Trap Handling Programmer's Guide

900 Series HP 3000 Computers

HEWLETT PACKARD

HP Part No. 32650-90026
Printed in U.S.A. 19871101

E1187

Printing History

New editions are complete revisions of the manual. Update packages, which are issued between editions, contain additional and replacement pages to be merged into the manual by the customer. The date on the title page and back cover of the manual changes only when a new edition is published. When an edition is reprinted, all the prior updates to the edition are incorporated. No information is incorporated into a reprinting unless it appears as a prior update.

First Edition	November 1987	A.01.00

List of Effective Pages

The List of Effective Pages gives the date of the current edition, and lists the dates of all changed pages. Unchanged pages are listed as "ORIGINAL". Within the manual, any page changed since the last edition is indicated by printing the date the changes were made on the bottom of the page. Changes are marked with a vertical bar in the margin. If an update is incorporated when an edition is reprinted, these bars and dates remain. No information is incorporated into a reprinting unless it appears as a prior update.

First Edition	November 1987

Documentation Map

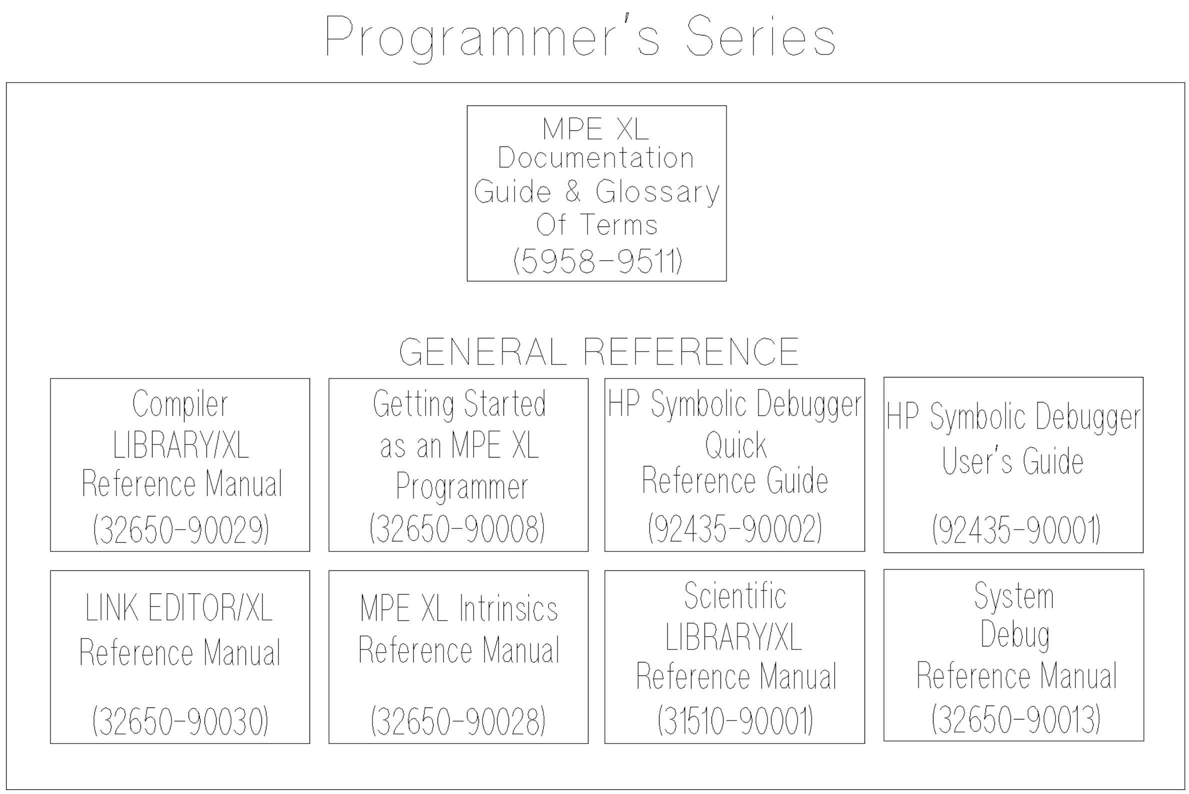

Figure 0-1. =center

width=6.5in height=3.9in border=box>

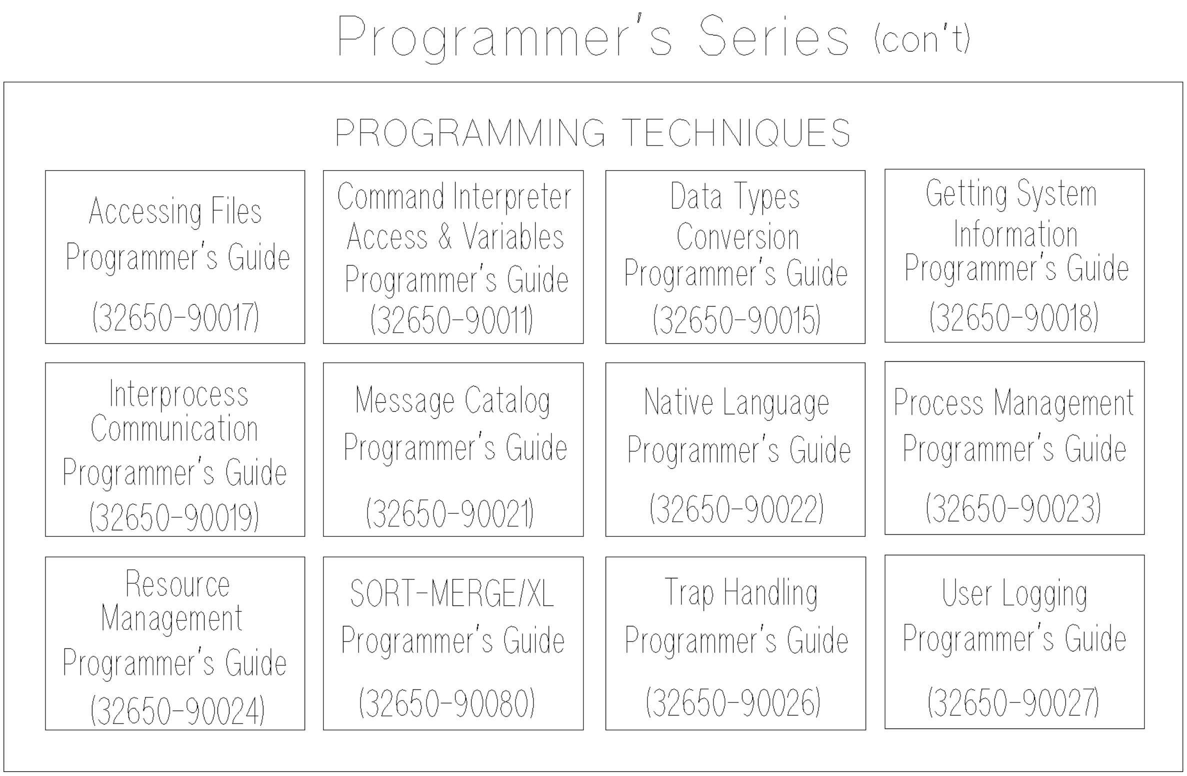

Figure 0-2. =center

width=6.5in height=4.4in border=box>

Preface

The *Trap Handling Programmer's Guide* explains how you can develop your own routines to handle interrupt events and thereby recover from errors and avoid a process abort.

Chapter 1	**Overview** defines traps and trap handling; summarizes trap handling on MPE XL.
Chapter 2	**MPE XL Arithmetic Traps** provides an overview of and reference information on the arithmetic trap handling intrinsics; describes the use of these intrinsics; includes example programs.
Chapter 3	**MPE XL Code-related Traps** provides information regarding how the MPE XL trap subsystem handles code-related traps.
Chapter 4	**MPE XL CONTROL-Y Traps** provides an overview of, and reference information on, the CONTROL-Y trap handling intrinsics; describes the use of these intrinsics; includes example programs.
Chapter 5	**MPE XL Software Library Traps** provides an overview of, and reference information on, the software library trap handling intrinsic; describes the use of this intrinsic; includes example programs.
Chapter 6	**MPE XL Software System Traps** provides an overview of, and reference information on, the software system trap handling intrinsic; describes the use of this intrinsic; includes example programs.
