start
```

The following procedure generates information to assist you with address referencing:

1. Use the /LIST qualifier to assemble the program and generate the list file.

   To generate the list file for the previous example, use the following command:

```
$ MACRO/LIST EXAMPLE
```

2. Use the /MAP qualifier with the link command to generate the full map file (.MAP file). Make sure that the default /DEBUG or /TRACEBACK qualifier is active for your link command. If not, specify /DEBUG or /TRACEBACK along with the /MAP qualifier.

   To generate the map file for the example program, use the following command:

```
$ LINK/MAP EXAMPLE
```

3. See the Program Section Synopsis of the map file, locate the section that you want to debug, and look up the base address.

For the example program, the map file is EXAMPLE.MAP. A portion of the Program Section Synopsis is shown below. The first section of the program has a base address of 200.

```
                +--------------------------+
                ! Program Section Synopsis !
                +--------------------------+

Psect Name      Module Name   Base     End          Length
----------      -----------   ----     ---          ------

. BLANK .                   00000200 0000020D 0000000E (         14.)
                EXAMPLE     00000200 0000020D 0000000E (         14.)
```

4. See the list file for the location of the specific instruction where you want to start debugging.

   For the example program, start with the second instruction (MOVL #5,R4) with an offset of 4.

5. Enable DELTA using the following commands:

   ```
   $ DEFINE LIB$DEBUG SYS$LIBRARY:DELTA
   $ RUN/DEBUG EXAMPLE
   ```

6. If you want to store the base address in a base register, use the ;X command to load the base register.

   For the example program, use the following DELTA/XDELTA command to store the base address 200 in base register 0.

   ```
   200,0;X Return
   ```

7. Now you can move to specific address locations.

   For example, if you want to place a breakpoint at the second instruction (MOVL #5,R4), you would calculate the address as 200 (base address) plus 4 (offset), or 204, and specify the ;B command as follows:

   ```
   204;B Return
   ```

   Alternatively, if you stored the base address in the base register, you could use the address expression X0+4 (or "X0 4", where the + sign is implied), as follows:

   ```
   X0+4;B Return
   ```

Reverse this technique to find an instruction displayed by DELTA/XDELTA in the .LIS file, as follows:

1. Note the address of the instruction you want to locate in the .LIS file.

   For example, DELTA/XDELTA displays the following instruction at address 020A:

   ```
   20A!  sobgtr  r4,00000207
   ```

   The following steps allow you to find the instruction at location 207:

2. See the .MAP file and identify the PSECT and MODULE where the address of the instruction is located. Check the base address value and the end address value of each PSECT and MODULE. When the instruction address is between the base and end address values, record the PSECT and MODULE names.

In the example, the instruction address is located in the EXAMPLE module (.BLANK. psect). The address instruction, 207, is between the base address 200 and the end address 20D.

3. Subtract the base address from the instruction address. Remember that all calculations are in hexadecimal and that you can use the DELTA/XDELTA = command to do the calculations. The result is the offset.

   For the example, subtract the base address 200 from the instruction address 207. The offset is 7.

4. See the .LIS file. Look up the MODULE and then find the correct PSECT. Look for the offset value you calculated in the previous step.

   In the example, there is only one PSECT and MODULE. Look up the instruction at offset 7. The program is branching to the following instruction:

```
10$:   addl r4,r3
```

3.2 Referencing Registers

When using DELTA or XDELTA to debug programs, you can view the contents of registers. The following sections describe the types of registers that are referenced by each OpenVMS platform.

3.2.1 Referencing Registers (I64 Only)

On I64, you can reference the following kinds of registers: integer, floating, application, branch, control, special purpose, and software equivilents of special OpenVMS symbolic locations.

Table 3–1 lists the Intel Itanium registers and symbols by which they are identified.

Table 3–1 Intel Itanium Registers and their Associated Symbols

Register	Symbol
General	R0 through R127
Floating	FP0 through FP127
Branch	BR0 through BR7
Predicate	P0 through P63
Application	AR16 (RSC), AR17 (BSP), AR18 (BSPSTORE), AR19 (RNAT), AR25 (CSD), AR26 (SSD), AR32 (CCV), AR36 (UNAT), AR64 (PFS), AR65 (LC), AR66 (EC)
Control	CR0 (DCR), CR1 (ITM), CR2 (IVA), CR8 (PTA), CR16 (IPSR), CR17 (ISR), CR19 (IIP), CR20 (IFA), CR21 (ITIR), CR22 (IIPA), CR23 (IFS), CR24 (IIM), CR25 (IHA),CR65 (IVR)

In addition, there is a program counter (PC) register, which is obtained from the hardware IP register and the `ri` field of the PSR register.

3.2.2 Referencing Registers (Alpha Only)

On Alpha, to view the contents of the 32 integer registers, the program counter (PC), the stack pointer (SP), the processor status (PS), the 32 floating point registers, the floating point control register (FPCR), and the internal processor registers (IPRs), use the same DELTA/XDELTA commands that you use to view the contents of any memory location. These commands include /, LINEFEED, and ESC. The symbols for identifying these registers follow:

- Integer registers are referenced by the symbol R and a decimal number from 0 to 31. For example, register 1_{10} is $R1_{10}$ and register 10_{10} is $R10_{10}$. (Decimal notation differs from the original implementation on VAX which uses hexadecimal notation.)
- PC is referenced symbolically by PC.
- PS is referenced symbolically by PS.
- FP is referenced by R29.
- SP is referenced by R30.
- Floating point registers are referenced by FP and a decimal number from 0 to 31. For example, floating point register 1_{10} is $FP1_{10}$ and floating point register 10_{10} is $FP10_{10}$.
- FPCR is treated like any other floating point register except, to explicitly open it, you specify FPCR/.
- Internal processor registers (IPRs) are accessed symbolically, for example, P(ASTEN). For IPR names, see the Alpha Architecture Reference Manual.

Floating point registers can be accessed from DELTA and from XDELTA but only if floating point arithmetic is enabled in the current process.

DELTA runs in the context of a process. Access to floating point registers is enabled as soon as the first floating point instruction in the code being examined is executed. Access is disabled as soon as that image completes execution.

When the system enters XDELTA, some process is the current process, and that current process may not be obvious. If that process happens to have floating point enabled at the time (because a floating point instruction had executed and the image containing the floating point instruction was still executing), then you can access the floating point registers. Otherwise, you cannot. XDELTA checks the FEN (floating point enable) IPR (internal processor register) to see if it needs to provide access to floating point registers.

3.2.3 Referencing Registers (VAX Only)

On VAX, to view the contents of the 16 general registers (including the program counter and the stack pointer) and the processor status longword (PSL), use the same DELTA/XDELTA commands as you use to view the contents of any memory location (for example, the /, LINEFEED, and the ESC commands). The symbols used to identify the locations of the registers and PSL are as follows:

- The general registers are referred to by the symbol R and a hexadecimal number from 0_{16} to F_{16} representing the number of the register. For example, general register 1_{10} is $R1_{16}$ and general register 10_{10} is RA_{16}. The stack pointer is located in general register 14_{10}, RE_{16}. The program counter is in general register 15_{10}, RF_{16}.

- Upon entry to DELTA or XDELTA, the PSL is stored in the longword directly following the longword representing general register F_{16}. Reference it by using the general register F_{16} symbol plus a longword (RF+4).

3.3 Interpreting the Error Message

When you make an error entering a command in DELTA or XDELTA, you get the "Eh?" error message. This is the only error message generated by DELTA and XDELTA. It is displayed under the following circumstances:

- You entered characters that DELTA/XDELTA does not recognize
- You entered a command incorrectly
- You exceeded the limits of the command (for example, trying to set another breakpoint when all breakpoints are used)
- You attempted to display a particular memory address and one or more of the following is true:
  - Location is not a valid memory address
  - You have no privilege to read the address
  - The process to which the read applies does not exist (DELTA only)
- You attempted to change a particular memory address (including setting a breakpoint) and one or more of the following is true:
  - The location is not a valid memory address
  - You have no privilege to write to the address
  - The process to which the write applies does not exist (DELTA only)

On I64, the error message is also displayed if you are unable to step over a subroutine call due to no write access to the address of the next instruction.

On Alpha, the error message is also displayed if you are unable to single-step or proceed due to no write access to the address of the next instruction.

3.4 Debugging Kernel Mode Code Under Certain Conditions

Some programs exist which, while running in process space, change mode to kernel and raise IPL. Typically, this code is debugged with both DELTA and XDELTA. DELTA is used to debug the kernel mode code at IPL zero. XDELTA is used to debug the code at elevated IPL. (DELTA does not work at elevated IPL.)

Before you can debug such code with XDELTA, you must complete some setup work.

3.4.1 Setup Required (I64 and Alpha Only)

On I64 and Alpha, some setup work is required before you can debug kernel mode code that runs in process space at an elevated IPL. Before you access XDELTA, do the following:

1. Ensure that page faults do not occur at elevated IPL by locking into memory (or the working set) the code that runs at elevated IPL.
2. Make the code writable. (By default, code pages are read only.) To do this, modify the code psect attributes in the link options file or set the affected code pages to writable with $SETPRT.

3. Make code pages copy-on-reference (CRF). You can do this when you make the code writable. If you modify the link options file, set the code psect attributes to be WRT, NOSHR. If you use $SETPRT, it automatically makes the pages CRF.

3.4.2 Setup Required (VAX Only)

On VAX, some setup work is required before you can debug kernel mode code that runs in process space at an elevated IPL. Before you access XDELTA, do the following:

1. Ensure that page faults do not occur at elevated IPL by locking into memory (or the working set) the code that runs at elevated IPL.
2. Make the code writable if you plan to do anything more than single-step through your code (such as set breakpoints, step-overs, and so forth). (By default, code pages are read only.) To make the code writable, modify the code psect attributes in the link options file or set the affected code pages to writable with $SETPRT.

3.4.3 Accessing XDELTA

After you set up the code for debugging, you are ready to access XDELTA. The most convenient method is to invoke INI$BRK from the code at elevated IPL. This causes a trap into XDELTA. You can then step out of the INI$BRK routine into the code to be debugged.

3.5 Debugging an Installed, Protected, Shareable Image

Some shareable images, such as user-written system services, must be linked and installed in a way that precludes debugging with DELTA unless you take further steps. Those steps are described in this section.

Typically, a user-written system service is linked and installed in such a way that the code is shared in a read-only global section, the data is copy-on-reference, and the default code psects are read-only and shareable. Such a shareable image is installed with the Install utility using a command like the following:

```
INSTALL> myimage.exe /share/protect/open/header
```

Other qualifiers can also be used.

When installed in this way, the shareable image code is read-only. However, to debug a user-written system service with DELTA, to single-step and to set breakpoints, the code must either be writable or DELTA must be able to change the code page protection to make it writable. Neither is possible when the code resides in a read-only global section.

Therefore, to debug a user-written system service, you must link and install it differently. In linking the image, the code psects must be set to writable and, preferably, to nonshareable (to force the code pages to be copy-on-reference). Multiple processes accessing this code through the global section will each have their own private copy. You can do this in the link options file by adding a line such as the following for each code psect:

```
PSECT=$CODE$,NOSHR,WRT
```

Then, the image must be installed writable with the /WRITE qualifier and without the /RESIDENT qualifier, as follows:

```
INSTALL> myimage.exe /share/protect/open/header/write
```

After you have installed the image in this way, you can use DELTA to set breakpoints in the shareable image code and single-step through it.

3.6 Using XDELTA on Multiprocessor Computers

On multiprocessor computers, only one processor can use XDELTA at a time. If a second processor attempts to enter XDELTA when another processor has already entered it, the second processor waits until the first processor has exited XDELTA. If the processor using XDELTA sets a breakpoint, other processors are aware of the breakpoint. Therefore, when the code with the XDELTA breakpoint is executed on another processor, that processor will enter XDELTA and stop at the specified breakpoint.

On Alpha and VAX systems, XDELTA uses its own system control block (SCB) to direct all interrupt handling to an error handling routine in XDELTA. Therefore, an error encountered by XDELTA does not affect any other processors that share the standard system SCB. On I64 systems, the implementation is different, but the outcome is the same: XDELTA avoids causing errors that could lead to unintended effects to other processors.

On VAX, when a breakpoint is taken by a processor in a multiprocessor environment, the processor's physical identification number is displayed on the XDELTA breakpoint message line as a 2-digit hexadecimal number. The following is an example of a breakpoint message in a multiprocessor environment:

```
1 BRK AT 00000400 ON CPU 03

00000400/movl #5,r4
```

On I64 and Alpha, the processor's physical identification number is similarly displayed but the number is decimal instead of hexadecimal with no leading zeros. For example:

```
BRK 1 AT 20000 ON CPU 2

20000! LDL R1,(R2)
```

3.7 Debugging Code When Single-Stepping Fails (Alpha Only)

On Alpha, the use of the S command to single-step occasionally fails and the error message Eh? is displayed. This can happen either when you are single-stepping through code or when you have stopped at a breakpoint. In each case, it fails because XDELTA does not have write access to the next instruction. Directions on how to continue debugging for both cases follow:

- You are single-stepping through your code and your single-step fails.

  You can set other breakpoints and proceed with the ;P command. If this occurs at a JSR or BSR instruction, you can first use the O command and then either single-step (with the S command) or proceed (with the ;P command).

- You have stopped at a breakpoint and your attempt to single-step fails.

  You can delete the breakpoint and then proceed with the ;P command. If this occurs at a JSR or BSR instruction, it may be possible to first use the O command and then either single-step (with the S command) or proceed (with the ;P command).

3.8 Debugging Code that Does Not Match the Compiler Listings (I64 and Alpha Only)

There are two cases when the code in your image does not exactly match your compiler listings. As long as you understand why these differences exist, they should not interfere with your debugging. The explanations follow:

- The compilers generate listings with mnemonics that replace some of the Alpha assembly language instructions. This makes the listings easier to read but can initially cause confusion because the code does not exactly match the code in your image. In every case, there is a 1–to–1 correlation between the line of code in your image and the line of code in your listing.
- In certain situations, the linker can modify the instructions in your image so that they do not exactly match your compiler listings. On Alpha, for example, the linker can replace JSR instructions and the call setup to use a BSR instruction for better performance. On I64, the linker sometimes generates code and performs jumps and calls.

4

DELTA/XDELTA Commands

This chapter describes how to use each DELTA and XDELTA command to debug a program. It also describes which commands are used only with DELTA. Table 4–1 provides a summary of the DELTA/XDELTA commands that are common to OpenVMS I64, Alpha, and VAX systems. Table 4–2 provides a summary of the DELTA/XDELTA commands that are available only on OpenVMS I64 and Alpha. Table 4–3 provides a summary of the DELTA/XDELTA commands that are available only on OpenVMS I64.

Many commands in this chapter include an example. The program used for all the examples, except those illustrating commands available only on OpenVMS I64 and Alpha, is listed in Appendix C.

4.1 Command Usage Summary

DELTA and XDELTA use the same commands, with the following exceptions:

- Only DELTA uses the EXIT and ;M commands and arguments that specify a process identification.
- XDELTA defines some base registers that DELTA does not (see Chapter 2).
- On I64 and Alpha, only DELTA uses the ;I command.

For all OpenVMS platforms, differences are noted in command descriptions.

Enter the LINEFEED, ESC, TAB, and RETURN commands by pressing the corresponding key.

Table 4–1 DELTA/XDELTA Command Summary (All platforms)

Command	Description
[	Set Display Mode
/	Open Location and Display Contents in Prevailing Width Mode
!	Open Location and Display Contents in Instruction Mode
LINEFEED	Close Current Location, Open Next
ESC	Open Location and Display Previous Location
TAB	Open Location and Display Indirect Location
"	Open Location and Display Contents in ASCII Mode
RETURN	Close Current Location
;B	Breakpoint
;P	Proceed from Breakpoint

(continued on next page)

Table 4–1 (Cont.) DELTA/XDELTA Command Summary (All platforms)

Command	Description
;G	Go
S	Step Instruction
O	Step Instruction over Subroutine
;D 'string'	Deposit ASCII String
;E	Execute Command String
;X	Load Base Register
=	Display Value of Expression
[1];M	Set All Processes Writable (available only on DELTA)
[2];M	Set All Processes Writable; also, set selected registers of other processes writable (available only on DELTA)
;L	Lists Names and Locations of Loaded Executive Images
EXIT	Exit from DELTA debugging session

[1]VAX specific
[2]I64 and Alpha specific

The commands in Table 4–2 are available only on OpenVMS I64 and Alpha.

Table 4–2 DELTA/XDELTA Command Summary (I64 and Alpha Only)

Command	Description
;D	Dumps a region of memory
;Q	Validate queue
;C	Force system to bugcheck and crash
;W	Locate and display the executive image that contains the specified address
;I	Locate and display information about the current main image that contains the specified address; also display information about all shareable images activated by the current main image (available only on DELTA)
;H	Display on video terminal or at hardcopy terminal
\ string\	Display the ASCII text string enclosed in backslashes

The commands in Table 4–3 are available only on OpenVMS I64.

Table 4–3 DELTA/XDELTA Command Summary (I64 Only)

Command	Description
;T	Display the address of the interrupt stack frame.

[(left angle bracket)—Set Display Mode

Sets the width mode of displays produced by DELTA/XDELTA commands.

Format

```
[ mode
```

Argument

mode
Specifies the display mode as follows:

Mode	Meaning
B	Byte mode. Subsequent open and display location commands display the contents of one byte of memory.
L	Longword mode. Subsequent open and display location commands display the contents of a longword of memory. This is the default mode.
W	Word mode. Subsequent open and display location commands display the contents of one word of memory.

On I64 and Alpha, the following modes are also available.

Mode	Meaning
A	Address display of 32-bit/64-bit. Subsequent address displays will be 64 bits.
Q	Quadword mode. Subsequent open and display location commands display the contents of a quadword of memory.

Description

The Set Display Mode command changes the prevailing display width to byte, word, longword, or quadword. The default display width is longword on Alpha and VAX, quadword on I64. The display mode remains in effect until you enter another Set Display Mode command.

Example

```
R0/   00000001    ❶
[B                ❷
R0/   01          ❸
```

❶ Contents of general register 0 (R0) are displayed using the / command. The display is the default mode, longword.

❷ Display mode is changed to byte mode using the [B command.

❸ Contents of R0 are displayed in byte mode. The least significant byte is displayed.

/ (forward slash)—Open Location and Display Contents in Prevailing Width Mode

Opens a location and displays its contents in the prevailing display mode.

Format

[pid:][start-addr-exp][,end-addr-exp]/ current-contents [new-exp]

Arguments

pid
The internal process identification (PID) of a process you want to access. If you specify zero or do not specify a PID, the default process is the current process. This argument cannot be used with XDELTA.

If you use the **pid** argument, every time you use this command during the debugging session the contents of the same process are displayed, unless you specify a different **pid**.

You can obtain the internal PID of processes by running the System Dump Analyzer utility (SDA). Use the SDA command SHOW SUMMARY to determine the external PID. Then use the SDA command SHOW PROCESS/INDEX to determine the internal PID. For more information about using SDA commands, see your operating system's System Dump Analyzer Utility Manual.

Note

The register examples in the descriptions of **start-addr-exp** and **end-addr-exp** apply to both VAX and Alpha registers. (Alpha register numbers are displayed in decimal, and VAX register numbers are displayed in hexadecimal.)

start-addr-exp
The address of the location to be opened, or the start of a range of addresses to be opened. If not specified, the address displayed is that currently specified by the symbol Q (last quantity displayed). Use the following syntax to display a single address location:

start-addr-exp/

You can also specify a register for this parameter. For example, if you want to view the contents of general register 3 (R3), enter the following DELTA/XDELTA command:

```
R3/
```

end-addr-exp
The address of the last location to be opened. Use the following syntax to display a range of address locations:

start-addr-exp,end-addr-exp/

You can also specify a range of registers. For example, if you want to view the contents of general registers 3 through 5, enter the following DELTA/XDELTA command:

```
R3,R5/
```

If you specify an address expression for **end-addr-exp** that is less than **start-addr-exp**, DELTA/XDELTA displays the contents of **start-addr-exp** only.

current-contents
You do not specify this parameter. It is a hexadecimal value, displayed by DELTA/XDELTA, of the contents of the location (or range of locations) you specified with the **pid** argument and the address expression. It is displayed in the prevailing width display mode.

new-exp
An expression, the value of which is deposited into the location just displayed. If you specify **new-exp** after a range of locations, the new value is placed only in the last location (specified by **end-addr-exp**).

When you specify **new-exp**, terminate the command by pressing the Return key.

If you want to deposit a new value into a location in another process (that is, you specified a PID other than the current process), you must have already set the target process to be writable using the ;M command.

If the value you deposit is longer than the last location where it will be deposited, the new value overwrites subsequent locations. For example, the values at address locations 202 and 204 are as follows:

```
202/ 05D053D4
204/ C05405D0
```

If you deposited the value FFFFFFFFF at address 202, the overflow value would overwrite the value stored at address location 204, as follows:

```
202/ 05D053D4  FFFFFFFFF Return
204/ C054FFFF
```

Description

The Open Location and Display Contents command opens the location or range of locations at **start-addr-exp** and displays **current-contents**, the contents of that location, in hexadecimal format. You can place a new value in the location by specifying an expression. A new value overwrites the last value displayed.

To display a range of locations, give the **start-addr-exp** argument as the first address in the range, followed by a comma, followed by the last address in the range (the **end-addr-exp** argument). For example, if you want to display all locations from 402 to 4F0, the command is as follows:

```
402,4F0/
```

This command changes the current address (. symbol) to the contents of the opened location. A subsequent Close Location command does not change the current address. However, a subsequent Close Current Location and Open Next command (ESC or LINEFEED) executes as follows:

- Writes any **new-exp** specified
- Closes the location opened by the / command

- Adds the number of bytes (defined by the prevailing display width mode) to the address just opened with the / command
- Changes the current address to the new value
- Opens the new location and displays the contents

The display mode remains hexadecimal until the next Open Location and Display Contents in Instruction Mode (!) command or Open Location and Display Contents in ASCII Mode (") command.

In DELTA, not XDELTA, processes having the CMKRNL privilege can examine the address space of any existing process. Use **pid** to specify the internal PID of the process you want to examine. For example, use the following command to view address location 402 in the process with a PID of 00010010:

```
00010010:402/
```

On I64 and Alpha, DELTA also permits the examination of general purpose registers in another process. The PID specifies the internal PID of the process you want to examine. For example, use the following command to examine R5 in the process with a PID of 00010010:

```
00010010:R5/
```

Example

```
R0,R9/00000001
R1/00000000
R2/00000226
R3/7FF2AD94
R4/000019B4
R5/00000000
R6/7FF2AA49
R7/8001E4DD
R8/7FFED052
R9/7FFED25A
```

Contents of all the general registers R0 through R9 are displayed.

! (exclamation mark)—Open Location and Display Contents in Instruction Mode

Displays an instruction and its operands.

Format

[pid:][start-addr-exp][,end-addr-exp] **!**

Arguments

pid
The internal process identification (PID) of a process you want to access. If you specify zero, or do not specify any PID, the default process is the current process. This argument cannot be used with XDELTA.

Subsequent open location and display contents commands, issued after using the **pid** argument, display the contents of the location of the specified process until you specify another PID with this command.

You can obtain the internal PID of processes by running the System Dump Analyzer utility (SDA). Use the SDA command SHOW SUMMARY to determine the external PID. Then use the SDA command SHOW PROCESS/INDEX to determine the internal PID. For more information about SDA commands, see your operating system's System Dump Analyzer Utility Manual.

start-addr-exp
The address of the instruction, or the first address of the range of instructions, to display. If you do not specify this parameter, the address displayed is that currently specified by Q (last quantity displayed). When you want to view just one location, the syntax is as follows:

start-addr-exp!

end-addr-exp
The address of the last instruction in the range to display. When you want to view several instructions, the syntax is as follows:

start-addr-exp,end-addr-exp!

Each location within the range is displayed with the address, a slash (/), and the machine instruction.

Description

The Open Location and Display Contents in Instruction Mode command displays the contents of a location or range of locations as a machine instruction. DELTA/XDELTA does not make any distinction between reasonable and unreasonable instructions or instruction streams.

This command does not allow you to modify the contents of the location. The command sets a flag that causes subsequent Close Current Location and Display Next (LINEFEED) and Open Location and Display Indirect Location (TAB) commands to display MACRO instructions. You can clear the flag by using the Open Location and Display Contents (/) command, which displays the contents of the location as a hexadecimal number, or Open Location and Display Contents in ASCII Mode ("), which displays the contents of the location in ASCII.

When an address appears as an instruction's operand, DELTA/XDELTA sets the Q symbol to that address. Then enter ! again to go to the address specified in the instruction operand. DELTA/XDELTA changes Q only for operands that use program-counter or branch-displacement addressing modes; Q is not altered for operands that use literal and register addressing modes. This feature is useful for branches that follow.

The following examples illustrate the command on each OpenVMS platform.

Examples

I64 example:

```
G0BF5D60!       add         r33 = 0008, r33 ;;      ❶
80BF5D62!       nop.i       000000 ;;               ❷
80BF5D70!       ld4         r2 = [r2] ;;
80BF5D71!       nop.m       000000
80BF5D72!       sxt4        r2 = r2 ;;
80BF5D80!       cmp.eq      p14, p0 = r2, r0
80BF5D81!       nop.f       000000
80BF5D82! (p14) br.cond.dpnt.few.clr 0000030 ;;
80BF5D90!       ld8         r14 = [r2], 008 ;;
80BF5D91!       nop.m       000000
80BF5D92!       mov         b7 = r14 ;;
80BF5DA0!       ld8         r1 = [r2]
```

❶ The instruction at the base address G0BF5D60 is displayed using the ! command. XDELTA displays an add instruction.

❷ After typing a LINEFEED command, XDELTA displays the next instruction location and the instruction at that address, and so on.

Alpha example:

```
30000!       LDA         SP,#XFFE0(SP) ❶
00030004!     BIS         R31,R31,R18 ❷
```

❶ The instruction at address 30000 is displayed using the ! command. DELTA/XDELTA displays a LDA instruction. Note that unlike on a VAX computer, an absolute address never appears in an instruction operand. So the value of Q has no use after an instruction display.

❷ After typing a LINEFEED command, DELTA/XDELTA displays the next instruction location and the instruction at that address.

VAX example:

```
69B!BRB     0000067A ❶
!CLRQ      -(SP)     ❷
```

❶ The instruction at address 69B is displayed using the ! command. DELTA/XDELTA displays a branch instruction and sets Q (last address location displayed) to the branch address 67A.

❷ The instruction at address 67A is displayed using the ! command. The value of Q is used as the address location.

" (double quote)—Open Location and Display Contents in ASCII

Displays the contents of a location as an ASCII string.

Format

[pid:] start-addr-exp[,end-addr-exp] "

Arguments

pid
The internal process identification (PID) of a process you want to access. If you specify zero, or do not specify any PID, the default process is the current process. This argument cannot be used with XDELTA.

Subsequent open location and display contents commands issued after using the **pid** argument, display the contents of the location of the specified process until you specify another PID with this command.

You can obtain the internal PID of processes by running the System Dump Analyzer utility (SDA). Use the SDA command SHOW SUMMARY to determine the external PID. Then use the SDA command SHOW PROCESS/INDEX to determine the internal PID. For more information about SDA commands, see your operating system's System Dump Analyzer Utility Manual.

start-addr-exp
The address of the location, or the start of a range of locations, to be displayed. If you want to view one location, the syntax is as follows:

start-add-exp"

end-addr-exp
The last address within a range of locations to be viewed. If you want to view a series of locations, the syntax is as follows:

start-add-exp,end-addr-exp"

Description

The Open Location and Display Contents in ASCII command opens the location or range of locations at **start-addr-exp** and displays the contents in ASCII format. This command does not change the width of the display (byte, word, longword) from the prevailing mode. If the prevailing mode is word mode, two ASCII characters are displayed; if byte mode, one character is displayed.

The display mode remains ASCII until you enter the next Open Location and Display Contents command (/) or Open Location and Display Contents in Instruction Mode command (!). These commands change the display mode to hexadecimal or instruction, respectively.

You can modify the contents of the locations, starting at **start-addr-exp**, with the Deposit ASCII string (') command.

Example

```
235FC2 [W/415A    ❶
235FC2" ZA        ❷    Linefeed  ❸
235FC4/PP
```

❶ The current display mode is word (displays one word in hexadecimal).

❷ The " command changes the prevailing display mode to ASCII but does not affect the width of the display.

❸ The next Close Current Location, Open Next command (LINEFEED), determines the address of the location to open by adding the width, in bytes, to the value contained in the symbol . (the current address). Then it opens the number of bytes equal to the width of the prevailing display mode, which in this example is two bytes.

The ASCII representation of the contents of the location presents the bytes left to right, while the hexadecimal representation presents them right to left.

' (single quote)—Deposit ASCII String

Deposits the ASCII string at the current address.

Format

'string'

Arguments

string
The string of characters to be deposited.

Description

The Deposit ASCII String command deposits **string** at the current location (.) in ASCII format. The second apostrophe is required to terminate the string. All characters typed between the first and second apostrophes are entered as ASCII character text. Avoid embedding an apostrophe (') within the string you want to deposit.

When you want to use key commands (LINEFEED, RETURN, ESC, or TAB), press the key. These commands are entered as text.

This command stores the characters in 8-bit bytes and increments the current address (.) by one for each character stored.

This command does not change the prevailing display mode.

Example

```
7FFE1600/'R0/ Linefeed Linefeed'
```

The ASCII string "R0/ Linefeed Linefeed" is stored at address 7FFE1600. This string, if subsequently executed with the ;E command, examines the contents of general register 0 (the command R0/), then examines two subsequent registers (using two LINEFEED commands).

= (equal sign)—Display Value of Expression

Evaluates an expression and displays its value.

Format

expression **=**

Argument

expression
The expression to be evaluated.

Description

The Display Value of Expression command evaluates an expression and displays its value in hexadecimal. The expression can be any valid DELTA/XDELTA expression. See Section 2.1 for a description of DELTA/XDELTA expressions.

All calculations and displays are in hexadecimal in the prevailing length mode.

Note

Because DELTA and XDELTA treat the space as an addition operator, do not enter an unnecessary space.

Example

```
FF+1=00000100  ❶
A-1=00000009   ❷
```

❶ FF_{16} and 1_{16} are added together. DELTA/XDELTA displays the sum in hexadecimal.

❷ 1_{16} is subtracted from A_{16}. DELTA/XDELTA displays the result in hexadecimal.

string\—Immediate mode text display command (I64 and Alpha Only)

Displays the ASCII text string enclosed in backslashes.

Format

string\

Description

This mode is useful when creating your own predefined command strings. Use the backslash to begin and end an ASCII text string. Follow the ending backslash with a terminator. When DELTA or XDELTA encounters the ending backslash and terminator, it prints the ASCII text string.

ESC (Escape key)—Open Location and Display Previous Location

Opens the previous location and displays its contents.

Format

ESC

Description

The Open Location and Display Previous Location command decrements the location counter (.) by the width (in bytes) of the prevailing display mode, opens that many bytes, and displays the contents on a new line. The address of the location is displayed on the new line in the prevailing mode, followed by a slash (/) and the contents of that address.

On VAX, this command is ignored if the prevailing display mode is instruction mode (set by the ! command).

On all platforms, use this command to move backwards through a series of locations. Set the address where you want to start (for example, with the / command). Then press the ESC key repeatedly to display each preceding location. ESC is echoed as a dollar sign ($) on the terminal.

On keyboards without a separate ESC key, press Ctrl/3 or the escape key sequence that you defined on your keyboard. The ESC key on LK201 keyboards (VT220, VT240, VT340, and workstation keyboards) generates different characters and cannot be used for the ESC command. You must use Ctrl/3.

Example

```
R1/00000000 ❶ $ ❷ ESC
R0/00000001
```

❶ The contents of general register 1 are displayed using the / command.

❷ The contents of general register 0, the location prior to general register 1, are displayed by pressing ESC.

EXIT—Exit from DELTA Debugging Session

Terminates the DELTA debugging session. Use with DELTA only.

Format

EXIT

Description

Use the EXIT command to terminate a DELTA debugging session. You cannot use EXIT in XDELTA.

You may have to enter EXIT twice, such as when your program terminates execution by the $EXIT system service or by the Return key (to DCL).

LINEFEED (Linefeed key or Ctrl/J)—Close Current Location, Open Next Location

Closes the currently open location and opens the next location, displaying its contents.

Format

LINEFEED

Description

The Close Current Location Open Next command closes the currently open location, then opens the next and displays its contents. This command accepts no arguments, and thus can only be used to open the next location. It is useful for examining a series of locations one after another. First, set the location where you want to start (for example, with the / or (!) command). Then, press the Linefeed key repeatedly to examine each successive location.

The LINEFEED command displays the contents of the next location in the prevailing display mode and display width. If the current display mode is hexadecimal (the / command was used) and the display width is word, the next location displayed is calculated by adding a word to the current location. Its contents are displayed in hexadecimal. If the current display mode is instruction, the next location displayed is the next instruction, and the contents are displayed as a MACRO instruction.

On keyboards without a separate Linefeed key, press CTRL/J . The Linefeed key on LK201 keyboards (VT220, VT240, VT340, and workstation keyboards) generates different characters and cannot be used for the LINEFEED command. You must use CTRL/J .

This command is useful for displaying a series of machine instructions, a series of register values, or a series of values on the stack or in memory.

The values in the symbol Q and the symbol . are changed automatically.

The following examples illustrate the command on each OpenVMS platform.

Examples

I64 example:

```
G0BF5D60!       add         r33 = 0008, r33 ;;   ❶
80BF5D62!       nop.i       000000 ;;            ❷
80BF5D71!       nop.m       000000
80BF5D72!       sxt4        r2 = r2 ;;
80BF5D80!       cmp.eq      p14, p0 = r2, r0
80BF5D81!       nop.f       000000
80BF5D82! (p14) br.cond.dpnt.few.clr 0000030 ;;
80BF5D90!       ld8         r14 = [r2], 008 ;;
80BF5D91!       nop.m       000000
80BF5D92!       mov         b7 = r14 ;;
80BF5DA0!       ld8         r1 = [r2]
```

❶ The instruction at the base address G0BF5D60 is displayed using the ! command. XDELTA displays an add instruction.

❷ Ten successive instructions are displayed by pressing the Linefeed key twelve times. The LINEFEED command is not echoed on the terminal.

Alpha example:

```
30000!          LDA             SP,#XFFE0(SP)  ❶
00030004!       BIS             R31,R31,R18    ❷
00030008!       STQ             R27,(SP)
0003000C!       BIS             R31,R31,R19
00030010!       STQ             R26,#X0008(SP)
00030014!       BIS             R31,#X04,R25
```

❶ Instruction at address 30000 is displayed using the ! command.

❷ Five successive instructions are displayed by pressing the Linefeed key five times. The LINEFEED command is not echoed on the terminal.

VAX example:

```
6B9!CLRQ          -(SP)  ❶  [Linefeed]  ❷
000006BB/CLRQ     -(SP)    [Linefeed]
000006BD/PUSHL    X1+002E  [Linefeed]
000006C1/PUSHAL X1+003A    [Linefeed]
000006C5/CLRQ     -(SP)    [Linefeed]
000006C7/PUSHL    #00
```

❶ Instruction at address 6B9 is displayed using the ! command.

❷ Five successive instructions are displayed by pressing the Linefeed key five times. The LINEFEED command is not echoed on the terminal.

RETURN (Return or Enter key)— Close Current Location

Closes a location that has been opened by one of the open location and display contents commands.

Format

RETURN

Description

If you have opened a location with one of the open location and display contents commands (/, LINEFEED, ESC, TAB, !, or "), press the Return key to close the location. Use this command to make sure that a specific location has not been left open with the possibility of being overwritten.

You also press the Return key to terminate the following DELTA/XDELTA commands:

- ;X
- ;E
- ;G
- ;P
- ;B
- ;M
- 'string'
- ;L
- EXIT (DELTA only)

On I64 and Alpha, the same is true for the commands that are specific to this implementation, as follow:

- ;Q
- ;C
- ;D
- ;H
- ;I
- ;T (I64 only)
- ;W
- \ string\

On all platforms, you can also use the Return key as an ASCII character in a quoted string. See the Deposit ASCII String command (').

TAB (Tab key)—Open Location and Display Indirect Location

Opens the location addressed by the contents of the current location and displays its contents.

Format

TAB

Description

The Open Location and Display Indirect Location command opens the location addressed by the contents of the current location and displays the contents of the addressed location on a new line. The display is in the prevailing display mode. This command is useful for examining data structures that have been placed in a queue, or the operands of instructions.

To execute this command, press the Tab key.

This command changes the current address (.) to the location displayed.

This command does not affect the display mode.

The following examples illustrate the command on each OpenVMS platform.

Examples

I64, Alpha, and VAX example:

```
10000/00083089 ❶
00010004/00000000 ❷
00010008/00030000 ❸
00030000/23DEFFE0
```

❶ The contents of location 10000 are displayed using the / command.

❷ The subsequent two locations are displayed using the LINEFEED command.

❸ After displaying the contents of location 10008 (30000), the TAB command is used to display the contents of location 30000.

VAX example:

```
69B!BRB    0000067A    ❶ Tab
0000067A/CLRQ    -(SP) ❷
```

❶ The instruction at 69B is displayed using the ! command. DELTA/XDELTA displays a branch instruction.

❷ The instruction at the address referred to by the branch instruction is displayed by pressing the Tab key. DELTA/XDELTA displays the instruction at address 67A.

;B—Breakpoint

Shows, sets, and clears breakpoints.

Format

[addr-exp][,n][,display-addr-exp][,cmd-string-addr] **;B**

Arguments

addr-exp
The address where you want the breakpoint.

n
The number to assign to the breakpoint. If you omit a number, DELTA/XDELTA assigns the first unused number to the breakpoint; if all numbers are in use, DELTA/XDELTA displays the error message, "Eh?".

On VAX, for XDELTA, the range is from 2 to 8. For XDELTA, breakpoint 1 is reserved for INI$BRK. For DELTA, the range is from 1 to 8.

On I64 and Alpha, for XDELTA, the range is from 1 to 8. For DELTA, the range is from 1 to 8.

display-addr-exp
The address of a location, the contents of which are to be displayed in hexadecimal in the prevailing width mode when the breakpoint is encountered. Omit this argument by specifying zero or two consecutive commas. If omitted, DELTA/XDELTA displays only the instruction that begins at the specified address.

cmd-string-addr
The address of the string of DELTA/XDELTA commands to execute when this breakpoint is encountered. See the Execute Command String (;E) command. DELTA/XDELTA displays the information requested before executing the string of commands associated with complex breakpoints. You must have previously deposited the string of commands using the ' command or have coded the string into an identifiable location in your program. If omitted, DELTA/XDELTA executes no commands automatically and waits for you to enter commands interactively.

Description

The breakpoint command shows, sets, and clears breakpoints. The action of this command depends on the arguments used with it. Each action is described below.

Displaying Breakpoints
To show all the breakpoints currently set, enter ;B. For each breakpoint, DELTA/XDELTA displays the following information:

- Number of the breakpoint
- Address of the breakpoint
- Address of a location the contents of which will be displayed when the breakpoint is encountered

- Address of the command string associated with this breakpoint (for complex breakpoints, see the section called "Setting Complex Breakpoints" in this Description)

Setting Simple Breakpoints

To set a breakpoint, enter an address expression followed by ;B. Then press the Return key, as follows:

addr-exp;B Return

DELTA/XDELTA sets a breakpoint at the specified location and assigns it the first available breakpoint number.

When DELTA/XDELTA reaches the breakpoint, it completes the following actions:

- Suspends instruction execution.
- Sets a flag to change the display mode to instruction mode. Any subsequent Close Current Location, Open Next (LINEFEED) commands, and Open and Display Indirect Location (TAB) commands will display locations as machine instructions.
- On VAX, the following message is displayed, listing the number of the breakpoint, the address of the breakpoint, and the instruction stored at the breakpoint location:

  ```
  n BRK at address

  address/decoded-instruction
  ```

  On I64 and Alpha, the format of the display differs slightly, as shown in the following example:

  ```
  Brk n at address [on CPUn] [new mode =]
  [new IPL =]

  address!decoded-instruction
  ```

- On I64 and Alpha systems, if the interrupt priority level (IPL) has changed, the new IPL is printed (XDELTA only). Also on I64 and Alpha systems, if the processor mode has changed, the new mode is printed (both XDELTA and DELTA).

If you are using XDELTA in a multiprocessor environment, the CPU ID of the processor where the break was taken is also displayed.

On VAX, the CPU ID is displayed as a 2-digit hexadecimal number.

On I64 and Alpha, the CPU ID is displayed as a decimal number with no leading zeros.

On all platforms, after the breakpoint message is displayed, you can enter other DELTA/XDELTA commands. You can reset the flag that controls the mode in which instructions are displayed by entering the Open Location and Display Contents (/) command.

Setting a Breakpoint and Assigning a Number to It

To set a breakpoint and assign it a number, enter the address where you want the breakpoint, a comma, a single digit for the breakpoint number, a semicolon (;), the letter B, and then press the Return key.

For example, if you wanted to set breakpoint 4 at address 408, the command is as follows:

```
408,4;B Return
```

DELTA/XDELTA sets a breakpoint at the specified location and assigns it the specified breakpoint number.

Clearing Breakpoints

To clear a breakpoint, enter zero (0), followed by a comma, the number of the breakpoint to remove, a semicolon (;), the letter B, and then press the Return key. DELTA/XDELTA clears the specified breakpoint. For example, if you wanted to clear breakpoint 4, the command is as follows:

```
0,4;B Return
```

On VAX, when using XDELTA, do not clear breakpoint 1. If you do, any calls to INI$BRK in your program will not result in entry into XDELTA.

Setting Complex Breakpoints

On all platforms, a complex breakpoint completes one or more of the following actions:

- Always displays the next instruction to be executed
- Optionally displays the contents of another, specified location
- Optionally executes a string of DELTA/XDELTA commands stored in memory

To use the complex breakpoint, you must first create the string of DELTA commands you want executed. Then deposit those commands at a memory location with the Deposit ASCII String command (').

To set a complex breakpoint, use the following syntax:

addr-exp,n,display-addr-exp,cmd-string-addr;B

Example

```
;B
1  00000690
2  00000699                    ❶
0,2;B                          ❷
;B
1  00000690                    ❸
;P ❹
```

❶ Two breakpoints have already been set and are displayed. Using ;B, DELTA/XDELTA displays each breakpoint number and the address location of each breakpoint.

❷ Breakpoint 2 is cleared.

❸ Current breakpoints are displayed. Because breakpoint 2 has been cleared, DELTA/XDELTA displays just breakpoint 1.

❹ Program execution is continued using the ;P command.

Displaying Breakpoints in a Multithreaded Application

To support the debugging of multithreaded applications, DELTA has the capability of displaying a thread ID at a breakpoint. When DELTA reaches a breakpoint in a multithreaded application, DELTA displays the thread ID and stops the execution of all other threads. (When DELTA reaches a breakpoint in a single-threaded application, the display and behavior is the same as in the past; DELTA displays the address and stops program execution.)

In the following example, a breakpoint is set in a multithreaded application with 30000;B and is followed by the ;P (Proceed from Breakpoint) command. The breakpoint is taken. Because it is a multithreaded application, the thread ID is included in the display.

```
30000;B  ;P
Brk 1 at 30000 on Thread 12
00030000!  LDA  SP,#XFF80(SP)
```

;C—Force System to Bugcheck and Crash (I64 and Alpha Only)

Force the system to bugcheck and crash.

Format

;C

Description

The ;C command forces the system to bugcheck and crash. You can do this from wherever you are in your debugging session. Although this command is for use primarily with XDELTA, you can also use it with DELTA, but only in kernel mode. When you issue this command, the following message is generated:

```
BUG$_DEBUGCRASH, Debugger forced system crash
```

;D—Dump (I64 and Alpha Only)

Dumps a region of memory.

Format

addr_exp, length **;D**

Parameters

addr-exp
The starting address of the dump.

length
The length of bytes to dump.

Description

On I64 and Alpha systems, the ;D command dumps a region of memory. The display is in a format similar to the DCL DUMP command.

Example

```
G,200;D                                                          ❶

Dump of 80000000 for 00000200 bytes                              ❷

00840008 80000200 0000241F 00E8401D  .@...$.......... : 80000000 ❸
00840008 80000200 00002400 0004401D  .@...$.......... : 80000010
00840008 80000200 00000001 0000001D  ................ : 80000020
00000000 00000000 00000000 00000000  ................ : 80000030
00040000 00203008 00202400 0260100B  ..'..$ ..0 ..... : 80000040
90000A00 40038004 10700001 00000001  ......p....@.... : 80000050
00800070 00000200 00001418 04200810  .. .........p... : 80000060
00000000 00000000 00000000 00000000  ................ : 80000070
00000000 00000000 00000000 00000000  ................ : 80000080
00000000 00000000 00000000 00000000  ................ : 80000090
00000000 00000000 00000000 00000000  ................ : 800000A0
00000000 00000000 00000000 00000000  ................ : 800000B0
00000000 00000000 00000000 00000000  ................ : 800000C0
00000000 00000000 00000000 00000000  ................ : 800000D0
00000000 00000000 00000000 00000000  ................ : 800000E0
00000000 00000000 00000000 00000000  ................ : 800000F0
00040000 00040000 00300580 02090001  ......0......... : 80000100
00840008 80000200 00000001 0000001D  ................ : 80000110
00840008 80000200 00000001 0000001D  ................ : 80000120
00840008 80000200 00002400 0004401D  .@...$.......... : 80000130
00840008 80000200 0000241C 0128401D  .@(..$.......... : 80000140
84000804 40006200 02000580 060D0800  .........b.@.... : 80000150
20000000 00000200 00002400 0000C81D  .....$.........  : 80000160
50000178 00000200 00000001 0000001D  ............x..P : 80000170
07000A00 00005501 08002100 44000802  ...D.!...U...... : 80000180
00840008 80000200 00000001 0000001D  ................ : 80000190
00840008 80000200 00002400 0004401D  .@...$.......... : 800001A0
00840008 80000200 00002400 0004401D  .@...$.......... : 800001B0
00840008 80000200 00002400 0004401D  .@...$.......... : 800001C0
00840008 80000200 00002400 0004401D  .@...$.......... : 800001D0
00840008 80000200 00002400 0004401D  .@...$.......... : 800001E0
00840008 80000200 00002400 0004401D  .@...$.......... : 800001F0

FFFFFFFF 8       ❹
```

❶ The dump command is issued.

❷ The dump output summarizes the operation.

❸ The memory dump is displayed. The output is in the same format as the DCL DUMP command.

❹ The starting location of the dump is printed.

;E—Execute Command String

Executes a string of DELTA/XDELTA commands stored in memory.

Format

address-expression **;E**

Arguments

address-expression
The address of the string of DELTA/XDELTA commands to execute.

Description

The Execute Command String command executes a string of DELTA/XDELTA commands. Load the ASCII text command string to a specific location in memory using the Deposit ASCII String command (') or code the string in your program into an identifiable location.

If you want DELTA/XDELTA to proceed with program execution after it executes the string of commands, end the command string with the ;P command. If you want DELTA/XDELTA to wait for you to enter a command after it executes the string of commands, end the command string with a null byte (a byte containing 0).

XDELTA, but not DELTA, provides two command strings in memory.

On Alpha, the addresses of these command strings are stored in base registers X14 and X15. The string addressed by X14 displays the physical page number (PFN) database for the PFN in X0. The string addressed by X15 copies the PFN in R0 to base register X0. It then displays the PFN database for that PFN.

On VAX, the addresses of these command strings are stored in base registers XE and XF. The string addressed by XE displays the physical page number (PFN) database for the PFN in X0. The string addressed by XF copies the PFN in R0 to base register X0. It then displays the PFN database for that PFN.

You can use the command strings provided with XDELTA to obtain the following information:

- Specified PFN
- PFN state and type
- PFN reference count
- PFN backward link or working-set-list index
- PFN forward link or share count
- Page table entry (PTE) address that points to the PFN
- PFN backing-store address
- On VAX, the virtual block number in the process swap image, the block to which the page's entry in the SWPVBN array points
- On Alpha, the virtual page number in process swap image, the collection of blocks containing the page as pointed to by the PFN database

Example

```
7FFE1600,0;X        ❶
7FFE1600            ❷
X0;E                ❸
R0/00000001         ❹
R1/00000000
R2/00000000
```

❶ The address (7FFE1600) where an ASCII string is stored is placed into base register 0 using ;X.

❷ DELTA/XDELTA displays the value in X0.

❸ The command string stored at address 7FFE1600, which is to examine the contents of R0, R1, and R2 (R0/Linefeed Linefeed), is executed with ;E.

❹ DELTA/XDELTA executes the commands and displays the contents of R0, R1, and R2.

;G—Go

Continues program execution.

Format

address-expression **;G**

Parameters

address-expression
The address at which to continue program execution.

Description

The Go command places the address you specified in **address-expression** into the PC and continues execution of the program at that address. It is useful when you want to ignore specific lines of code or return to a previous program location to repeat execution.

Example

```
6A2;G
```

Program execution is started at address 6A2.

;H—Video Terminal Display Command (I64 and Alpha Only)

Specifies the display mode, either hardcopy terminal mode or DEC-CRT.

Format

;H

Description

The ;H command enables you to choose the display mode of DELTA/XDELTA output. You can display output either in hardcopy terminal mode or DEC-CRT mode. The default display is DEC-CRT mode. You can toggle back and forth from one display mode to the other by repeating the ;H command.

;I—List Current Main Image and Its Shareable Images (I64 and Alpha Only)

Lists information about the current main image and all shareable images that were activated, including those that were installed /RESIDENT.

Format

;I

Description

The ;I command peruses the image control block (IMCB) list and displays information about the current main image and all shareable images that were activated, including those that were installed /RESIDENT. The ;I command differs from the ;L command which displays information about the loadable image database.

The display of the ;I command is similar to the ;L command display. It shows the image name, the starting and ending addresses, the symbol vector address, and some flags. The command is useful for debugging shareable images. For example, the display enables you to determine where LIBRTL is mapped.

The field flags are M, S, and P. The flag M indicates the main image; S or P indicates images that are installed as shareable or protected, respectively.

Unlike the ;L command, which only works from kernel mode or when you have CMEXEC or CMKRNL privileges, the ;I command works from any mode. However, to modify the IMCB database, you must be in executive or kernel mode.

For resident main and shareable images, the ;I command also includes an entry for each resident code section and each compressed data section, which shows the base and end address for each section.

The ;I command is implemented only for DELTA.

Example

```
$ define lib$debug delta
$ run/debug hello
OpenVMS Alpha DELTA Version 1.5

Brk 0 at 00020040

00020040!     LDA               SP,#XFFD0(SP) ;i
Image Name                        Base     End      Symbol-Vector  Flags

HELLO                             00010000 000301FF                M
DECC$SHR                          00032000 001233FF 00106B90       S
DPML$SHR                          0012C000 001AC5FF 0019DED0       S
LIBRTL                            001AE000 0025E7FF 00240790       S
Resident Code Sections:
                                  8015A000 801BBA00
LIBOTS                            00124000 0012A1FF 00128000       S
```

;I—List Current Main Image and Its Shareable Images (I64 and Alpha Only)

```
Resident Code Sections:
                                801BC000 801C2C00
Compressed Data Sections:
                                00124000 00124A00
                                00126000 00126800
                                00128000 00128600
                                0012A000 0012A200
SYS$PUBLIC_VECTORS              80401C98 80403028 80401C98
DELTA                           00260000 002943FF 00260000
SYS$BASE_IMAGE                  8040C5B0 804163E0 8040C5B0
```

;L—List Names and Locations of Loaded Executive Images

Lists the names and virtual addresses of all loaded executive images.

Format

[sequence number]**;L**

Argument

sequence number
On I64 and Alpha, specifies a single executive image.

Description

Use the ;L command when you are debugging code that resides in system space. Although you use this command mostly with XDELTA, you can use it with DELTA if your process has change-mode-to-executive (CMEXEC) privilege and you are running a program in executive mode.

This command lists the names and locations of the loaded modules of the executive. A loading mechanism maps a number of images of the executive into system space. The ;L command lists the currently loaded images with their starting and ending virtual addresses. If you enter ;L before all the executive images are loaded (for example, at an XDELTA initial breakpoint), only those images that have been loaded will be displayed.

On Alpha, this command displays additional information and provides a second use, based on the additional information. For each loaded executive image that is sliced into discontiguous image sections, the display shows the sequence number for the executive image and the base and ending addresses of each image section. A second use of this command is to display the base and ending addresses of a single image if you specify its sequence number.

The following examples illustrate the command on each platform.

Examples

The following I64 example shows the names, the starting and ending virtual addresses, and the sequence number for the specified loaded executive image. Images are split into image sections, showing the name and the base, link, and ending address of each respective section. In these examples, sequence number 24 selects the PROCESS_MANAGEMENT; sequence number selects SYS$PUBLIC_VECTORS; and sequence number 32 selects RMS.

```
24;L

Seq# LDRISD   Image Name                      Base      End     Link      End

0024 83881B80 PROCESS_MANAGEMENT
   0 83881C70 Read Write                 83203800 83203808 00010000 00010008
   1 83881CB8 Read       Execute         805AF300 806E4D70 00014000 00149A70
   2 83881D00 Read                       83203A00 83230C78 0014C000 00179278
   3 83881D48 Read Write                 83230E00 8323C120 0017C000 00187320
   4 83881D90 Read Write                 8323C200 8323C214 00188000 00188014
   7 83881E68 Read Write                 8323C400 8323C414 00194000 00194014
   8 83881EB0 Read Write                 8323C600 8323C604 00198000 00198004
   9 83881EF8 Read Write                 8323C800 83240660 0019C000 0019FE60

0;L
```

```
Seq# LDRISD   Image Name                    Base      End     Link      End

0000 83868580 SYS$PUBLIC_VECTORS
   0 83868670 Read       Execute          80000000 80000070 00010000 00010070
   1 838686B8 Read                        83000000 830000B0 00014000 000140B0
   2 83868700 Read Write                  83000200 83000218 00018000 00018018
   3 83868748 Read                        83000400 83008788 0001C000 00024388
              Symbol Vector               83000400

32;L

Seq# LDRISD   Image Name                    Base      End     Link      End

0032 83885500 RMS
   0 838855E0 Read Write                  832B5800 832B5F40 00010000 00010740
   1 83885628 Read       Execute          8014E900 8014FAE0 00014000 000151E0
   2 83885670 Read       Execute          8098D100 80B9C8A0 00018000 002277A0
   3 838856B8 Read                        832B6000 832EC400 00228000 0025E400
   4 83885700 Read Write                  832EC400 832EFAE8 00260000 002636E8
   5 83885748 Read Write                  832EFC00 832EFC14 00264000 00264014
   6 83885790 Read Write                  832EFE00 832EFE50 00268000 00268050
   9 83885868 Read Write                  832F0000 832F0014 00274000 00274014
   A 838858B0 Read Write                  832F0200 832F0204 00278000 00278004
   B 838858F8 Read Write                  832F0400 832F3DC0 0027C000 0027F9C0
```

The following Alpha example shows the names, the starting and ending virtual addresses, and the sequence numbers for all the loaded executive images. Only one image, EXEC_INIT.EXE, was not split into image sections. For every image that was split into image sections, it also shows the name and the base and ending address of each section.

```
;L

Seq#  Image Name                          Base      End

0012  EXEC_INIT.EXE                       8080C000  80828000
0010  SYS$CPU_ROUTINES_0101.EXE
          Nonpaged read only               80038000  8003A200
          Nonpaged read/write              80420200  80420A00
          Initialization                   80808000  80808400
000E  ERRORLOG.EXE
          Nonpaged read only               8002E000  80036600
          Nonpaged read/write              8041BE00  80420200
          Initialization                   80804000  80804800
000C  SYSTEM_SYNCHRONIZATION.EXE
          Nonpaged read only               80024000  8002C800
          Nonpaged read/write              8041A000  8041BE00
          Initialization                   80800000  80800800
              .       .     .
              .       .     .
              .       .     .
0002  SYS$BASE_IMAGE
          Nonpaged read only               80002000  80009400
          Nonpaged read/write              80403000  80414C00
          Fixup                            80620000  80620600
          Symbol Vector                    8040B010  80414560
0000  SYS$PUBLIC_VECTORS.EXE
          Nonpaged read only               80000000  80001C00
          Nonpaged read/write              80400000  80403000
          Fixup                            8061E000  8061E200
          Symbol Vector                    80401BF0  80402ED0
```

The following Alpha example illustrates the use of the sequence number with the ;L command to display information about one image. In this example, the sequence number C for the SYSTEM_SYNCHRONIZATION.EXE module is specified with the ;L command. (It is not necessary to specify the leading zeros in the command.) The resulting display shows only the SYSTEM_SYNCHRONIZATION.EXE module (whose sequence number is 000C). The display includes the names of the image sections within the module and their base and ending addresses.

```
C;L
Seq#  Image Name                                Base      End

000C  SYSTEM_SYNCHRONIZATION.EXE
          Nonpaged read only                    80024000  8002C800
          Nonpaged read/write                   8041A000  8041BE00
          Initialization                        80800000  80800800
```

VAX example, showing the names and the starting and ending virtual addresses of the three executive images that are loaded in memory.

```
;L
PRIMITIVE_IO.EXE              800EAA00  800EBC00
SYSTEM_SYNCHRONIZATION.EXE    800EBC00  800ED400
SYSTEM_PRIMITIVES.EXE         800ED400  800F1000
```

;M—Set All Processes Writable

Sets the address spaces of all processes to be writable or read-only by your DELTA process. This command can be used only with DELTA. Use of this command requires CMKRNL privilege.

On Alpha, this command also sets writable the general purpose registers of other processes, if, after issuing the ;M command, you specify another process with any command that takes the PID argument, such as the / command.

Format

n**;M**

Argument

n
Specifies your process privileges for reading and writing at other processes. If 0, your DELTA process can only read locations in other processes; if 1, your process can read or write any location in any process. If not specified, DELTA returns the current value of the M (modify) flag (0 or 1).

Description

The Set All Processes Writable command is useful for changing values in the running system.

Note

Use this activity very carefully during timesharing. It affects all processes on the system. For this reason, your process must have change-mode-to-kernel (CMKRNL) privilege to use this command. It is safest to use this command only on a standalone system.

;P—Proceed from Breakpoint

Continues program execution following a breakpoint.

Format

;P

Description

The Proceed from Breakpoint command continues program execution at the address contained in the PC of the program. Program execution continues until the next breakpoint or until program completion.

Note

If DELTA/XDELTA does not have write access to the target of a JSR instruction, you cannot use the S or ;P command at the JSR instruction. First, you must use the O command; then you can use the S or ;P command.

The following examples illustrate the command on each OpenVMS platform.

Example

I64 example:

```
G0BF5D60,0;X                          ❶
G0BF5D60

X0+60;B
 1 00000060                           ❷

;P                                    ❸

Brk 1 at X0+00000060 on CPU 0         ❹

X0+00000060!       alloc       r53 = ar.pfs, 18, 08, 00  (New IPL = 0) -
                               (New mode = USER)
```

❶ Set the base register.

❷ Set a breakpoint at address X0+00000060 using ;B.

❸ Program execution is continued using the ;P command.

❹ Program execution halts at breakpoint 1. DELTA/XDELTA displays the breakpoint message (the breakpoint number and the address) and the instruction.

Alpha example:

```
;B
 1 00030010      ❶
```

```
;P          ❷
Brk 1 at 00030010
00030010!       STQ             R26,#X0008(SP)  ❸
```

❶ Current breakpoints are displayed using ;B (breakpoint 1 at address 30010).

❷ Program execution is continued using the ;P command.

❸ Program execution halts at breakpoint 1. DELTA/XDELTA displays the breakpoint message (the breakpoint number and the address) and the instruction.

VAX example:

```
;B
2  00000699                ❶
;P                         ❷
2 BRK AT 00000699
00000699/BSBB    000006A2  ❸
```

❶ Current breakpoints are displayed using ;B (breakpoint 2 at address 699).

❷ Program execution is continued using the ;P command.

❸ Program execution halts at breakpoint 2. DELTA/XDELTA displays the breakpoint message (the breakpoint number and the address) and the instruction.

;Q—Validate Queue (I64 and Alpha Only)

Analyzes absolute and self-relative longword queues and displays the results of the analysis.

Format

queue_header_address[,queue_type]**;Q**

Argument

queue_header_address
The queue header must be at least longword aligned.

queue_type
A queue type of zero (the default) represents an absolute queue. A queue type of 1 indicates a self-relative queue.

Description

The validate queue function is similar to the one in the OpenVMS System Dump Analyzer Utility. It can analyze both absolute and self-relative longword queues and display the results of the analysis. This function identifies various problems in the queue headers and invalid backward links for queue entries and evaluates the readability of both. For valid queues, it tells you the total number of entries. For invalid queues, it tells you the queue entry number and the address that is invalid and why.

Example

```
FFFFFFFF8000F00D;Q      !Absolute at GF00D
GF00D,0;Q               !Absolute at GF00D
GF00,1;Q                !Self-relative at GF00
```

;T—Display Interrupt Stack Frame on XDELTA (I64 Only)

XDELTA only; displays contents of an interrupt stack frame.

Format

addr_exp **;T**

Parameters

addr-exp
The address of the stack frame. This is an optional argument. If not specified, the ;T command without any argument displays the interrupt stack frame with which XDELTA was invoked.

Description

On I64 systems, the XDELTA ;T command displays the contents of an interrupt stack frame.

Example

In the following example, the ;T command displays the machine state at the time of the exception.