Appendix A	**MPE XL Trap Subsystem Escape Codes** lists the MPE XL trap subsystem escape codes in decimal and hexadecimal and supplies the meanings for those codes.
Appendix B	**Intrinsic Numbers** lists the MPE XL system intrinsics and their associated numbers (used in handling software system traps).

Conventions

NOTATION	DESCRIPTION
`UPPERCASE`	Within syntax statements, characters in uppercase must be entered in exactly the order shown, though you can enter them in either uppercase or lowercase. For example: `SHOWJOB` Valid entries: `showjob` `ShowJob` `SHOWJOB` Invalid entries: `shojwob` `ShoJob` `SHOW_JOB`
italics	Within syntax statements, a word in italics represents a formal parameter or argument that you must replace with an actual value. In the following example, you must replace *filename* with the name of the file you want to release: `RELEASE` *filename*
punctuation	Within syntax statements, punctuation characters (other than brackets, braces, vertical parallel lines, and ellipses) must be entered exactly as shown.
`{ }`	Within syntax statements, braces enclose required elements. When several elements within braces are stacked, you must select one. In the following example, you must select `ON` or `OFF`: $\texttt{SETMSG} \begin{Bmatrix} \texttt{ON} \\ \texttt{OFF} \end{Bmatrix}$
`[ ]`	Within syntax statements, brackets enclose optional elements. In the following example, brackets around `,TEMP` indicate that the parameter and its delimiter are optional: `PURGE {`*filename*`} [,TEMP]` When several elements with brackets are stacked, you can select any one of the elements or none. In the following example, you can select *devicename* or *deviceclass* or neither: $\texttt{SHOWDEV} \begin{bmatrix} \textit{devicename} \\ \textit{deviceclass} \end{bmatrix}$

NOTATION	DESCRIPTION

`[ ... ]`	Within syntax statements, a horizontal ellipsis enclosed in brackets indicates that you can repeatedly select elements that appear within the immediately preceding pair of brackets or braces. In the following example, you can select *itemname* and its delimiter zero or more times. Each instance of *itemname* must be preceded by a comma:
	
	`[,`*itemname*`][ ... ]`
	
	If a punctuation character precedes the ellipsis, you must use that character as a delimiter to separate repeated elements. However, if you select only one element, the delimiter is not required. In the following example, the comma cannot precede the first instance of *itemname*:
	