```
;T

* Exception Frame Display: *

Exception taken at IP FFFFFFFF.8063D830, slot 01
from Kernel mode Exception Frame at FFFFFFFF.89DA1CE0
Trap Type   00000080 (External Interrupt)
IVT Offset  00003000 (External Interrupt)
External Interrupt Vector  00000000

* = Value read directly from the register rather than the frame

Control Registers:
CR0   Default Control Register (DCR)          00000000.00007F00
CR1   Interval Timer Match Register (ITM)   * 0000C6F7.31F82D5B
CR2   Interruption Vector Address (IVA)     * FFFFFFFF.801D0000
CR8   Page Table Address (PTA)              * FFFFFFFF.7FFF013D
CR16  Processor Status Register (IPSR)        00001210.0A026010
CR17  Interrupt Status Register (ISR)         00000200.00000000
CR19  Instruction Pointer (IIP)               FFFFFFFF.8063D830
CR20  Faulting Address (IFA)                  FFFFFFFF.88580078
CR21  TLB Insertion Register (ITIR)           00000000.00000334
CR22  Instruction Previous Address (IIPA)     FFFFFFFF.8063D830
CR23  Function State (IFS)                    80000000.00000FA7
CR24  Instruction immediate (IIM)             FFFFFFFF.88580078
CR25  VHPT Hash Address (IHA)                 FFFFFFFF.7FFF5860
CR64  Local Interrupt ID (LID)              * 00000000.00000000
CR66  Task Priority Register (TPR)          * 00000000.00010000
CR68  External Interrupt Req Reg 0 (IRR0)   * 00000000.00000000
CR69  External Interrupt Req Reg 1 (IRR1)   * 00000000.00000000
CR70  External Interrupt Req Reg 2 (IRR2)   * 00000000.00000000
CR71  External Interrupt Req Reg 3 (IRR3)   * 00020000.00000000
CR72  Interval Time Vector (ITV)            * 00000000.000000F1
CR73  Performance Monitoring Vector (PMV)   * 00000000.000000FB
CR74  Corrected Machinecheck Vector (CMCV)  * 00000000.00010000
CR80  Local Redirection Register 0 (LRR0)   * 00000000.00010000
CR81  Local Redirection Register 1 (LRR1)   * 00000000.00010000
```

```
Application Registers:
AR0   Kernel Register (KR0)                 * 00000000.20570000
AR1   Kernel Register (KR1)                 * 00000000.60000000
AR2   Kernel Register (KR2)                 * 00000000.00000000
AR3   Kernel Register (KR3)                 * 00000000.00000000
AR4   Kernel Register (KR4)                 * 00000000.00000000
AR5   Kernel Register (KR5)                 * 0000C6F7.31F82D5B
AR6   Kernel Register (KR6)                 * FFFFFFFF.84C3E000
AR7   Kernel Register (KR7)                 * FFFFFFFF.89D4B000
AR16  Register Stack Config Reg (RSC)         00000000.00000000
AR17  Backing Store Pointer (BSP)             FFFFF802.A3EAC300
AR18  Backing Store for Mem Store (BSPSTORE) FFFFF802.A3EAC300
AR19  RSE NaT Collection Register (RNAT)      00000000.00000000
AR32  Compare/Exchange Comp Value Reg (CCV)   FFFFFFFF.84132680
AR36  User NaT Collection Register (UNAT)     00000000.00000000
AR40  Floating-point Status Reg (FPSR)        0009804C.0270033F
AR44  Interval Time Counter (ITC)           * 0000C6FB.A91997B5
AR64  Previous Function State (PFS)           00000000.00000FA7
AR65  Loop Count Register (LC)                00000000.00000000
AR66  Epilog Count Register (EC)              00000000.00000000

Processor Status Register (IPSR):
AC = 0   MFL= 1   MFH= 0   IC = 1   I  = 1   DT = 1
DFL= 0   DFH= 0   RT = 1   CPL= 0   IT = 1   MC = 0   RI = 1
Interrupt Status Register (ISR):
Code 00000000     X  = 0   W  = 0   R  = 0   NA = 0   SP = 0
RS = 0   IR = 0   NI = 0   SO = 0   EI = 1   ED = 0

Branch Registers:                   Region Registers:
B0        FFFFFFFF.8063C570         RR0     * 00000000.00000035
B1        00000000.00000000         RR1     * 00000000.00000030
B2        00000000.00000000         RR2     * 00000000.00000030
B3        00000000.00000000         RR3     * 00000000.00000030
B4        00000000.00000000         RR4     * 00000000.00000030
B5        00000000.00000000         RR5     * 00000000.00000030
B6        FFFFFFFF.80001580         RR6     * 00000000.00000030
B7        FFFFFFFF.806F4D30         RR7     * 00000000.00000335

Floating Point Registers:           FPSR      0009804C.0270033F
F6        00000000.0001003E.00000000.0000FCBE
F7        00000000.0001003E.00000000.00000040
F8        00000000.0001003E.00000000.003F2F80
F9        00000000.00010003.80000000.00000000
F10       00000000.0000FFFB.80000000.00000000
F11       00000000.0000FFFB.80000000.00000000

Miscellaneous Registers:
Processor Identifier (CPUID 0,1)            GenuineIntel
                     (CPUID 3)              00000000.1F010504
Interrupt Priority Level (IPL)                       00000003
Stack Align                                          000002D0
NaT Mask                                                 001C
PPrev Mode                                                 00
Previous Stack                                             00
Interrupt Depth                                            00
Preds                                       00000000.FF65CCA3
Nats                                        00000000.00000000
Context                                     00000000.FF61CEA3
```

;T—Display Interrupt Stack Frame on XDELTA (I64 Only)

```
General Registers:
R0   00000000.00000000     GP   FFFFFFFF.8442E200     R2  FFFFFFFF.84132688
R3   FFFFFFFF.8442E200     R4   FFFFFFFF.8442E200     R5  00000000.00000001
R6   FFFFFFFF.84C3E000     R7   00000000.00000000     R8  00000000.00000003
R9   00000000.00000009     R10  00000000.00000008     R11 00000000.00000000
SP   FFFFFFFF.89DA0D18     TP   00000000.00000000     R14 00000000.00000001
R15  FFFFFFFF.8401BD90     R16  FFFFFFFF.84017508     R17 FFFFFFFF.84009E98
R18  FFFFFFFF.84C3F274     R19  00000000.00000000     R20 FFFFFFFF.84009E00
R21  FFFFFFFF.84132627     R22  FFFFFFFF.84C3E01C     R23 00000000.0000000F
R24  00000000.00011F90     R25  00000000.00000003     R26 00000000.00000000
R27  FFFFFFFF.84132668     R28  FFFFFFFF.8416D7C8     R29 FFFFFFFF.89DA1FB0
R30  00000000.7FF2E318     R31  00000000.00000000

Interrupted Frame RSE Backing Store , Size = 39 registers

 FFFFF802.A3EAC300:  FFFFFFFF.84C3E080 (R32)
 FFFFF802.A3EAC308:  E0000000.00000000 (R33)
 FFFFF802.A3EAC310:  FFFFFFFF.84132628 (R34)
 FFFFF802.A3EAC318:  FFFFFFFF.88598080 (R35)
 FFFFF802.A3EAC320:  00000000.00000001 (R36)
 FFFFF802.A3EAC328:  FFFFFFFF.806029A0 (R37)
 FFFFF802.A3EAC330:  00000000.FF65C563 (R38)
 FFFFF802.A3EAC338:  00000000.00000000 (R39)
 FFFFF802.A3EAC340:  FFFFFFFF.8442E200 (R40)
 FFFFF802.A3EAC348:  FFFFFFFF.806029C0 (R41)
 FFFFF802.A3EAC350:  FFFFFFFF.8442E200 (R42)
 FFFFF802.A3EAC358:  FFFFFFFF.88598080 (R43)
 FFFFF802.A3EAC360:  FFFFFFFF.84191000 (R44)
 FFFFF802.A3EAC368:  00000000.00000009 (R45)
 FFFFF802.A3EAC370:  FFFFFFFF.8416D7C8 (R46)
 FFFFF802.A3EAC378:  FFFFFFFF.8442E200 (R47)
 FFFFF802.A3EAC380:  00000000.00000000 (R48)
 FFFFF802.A3EAC388:  FFFFFFFF.84132668 (R49)
 FFFFF802.A3EAC390:  00000000.00000008 (R50)
 FFFFF802.A3EAC398:  00000000.00000000 (R51)
 FFFFF802.A3EAC3A0:  00000000.7FF2E318 (R52)
 FFFFF802.A3EAC3A8:  00000000.00000000 (R53)
 FFFFF802.A3EAC3B0:  00000000.00000FB2 (R54)
 FFFFF802.A3EAC3B8:  FFFFFFFF.84132627 (R55)
 FFFFF802.A3EAC3C0:  00000000.00000003 (R56)
 FFFFF802.A3EAC3C8:  FFFFFFFF.89DA1FB0 (R57)
 FFFFF802.A3EAC3D0:  FFFFFFFF.801D9BD0 (R58)
 FFFFF802.A3EAC3D8:  FFFFFFFF.806029C0 (R59)
 FFFFF802.A3EAC3E0:  00000000.00000001 (R60)
 FFFFF802.A3EAC3E8:  FFFFFFFF.89DA1FB0 (R61)
 FFFFF802.A3EAC3F0:  FFFFFFFF.8442E200 (R62)
 FFFFF802.A3EAC400:  00000000.00000003 (R63)
 FFFFF802.A3EAC408:  FFFFFFFF.8063C570 (R64)
 FFFFF802.A3EAC410:  00000000.00000008 (R65)
 FFFFF802.A3EAC418:  00000000.00000008 (R66)
 FFFFF802.A3EAC420:  FFFFFFFF.84132668 (R67)
 FFFFF802.A3EAC428:  FFFFFFFF.8416D7C8 (R68)
 FFFFF802.A3EAC430:  00000000.00000008 (R69)
 FFFFF802.A3EAC438:  FFFFFFFF.8416DAA0 (R70)
```

;W—List Name and Location of a Single Loaded Image (I64 and Alpha Only)

Lists information about an image that contains the address you supplied.

Format

address-expression**;W**

sequence number, offset**;W**

Arguments

address-expression
An address contained within an executive image or a user image.

sequence number
The identifier assigned to an executive image.

offset
The distance from the base address of the image.

Description

The ;W command is used for debugging code that resides in system or user space. You can use this command with XDELTA for debugging an executive image. You can also use this command with DELTA.

To examine the executive image list, you must be running in executive mode or your process must have change-mode-to-executive (CMEXEC) privilege.

This command can be used in two ways. In the first way, if you supply an address that you are trying to locate, the command lists the name of the executive or user image that contains the address, its base and ending addresses, and the offset of the address from the base of the image. For any executive image that has been sliced, it also displays its sequence number. The offset can be used with the link map of the image to locate the actual code or data. This offset is saved in the value Q.

In the second way, if you supply the sequence number of a sliced executive image and an offset, the command computes and displays the address in memory. The address is saved in the value Q.

Examples

The first form of the command takes a system space address as a parameter and attempts to locate that address within the loaded executive images. This command works for both sliced and unsliced loadable executive images. The output is very similar to ;L, except the offset is displayed for you, as shown in the following example:

```
80026530;W
Seq#  Image Name                          Base      End       Image Offset
000C  SYSTEM_SYNCHRONIZATION.EXE
          Nonpaged read only               80024000  8002C800  00002530
```

The second form of the command takes a loadable executive image sequence number and an image offset from the map file as parameters. The output, again, is very similiar to ;L, except that the system space address that corresponds to the image offset is displayed, as shown in the following example:

```
C,2530;W
Seq#  Image Name                                Base      End       Address

000C  SYSTEM_SYNCHRONIZATION.EXE
          Nonpaged read only                    80024000  8002C800  80026530
```

;X—Load Base Register

Places an address in a base register.

Format

address-expression,n[,y]**;X**

Arguments

address-expression
The address to place in the base register.

n
The number of the base register.

y
On I64 and Alpha, a parameter for modifying the default offset of 10000_{16}. The valid range is 1 to FFFFFFFF.

Description

On I64 and Alpha, to place an address in a base register, enter:

- An expression followed by a comma (,), or
- A number from 0 to 15_{10}, or optionally, a number from 1 to FFFFFFFF, a semicolon (;)
- The letter X.
-

On VAX, to place an address in a base register, enter an expression followed by a comma (,), a number from 0 to F_{16}, a semicolon (;), and the letter X.

On all platforms, DELTA/XDELTA places the address in the base register. DELTA/XDELTA confirms that the base register is set by displaying the value deposited in the base register.

For example, the following command places the address 402 in base register 0. DELTA/XDELTA then displays the value in the base register to verify it.

```
402,0;X Return
00000402
```

Whenever DELTA/XDELTA displays an address, it will display a relative address if the address falls within the computer's valid range for an offset from a base register. The relative address consists of the base register identifier (Xn), followed by an offset. The offset gives the address location in relation to the address stored in the base register.

For example, if base register 2 contains 800D046A, the address that would be displayed is X2+C4, the base register identifier followed by the offset.

Relative addresses are computed for both opened and displayed locations and for addresses that are instruction operands.

If you have defined several base registers, the offset will be relative to the closest base register. If an address falls outside the valid range, it is displayed as a hexadecimal value.

On I64, the default offset is 100000_{16}, which can be modified.

On Alpha, the default offset is 10000_{16}, which can be modified.

On VAX, the default offset is 2000_{16} bytes. It cannot be modified.

The following examples illustrate the command on each platform.

Examples

I64 example:

```
G0BF5D60,0,200;X 1

;X

0 80BF5D60  00000200
4 8392A900
5 83009DE0
13 FFFFF802 06C00000
14 830937F0
15 83093700

G0BF5D60,0,200;X                    ❶

;X

  0 80BF5D60  00000200
  4 8392A900
  5 83009DE0
 13 FFFFF802 06C00000
 14 830937F0
 15 83093700                              ❷
```

❶ Set the base register, with an offset.

❷ The ;X command with no arguments displays the existing base register values. Offset values are also displayed, if their value is other than the default offset.

Alpha example:

```
30000,0;X       ❶
00030000
30070,1,200;X       ❷
00030070
;X       ❸
  0 00030000
  1 00030070  00000200

S       ❹
X0+00000004!    BIS             R31,R31,R18

x1+10!  STQ             FP,#X0020(SP)    ❺
```

❶ The base address of the program (determined from the map file) is virtual address 30000. The base address is stored in base register 0 with ;X, using the default offset. DELTA/XDELTA displays the value in base register 0 just loaded, 30000.

❷ The address of a subroutine, 30070, is stored in base register 1, specifying a new offset of 200 (to override the default value of 100000). Note that this command could also have been expressed as "x0+70,1,200;X". DELTA/XDELTA displays the value in base register 1 just loaded, 30070.

❸ The ;x command is used to display the current base registers. Note that for those not using the default offset, the offset is also displayed.

❹ The S command is used to execute the first instruction in the main routine. DELTA/XDELTA displays the address of the next instruction, 30004, as x0+00000004 and then displays the instruction at that address.

❺ The instruction at offset 10 from base register 1 is displayed in instruction mode using the ! command.

VAX example:

```
00000664/CLRQ    -(SP)    200,1;X   ❶
00000200                            ❷

X1 490!CMPL    R0,#000009A8         ❸
X1 499!BSBB    X1+04A2              ❹
```

❶ The base address of the program (determined from the map file) is virtual address 200. The base address is stored in base register 1 with ;X.

❷ DELTA/XDELTA displays the value in base register 1 just loaded, 200.

❸ The instruction at offset 490 is displayed in instruction mode using the ! command. The address reference is X1+490 (the + sign is implied when not specified). DELTA/XDELTA displays the instruction at address X1+490.

❹ The instruction at offset 499 is displayed. This instruction is a branch instruction. DELTA/XDELTA displays the address of the branch in offset notation.

O—Step Instruction over Subroutine

Executes one instruction, steps over a subroutine by executing it, and displays the instruction to which the subroutine returns control.

Format

O

Description

The Step Instruction over Subroutine command executes one instruction and displays the address of the next instruction. If the instruction executed is a call to a subroutine, the subroutine is executed and the next instruction displayed is the instruction to which the subroutine returns control. Use this command to do single-step instruction execution excluding single-stepping of instructions within subroutines. If you want to do single-step execution of all instructions, including those in subroutines, use the S command.

This command sets a flag to change the display mode to instruction mode. Any subsequent Close Current Location, Open Next (LINEFEED) commands and Open and Display Indirect Location (TAB) commands will display locations as machine instructions. The Open Location and Display Contents (/) command clears the flag, causing the display mode to revert to longword, hexadecimal mode.

On I64, the subroutine call instruction is br.call.

On Alpha, the subroutine call instructions are JSR and BSR.

On VAX, the subroutine call instructions are BSBB, BSBW, JSB, CALLG, and CALLS.

On all platforms, if you set a breakpoint in the subroutine and enter the O command, program execution breaks at the subroutine breakpoint. When you enter a Proceed command (;P), and program execution returns to the instruction to which the subroutine returns control, a message is displayed, as follows:

On I64 and Alpha systems:

```
Step-over at nnnnnnnn
instruction
```

On VAX systems:

```
STEPOVER BRK AT nnnnnnnn
instruction
```

The message informs you that program execution has returned from a subroutine.

If you are using XDELTA in a multiprocessor environment, the CPU ID of the processor where the break was taken is also displayed.

On I64 and Alpha, the CPU ID is displayed as a decimal number with no leading zeros.

On VAX, the CPU ID is displayed as a 2-digit hexadecimal number.

The following examples illustrate the command on each OpenVMS platform.

Examples

I64 example:

```
X0+00000380! mov         r7 = r23S              ❶
X0+00000381! nop.f       000000S
X0+00000382! br.call.sptk.many b0 = 0000E30 O ❷
X0+00000390! mov         r29 = r41S             ❸
X0+00000391! mov         r1 = r40S
```

❶ Program execution is currently at Base Register X0, plus offset 00000380. The instruction at X0+380 is a Move Application Register instruction. Step execution is then continued using the S command.

❷ Program execution is stopped at Base Register X0, plus offset 00000381. The instruction at offset 00000381 is a No Operation instruction. Step execution is then continued using the S command.

❸ Program execution is stopped at offset 00000382. The instruction at 00000382 is a "br.call" instruction. Execution is continued using the O command, thus skipping the routine(s) being called.

Alpha example:

```
30040;B        ❶
30070;B        ❷

;B
 1 00030040
 2 00030070

;P        ❸

Brk 1 at 00030040

00030040!       LDA             R27,#XFFC8(R2) O ❹
00030044!       BSR             R26,#X00000A O   ❺
Brk 2 at 00030070

00030070!       LDA             SP,#XFFD0(SP) ;P ❻

Step-over at 30048
00030048!       LDQ             R26,#X0048(R2) S ❼
0003004C!       BIS             R31,R31,R17
```

❶ A simple breakpoint is set in the main routine at address 30040, just prior to the subroutine call.

❷ A simple breakpoint is set in the subroutine at address 30070. The breakpoints are displayed using the ;B command.

❸ Program execution continues using ;P.

❹ Program execution stops at breakpoint 1. DELTA/XDELTA displays the breakpoint message and the instruction at the breakpoint address. The O command is used to single-step (DELTA/XDELTA recognizes that this is not a call instruction and turns it into a single-step instead).

❺ The next instruction is a subroutine call (BSR). The subroutine is stepped over using the O command.

❻ Ordinarily, the step-over would continue execution at the instruction following the subroutine call. However, in this case, program execution stops at breakpoint 2 inside the subroutine at address 30070. Program execution continues with the ;P command.

❼ The subroutine completes execution. DELTA/XDELTA displays a step-over break message that indicates that the O command has been completed, returning control at address 30048.

VAX example:

```
6D5;B                                                    ❶
;P                                                       ❷
1 BRK AT 000006D5
000006D5/CALLS    #0C,@#7FFEDE00              ;P        ❸
        PID=  0006      LOGINTIME=  12:50:29.45
2 BRK AT 00000699
00000699/BSBB    000006A2       ;P                       ❹
1 BRK AT 000006D5
000006D5/CALLS    #0C,@#7FFEDE00              ;P        ❺
        PID=  0007      LOGINTIME=  12:50:37.08
2 BRK AT 00000699
00000699/BSBB    000006A2       O                        ❻
1 BRK AT 000006D5
000006D5/CALLS    #0C,@#7FFEDE00              ;P        ❼
        PID=  0008      LOGINTIME=  12:50:45.64
STEPOVER BRK AT 0000069B                                 ❽
0000069B/BRB    X1+047A
```

❶ One breakpoint has been set at address 699 in the main routine. A simple breakpoint is set at 6D5 using ;B. This breakpoint is in a subroutine.

❷ Program execution continues using ;P.

❸ Program execution stops at breakpoint 1, which is in the subroutine. DELTA/XDELTA displays the breakpoint message and the instruction at the new breakpoint. Program execution continues using ;P.

❹ The subroutine completes and displays some output. Program execution continues until breakpoint 2. DELTA/XDELTA displays the breakpoint message and the breakpoint 2 instruction. Program execution continues with the ;P command.

❺ Program execution stops at breakpoint 1. Program execution continues with the ;P command. The subroutine completes execution and displays the output.

❻ Program execution stops at breakpoint 2. The subroutine is stepped over to the next instruction using the O command.

❼ Program execution stops at breakpoint 1 in the subroutine. Program execution continues using the ;P command.

❽ The subroutine completes execution and displays output. DELTA/XDELTA displays a STEPOVER break message that states the O command has been completed, returning control at address 69B.

S—Step Instruction

Executes one instruction and displays the next. If the executed instruction is a call to a subroutine, it steps into the subroutine and displays the next instruction to be executed in the subroutine.

Format

S

Description

The Step Instruction command executes one instruction and displays the next instruction (in instruction mode) and its address. Use this command to single-step instructions, including single-stepping all instructions in subroutines. If you want to exclude single-stepping instructions in subroutines, use the O command.

The instruction displayed has not yet been executed. This command sets a flag to change the display mode to instruction mode. Any subsequent Close Current Location, Open Next (LINEFEED) commands and Open and Display Indirect Location (TAB) commands will display locations as machine instructions. The Open Location and Display Contents (/) command clears the flag, causing the display mode to revert to longword, hexadecimal mode.

On I64, if the instruction is a br.call instruction, Step moves to the subroutine called by these instructions and displays the first instruction within the subroutine.

On Alpha, if the instruction being executed is a JSR or BSR instruction, Step moves to the subroutine called by these instructions and displays the first instruction within the subroutine.

Note

If DELTA/XDELTA does not have write access to the target of a JSR instruction, you cannot use the S or ;P command at the JSR instruction. First, you must use the O command; then you can use the S or ;P command.

On VAX, if the instruction being executed is a BSBB, BSBW, JSB, CALLG, or CALLS instruction, Step moves to the subroutine called by these instructions and displays the first instruction within the subroutine.

On Alpha and VAX, in general, you move to the instruction where you want to start single-step execution by placing a breakpoint at that instruction and typing ;P. Then press S to execute the first instruction and display the next one.

Examples

I64 example:

```
X0+00000061!      mov         r52 = b0 S  ❶
X0+00000062!      mov         r40 = r1 S  ❷
X0+00000070!      st8         [r12] = r0 ;;  ❸
```

❶ Program execution has been stopped at base register X0 plus offset 0000061. The instruction at this address is a Move Branch Register. Step execution is continued using the S command.

❷ Program execution is now stopped at base register X0 plus offset 0000062. The instruction at this address is a Move Application Register. Step execution is then continued using the S command.

❸ The instruction at offset 0000070 is displayed.

Alpha example:

```
0003003C!       BLBC            R0,#X000006 S   ❶
00030040!       LDQ             R16,#X0050(R2) S ❷
00030044!       BIS             R31,R31,R17 S   ❸
00030048!       LDQ             R26,#X0040(R2)
```

❶ Step program execution is started at address 3003C. The instruction at 3003C is a conditional branch instruction. Step execution is continued using the S command.

❷ Because the condition (BLBC) was not met, program execution continued at the next instruction at address 30040. Had the branch been taken, execution would have continued at address 30058. The second S command causes the LDQ instruction to be executed.

❸ The instruction at address 30044 is displayed. The S command is executed.

OpenVMS VAX example:

```
00000690/CMPL    R0,#000009A8    S    ❶
00000697/BEQL    0000069D        S    ❷
00000699/BSBB    000006A2        S    ❸
000006A2/PUSHL   R2                   ❹
```

❶ Step program execution is started at address 690. The instruction at 690 is executed and the next instruction is displayed. Step execution is continued using S.

❷ At address 697, there is a branch instruction to the instruction at address 69D. However, because the condition (BEQL) is not met, program execution continues at the next instruction. The next S command is executed.

❸ At address 699, there is a branch instruction to the instruction at address 6A2, a subroutine. The next S command is executed.

❹ Program execution moves to the subroutine.

A

Sample DELTA Debug Session on I64

This appendix gives an example of how you would use DELTA to debug a program executing on OpenVMS I64. The example C program named LOG uses the system service SYS$GETJPIW to obtain the PID, process name, and login time of each process. To run the example program without error, you need WORLD privilege.

Note

Although this example debugging session demonstrates using the DELTA debugger, you could use most of the commands in the example in an XDELTA debugging session as well.

This appendix consists of two sections:

- Section A.1 shows the source and machine listing files for the example C program.
- Section A.2 shows the example DELTA debugging session and explains the various commands used and information provided.

A.1 Listing File for C Example Program

This section shows the listing file for the C program, LOG, in two parts:

- Section A.1.1—C source code
- Section A.1.2—Machine code

See Section A.2 for the corresponding sample debugging session using this program.

A.1.1 Source Listing for I64 Debugging Example

Example A-1 shows the C source code for the example file, LOG.

Example A–1 Listing File for LOG: C Source Code

```
    1 #include <descrip.h>
  973 #include <jpidef.h>
 1378 #include <ssdef.h>
 5641 #include <starlet.h>
19024 #include <stdio.h>
20606 #include <stdlib.h>
22406
22407 void print_line(unsigned long int pid,
22408   char *process_name,
22409   unsigned long int *time_buffer);
22410
22411 typedef struct {
22412     unsigned short int il3_buffer_len;
22413     unsigned short int il3_item_code;
22414     void *il3_buffer_ptr;
22415     unsigned short int *il3_return_len_ptr;
22416 } item_list_3;
22417
22418 #define NUL '\0'
22419
22420 main(void)
22421 {
22422     static char name_buf[16];
22423     static unsigned long int pid, time_buf[2];
22424     static unsigned short int name_len;
22425
22426     unsigned short int pidadr[2] = {-1, -1};
22427     unsigned long int ss_sts;
22428     item_list_3 jpi_itmlst[] = {
22429  /* Get login time */
22430  { sizeof(time_buf),
22431    JPI$_LOGINTIM,
22432    (void *) time_buf,
22433    NULL
22434  },
22435
22436  /* Get process name */
22437  { sizeof(name_buf) - 1,
22438    JPI$_PRCNAM,
22439    (void *) name_buf,
22440    &name_len
22441  },
22442
22443  /* Get process ID (PID) */
22444  { sizeof(pid),
22445    JPI$_PID,
22446    (void *) &pid,
22447    NULL
22448  },
```

(continued on next page)

Example A–1 (Cont.) Listing File for LOG: C Source Code

```
22449
22450  /* End of list */
22451  { 0,
22452    0,
22453    NULL,
22454    NULL
22455  }
22456     };
22457
22458 /*
22459  * While there's more GETJPI information to process and a catastrophic
22460  * error has not occurred then
22461  *    If GETJPI was successful then
22462  *        NUL terminate the process name string and
22463  * print the information returned by GETJPI
22464  */
22465
22466     while((ss_sts = sys$getjpiw(0, &pidadr, 0, &jpi_itmlst, 0, 0, 0)) != SS$_NOMOREPROC &&
22467    ss_sts != SS$_BADPARAM &&
22468    ss_sts != SS$_ACCVIO) {
22469
22470         if (ss_sts == SS$_NORMAL) {
22471   *(name_buf + name_len) = NUL;
22472   print_line(pid, name_buf, time_buf);
22473  }
22474     }
22475     exit(EXIT_SUCCESS);
22476 }
22477
22478 void print_line(unsigned long int pid,
22479   char *process_name
22480   unsigned long int *time_buffer)
22481 {
22482     static char ascii_time[12];
22483
22484     struct dsc$descriptor_s time_dsc = {
22485      sizeof(ascii_time) - 1,
22486      DSC$K_DTYPE_T,
22487      DSC$K_CLASS_S,
22488      ascii_time
22489          1   22490     unsigned short int time_len;
22491
22492 /*
22493 Convert the logged in time to ASCII and NUL terminate it
22494 */
22495     sys$asctim(&time_len, &time_dsc, time_buffer, 1);
22496     *(ascii_time + time_len) = NUL;
22497
22498 /*
22499 Output the PID, process name and logged in time
22500 */
22501     printf("\n\tPID= %08.8X\t\tPRCNAM= %s\tLOGINTIM= %s",
22502     pid,
22503     process_name,
22504     ascii_time);
22505
22506     return;
22507 }
```

A.1.2 Machine Code Listing for I64 Debugging Example

Example A–2 through Example A–4 show machine code listings for the procedures in the example program, LOG.

Example A–2 Listing File for LOG: Machine Code _MAIN Procedure

```
   .psect $CODE$, CON, LCL, SHR, EXE, NOWRT, NOVEC, NOSHORT
   .proc   __MAIN
         .align 32
   .global   __MAIN
   .personality  DECC$$SHELL_HANDLER
   .handlerdata
  __MAIN:        // 02242
     { .mii
002C00F2EB40     0000      alloc   r45 = rspfs, 6, 9, 8, 0
010800C00080     0001      mov     r2 = sp  // r2 = r12
0120000A0380     0002      mov     r14 = 80 ;;
     }
     { .mmi
010028E183C0     0010      sub     r15 = sp, r14 ;; // r15 = r12, r14
0080C0F00380     0011      ld8     r14 = [r15]
010800F00300     0012      mov     sp = r15 ;; // r12 = r15
     }
     { .mii
000008000000     0020      nop.m   0
000188000B00     0021      mov     r44 = rp  // r44 = br0
010800100B80     0022      mov     r46 = gp ;; // r46 = r1
     }
   .
   .
   .
     { .mii
010802E00040     0100      mov     gp = r46  // r1 = r46
00015405A000     0101      mov.i   rspfs = r45
000E00158000     0102      mov     rp = r44 ;; // br0 = r44
     }
     { .mbb
010800CA0300     0110      adds    sp = 80, sp // r12 = 80, r12
000108001100     0111      br.ret.sptk.many rp // br0
004000000000     0112      nop.b   0 ;;
     }
   .endp   __MAIN
Routine Size: 288 bytes,    Routine Base: $CODE$ + 0000
```

Example A–3 Listing File for LOG: Machine Code MAIN Procedure

```
   .proc   MAIN
   .align 32
   .global  MAIN
  MAIN:       // 022420
     { .mii
002C00A22A00     0120      alloc   r40 = rspfs, 0, 10, 7, 0
010800C00080     0121      mov     r2 = sp  // r2 = = r12
012000080380     0122      mov     r14 = 64 ;;
     }
     { .mmi
010028E183C0     0130      sub     r15 = sp, r14 ;; // r5 = r12, r14
0080C0F00380     0131      ld8     r14 = [r15]
010800F00300     0132      mov     sp = r15 ;; // r12 = r15
     }
     { .mii
000008000000     0140      nop.m   0
0001880009C0     0141      mov     r39 = rp  // r39 = br0
010800100A40     0142      mov     r41 = gp ;; // r41 = r1
     }
   .
   .
   .
     { .mbb
01C4321401C0     0280      cmp4.eq pr7, pr6 = ss_sts, r33  // pr7, pr6 = r32, r33
008600018007     0281      (pr7)  br.cond.dpnt.many L$12
004000000000     0282      nop.b   0 ;;
     }
   .
   .
   .
     { .mib
0080C2B00AC0     0320      ld8.mov out1 = [r43], name_buf
012000100B00     0321      add     out2 = @gprel(time_buf), gp  // r44 = @gprel(time_buf), r1
00A000001000     0322      br.call.sptk.many rp = PRINT_LINE ;; // br0 = PRINT_LINE
     }
     { .bbb
0091FFFDD000     0330      br.sptk.many L$10    // 022473
004000000000     0331      nop.b   0
004000000000     0332      nop.b   0 ;;
     }
   .
   .
   .
     { .mii
000008000000     0370      nop.m   0
000E0014E000     0371      mov     rp = r39     // br0 = r39
010800C80300     0372      adds    sp = 64, sp ;;    // r12 = 64, r12
     }
     { .bbb
000108001100     0380      br.ret.sptk.many rp    // br0
004000000000     0381      nop.b   0
004000000000     0382      nop.b   0 ;;
     }
   .endp   MAIN
Routine Size: 624 bytes,    Routine Base: $CODE$ + 0120
```

Example A–4 Listing File for LOG: Machine Code PRINT_LINE Procedure

```
   .proc   PRINT_LINE
   .align 32
         .global  PRINT_LINE
  PRINT_LINE:       // 022478
     { .mii
002C0091A9C0     0390      alloc   r39 = rspfs, 3, 6, 4, 0
010800C00080     0391      mov     r2 = sp    // r2 = r12
012000020380     0392      mov     r14 = 16 ;;
     }
     { .mmi
010028E183C0     03A0      sub     r15 = sp, r14 ;;   // r15 = r12, r14
0080C0F00380     03A1      ld8     r14 = [r15]
010800F00300     03A2      mov     sp = r15 ;;   // r12 = r15
     }
   .
   .
   .
    { .mmi
012000100B00     0490      add     out3 = @ltoffx(ascii_time), gp ;; //r44 = @ltoffx(ascii_time), r1
0080C2C00B00     0491      ld8.mov out3 = [r44], ascii_time
012000008640     0492      mov     ai = 4 ;;       // r25 = 4
    }
    { .bbb
00A000001000     04A0      br.call.sptk.many rp = DECC$TXPRINTF // br0 = DECC$TXPRINTF
004000000000     04A1      nop.b   0
004000000000     04A2      nop.b   0 ;;
    }
    { .mii
010802800040     04B0      mov     gp = r40  // r1 = r40
00015404E000     04B1      mov.i   rspfs = r39       // 022506
000E0014C000     04B2      mov     rp = r38 ;;       // br0 = r38
    }
    { .mbb
010800C20300     04C0      adds    sp = 16, sp // r12 = 16, r12
000108001100     04C1      br.ret.sptk.many rp // br0
004000000000     04C2      nop.b   0 ;;
    }
   .endp   PRINT_LINE
Routine Size: 320 bytes,    Routine Base: $CODE$ + 0390
```

The .MAP file for the sample program is shown in Example A–5. Only the Program Section Synopsis with the psect, module, base address, end address, and length are listed.

Example A–5 .MAP File for the Sample Program

```
                                          +--------------------------+
                                          ! Program Section Synopsis !
                                          +--------------------------+

Psect Name      Module/Image      Base     End          Length
----------      ------------      ----     ---          ------

$BSS$                           00010000 0001001F 00000020 (         32.)
                LOG             00010000 0001001F 00000020 (         32.)

$CODE$                          00020000 0002061F 00000620 (       1568.)
                LOG             00020000 000204CF 000004D0 (       1232.)
                <Linker>        000204D0 0002061F 00000150 (        336.)

$LITERAL$                       00030000 00030058 00000059 (         89.)
                LOG             00030000 00030058 00000059 (         89.)

$READONLY$                      00030060 00030087 00000028 (         40.)
                LOG             00030060 00030087 00000028 (         40.)

$LINK$                          00040000 00040000 00000000 (          0.)
                LOG             00040000 00040000 00000000 (          0.)

$LINKER UNWIND$                 00040000 00040047 00000048 (         72.)
                LOG             00040000 00040047 00000048 (         72.)

$LINKER UNWINFO$                00040048 000400B7 00000070 (        112.)
                LOG             00040048 000400B7 00000070 (        112.)

.sbss                           00050000 00050013 00000014 (         20.)
                LOG             00050000 00050013 00000014 (         20.)