	`[`*itemname*`][, ... ]`

`| ... |`	Within syntax statements, a horizontal ellipsis enclosed in parallel vertical lines indicates that you can select more than one element that appears within the immediately preceding pair of brackets or braces. However, each element can be selected only one time. In the following example, you must select `,A` or `,B` or `,A,B` or `,B,A` : $\left\{\begin{matrix}\texttt{,A}\\\texttt{,B}\end{matrix}\right\}\texttt{\| ... \|}$ If a punctuation character precedes the ellipsis, you must use that character as a delimiter to separate repeated elements. However, if you select only one element, the delimiter is not required. In the following example, you must select `A` or `B` or `AB` or `BA`. The first element cannot be preceded by a comma: $\left\{\begin{matrix}\texttt{A}\\\texttt{B}\end{matrix}\right\}\texttt{\|, ... \|}$
...	Within examples, horizontal or vertical ellipses indicate where portions of the example are omitted.
⊔	Within syntax statements, the space symbol ⊔ shows a required blank. In the following example, you must separate *modifier* and *variable* with a blank: `SET[(`*modifier*`)]⊔(`*variable*`);`
`shading`	Within an example of interactive dialog, **`shaded`** characters indicate user input or responses to prompts. In the following example, **`OMEGA`** is the user's response to the `NEW NAME` prompt: `NEW NAME?` **`OMEGA`**

NOTATION	DESCRIPTION
( )	The symbol ( ) indicates a key on the terminal's keyboard. For example, (CTRL) indicates the Control key.
(CTRL) *char*	(CTRL) *char* indicates a control character. For example, (CTRL) `Y` means you have to simultaneously press the Control key and the Y key on the keyboard.
base prefixes	The prefixes `%`, `#`, and `$` specify the numerical base of the value that follows: `%`*num* specifies an octal number. `#`*num* specifies a decimal number. `$`*num* specifies a hexadecimal number. When no base is specified, decimal is assumed.

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

```
most significant                   least significant

|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
| 0|  |  |  |  |  |  |  |  |  |  |  |  |13|14|15|
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|

Bit (0:1)                                 Bits(13:3)
```

Contents

Figures

Tables

1

Overview

The following paragaphs describe traps and trap handling routines. The discussion also introduces the concepts of enabling/disabling and arming/disarming traps.

What are Traps?

A trap involves the interruption, and possible termination, of a running process as the result of some event. Some events are detected in the hardware and, therefore, are known as hardware traps. Division by zero, result overflow, and data memory protection traps are examples of hardware traps. Other events are detected by the operating system or by subsystems. These events are called software traps. Examples of software traps are stack overflow and violations of callable intrinsics, such as passing illegal parameters or invoking an intrinsic without having the required capability class.

What You See When a Trap Aborts a Process

The following sequence of events occurs when a process is aborted because of a trap:

1. You see two messages (see Examples 1-1 and 1-2):
    a. The first message indicates the error condition that caused the abort.
    b. The second message contains the program name and the type of abort.
2. The `JCW` Job Control Word is set to indicate an abort occurred.
3. Depending on whether the `SETDUMP` intrinsic is armed, the following happen:

a. If SETDUMP is not armed, then a minimal stack trace is printed, and the process is terminated. This stack trace contains all calls in which the destination (called) routine is in a different executable library.

b. If SETDUMP is armed, the debugger is called with the SETDUMP command string. Depending on the cause of the abort, the debugger could be called again to allow interactive dialog.

Examples 1-1 and 1-2 illustrate a minimal (SETDUMP unarmed) and a full (SETDUMP armed) stack trace, respectively:

```
**** FILE NOT OPENED (PASCERR 706)
ABORT: MYPROG.GROUP.ACCT
SYS     a.4147c    TRAP_HANDLER+3c4
XL     31.3361bc   P_PASCFERROR+a0
USER   30.30d8     proc_in_user_library+324
PROG   25.4d30     proc_in_program+108
```

Figure 1-1. Example 1-1: Minimal Stack Trace

```
**** FILE NOT OPENED (PASCERR 706)
ABORT: MYPROG.GROUP.ACCT
SYS     a.4147c    TRAP_HANDLER+3c4
XL     31.361bc    P_PASCFERROR+a0
XL     31.30044    P_FERR+68
XL     31.36a84    P_READINT+20
USER   30.30d8     proc_in_user_library+324
USER   30.54       first_proc_in_user_library+16
PROG   25.4d30     proc_in_program+108
PROG   25.8920     middle_proc_in_program+70
PROG   25.234      first_proc_in_program+5c
PROG   25.f938     PROGRAM+94
```

Figure 1-2. Example 1-2: Full Stack Trace

What is Trap Handling?

When a trap situation is detected and process execution is interrupted, control passes to the MPE XL trap subsystem, which determines what action is to be taken in response to the trap. In most instances, the trap subsystem outputs an appropriate error message and aborts the offending process. Generally, this is what you want to happen. You can, however, avoid the abort by writing and enabling/arming one or more of the types of trap handlers allowed by MPE XL:

- Arithmetic trap handler, for errors of an arithmetic nature
- Library trap handler, for errors detected during the execution of a system library procedure
- System trap handler, for errors detected during execution of a system intrinsic
- CONTROL-Y trap handler, for handling subsystem breaks during interactive sessions

You can either arm or disarm your trap handlers, and in some instances, enable or disable them. If your trap handler is armed and enabled, the MPE XL trap subsystem does the following when an error is detected:

- Suppresses output of the normal error message
- Transfers control to a trap handling procedure defined by you
- Passes one or more parameters describing the error to your trap handling procedure

Your trap handling procedure may attempt to analyze or recover from the error, or it may execute some other programming path. Upon exit from the trap handling procedure, control usually returns to the instruction following the one that activated the trap. In the case of library traps, however, you can specify that the process should abort when control exits from the trap handler.

User-defined trap handling procedures are armed or disarmed, enabled or disabled by means of the system intrinsics described later. **If a user-defined trap handler is not both armed and enabled, the MPE XL trap subsystem is invoked when a trap occurs.**

There is a difference between arming and enabling traps. Enabling a trap means that the occurrence of a trap condition is not ignored. Arming a trap is required so that, on a trap condition, a user-written routine is nvoked and can take appropriate recovery actions. The following list summarizes what can occur when an arithmetic trap condition arises:

1. If a trap is both enabled and armed, the user-written trap handler is invoked whenever a trap condition occurs.
2. If a trap is enabled but not armed, one of two situations applies:
   a. If you have executed an HP Pascal/XL TRY statement, control is passed to the RECOVER block by doing an ESCAPE.
   b. If you have not executed an HP Pascal/XL TRY statement, an error message is output and the process aborts.
3. If a trap is disabled, irrespective of whether it is armed, the trap is ignored, and execution of the process continues without any interruption.

Although trap situations are usually involuntary, there is one kind of trap that you can cause intentionally. This is the subsystem break (CONTROL-Y), and it can bge entered during interactive sessions. You can write a trap handling procedure for such situations and arm or disarm your trap handler by using an intrinsic.

Refer to individual programming language manuals for language-specific information on trap handling.

MPE XL Trap Handling Intrinsics

In MPE XL, all access to user-defined trap handling routines is controlled by system intrinsics. The MPE XL trap handling intrinsics deal with the following types of trap situations:

- Arithmetic trap
- Software Library trap
- Software System trap
- CONTROL-Y trap

Trap intrinsics can be invoked from within trap handling procedures.

Arithmetic Traps

An arithmetic trap handler reacts to arithmetic errors that occur as a result of arithmetic operations. The `ARITRAP` intrinsic enables or disables arithmetic traps. The `XARITRAP` intrinsic arms or disarms the user-written arithmetic trap handler. The `HPENBLTRAP` is similar in function to `ARITRAP`, but allows you greater flexibility in selectively enabling or disabling arithmetic traps.

Software Library Traps

A software library trap handler reacts to errors that occur during execution of procedures from the compiler libraries. You can arm or disarm your software library trap handler with the `XLIBTRAP` intrinsic.

Software System Traps

A software system trap handler reacts to certain errors that occur when user programs call intrinsics incorrectly. You arm or disarm your software system trap handler by means of the `XSYSTRAP` intrinsic.

CONTROL-Y Traps

If you are running a program in an interactive session, you can invoke a special trap that transfers control in the program to an armed, user-defined trap handling procedure when you enter a CONTROL-Y subsystem break signal from the terminal. On most terminals, this signal is transmitted by pressing the [Y] key while holding down the [CTRL] key. The `XCONTRAP` intrinsic arms or disarms the user-written CONTROL-Y trap handling procedure.

If you send another CONTROL-Y signal, MPE XL ignores it unless you call the `RESETCONTROL` intrinsic at some point prior to the signal.

Code-related Traps

Code-related traps are handled solely by the MPE XL trap subsystem (that is, users cannot define their own routines to handle these traps). Code-related traps can occur as a result of illegal or incorrect coding practices.

Operating Mode Considerations

A Native Mode (NM) trap handling routine often differs from the corresponding Compatibility Mode (CM) trap handler in its calling sequence [refer to the *Introduction to MPE XL for MPE V Programmers* (30367-90005)] and the method by which the trap routine obtains error information. However, you need supply only the version for the mode in which you invoke intrinsics.

You do not need to modify existing CM applications that use a trap handler in order to run them on MPE XL. Likewise, new NM applications need not specify a CM version of their NM trap handling routine. Only those doing mixed-mode programming, who invoke intrinsics in both modes, need to specify and arm trap handling routines in both modes in order to capture all possible traps.

This manual documents trap handling from the Native Mode perspective only. For information concerning trap handling in the Compatibility Mode environment, refer to the *MPE V Intrinsics Reference Manual* (32033-90007). For information concerning trap handling when doing mixed-mode programming, refer to the *Introduction to MPE XL for MPE V Programmers* (30367-90005).

2

MPE XL Arithmetic Traps

There are two types of arithmetic traps:

- Hardware arithmetic traps
- Software arithmetic traps

Each trap in the arithmetic trap set detects a particular type of arithmetic error, such as division by zero or result overflow.

The user-written trap handler, if enabled and armed, receives an interrupt from the trap when an error is encountered, and control transfers to the user-written trap handling procedure.

When a user process begins execution, the following hold:

- All arithmetic traps, except the IEEE floating-point exceptions, are enabled automatically.
- The software trap handler is disarmed. This allows any arithmetic error to abort the process (unless a TRY/RECOVER block assumes control).