$LINKER SDATA$                  00060000 000600CF 000000D0 (        208.)
                <Linker>        00060000 000600CF 000000D0 (        208.)
```

A.2 Example DELTA Debugging Session on I64

The DELTA debugging session on OpenVMS I64 for the sample program is shown in the three example segments that follow.

DELTA Debugging Session Example on I64 - Part 1

In the first part of the example session, DELTA is enabled and the LOG program is invoked. The example shows version information displayed by DELTA and the use of several key Delta commands, including !, ;B, and ;P.

The callout list following the example provides details for this example segment.

Example A–6 DELTA Debugging Session on I64 - Part 1

```
$ DEFINE LIB$DEBUG SYS$SHARE:DELTA  ❶
$ RUN/DEBUG LOG  ❷
hp OpenVMS Industry Standard 64 DELTA Debugger  ❸

Brk 0 at 00020000

00020000!            alloc       r45 = ar.pfs, 0F, 08, 00 20000,1;X
00020000
X1 280!            cmp4.eq     p7, p6 = r32, r33 .;B  ❹

X1 322!            br.call.sptk.many b0 = 0000070 ;; .;B  ❺

;P

Brk 1 at X1+00000280  ❻

X1+00000280!            cmp4.eq     p7, p6 = r32, r33 R32/00000000 00000001 ;P

Brk 2 at X1+00000322

X1+00000322!            br.call.sptk.many b0 = 0000070 ;; O

        PID= 37E00401           PRCNAM= SWAPPER LOGINTIM= 00:00:00.00  ❼
```

❶ DELTA is enabled as the debugger.

❷ The example program LOG is invoked with DELTA.

❸ DELTA displays a banner and the first executable instruction. The base address of the program (determined from the .MAP file) is virtual address 20000. The base address is placed in base register 1 with the ;X command. Now, references to an address can use the address offset notation. For example, a reference to the first instruction in routine `main` is X1+0120. Also, DELTA displays some address locations as offsets to the base address.

❹ The instruction at address 20280 is displayed in instruction mode using the ! command. Its address location is expressed as the base address plus an offset. In the listing file, the offset is 280. (This is the point where the return status from SYS$GETJPIW is checked.) The base address in base address register X1 is 20000. The address reference, then, is X1+280. Note that the + sign is implied when not specified.

A simple breakpoint is set at that address using the ;B command. The address reference for ;B is the dot (.) symbol, representing the current address. (X1+280;B would have produced the same thing.)

❺ The same commands (that is, the ! command to view the instructions and the ;B command to set a breakpoint) are repeated for the instruction at offset 322. (This is the point at which the `print_line` function is called.)

❻ Program execution halts at the first breakpoint. DELTA displays the breakpoint message (Brk 1 at X1+00000280) with the breakpoint number 1 and the address at which the break occurred. The virtual address is 20280, which is the base address (20000) plus the offset 280. DELTA then displays the instruction in instruction mode (cmp4.eq p7, p6 = r32, r33). The contents of general register 32 are displayed with the forward slash (/) command (register 32 contains the value of the `ss_sts` variable). DELTA displays the contents of R32, which is 1. Program execution continues using the ;P command.

❼ The function `print_line` is executed and the output (PID, process name, and login time) is displayed.

DELTA Debugging Session Example on I64 - Part 2

In the second part of the example session, program execution continues and DELTA stops at the next breakpoint and displays information. User interaction allows DELTA to continue subsequent breakpoints. Use of the O command is demonstated to halt program execution and step over a routine call.

The callout list following the example provides details for this example segment.

Example A–7 DELTA Debugging Session on I64 - Part 2

```
X1+00000330!            br.many     1FFFEE0 ;P ❶
Brk 1 at X1+00000280
X1+00000280!            cmp4.eq     p7, p6 = r32, r33 ;P
Brk 2 at X1+00000322
X1+00000322!            br.call.sptk.many b0 = 0000070 ;; O   ❷
        PID= 37E00407       PRCNAM= CLUSTER_SERVER  LOGINTIM= 13:48:49.48
X1+00000330!            br.many     1FFFEE0 ;P
Brk 1 at X1+00000280
X1+00000280!            cmp4.eq     p7, p6 = r32, r33
;B
 1 X1+00000280
 2 X1+00000322
0,1;B
;B
 2 X1+00000322
;P
Brk 2 at X1+00000322   ❸
X1+00000322!            br.call.sptk.many b0 = 0000070 ;; O
        PID= 37E00408           PRCNAM= CONFIGURE       LOGINTIM= 13:48:52.06
X1+00000330!            br.many     1FFFEE0 ;P
Brk 2 at X1+00000322   ❹
X1+00000322!            br.call.sptk.many b0 = 0000070 ;; O
        PID= 37E00409           PRCNAM= USB$UCM_SERVER  LOGINTIM= 13:48:54.80
X1+00000330!            br.many     1FFFEE0 ;P
Brk 2 at X1+00000322   ❺
X1+00000322!            br.call.sptk.many b0 = 0000070 ;; X1 491! ld8 r44 = [r44]
Linefeed  ❻
X1+00000492!            mov         r25 = 000004 ;; Linefeed
X1+000004A0!            br.call.sptk.many b0 = 0000150 .;B   ❼
;B
 1 X1+000004A0
 2 X1+00000322
;P   ❽
```

(continued on next page)

Example A–7 (Cont.) DELTA Debugging Session on I64 - Part 2

```
Brk 1 at X1+000004A0    ❾

X1+000004A0!            br.call.sptk.many b0 = 0000150 O
        PID= 37E0040A           PRCNAM= LANACP  LOGINTIM= 13:48:54.84
X1+000004B0!            mov         r1 = r40 ;P

Brk 2 at X1+00000322

X1+00000322!            br.call.sptk.many b0 = 0000070 ;; ;P

Brk 1 at X1+000004A0    ❿

X1+000004A0!            br.call.sptk.many b0 = 0000150 O
        PID= 37E0040C           PRCNAM= FASTPATH_SERVER LOGINTIM= 13:48:55.01
X1+000004B0!            mov         r1 = r40 ;P

Brk 2 at X1+00000322

X1+00000322!            br.call.sptk.many b0 = 0000070 ;;

;B
 1 X1+000004A0
 2 X1+00000322

0,2;B

0,1;B

;B

;P
 PID= 37E0040D  PRCNAM= IPCACP LOGINTIM= 13:48:55.05
 PID= 37E0040E  PRCNAM= ERRFMT LOGINTIM= 13:48:55.14
 PID= 37E0040F  PRCNAM= CACHE_SERVER LOGINTIM= 13:48:55.19
 PID= 37E00410  PRCNAM= OPCOM LOGINTIM= 13:48:55.24
 PID= 37E00411  PRCNAM= AUDIT_SERVER LOGINTIM= 13:48:55.31
 PID= 37E00412  PRCNAM= JOB_CONTROL LOGINTIM= 13:48:55.39
 PID= 37E00414  PRCNAM= SECURITY_SERVER LOGINTIM= 13:48:55.84
 PID= 37E00415  PRCNAM= ACME_SERVER LOGINTIM= 13:48:55.88
 PID= 37E00416  PRCNAM= SMISERVER LOGINTIM= 13:49:02.26
 PID= 37E0041E  PRCNAM= NETACP LOGINTIM= 13:49:04.54
 PID= 37E0041F  PRCNAM= EVL LOGINTIM= 13:49:05.68
 PID= 37E00420  PRCNAM= REMACP LOGINTIM= 13:49:13.39
 PID= 37E00424  PRCNAM= TCPIP$INETACP LOGINTIM= 13:50:05.71
 PID= 37E00425  PRCNAM= TCPIP$PORTM_1 LOGINTIM= 13:50:08.40
 PID= 37E00426  PRCNAM= TCPIP$FTP_1 LOGINTIM= 13:50:08.77
 PID= 37E0042A  PRCNAM= LATACP LOGINTIM= 13:50:12.00
 PID= 37E008E5  PRCNAM= SYSTEM LOGINTIM= 13:32:01.42
 PID= 37E008E7  PRCNAM= JNELSON LOGINTIM= 13:41:17.48
$
```

❶ Program execution continues with the ;P command. DELTA stops at the next breakpoint.

❷ The O command halts program execution at the instruction where the function returns control (br.many 1FFFEE0). (This is the point at which control passes to checking the conditions of the while loop.) Program execution continues with ;P.

❸ Breakpoint 2 is encountered. DELTA displays the breakpoint message and the instruction. The function is executed with the O command and the function output is displayed. The next instruction where the function returns control is displayed. Program execution continues with the ;P command.

❹ Breakpoint 2 is encountered again. DELTA displays the breakpoint message and the instruction. The function is executed with the O command and the function output is displayed. The next instruction where the function returns control is displayed. Program execution continues with the ;P command.

❺ Breakpoint 2 is encountered again. The instruction at offset 491 (located in `print_line`) is displayed using the ! command. This instruction is part of the setup for the call to the `printf` function.

❻ Successive address locations are displayed by pressing the Linefeed key (CTRL/J) twice. These instructions are the remainder of the setup and the call to `printf`.

❼ A breakpoint at X1+4A0 (the current address) is set using the ;B command. This breakpoint is in the function `print_line`. The dot (.) symbol represents the current address. Note that breakpoint 1 was cleared earlier and is now reused by DELTA for the new breakpoint.

❽ Program execution continues with the ;P command.

❾ Program execution stops at the new breakpoint 1, which is in the `print_line` function. DELTA displays the breakpoint message and the instruction at the new breakpoint. The O command halts program execution at the instruction where the function returns control, stepping over the routine call. Program execution is continued with the ;P command.

❿ Program execution stops at breakpoint 1 in the `print_line` function. Program execution is continued using a combination of the O and ;P commands.

B

Sample DELTA Debug Session on Alpha

This appendix gives an example of how you would use DELTA to debug a program executing on OpenVMS Alpha. The example C program named LOG uses the system service SYS$GETJPIW to obtain the PID, process name, and login time of each process. To run the example program without error, you need WORLD privilege.

Note

Although this example debugging session demonstrates using the DELTA debugger, you could use most of the commands in the example in an XDELTA debugging session as well.

This appendix consists of two sections:

- Section B.1 shows the source and machine listing files for the example C program
- Section B.2 shows the example DELTA debugging session and explains the various commands used and information provided.

B.1 Listing File for C Example Program

This section shows the listing file for the C program, LOG, in two parts:

- Section B.1.1—C source code
- Section B.1.2—Machine code

See Section B.2 for the corresponding sample debugging session using this program.

B.1.1 Source Listing for Alpha Debugging Example

Example B-1 shows the C source code for the example file, LOG.

Example B–1 Listing File for LOG: C Source Code

```
1    #include <descrip.h>
434  #include <jpidef.h>
581  #include <ssdef.h>
1233 #include <starlet.h>
3784 #include <stdio.h>
4117 #include <stdlib.h>
4345
4346 void print_line(unsigned long int pid, char *process_name,
4347   unsigned long int *time_buffer);
4348
4349 typedef struct {
4350   unsigned short int il3_buffer_len;
4351   unsigned short int il3_item_code;
4352   void *il3_buffer_ptr;
4353   unsigned short int *il3_return_len_ptr;
4354         } item_list_3;
4355
4356 #define NUL '\0'
4357
4358 main()
4359   {
4360  static char name_buf[16];
4361  static unsigned long int pid, time_buf[2];
4362  static unsigned short int name_len;
4363
4364  unsigned short int pidadr[2] = {-1, -1};
4365  unsigned long int ss_sts;
4366  item_list_3 jpi_itmlst[] = {
4367          /* Get's login time */
4368          {sizeof(time_buf),
4369           JPI$_LOGINTIM,
4370           (void *) time_buf,
4371           NULL},
4372
4373          /* Get's process name */
4374          {sizeof(name_buf) - 1,
4375           JPI$_PRCNAM,
4376           (void *) name_buf,
4377           &name_len},
4378
4379          /* Get's process ID (PID) */
4380          {sizeof(pid),
4381           JPI$_PID,
4382           (void *) &pid,
4383           NULL},
4384
4385          /* End of list */
4386          {0,
4387           0,
4388           NULL,
4389           NULL}
4390         };
4391
```

(continued on next page)

Example B–1 (Cont.) Listing File for LOG: C Source Code

```
4392 /*
4393 While there's more GETJPI information to process and a catastrophic
4394 error has not occurred then
4395     If GETJPI was successful then
4396         NUL terminate the process name string and
4397  print the information returned by GETJPI
4398 */
4399
4400  while(
4401           (ss_sts = sys$getjpiw(0, &pidadr, 0, &jpi_itmlst, 0, 0, 0)) != SS$_NOMOREPROC &&
4402    ss_sts != SS$_BADPARAM &&
4403    ss_sts != SS$_ACCVIO)
4404    {
4405   if (ss_sts == SS$_NORMAL)
4406     {
4407    *(name_buf + name_len) = NUL;
4408    print_line(pid, name_buf, time_buf);
4409     }
4410    }
4411  exit(EXIT_SUCCESS);
4412   }
4413
4414 void print_line(unsigned long int pid, char *process_name,
4415   unsigned long int *time_buffer)
4416   {
4417  static char ascii_time[12];
4418
4419  struct dsc$descriptor_s time_dsc = {
4420           sizeof(ascii_time) - 1,
4421                  DSC$K_DTYPE_T,
4422                  DSC$K_CLASS_S,
4423                  ascii_time
4424                };
4425  unsigned short int time_len;
4426
4427 /*
4428 Convert the logged in time to ASCII and NUL terminate it
4429 */
4430  sys$asctim(&time_len, &time_dsc, time_buffer, 1);
4431  *(ascii_time + time_len) = NUL;
4432
```

(continued on next page)

Example B–1 (Cont.) Listing File for LOG: C Source Code

```
4433 /*
4434 Output the PID, process name and logged in time
4435 */
4436  printf("\n\tPID= %08.8X\t\tPRCNAM= %s\tLOGINTIM= %s", pid,
4437    process_name, ascii_time);
4438
4439  return;
4440   }
4441 _main(void *p1, void *p2, void *p3, void *p4, void *p5, void *p6)
4442 {
4443     void decc$exit(int);
4444     void decc$main(void *, void *, void *, void *, void *, void *, int *, void **, void **);
4445     int status;
4446     int argc;
4447     void *argv;
4448     void *envp;
4449
4450     decc$main(p1, p2, p3, p4, p5, p6, &argc, &argv, &envp);
4451
4452     status = main
4453                 (
4454
4455
4456
4457                 );
4458
4459     decc$exit(status);
4460 }
```

B.1.2 Machine Code Listing for Alpha Debugging Example

Example B–2 shows the machine code listing for the example program.

Example B–2 Listing File for LOG: Machine Code

```
                        .PSECT  $CODE, OCTA, PIC, CON, REL, LCL, SHR,-
                                EXE, NORD, NOWRT
0000    print_line::                                                ; 004414
0000            LDA     SP, -80(SP)             ; SP, -80(SP)
0004            MOV     1, R19                  ; 1, R19            ; 004430
0008            STQ     R27, (SP)               ; R27, (SP)         ; 004414
000C            MOV     4, R25                  ; 4, R25            ; 004430
0010            STQ     R26, 32(SP)             ; R26, 32(SP)       ; 004414
0014            STQ     R2, 40(SP)              ; R2, 40(SP)
0018            STQ     R3, 48(SP)              ; R3, 48(SP)
001C            STQ     R4, 56(SP)              ; R4, 56(SP)
0020            STQ     FP, 64(SP)              ; FP, 64(SP)
0024            MOV     SP, FP                  ; SP, FP
0028            MOV     R27, R2                 ; R27, R2
002C            STL     R17, process_name       ; R17, 16(FP)
0030            LDQ     R0, 40(R2)              ; R0, 40(R2)        ; 004419
0034            MOV     R16, pid                ; R16, R3           ; 004414
0038            LDQ     R26, 48(R2)             ; R26, 48(R2)       ; 004430
003C            LDA     R16, time_len           ; R16, 8(FP)
```

(continued on next page)

Example B–2 (Cont.) Listing File for LOG: Machine Code

```
0040            LDQ     R4, 32(R2)              ; R4, 32(R2)    ; 004423
0044            LDA     R17, time_dsc           ; R17, 24(FP)   ; 004430
0048            STQ     R0, time_dsc            ; R0, 24(FP)    ; 004419
004C            LDQ     R27, 56(R2)             ; R27, 56(R2)   ; 004430
0050            STL     R4, 28(FP)              ; R4, 28(FP)    ; 004419
0054            JSR     R26, SYS$ASCTIM         ; R26, R26      ; 004430
0058            LDL     R0, time_len            ; R0, 8(FP)     ; 004431
005C            MOV     pid, R17                ; R3, R17       ; 004436
0060            LDQ     R27, 88(R2)             ; R27, 88(R2)
0064            MOV     R4, R19                 ; R4, R19
0068            LDQ     R26, 80(R2)             ; R26, 80(R2)
006C            MOV     4, R25                  ; 4, R25
0070            ZEXTW   R0, R0                  ; R0, R0        ; 004431
0074            ADDQ    R4, R0, R0              ; R4, R0, R0
0078            LDQ_U   R16, (R0)               ; R16, (R0)
007C            MSKBL   R16, R0, R16            ; R16, R0, R16
0080            STQ_U   R16, (R0)               ; R16, (R0)
0084            LDQ     R16, 64(R2)             ; R16, 64(R2)   ; 004436
0088            LDL     R18, process_name       ; R18, 16(FP)
008C            JSR     R26, DECC$GPRINTF       ; R26, R26
0090            MOV     FP, SP                  ; FP, SP        ; 004439
0094            LDQ     R28, 32(FP)             ; R28, 32(FP)
0098            LDQ     R2, 40(FP)              ; R2, 40(FP)
009C            LDQ     R3, 48(FP)              ; R3, 48(FP)
00A0            LDQ     R4, 56(FP)              ; R4, 56(FP)
00A4            LDQ     FP, 64(FP)              ; FP, 64(FP)
00A8            LDA     SP, 80(SP)              ; SP, 80(SP)
00AC            RET     R28                     ; R28

Routine Size: 176 bytes,    Routine Base: $CODE + 0000
00B0     main::                                                 ; 004358
00B0            LDA     SP, -144(SP)            ; SP, -144(SP)
00B4            MOV     48, R17                 ; 48, R17       ; 004366
00B8            STQ     R27, (SP)               ; R27, (SP)     ; 004358
00BC            STQ     R26, 64(SP)             ; R26, 64(SP)
00C0            STQ     R2, 72(SP)              ; R2, 72(SP)
00C4            STQ     R3, 80(SP)              ; R3, 80(SP)
00C8            STQ     R4, 88(SP)              ; R4, 88(SP)
00CC            STQ     R5, 96(SP)              ; R5, 96(SP)
00D0            STQ     R6, 104(SP)             ; R6, 104(SP)
00D4            STQ     R7, 112(SP)             ; R7, 112(SP)
00D8            STQ     R8, 120(SP)             ; R8, 120(SP)
00DC            STQ     FP, 128(SP)             ; FP, 128(SP)
00E0            MOV     SP, FP                  ; SP, FP
00E4            MOV     R27, R2                 ; R27, R2
00E8            LDA     SP, -16(SP)             ; SP, -16(SP)
00EC            LDQ     R26, 40(R2)             ; R26, 40(R2)   ; 004366
00F0            LDQ     R18, 64(R2)             ; R18, 64(R2)
00F4            LDA     R16, jpi_itmlst         ; R16, 16(FP)
00F8            JSR     R26, OTS$MOVE           ; R26, R26
00FC            LDA     R6, jpi_itmlst          ; R6, 16(FP)    ; 004401
0100            LDQ     R3, -64(R2)             ; R3, -64(R2)   ; 004370
0104            LDA     R7, pidadr              ; R7, 8(FP)     ; 004401
0108            LDQ     R0, 32(R2)              ; R0, 32(R2)    ; 004364
010C            MOV     2472, R8                ; 2472, R8      ; 004401
0110            STL     R0, pidadr              ; R0, 8(FP)     ; 004364
0114            LDA     R3, time_buf            ; R3, 16(R3)    ; 004370
```

(continued on next page)

Example B–2 (Cont.) Listing File for LOG: Machine Code

```
0118            MOV     R3, R5                  ; R3, R5
011C            STL     R5, 20(FP)              ; R5, 20(FP)    ; 004366
0120            LDA     R4, 8(R3)               ; R4, 8(R3)     ; 004376
0124            STL     R4, 32(FP)              ; R4, 32(FP)    ; 004366
0128            LDA     R17, 24(R3)             ; R17, 24(R3)
012C            STL     R17, 36(FP)             ; R17, 36(FP)
0130            LDA     R19, 28(R3)             ; R19, 28(R3)
0134            STL     R19, 44(FP)             ; R19, 44(FP)
0138    L$6:                                                    ; 004400
0138            LDQ     R26, 48(R2)             ; R26, 48(R2)   ; 004401
013C            CLR     R16                     ; R16
0140            LDQ     R27, 56(R2)             ; R27, 56(R2)
0144            MOV     R7,  R17                ; R7, R17
0148            STQ     R31, (SP)               ; R31, (SP)
014C            CLR     R18                     ; R18
0150            MOV     R6, R19                 ; R6, R19
0154            CLR     R20                     ; R20
0158            CLR     R21                     ; R21
015C            MOV     7, R25                  ; 7, R25
0160            JSR     R26, SYS$GETJPIW        ; R26, R26
0164            CMPEQ   ss_sts, 20, R16         ; R0, 20, R16   ; 004402
0168            CMPEQ   ss_sts, R8, R17         ; R0, R8, R17   ; 004401
016C            CMPEQ   ss_sts, 12, R18         ; R0, 12, R18   ; 004403
0170            BIS     R17, R16, R17           ; R17, R16, R17 ; 004401
0174            BIS     R17, R18, R18           ; R17, R18, R18
0178            BNE     R18, L$10               ; R18, L$10     ; 004400
017C            CMPEQ   ss_sts, 1, R0           ; R0, 1, R0     ; 004405
0180            BEQ     R0, L$6                 ; R0, L$6
0184            MOV     R4, R17                 ; R4, R17       ; 004408
0188            LDQ_U   R19, 24(R3)             ; R19, 24(R3)   ; 004407
018C            MOV     R5, R18                 ; R5, R18       ; 004408
0190            LDA     R27, -96(R2)            ; R27, -96(R2)
0194            EXTWL   R19, R3, R19            ; R19, R3, R19  ; 004407
0198            ADDQ    R4, R19, R19            ; R4, R19, R19
019C            LDQ_U   R22, (R19)              ; R22, (R19)
01A0            MSKBL   R22, R19, R22           ; R22, R19, R22
01A4            STQ_U   R22, (R19)              ; R22, (R19)
01A8            LDL     R16, 28(R3)             ; R16, 28(R3)   ; 004408
01AC            BSR     R26, print_line         ; R26, print_line
01B0            BR      L$6                     ; L$6           ; 004405
01B4            NOP                             ;
01B8    L$10:                                                   ; 004400
01B8            LDQ     R26, 80(R2)             ; R26, 80(R2)   ; 004411
01BC            CLR     R16                     ; R16
01C0            LDQ     R27, 88(R2)             ; R27, 88(R2)
01C4            MOV     1, R25                  ; 1, R25
01C8            JSR     R26, DECC$EXIT          ; R26, R26
01CC            MOV     FP, SP                  ; FP, SP        ; 004412
01D0            LDQ     R28, 64(FP)             ; R28, 64(FP)
01D4            MOV     1, R0                   ; 1, R0
01D8            LDQ     R2, 72(FP)              ; R2, 72(FP)
01DC            LDQ     R3, 80(FP)              ; R3, 80(FP)
01E0            LDQ     R4, 88(FP)              ; R4, 88(FP)
01E4            LDQ     R5, 96(FP)              ; R5, 96(FP)
```

(continued on next page)

Example B–2 (Cont.) Listing File for LOG: Machine Code