Through intrinsic calls, however, you can alter the ability of the arithmetic traps to occur, and that of the software trap handler to be invoked from any particular arithmetic trap. Only enabled traps can invoke a user-written trap handling procedure.

There are three MPE XL intrinsics used in dealing with arithmetic traps:

- `ARITRAP`
- `HPENBLTRAP` (NM only)
- `XARITRAP`

The `ARITRAP` intrinsic collectively enables or disables arithmetic traps (except IEEE Floating Point and Inexact Result).

The `HPENBLTRAP` intrinsic lets you selectively enable or disable specific arithmetic traps. This intrinsic provides you with more flexibility than the `ARITRAP` intrinsic.

The `XARITRAP` intrinsic arms or disarms the user-written arithmetic trap handling procedure and enables/disables whatever traps that procedure accommodates. For some or all arithmetic traps, you can then replace the MPE XL software arithmetic trap handler with your own trap handling routine.

Note `ARITRAP` is provided for compatibility with MPE V/E systems. `HPENBLTRAP` is provided on MPE XL to make available the full power of MPE XL trap handling.

The interrupts listed below are the arithmetic traps that the `ARITRAP`, `HPENBLTRAP`, and `XARITRAP` intrinsics let you enable/disable and arm/disarm:

- Integer Overflow
- 3000 Mode Double Precision Divide By Zero
- 3000 Mode Floating Point Overflow
- Decimal Overflow
- 3000 Mode Floating Point Underflow
- Invalid ASCII Digit
- Integer Divide By Zero
- Invalid Decimal Digit
- 3000 Mode Floating Point Divide By Zero
- 3000 Mode Double Precision Overflow
- 3000 Mode Double Precision Underflow
- Decimal Divide By Zero
- IEEE Floating Point Divide By Zero
- IEEE Floating Point, Inexact Result
- IEEE Floating Point Underflow
- IEEE Floating Point Overflow
- IEEE Floating Point, Invalid Operation
- Range Errors
- Software-detected NIL Pointer Reference
- Software-detected Result of Pointer Arithmetic Misaligned or Error in Conversion From Long Pointer to Short Pointer
- Unimplemented Condition Traps

- Paragraph Stack Overflow

Discussions of the intrinsics follow.

ARITRAP

Collectively enables or disables arithmetic traps.

Note When arithmetic traps are ignored (disabled) on MPE XL, the results are not guaranteed to be identical to those on MPE V/E. On MPE XL, a better way to disable arithmetic traps is to use compiler directives, for example, `$ovflcheck off$` in HP Pascal/XL.

When compiler directives are used, the compiler generates arithmetic instructions that do not trap on overflow. This is a more natural way of ignoring arithmetic traps. When arithmetic traps are ignored using `ARITRAP`, the trap actually takes place, but the MPE XL trap subsystem recovers from the trap and takes the action required to continue execution.

Syntax

The `ARITRAP` intrinsic is called as follows:

`ARITRAP(`**trapstate**`);`

Parameters

trapstate **32-bit signed integer by value (required)**

A value enabling or disabling arithmetic traps.

Enter 0 if you want to disable arithmetic traps or 1 if you want to enable all traps except the IEEE inexact result trap.

Note By default, all traps except IEEE floating-point exceptions are enabled, and no trap is armed. Many floating-point operations result in an inexact result. Consequently, most compiler libraries doing floating-point operations will result in an inexact trap if the IEEE Inexact Result trap is enabled. Therefore, you should enable the IEEE Inexact Result trap (using the `HPENBLTRAP` intrinsic) only if absolutely necessary.

Condition Codes

CCE	Request granted. The arithmetic traps were originally disabled.
CCG	Request granted. The arithmetic traps were originally enabled.
CCL	Not returned by this intrinsic.

HPENBLTRAP

Selectively enables or disables arithmetic traps.

Note By default, all traps except IEEE floating-point exceptions are enabled, and no trap is armed. Many floating-point operations result in an inexact result. Consequently, most compiler libraries doing floating-point operations will result in an inexact trap if the IEEE Inexact Result trap is enabled. Therefore, you should enable the IEEE Inexact Result trap only if absolutely necessary.

Syntax

The `HPENBLTRAP` intrinsic is called as follows:

`HPENBLTRAP(`**mask**`,`**oldmask**`);`

Parameters

mask **32-bit signed integer by value (required)**

A value determining which arithmetic traps are enabled and which are not.

If a bit is on (=1), the corresponding trap is enabled. If a bit is off (=0), the corresponding trap is disabled. The bit and their associated arithmetic errors are as follows:

Bit	Trap Condition
(31:1)	3000 Mode Floating Point Divide By Zero.
(30:1)	Integer Divide By Zero.
(29:1)	3000 Mode Floating Point Underflow.
(28:1)	3000 Mode Floating Point Overflow.
(27:1)	Integer Overflow.
(26:1)	3000 Mode Double Precision Overflow.
(25:1)	3000 Mode Double Precision Underflow.
(24:1)	3000 Mode Double Precision Divide By Zero.
(23:1)	Decimal Overflow.
(22:1)	Invalid ASCII Digit.
(21:1)	Invalid Decimal Digit.
(19:2)	Reserved for future use. You should set these to 0.
(18:1)	Decimal Divide By Zero.
(17:1)	IEEE Floating Point, Inexact Result.
(16:1)	IEEE Floating Point Underflow.
(15:1)	IEEE Floating Point Overflow.
(14:1)	IEEE Floating Point Divide By Zero.
(13:1)	IEEE Floating Point, Invalid Operation.

(12:1)	Range Errors.
(11:1)	Software-detected NIL Pointer Reference.
(10:1)	Software-detected Misaligned Result Of Pointer Arithmetic or Error In Conversion From Long Pointer To Short Pointer.
(9:1)	Unimplemented Condition Traps.
(8:1)	Paragraph Stack Overflow.
(0:8)	Reserved. You should set these to 0.

Note

The following apply to the preceding trap conditions represented in the mask:

1. Native Mode supports two floating-point formats: IEEE and 3000 Mode. Both execute in Native Mode, but 3000 Mode performs HP 3000 type manipulations. Since it is possible to use both formats during program execution, there are separate bits in the mask for enabling/disabling traps of these formats.
2. Some of the error conditions specified here are not strictly arithmetic traps (for example, range errors, nil pointers, and paragraph stack overflow). However, they and many arithmetic traps are caught by reserved instructions that raise the conditional traps. For this reason, all are enabled/disabled by `HPENBLTRAP`.
3. Some of the instructions that raise conditional traps are reserved to indicate some of the above trap conditions. A nonreserved instruction is one not generated by a compiler. If a nonreserved instruction causes a conditional trap, this is reported as an Unimplemented Condition Trap. Only assembly language programmers can generate such a trap.

---

oldmask **32-bit signed integer by reference (required)**

Returns the value of the previous **mask** to your program.

Condition Codes

CCE	Request granted. All traps were originally disabled.
CCG	Request granted. At least one trap was originally enabled.
CCL	Not returned by this intrinsic.

XARITRAP

Arms or disarms the user-written arithmetic trap handling procedure. Although you can arm the trap for any desired combination of events, at any given time there is only one user-written trap handler for all armed traps.

Note By default, all traps except IEEE floating-point exceptions are enabled, and no trap is armed. Many floating-point operations result in an inexact result. Consequently, most compiler libraries doing floating-point operations will result in an inexact trap if the IEEE Inexact Result trap is enabled. Therefore, you should enable the IEEE Inexact Result trap only if absolutely necessary.

Syntax

The `XARITRAP` intrinsic is called as follows:

`XARITRAP(`**mask,plabel,oldmask,oldplabel**`);`

Parameters

mask **32-bit signed integer by value (required)**

A value determining which trap conditions, if enabled, invoke the user-written software trap handler, and which do not.

If a bit is on (=1), the corresponding trap condition becomes armed. If a bit is off (=0), the corresponding trap condition is disarmed. The bits and their associated arithmetic errors are as follows:

Bit	Trap Condition
(31:1)	3000 Mode Floating Point Divide By Zero.
(30:1)	Integer Divide By Zero.
(29:1)	3000 Mode Floating Point Underflow.
(28:1)	3000 Mode Floating Point Overflow.
(27:1)	Integer Overflow.
(26:1)	3000 Mode Double Precision Overflow.
(25:1)	3000 Mode Double Precision Underflow.
(24:1)	3000 Mode Double Precision Divide By Zero.
(23:1)	Decimal Overflow.
(22:1)	Invalid ASCII Digit.
(21:1)	Invalid Decimal Digit.
(19:2)	Reserved for future use. You should set these to 0.
(18:1)	Decimal Divide By Zero.
(17:1)	IEEE Floating Point, Inexact Result.
(16:1)	IEEE Floating Point Underflow.
(15:1)	IEEE Floating Point Overflow.
(14:1)	IEEE Floating Point Divide By Zero.
(13:1)	IEEE Floating Point, Invalid Operation.
(12:1)	Range Errors.
(11:1)	Software-detected NIL Pointer Reference.
(10:1)	Software-detected Misaligned Result Of Pointer Arithmetic or Error In Conversion From Long Pointer To Short Pointer.

(9:1)	Unimplemented Condition Traps.
(8:1)	Paragraph Stack Overflow.
(0:8)	Reserved. You should set these to 0.

Note

The following apply to the preceding trap conditions represented in the mask:

1. Native Mode supports two floating-point formats: IEEE and 3000 Mode. Both execute in Native Mode, but 3000 Mode performs HP 3000 type manipulations. Since it is possible to use both formats during program execution, there are separate bits in the mask for enabling/disabling traps of these formats.
2. Some of the error conditions specified here are not strictly arithmetic traps (for example, range errors, nil pointers, and paragraph stack overflow). However, they and many arithmetic traps are caught by reserved instructions that raise the conditional traps. For this reason, all are enabled/disabled by `HPENBLTRAP`.
3. Some of the instructions that raise conditional traps are reserved to indicate some of the above trap conditions. A nonreserved instruction is one not generated by a compiler. If a nonreserved instruction causes a conditional trap, this is reported as an Unimplemented Condition Trap. Only assembly language programmers can generate such a trap.

plabel **32-bit signed integer by value (required)**

The address of your trap handling procedure. If the value is 0, the user-written arithmetic trap handler is disarmed.

oldmask **32-bit signed integer by reference (required)**

Returns the value of the previous **mask** to your program.

oldplabel **32-bit signed integer by reference (required)**

Returns the plabel of your process' previous arithmetic trap handler. If no plabel was previously configured, **oldplabel** returns 0 (indicating the MPE XL trap subsystem was in effect).