```
01E8            LDQ     R6, 104(FP)             ; R6, 104(FP)
01EC            LDQ     R7, 112(FP)             ; R7, 112(FP)
01F0            LDQ     R8, 120(FP)             ; R8, 120(FP)
01F4            LDQ     FP, 128(FP)             ; FP, 128(FP)
01F8            LDA     SP, 144(SP)             ; SP, 144(SP)
01FC            RET     R28                     ; R28

Routine Size: 336 bytes,    Routine Base: $CODE + 00B0
0200    __main::                                                ; 004441
0200            LDA     SP, -48(SP)             ; SP, -48(SP)
0204            MOV     9, R25                  ; 9, R25        ; 004450
0208            STQ     R27, (SP)               ; R27, (SP)     ; 004441
020C            STQ     R26, 24(SP)             ; R26, 24(SP)
0210            STQ     R2, 32(SP)              ; R2, 32(SP)
0214            STQ     FP, 40(SP)              ; FP, 40(SP)
0218            MOV     SP, FP                  ; SP, FP
021C            LDA     SP, -32(SP)             ; SP, -32(SP)
0220            MOV     R27, R2                 ; R27, R2
0224            LDA     R0, argc                ; R0, 16(FP) ; 004450
0228            LDQ     R26, 48(R2)             ; R26, 48(R2)
022C            LDA     R1, argv                ; R1, 12(FP)
0230            STQ     R0, (SP)                ; R0, (SP)
0234            LDA     R0, envp                ; R0, 8(FP)
0238            STQ     R1, 8(SP)               ; R1, 8(SP)
023C            LDQ     R27, 56(R2)             ; R27, 56(R2)
0240            STQ     R0, 16(SP)              ; R0, 16(SP)
0244            JSR     R26, DECC$MAIN          ; R26, R26
0248            LDA     R27, -96(R2)            ; R27, -96(R2) ; 004452
024C            BSR     R26, main               ; R26, main
0250            LDQ     R27, 40(R2)             ; R27, 40(R2)   ; 004459
0254            MOV     status, R16             ; R0, R16
0258            MOV     1, R25                  ; 1, R25
025C            LDQ     R26, 32(R2)             ; R26, 32(R2)
0260            JSR     R26, DECC$EXIT          ; R26, R26
0264            MOV     FP, SP                  ; FP, SP        ; 004460
0268            LDQ     R28, 24(FP)             ; R28, 24(FP)
026C            LDQ     R2, 32(FP)              ; R2, 32(FP)
0270            LDQ     FP, 40(FP)              ; FP, 40(FP)
0274            LDA     SP, 48(SP)              ; SP, 48(SP)
0278            RET     R28                     ; R28

Routine Size: 124 bytes,    Routine Base: $CODE + 0200
```

The .MAP file for the sample program is shown in Example B–3. Only the Program Section Synopsis with the psect, module, base address, end address, and length are listed.

Example B–3 .MAP File for the Sample Program

```
                                          +--------------------------+
                                          ! Program Section Synopsis !
                                          +--------------------------+

Psect Name      Module Name       Base     End          Length
----------      -----------       ----     ---          ------

$LINKAGE                        00010000 000100FF 00000100 (       256.)
                LOG             00010000 000100FF 00000100 (       256.)

$LITERAL                        00010100 00010158 00000059 (        89.)
                LOG             00010100 00010158 00000059 (        89.)

$READONLY                       00010160 00010160 00000000 (         0.)
                LOG             00010160 00010160 00000000 (         0.)

$INIT                           00020000 00020000 00000000 (         0.)
                LOG             00020000 00020000 00000000 (         0.)

$UNINIT                         00020000 0002002F 00000030 (        48.)
                LOG             00020000 0002002F 00000030 (        48.)

$CODE                           00030000 0003027B 0000027C (       636.)
                LOG             00030000 0003027B 0000027C (       636.)
```

B.2 Example DELTA Debugging Session on Alpha

The DELTA debugging session on OpenVMS Alpha for the sample program is shown in the three example segments that follow.

B.2.1 DELTA Debugging Session Example on Alpha - Part 1

In the first part of the example session, DELTA is enabled and the LOG program is invoked. The example shows version information displayed by DELTA and the use of the ;B and ;P commands.

The callout list following the example provides details for this example segment.

Example B–4 DELTA Debugging Session on Alpha- Part 1

```
$ DEFINE LIB$DEBUG SYS$LIBRARY:DELTA  ❶
$ RUN/DEBUG LOG  ❷
Alpha/VMS DELTA Version 1.5    ❸

Brk 0 at 00030200

00030200!    LDA         SP,#XFFD0(SP)  30000,1;X
X1 164! CMPEQ           R0,#X14,R16  .;B ❹

X1 1AC! BSR             R26,#XFFFF94  .;B  ❺
```

❶ DELTA is enabled as the debugger.

❷ The example program LOG is invoked with DELTA.

❸ DELTA displays a version number and the first executable instruction. The base address of the program (determined from the map file) is virtual address 30000. The base address is placed in base register 1 with ;X. Now references to an address can use the address offset notation. For example, a reference to the first instruction is X1+200 (or the base address 30000 + offset 200). Also, DELTA displays some address locations as offsets to the base address.

❹ The instruction at address 30164 is displayed in instruction mode using !. Its address location is expressed as the base address plus an offset. In the listing file, the offset is 164. (This is the point where the return status from SYS$GETJPIW is checked.) The base address in base address register X1 is 30000. The address reference, then, is X1+164. Note the + sign is implied when not specified.

A simple breakpoint is set at that address using the ;B command. The address reference for ;B is the . symbol, representing the current address. X1+164;B would have done the same thing.

❺ The ! command to view the instruction and ;B to set a breakpoint are repeated for the instruction at offset 1AC. (This is the point at which the print_line function is called.)

B.2.2 DELTA Debugging Session Example on Alpha - Part 2

In the second part of the example session, program execution continues with ;P, then halts at the first breakpoint and displays information. User interaction allows DELTA to continue subsequent breakpoints.

The callout list following the example provides details for this example segment.

Example B–5 DELTA Debugging Session on Alpha - Part 2

```
;P ❶
Brk 1 at 00030164    ❷
X1+00000164!    CMPEQ           R0,#X14,R16  R0/ 00000001 ;P
Brk 2 at 000301AC
X1+000001AC!    BSR             R26,#XFFFF94  O
        PID= 00000021           PRCNAM= SWAPPER LOGINTIM= 00:00:00.00  ❸
X1+000001B0!    BR              R31,#XFFFFE1  ;P
Brk 1 at 00030164
X1+00000164!    CMPEQ           R0,#X14,R16  R0/ 00000001 ;P
```

(continued on next page)

Example B–5 (Cont.) DELTA Debugging Session on Alpha - Part 2

```
Brk 2 at 000301AC

X1+000001AC!    BSR             R26,#XFFFF94  O ❹
        PID= 00000024           PRCNAM= ERRFMT  LOGINTIM= 16:24:01.03
X1+000001B0!    BR              R31,#XFFFFE1  ;P

Brk 1 at 00030164

X1+00000164!    CMPEQ           R0,#X14,R16

;B
 1 00030164
 2 000301AC

0,1;B

;B
 2 000301AC

;P

Brk 2 at 000301AC  ❺

X1+000001AC!    BSR             R26,#XFFFF94  O
        PID= 00000025           PRCNAM= OPCOM   LOGINTIM= 16:24:02.56
X1+000001B0!    BR              R31,#XFFFFE1  ;P

Brk 2 at 000301AC  ❻

X1+000001AC!    BSR             R26,#XFFFF94  O
      PID= 00000026          PRCNAM= AUDIT_SERVER    LOGINTIM=16:24:03.66
X1+000001B0!    BR              R31,#XFFFFE1  ;P

Brk 2 at 000301AC  ❼

X1+000001AC!    BSR         R26,#XFFFF94 X1 84!  LDQ   R16,#X0040(R2)
```

❶ The ;P command lets you proceed from the breakpoint.

❷ Program execution halts at the first breakpoint. DELTA displays the breakpoint message (Brk 1 at 00030164) with the breakpoint number 1 and the virtual address. The virtual address is 30164, which is the base address (30000) plus the offset 164. DELTA then displays the instruction in instruction mode (CMPEQ R0,#X14,R16). The contents of the general register 0 are displayed with the / command. DELTA displays the contents of R0, which is 1. Program execution continues using the ;P command.

❸ The function print_line is executed, and the output (PID, process name, and login time) is displayed.

❹ The O command halts program execution at the instruction where the function returns control (BR R31,#XFFFFE1). (This is the point at which control passes to checking the conditions of the while loop.) Program execution continues with ;P.

❺ Breakpoint 2 is encountered. DELTA displays the breakpoint message, and the instruction. The function is executed with the O command and the function output is displayed. The next instruction where the function returns control is displayed. Program execution continues with the ;P command.

❻ Breakpoint 2 is encountered again. DELTA displays the breakpoint message, and the instruction. The function is executed with the O command and the function output is displayed. The next instruction where the function returns control is displayed. Program execution continues with the ;P command.

❼ Breakpoint 2 is encountered again. The instruction at offset 84 (in print_line) is displayed using !. This instruction is part of the setup for the call to the printf function.

B.2.3 DELTA Debugging Session Example on Alpha - Part 3

In the third part of the example session, successive address locations are specified when the user presses Linefeed. Another breakpoint is set, and program execution continues. DELTA stops at the break point, and the ;O command is used to halt execution and step over a routine call. Program execution continues through more breakpoints to a final exit.

The callout list following the example provides details for this example segment.

Example B–6 DELTA Debugging Session Example on Alpha - Part 3

```
Linefeed ❶

X1+00000088!    LDL             R18,#X0010(FP)   Linefeed

X1+0000008C!    JSR             R26,(R26)  .;B ❷

;B
 1 0003008C
 2 000301AC

;P ❸

Brk 1 at 0003008C    ❹

X1+0000008C!    JSR             R26,(R26)  0
     PID= 00000027          PRCNAM= JOB_CONTROL     LOGINTIM= 16:24:06.83
X1+00000090!    BIS             R31,FP,SP  ;P

Brk 2 at 000301AC

X1+000001AC!    BSR             R26,#XFFFF94  ;P
Brk 1 at 0003008C ❺

X1+0000008C!    JSR             R26,(R26)  0
     PID= 00000028          PRCNAM= NETACP  LOGINTIM= 16:24:22.86
X1+00000090!    BIS             R31,FP,SP  ;P

Brk 2 at 000301AC

X1+000001AC!    BSR             R26,#XFFFF94

;B
 1 0003008C
 2 000301AC

0,2;B

0,1;B

;B

;P
```

(continued on next page)

Example B–6 (Cont.) DELTA Debugging Session Example on Alpha - Part 3

```
        PID= 00000029        PRCNAM= EVL     LOGINTIM= 16:24:26.67
        PID= 0000002A        PRCNAM= REMACP  LOGINTIM= 16:24:38.21
        PID= 0000002B        PRCNAM= LATACP  LOGINTIM= 16:24:43.18
        PID= 0000004C        PRCNAM= GODDARD LOGINTIM= 07:40:49.34
        PID= 0000002D        PRCNAM= SYMBIONT_0001   LOGINTIM= 16:25:47.54
        PID= 0000002F        PRCNAM= MCCORMICK       LOGINTIM= 16:27:45.27 ❻
Exit 00000001       ❼

8002228C!       ADDL            R15,SP,SP EXIT ❽
```

❶ Successive address locations are displayed by pressing the Linefeed key two times. These instructions are the remainder of the setup and the call to printf.

❷ A breakpoint at X1+8C (the current address) is set using the ;B command. This breakpoint is in the function print_line. The . symbol represents the current address. Note that breakpoint 1 was cleared earlier and is now reused by DELTA for the new breakpoint.

❸ Program execution continues with the ;P command.

❹ Program execution stops at the new breakpoint 1, which is in the print_line function. DELTA displays the breakpoint message and the instruction at the new breakpoint. The O command halts program execution at the instruction where the function returns control, stepping over the routine call. Note the O command must be used in this case, as opposed to the ;P command, because the printf function resides in read-only protected memory. Program execution is continued with the ;P command.

❺ Program execution stops at breakpoint 1 in the print_line function. Program execution is continued using a combination of the O and ;P commands.

❻ All current process login times are displayed.

❼ Final exit status is displayed.

❽ The DELTA EXIT command is entered to terminate the debugging session and leave DELTA.

C

Sample DELTA Debug Session on VAX

This appendix provides an example of how you would use DELTA to debug a program executing on OpenVMS VAX. The example program, named LOGINTIM, uses the system service SYS$GETJPI to obtain the login times of each process. To run the example program without error, you need WORLD privilege.

Note

Although this example debugging session demonstrates using the DELTA debugger, you could use most of the commands in the example in an XDELTA debugging session as well.

This appendix consists of two sections:

- Section C.1 shows the source and machine listing files for the example program
- Section C.2 shows the example DELTA debugging session and explains the various commands used and information provided.

C.1 Listing Files for Example Program

This section shows the listing files for the example program, LOGINTIM, in two parts:

- Section C.1.1—Listing file for example source code
- Section C.1.2—Map file program section synopsis

See Section C.2 for the corresponding sample debugging session using this program.

C.1.1 Source Listing for VAX Debugging Example

The .LIS file for the DELTA debugging example on OpenVMS VAX is shown in Example C–1. Only the offsets and source code are shown.

Example C–1 Program for Getting LOGINTIMs

```
0000     1 ;++
0000     2 ; This sample program uses the wildcard feature of GETJPI to get the
0000     3 ; LOGINTIM for each active process.  It outputs the PID and LOGINTIM
0000     4 ; for each and exits when there are NOMOREPROCs.
0000     5 ;--
0000     6
0000     7 ;
0000     8 ; Data areas.
0000     9 ;
0000    10 DEVNAM: .ASCID /SYS$OUTPUT/         ;Output device specifier
000E
0012    11
0012    12 CHAN:   .LONG  0                    ;Assigned output channel
0016    13
0016    14 ITMLST:                             ;Item list for GETJPI call
0016    15         .WORD  8                    ; Byte length of output buffer
0018    16         .WORD  JPI$_LOGINTIM        ; Specify LOGINTIM item code
001A    17         .ADDRESS      TIME          ; Address of output buffer
001E    18         .LONG  0                    ; Not interested in return length
0022    19         .LONG  0                    ;Item list terminator
0026    20
0026    21 TIME:   .QUAD  0                    ;Buffer to hold LOGINTIM
002E    22
002E    23 OUTLEN: .LONG  0                    ;FAO buffer length
0032    24 OUTBUF: .LONG  1024                 ;FAO buffer descriptor
0036    25         .ADDRESS BUF
003A    26 BUF:    .BLKB  1024                 ;FAO buffer
043A    27
043A    28 CTRSTR: .ASCID *!/!_PID= !XW!_LOGINTIME= !%T* ;FAO control string
0448
0454
045E    29
045E    30 PIDADR: .LONG  -1                   ;Wildcard PID control longword
0462    31
0462    32 ;++
0462    33 ; Start of program.
0462    34 ;--
0462    35 S:       .WORD   0                  ;Entry mask
0464    36          $ASSIGN_S   DEVNAM,CHAN    ;Assign output channel
0475    37          MOVAB   TIME,R2            ;Load pointer to LOGINTIM
047A    38                                     ;     output buffer
047A    39 LOOP:    $GETJPI_S   ITMLST=ITMLST,-;Get LOGINTIM for a process
047A    40                  PIDADR=PIDADR
0490    41          CMPL    R0,#SS$_NOMOREPROC ;Are we done?
0497    42          BEQL    5$                 ;If EQL yes
0499    43          BSBB    GOT_IT             ;Process data for this process
049B    44          BRB     LOOP               ;Look for another process
049D    45
049D    46 5$:      MOVZBL  #SS$_NORMAL,R0     ;Set successful completion code
04A1    47          RET                        ;Return, no more processes
04A2    48
04A2    49 GOT_IT:  $FAO_S  CTRSTR,-           ;Format the output data
04A2    50                  OUTLEN,-
04A2    51                  OUTBUF,-
04A2    52                  PIDADR,R2
```

(continued on next page)

Example C–1 (Cont.) Program for Getting LOGINTIMs

```
04B9     53          $QIOW_S CHAN=CHAN,-        ;Output to SYS$OUTPUT
04B9     54                  FUNC=#IO$_WRITEVBLK,-
04B9     55                       P1=BUF,-
04B9     56                       P2=OUTLEN
04DC     57           RSB                       ;Done with this process data
04DD     58
04DD     59           .END S
```

C.1.2 Map File for VAX Debugging Example

The .MAP file is shown in Example C–2. Only the Program Section Synopsis with the PSECT, MODULE, base address, end address, and length are listed.

Example C–2 LOGINTIM Program .Map File

```
                          +--------------------------+
                          ! Program Section Synopsis !
                          +--------------------------+

Psect Name      Module Name       Base     End          Length
----------      -----------       ----     ---          ------
. BLANK .                      00000200 000006E2 000004E3 (      1251.)
                .MAIN.         00000200 000006E2 000004E3 (      1251.)
```

C.2 Example DELTA Debugging Session on VAX

The DELTA debugging session on OpenVMS VAX for the sample program is shown in the four example segments that follow.

C.2.1 DELTA Debugging Session Example on VAX - Part 1

In the first part of the example session, DELTA is enabled and the LOGINTM program is invoked. The example shows version information displayed by DELTA and the use of the ;B and ;P commands.

The callout list following the example provides details for this example segment.

Example C–3 DELTA Debugging Session Example on VAX - Part 1

```
$ DEFINE LIB$debugging SYS$LIBRARY:DELTA ❶
$ RUN/debugging LOGINTIM ❷
DELTA Version 6.0

00000664/CLRQ    -(SP)    200,1;X ❸
00000200 ❹

X1 490!CMPL    R0,#000009A8   .;B  ❺
X1 499!BSBB    X1+04A2        .;B  ❻
```

❶ DELTA is enabled as the debugger.

❷ The example program LOGINTIM is invoked with DELTA.

❸ DELTA displays a version number and the first executable instruction. The base address of the program (determined from the map file) is virtual address 200. The base address is placed in base register 1 with ;X. Now references to an address can use the address offset notation. For example, a reference to the first instruction is X1+464 (or base address 200 + offset 464). Also, DELTA displays some address locations as offsets to the base address.

❹ DELTA displays the value in base register 1, just loaded 200.

❺ The instruction at address 690 is displayed in instruction mode using !. Its address location is expressed as the base address plus an offset. In the listing file, the offset is 490. The base address in base register X1 is 200. The address reference, then, is X1+490. (Note that the + sign is implied when not specified.)

A simple breakpoint is set at that address using the ;B command. The address reference for ;B is the . symbol, representing the current address. X1+490;B would have done the same thing.

❻ The same commands (! command to view the instruction and ;B to set a breakpoint) are repeated for the instruction at offset 499. When DELTA displays the instruction (BSBB GOT_IT), it displays the destination of the branch (GOT_IT) as the address location. DELTA displays the value as an offset to base register 1.

C.2.2 DELTA Debugging Session Example on VAX - Part 2

In the second part of the example session, program execution begins. DELTA halts at the first breakpoint and displays information. User interaction allows DELTA to continue to the next breakpoint.

The callout list following the example provides details for this example segment.

Example C–4 DELTA Debugging Session Example on VAX - Part 2