Condition Codes

CCE Request granted. The desired traps are now armed.

CCG Request granted. All traps are now disarmed.

CCL Not returned by this intrinsic.

Caution CCL is defined differently on MPE V/E systems.

Handling Arithmetic Traps

When you invoke a user-written trap handling procedure, it is passed a pointer to a record that contains some useful information. This record has different fields depending upon the trap condition. The following fields are supplied for all trap conditions:

`instruction` `32-bit integer`

The offending instruction.

`offset` `32-bit integer`

Offset of the offending instruction.

`space_id` `32-bit integer`

Space ID of the offending instruction.

`error_code` `32-bit integer`

Trap type. The error-code is forward by setting the bit corresponding to the trap condition in a 32-bit integer. These bits are described in the discussion of the **mask** parameter of the XARITRAP intrinsic.

Note If two exceptions occur simultaneously, the error-code is the inclusive-OR of the error-code for each exception. The only exceptions that coincide are IEEE inexact with IEEE overflow, and IEEE inexact with IEEE underflow.

The following paragraphs describe the contents of this record structure for the various arithmetic trap conditions.

Integer Divide by Zero

The record structure for a trap condition resulting from an integer divide by zero is as follows:

```
 0  |--------------------------------|
    |     instruction                |
 4  |--------------------------------|
    |     offset                     |
 8  |--------------------------------|
    |     space_id                   |
12  |--------------------------------|
    |     error_code                 |
    |--------------------------------|
```

Corrective action for this trap condition is not supported. You can allow execution to resume, but the result will be unpredictable.

Note If the trap is ignored, or if you continue execution from the trap handler, then the result of the illegal division is zeroed.

Range Errors

The record structure for a trap condition resulting from a range error is as follows:

```
 0  |--------------------------------|
    |     instruction                |
 4  |--------------------------------|
    |     offset                     |
 8  |--------------------------------|
    |     space_id                   |
12  |--------------------------------|
    |     error_code                 |
    |--------------------------------|
```

Corrective action for this trap condition is not supported. You can allow execution to resume, but the result will be unpredictable.

Nil Pointer Reference

The record structure for a trap condition resulting from a nil pointer reference is as follows:

```
 0  |--------------------------------|
    |     instruction                |
 4  |--------------------------------|
    |     offset                     |
 8  |--------------------------------|
    |     space_id                   |
12  |--------------------------------|
    |     error_code                 |
    |--------------------------------|
```

Corrective action for this trap condition is not supported. You can allow execution to resume, but the result will be unpredictable.

Pointer Arithmetic Errors

The record structure for a trap condition resulting from a pointer arithmetic error is as follows:

```
 0  |--------------------------------|
    |     instruction                |
 4  |--------------------------------|
    |     offset                     |
 8  |--------------------------------|
    |     space_id                   |
12  |--------------------------------|
    |     error_code                 |
    |--------------------------------|
```

Corrective action for this trap condition is not supported. You can allow execution to resume, but the result will be unpredictable.

Paragraph Stack Overflow

The record structure for a trap condition resulting from a paragraph stack overflow is as follows:

```
 0  |--------------------------------|
    |     instruction                |
 4  |--------------------------------|
    |     offset                     |
 8  |--------------------------------|
    |     space_id                   |
12  |--------------------------------|
    |     error_code                 |
    |--------------------------------|
```

Corrective action for this trap condition is not supported. You can allow execution to resume, but the result will be unpredictable.

Integer Overflow

The record structure for a trap condition resulting from an integer overflow is as follows:

```
 0  |--------------------------------|
    |     instruction                |
 4  |--------------------------------|
    |     offset                     |
 8  |--------------------------------|
    |     space_id                   |
12  |--------------------------------|
    |     error_code                 |
16  |--------------------------------|
    |     subcode                    |
    |--------------------------------|
```

The `subcode` field is a 32-bit integer that tells what type of integer overflow has occurred. The following table summarizes the possible values for this field and their associated overflow types:

Value	Type of Overflow
1	32-bit overflow
2	16-bit overflow
4	Overflow on conversion from a 3000 Mode floating-point number
5	Overflow on conversion from an IEEE floating-point number

Corrective action for this trap condition is not supported. You can allow execution to resume, but the result will be unpredictable.

Decimal Overflow

The record structure for a trap condition resulting from a decimal overflow is as follows:

```
 0  |--------------------------------|
    |     instruction                |
 4  |--------------------------------|
```

```
     |     offset                       |
   8 |--------------------------------|
     |     space_id                     |
  12 |--------------------------------|
     |     error_code                   |
  16 |--------------------------------|
     |     subcode                      |
     |--------------------------------|
```

The `subcode` field is a 32-bit integer that tells what type of decimal overflow has occurred. The following table summarizes the possible values for this field and their associated overflow types:

Value	Type of Overflow
1	Decimal arithmetic operation resulted in overflow
2	Conversion from integer to ASCII resulted in overflow

Corrective action for this trap condition is not supported. You can allow execution to resume, but the result will be unpredictable.