```
;P ❶
1 BRK AT 00000690
X1+0490/CMPL    R0,#000009A8  R0/00000001 ;P ❷
2 BRK AT 00000699
X1+499/BSBB    X1+04A2       O ❸
        PID=  0000      LOGINTIME=  00:00:00.00  ❹
X1+049B/BRB    X1+047A        ;P ❺
1 BRK AT 00000690
X1+0490/CMPL    R0,#000009A8  R0/00000001 ;P ❻
2 BRK AT 00000699
X1+0499/BSBB    X1+04A2       O ❼
        PID=  0001      LOGINTIME=  00:00:00.00
X1+049B/BRB    X1+047A        ;P
1 BRK AT 00000690
X1+0490/CMPL    R0,#000009A8   ❽
;B ❾
1  00000690
2  00000699 ❿
```

❶ Program execution begins with the ;P command.

❷ Program execution halts at the first breakpoint. DELTA displays the breakpoint message (1 BRK AT 00000690) with the breakpoint number 1 and the virtual address. The virtual address is 00000690, which is the base address (200) plus the offset 490. DELTA then displays the instruction in instruction mode (CMPL R0,#000009A8). The contents of general register 0 are displayed with the / command. DELTA displays the contents of R0, which is 1. Program execution continues using the ;P command.

❸ Program execution halts at breakpoint 2. DELTA displays the breakpoint message, then the instruction. Step-instruction execution, excluding instructions in subroutines, is initiated with O.

❹ The subroutine GOT_IT is executed, and the output (PID and login time) is displayed.

❺ The O command halts program execution at the instruction where the subroutine returns control (BRB LOOP). DELTA displays the instruction in instruction mode (BRB X1+047A), where X1+047A is the address of the first instruction in LOOP. Program execution continues with ;P.

❻ Breakpoint 1 is encountered again; DELTA displays the breakpoint message and the instruction. The contents of R0 are examined (/ command) and program execution continues (;P).

❼ Breakpoint 2 is encountered again; DELTA displays the breakpoint message and the instruction. The subroutine is stepped over again with the O command. The subroutine is executed, and the output is displayed. The instruction where the subroutine returns control is displayed. Program execution continues (;P command).

❽ Breakpoint 1 is encountered; DELTA displays the breakpoint message and the instruction.

❾ All breakpoints in the program are listed with the ;B command.

❿ DELTA displays the breakpoints (by breakpoint number) and the address locations.

C.2.3 DELTA Debugging Session Example on VAX - Part 3

In the third part of the example session, the first breakpoint is cleared, then all breakpoints are listed. The program continues until the next breakpoint is encountered, and the user sets a new breakpoint.

The callout list following the example provides details for this example segment.

Example C–5 DELTA Debugging Session Example on VAX - Part 3

```
0,1;B   ❶
;B  ❷
2  00000699  ❸
;P  ❹
2 BRK AT 00000699
X1+0499/BSBB    X1+04A2       O
        PID=  0004      LOGINTIME=  12:50:20.40
X1+049B/BRB    X1+047A        ;P  ❺
2 BRK AT 00000699
X1+0499/BSBB    X1+04A2       ;P
        PID=  0005      LOGINTIME=  12:50:25.61  ❻
2 BRK AT 00000699
X1+0499/BSBB    X1+04A2       X1 4B9!CLRQ    -(SP)  ❼
[Linefeed]  ❽
X1+04BB/CLRQ    -(SP)   [Linefeed]
X1+04BD/PUSHL    X1+002E   [Linefeed]
X1+04C1/PUSHAL    X1+003A   [Linefeed]
X1+04C5/CLRQ    -(SP)   [Linefeed]
X1+04C7/PUSHL    #00   [Linefeed]
X1+04C9/MOVZWL    #0030;-(SP)   [Linefeed]
X1+04CE/MOVZWL    X1+0012,-(SP)   [Linefeed]
X1+04D3/PUSHL     #00   [Linefeed]
X1+04D5/CALLS    #0C,@#7FFEDE00              .;B  ❾
;B  ❿
1  000006D5
2  00000699
```

❶ Breakpoint 1 is cleared using 0,[breakpoint #];B. (Never clear breakpoint 1 in XDELTA.)

❷ All breakpoints are listed again with ;B command.

❸ DELTA displays breakpoint 2 (breakpoint 1 cleared).

❹ Program execution continues using the ;P command.

❺ Breakpoint 2 is encountered; DELTA displays the breakpoint message and the instruction. The subroutine is executed with the O command and the subroutine output is displayed. The next instruction where the subroutine returns control is displayed. Program execution continues with the ;P command.

❻ Breakpoint 2 is encountered; DELTA displays the breakpoint message and the instruction. Program execution continues to the next breakpoint with the ;P command. The subroutine is executed, and the subroutine output is displayed.

❼ Breakpoint 2 is encountered again; the instruction at offset 4B9 (in the subroutine) is displayed using !. This instruction is part of the setup for the call to the system service $QIOW.

❽ Successive address locations are displayed by pressing the Linefeed key nine times. These instructions are the remainder of the setup and the call to the system service $QIOW.

❾ A breakpoint at X1+04D5 (the current address) is set using the ;B command. This breakpoint is in the subroutine. The . symbol represents the current address.

❿ The current breakpoints in the program are listed. The new breakpoint is assigned breakpoint 1.

C.2.4 DELTA Debugging Session Example on VAX - Part 4

In the final part of the example session, program execution continues and stops at the new breakpoint set in the previous example segment. DELTA executes the subroutine where the breakpoint was encountered and displays the output. The next breakpoint is reached and the use enters the ;O command to step over the subroutine. When there are no more breakpoints, the program completes and final exit status is displayed.

The callout list following the example provides details for this example segment.

Example C–6 DELTA Debugging Session Example on VAX - Part 4

```
;P ❶
1 BRK AT 000006D5
X1+04D5/CALLS    #0C,@#7FFEDE00              ;P ❷
        PID=  0006      LOGINTIME=  12:50:29.45
2 BRK AT 00000699
X1+0499/BSBB    X1+04A2       ;P ❸
1 BRK AT 000006D5
X1+04D5/CALLS    #0C,@#7FFEDE00              ;P ❹
        PID=  0007      LOGINTIME=  12:50:37.08
2 BRK AT 00000699
X1+0499/BSBB    X1+04A2       O ❺
1 BRK AT 000006D5
```

(continued on next page)

Example C–6 (Cont.) DELTA Debugging Session Example on VAX - Part 4

```
X1+04D5/CALLS    #0C,@#7FFEDE00                ;P  ❻
        PID=  0008      LOGINTIME=  12:50:45.64
STEPOVER BRK AT 0000069B  ❼
X1+049B/BRB    X1+047A                         ;B  ❽
1  000006D5
2  00000699  ❾
0,2;B  ❿
0,1;B  ⓫
;B  ⓬
;P  ⓭
        PID=  0009      LOGINTIME=  12:51:22.51
        PID=  000A      LOGINTIME=  12:51:30.26
        PID=  000B      LOGINTIME=  12:51:36.21
        PID=  000C      LOGINTIME=  12:51:58.86  ⓮
EXIT 00000001  ⓯
80187E7E/POPR    #03                           EXIT  ⓰
```

❶ Program execution continues with the ;P command.

❷ Program execution stops at the new breakpoint 1, which is in the subroutine GOT_IT. DELTA displays the breakpoint message and the instruction at the new breakpoint. Program execution continues with the ;P command.

❸ The subroutine completes and displays the output, and program execution continues until breakpoint 2. DELTA displays the breakpoint message and the breakpoint 2 instruction. Program execution continues with the ;P command.

❹ Program execution stops at breakpoint 1 in the subroutine. Program execution continues with the ;P command. The subroutine is executed, and the output is displayed.

❺ Program execution stops at breakpoint 2. The O command is entered to execute and step over the subroutine.

❻ Program execution stops at breakpoint 1 in the subroutine. Program execution continues with the ;P command.

❼ The subroutine completes execution and displays output. DELTA displays a STEPOVER break message to state that the O command has been completed, returning control at address 69B (an instruction in the main routine).

❽ The instruction where the subroutine returns is displayed, and program execution is halted. The ;B command is entered to display all current breakpoints.

❾ The two current breakpoints are listed.

❿ The command 0,2;B clears breakpoint 2.

⓫ The command 0,1;B clears breakpoint 1.

⓬ The ;B command is entered to display all current breakpoints. Because all breakpoints have been cleared, DELTA does not display any.

⓭ Program execution continues with the ;P command. Because there are no longer any breakpoints, the program executes to the end.

⓮ All current process login times are displayed.

⓯ Final exit status is displayed.

⓰ The DELTA EXIT command is entered to terminate the debugging session and leave DELTA.

Index

A

Address location
  changing the value, 4–5
  closing current, 4–16, 4–18
  command strings (XDELTA), 2–4, 4–27
  displaying contents of current, 4–4
  displaying in ASCII, 4–9
  displaying location pointed to by current location, 4–19
  displaying next, 4–16
  displaying previous, 4–14
  displaying range, 4–4
  displaying, from other processes, 4–4
  listing for executive images, 4–33
  PCB, 2–2, 2–3
  referencing, 3–1
  using base address and offsets, 3–2, 3–4
Address symbol
  current, 2–1, 2–2, 2–3
Alpha
  requesting interrupts, 1–7
Application register (I64)
  See AR symbol
AR symbol (Application register)
  I64 systems, 2–1
Arithmetic operators, 2–5
Arithmetic shift, 2–5
ASCII
  depositing string, 4–11
  displaying contents, 4–9

B

;B command, 4–20
Base register
  loading, 4–45
  symbol, 2–3
Boot command
  qualifiers for XDELTA, 1–4
Boot procedures for XDELTA, 1–5
BR symbol (Branch register)
  I64 systems, 2–1
Branch register (I64)
  See BR symbol
Breakpoint
  accessing initial in XDELTA, 1–7
  accessing initial on Alpha, 1–7
  accessing initial on I64, 1–7
  accessing initial on VAX, 1–7
  clearing, 4–20, 4–22
  complex, 4–22
  initial in multiprocessor environment, 1–8
  multiprocessor environment, 1–8, 3–10, 4–48
  proceeding from, 4–37
  proceeding from initial XDELTA, 1–8
  range for DELTA, 4–20
  range for XDELTA, 4–20
  setting, 4–20, 4–21
  showing, 4–20
  simple, 4–21
  XDELTA restriction on breakpoint 1, 1–5
Breakpoint command, 4–20
Bugcheck information, 1–9

C

;C command, 4–24
Change-mode-to-kernel (CMKRNL) privilege, 1–1
Close Current Location, Open Next command, 4–16
Code pages
  making writable, 3–8
' command, 4–11
/ command, 4–4
= command, 4–12
[command, 4–3
Command list, 4–1
Complex breakpoint, 4–22
Console terminal, 1–2
Control register (I64)
  See CR symbol
Copy-on-reference
  See CRF
CPU ID, 4–21, 4–48
CR symbol (Control register)
  I64 systems, 2–1
Crash command, 4–24
CRF (copy-on-reference), 3–8

D

;D command, 4–25
Debugging
  at elevated IPL, 1–1
  at IPL 0, 1–1
  code that does not match compiler listing, 3–11
  kernel mode code in process space, 3–8
  privileged code, 1–1
  user-mode programs, 1–1
Debugging session example
  source listings on Alpha, B–1
  source listings on I64, A–1
  source listings on VAX, C–1
  using DELTA on Alpha systems, B–1, B–8
  using DELTA on I64 systems, A–7
  using DELTA on VAX systems, C–4
DELTA
  exiting, 1–3
  invoking, 1–2
  privileges required for running, 1–1
  sample debugging session on Alpha, B–1
  sample debugging session on I64, A–1
  sample debugging session on VAX, C–1
DELTA/XDELTA debugger
  command reference, 4–1
  debugging an installed, protected, shareable image, 3–9
  exiting from DELTA, 1–3
  exiting from XDELTA, 1–9
  invoking DELTA, 1–2
  invoking XDELTA, 1–3
  overview, 1–1
  restrictions for XDELTA on I64 systems, 1–2
  symbols and expressions, 2–1
Deposit ASCII String command, 4–11
Display information commands
  See List commands
Display mode
  how to set, 4–3
Display Value of Expression command, 4–12
dot (.) symbol (Alpha), 2–2
dot (.) symbol (I64), 2–1
dot (.) symbol (VAX), 2–3
Double quote (") command, 4–9
Dump command, 4–25

E

;E command, 4–27
Eh? error message, 3–8
Equal (=) command, 4–12
ESC command, 4–14
ESC key equivalent, 4–14
Evaluation precedence, 2–4
Exclamation Point (!) command, 4–7
Execute Command String command, 4–27
Executive images
  listing names and addresses, 4–33, 4–43
Exit command, 4–15
Exiting
  from DELTA, 1–3, 4–15
  from XDELTA, 1–9
Expressions
  See also Numeric expressions
  forming numeric, 2–4
  precedence in, 2–4

F

Floating point control register (Alpha)
  See FPCR symbol
Floating point registers (Alpha)
  See FP symbol
Floating point registers (I64)
  See FP symbol
Floating point status register (I64)
  See FPSR symbol
Forward slash (/) command, 4–4
FP symbol (Floating point registers)
  Alpha systems, 2–2, 2–4
  I64 systems, 2–1, 2–4
FPCR symbol (Floating point control register)
  Alpha systems, 2–2
FPSR symbol (Floating point status register)
  I64 systems, 2–1

G

;G command, 4–29
G symbol (Alpha), 2–2
G symbol (I64), 2–1
G symbol (VAX), 2–3
General register (I64)
  See Rn symbol
General register symbol, 3–7
Go command, 4–29

H

;H command, 4–30
H symbol (Alpha), 2–2
H symbol (I64), 2–1
H symbol (VAX), 2–3
Hardcopy output command, 4–30

I

;I command, 4–31
I64
  requesting interrupts, 1–7
Image code
  does not match compiler listing, 3–11
images
  single
    names and addresses of, 4–43
Images
  executive
    listing names and addresses, 4–33
  shareable
    listing current information, 4–31
  sliced, 4–33
Images, sliced, 4–31, 4–43
Immediate mode text display command, 4–13
INI$BRK routine, 1–5, 3–9, 4–21
  Alpha, 1–6, 1–7
  I64, 1–6, 1–7
  VAX, 1–5, 1–7
Initial breakpoint
  See Breakpoint
Instructions
  how to display, 4–7
Interrupt request
  on Alpha, 1–7
  on I64, 1–7
  on VAX, 1–6
  XDELTA, 1–5
Interrupt stack frame
  displaying contents, 4–40
Invoking
  See also Boot procedures for XDELTA; Interrupt request for XDELTA
  DELTA, 1–2
  XDELTA, 1–3
IPID, 4–4, 4–7, 4–9

K

Kernel mode code in process space
  debugging, 3–8

L

;L command, 4–33
  privileges required, 1–1
LINEFEED command, 4–16
Linefeed key equivalent, 4–16
Linker options file
  used with XDELTA, 3–8
LIS file, 3–1, 3–5, 3–6
List commands
  Information about current main image associated shareable images, 4–31
List commands (cont'd)
  name and location of single image, 4–43
  names and addresses of loaded executive images, 4–33
Load Base Register command, 4–45

M

;M command, 4–36
  privileges required, 1–1
MAP file, 3–1, 3–4, 3–5
Memory management
  dumping region of memory, 4–25
Multiprocessor environment
  initial breakpoint, 1–8
  XDELTA breakpoints, 1–8, 3–10, 4–21, 4–48
  XDELTA operation, 3–10

N

Numeric expressions, 2–4, 4–12

O

O command, 4–48
Open Location and Display Contents command, 4–4
Open Location and Display Contents in Instruction Mode command, 4–7
Open Location and Display Indirect Location command, 4–19
Open Location and Display Previous Location command, 4–14
Operators
  arithmetic, 2–5
Output
  from DELTA, 1–1
  from XDELTA, 1–2

P

;P command, 4–37
P(ipr) symbol
  internal processor register, 2–1
Page faults
  preventing, 3–8
PC symbol (program counter)
  Alpha systems, 2–3
  I64 systems, 2–2
  VAX systems
    See RF symbol, 2–3
PCB address location, 2–2, 2–3
PCB vector start symbolic address, 2–2, 2–3
PFN (physical page number), 4–27
Physical page number
  See PFN

PID (process ID), 2–2, 2–3
pid:PC symbol (Alpha), 2–3
pid:Rn symbol (Alpha), 2–3
pid:Rn symbol (I64), 2–2
Pn symbol (Predicate register)
  I64 systems, 2–2
Pn symbol (Processor status register)
  VAX systems, 2–3
Predicate register (I64)
  See Pn symbol
Printed output command, 4–30
Privileged code
  debugging, 1–1
Privileges
  DELTA, 1–1
  XDELTA, 1–2
Proceed from Breakpoint command, 4–37
Processes
  how to set writable, 4–36
Processor status register
  See PS symbol
Processor status register (Alpha)
  See PS symbol
Processor status register (I64)
  See PS symbol
Processor status register (VAX)
  See Pn symbol
Program counter (Alpha)
  See PC symbol
Program counter (I64)
  See PC symbol
Program execution
  continuing, 4–29
  proceeding from breakpoint, 4–37
  step execution, 4–51
  step over subroutine execution, 4–48
PS symbol (Processor status register)
  Alpha systems, 2–3, 3–7
  I64 systems, 2–2
PSL (processor status longword), 3–7

Q

;Q command, 4–39
Q symbol (Alpha), 2–3
Q symbol (I64), 2–2
Q symbol (VAX), 2–3
Queue
  validate, 4–39

R

Radix, 2–1
Redirecting output
  DELTA, 1–1
  XDELTA, 1–2
Registers
  display contents, 4–4
  examining general purpose registers of another process, 4–6
  loading base, 4–45
  referencing, 3–6
  symbol for base, 2–3
  symbol for general, 3–7
RETURN command, 4–18
RF symbol<VAX systems>
  as program counter, 2–3
Right angle bracket (]) command, 4–3
Rn symbol (general register)
  Alpha systems, 2–3
  I64 systems, 2–2
  VAX systems, 2–3

S

S command, 4–51
SCH$GL_PCBVEC symbolic address, 2–2, 2–3
Set All Processes Writable command, 4–36
Set Display Mode command, 4–3
SET HOST/LAT/LOG command, 1–2
$SETPRT
  used with XDELTA, 3–8
Shareable images
  debugging installed, protected, 3–9
  listing current information, 4–31
Simple breakpoint, 4–21
Single quote (') command, 4–11
Single-step
  fails, 3–10
Sliced images, 4–31, 4–33, 4–43
Stack pointer symbol, 3–7
Step Instruction command, 4–51
Step Instruction over Subroutine command, 4–48
String
  depositing ASCII, 4–11
\ string\ command, 4–13
Symbol
  DELTA, 2–1
  XDELTA, 2–1
. symbol (Alpha), 2–2
. symbol (I64), 2–1
. symbol (VAX), 2–3
System space prefix symbol, 2–1, 2–2, 2–3

T

;T command, 4–40
TAB command, 4–19
Terminating DELTA, 4–15
Terminating DELTA/XDELTA commands, 4–18

U

User-mode program
  debugging, 1–1

V

Validate queue command, 4–39
Value (last) displayed symbol, 2–2, 2–3
VAX
  requesting interrupts, 1–6
Video Terminal Display command, 4–30

W

;W command, 4–43

X

;X command, 4–45
XDELTA
  boot procedures, 1–5
  exiting, 1–9
  guidelines for using, 1–2
  invoking, 1–3
  proceeding from initial breakpoint, 1–8
  redirecting output, 1–2
XE base register, 4–27
XE base register (VAX), 2–4
XF base register, 4–27
XF base register (VAX), 2–4
Xn symbol (Alpha)
  X4 symbol, 2–3
  X5 symbol, 2–3
Xn symbol (I64)
  X4 symbol, 2–2
  X5 symbol, 2–2
Xn symbol(VAX)
  X4 symbol, 2–3
  X5 symbol, 2–3