Invalid ASCII Digit

The record structure for a trap condition resulting from an invalid ASCII digit is as follows:

```
   0 |--------------------------------|
     |     instruction                  |
   4 |--------------------------------|
     |     offset                       |
   8 |--------------------------------|
     |     space_id                     |
  12 |--------------------------------|
     |     error_code                   |
  16 |--------------------------------|
     |        subcode                   |
  20 |--------------------------------|
     |     source_address               |
  24 |--------------------------------|
     |     digit_count                  |
```

```
                 |--------------------------------|
```

The `subcode` field can assume the following values with the associated meanings:

0 There is an illegal digit and/or sign.

1 The number is not signed.

2 The input number on an unsigned-to-unsigned operation is signed.

3 The input number on an unsigned-to-signed operation is signed.

The `source_address` and `digit_count` record fileds supply the following information:

- Address of the first digit (32-bit address).
- Digit count (32-bit integer) is the number of bytes.

With this information, you can correct the invalid digits, and then allow execution to proceed. Validation of the data occurs before the actual operation begins. This lets you correct invalid data and continue from the beginning of the actual operation.

Invalid Decimal Digit

The record structure for a trap condition resulting from an invalid decimal digit is as follows:

```
              0  |--------------------------------|
                 |     instruction                |
              4  |--------------------------------|
                 |     offset                     |
              8  |--------------------------------|
                 |     space_id                   |
             12  |--------------------------------|
                 |     error_code                 |
             16  |--------------------------------|
                 |        subcode                 |
             20  |--------------------------------|
```

```
     |     source_address              |
  24 |---------------------------------|
     |     digit_count                 |
     |---------------------------------|
```

The `subcode` field can assume the following values with the associated meanings:

0	There is an illegal digit and/or sign.
1	The number is not signed.
2	The input number on an unsigned-to-unsigned operation is signed.
3	The input number on an unsigned-to-signed operation is signed.

The `source_address` and `digit_count` record fields supply the following information:

- Address of the first digit (32-bit address).
- Digit count (32-bit integer) is the number of decimal digits, including the sign bit.

With this information, you can correct the invalid digits, and then allow execution to proceed. Validation of the data occurs before the actual operation begins. This allows you to correct invalid data and continue from the beginning of the actual operation. For complete information on decimal format, refer to *Data Types Conversion Programmer's Guide* (32650-90015).

Decimal Divide by Zero

The record structure for a trap condition resulting from a decimal divide by zero is as follows:

```
   0 |---------------------------------|
     |     instruction                 |
   4 |---------------------------------|
     |     offset                      |
   8 |---------------------------------|
     |     space_id                    |
```

```
12  |--------------------------------|
    |     error_code                 |
16  |--------------------------------|
    |     result address             |
20  |--------------------------------|
    |     digit count                |
    |--------------------------------|
```

The `result_address` and `digit_count` record fields supply the following information:

- Address of the result (32-bit address)
- Number of digits in the result (32-bit integer)

With this information, you can assign the desired value to the result, and allow execution to resume with the operation following the division.

IEEE Floating Point Traps

The IEEE floating-point traps include the following conditions:

- IEEE Floating Point Divide By Zero
- IEEE Floating Point, Inexact Result
- IEEE Floating Point Underflow
- IEEE Floating Point Overflow
- IEEE Floating Point, Invalid Operation

The record structure for an IEEE floating-point trap condition is as follows:

```
 0  |--------------------------------|
    |     instruction                |
 4  |--------------------------------|
    |     offset                     |
 8  |--------------------------------|
    |     space_id                   |
12  |--------------------------------|
    |     error_code                 |
16  |--------------------------------|
    |     status                     |
```

```
20  |--------------------------------|
    |     operation                  |
24  |--------------------------------|
    |     format                     |
28  |--------------------------------|
    |     source_op1_ptr             |
32  |--------------------------------|
    |     source_op2_ptr             |
36  |--------------------------------|
    |     result_ptr                 |
    |--------------------------------|
```

The following table explains the information supplied in the additional record fields associated with IEEE traps:

`status` `32-bit integer`

Value of the status register of the IEEE corprocessor. You can assign a new value to this field if you want to change the contents of the status register. The most common use of this field is to change the rounding mode.

`operation` `32-bit integer`

Indicates which floating-point operation caused the trap. The values of this field are associated with IEEE floating-point operations as follows:

Hex Value	**Operation**
18	ADD
19	SUB
1A	MPY
1B	DIV
1C	REM
04	SQRT
03	ABS
05	RND
08	CNVFF
0A	CNVFX

09	CNVXF
10	CMP

Note The value of the `operation` record field is formed by extracting the `OP` and `CLASS` fields from the instruction that caused the trap. The instruction appears as follows:

```
0                          26 27  28 29  31
 |--------------------------|------|------|
 |     all zeros            |Class |  OP  |
 |--------------------------|------|------|
```

`format` `32-bit integer`

Indicates whether the operation that caused the trap had 32-bit, 64-bit, or 128-bit operands.

If the operation is not a `CONVERT`, then the following are the values of the `format` field:

Value	Format
0	32-bit
1	64-bit
3	128-bit

If the operation is a `CONVERT`, then these are the values of the `format` field:

Value	Format
1	The source is 64-bit; the result is 32-bit.
3	The source is 128-bit; the result is 32-bit.
4	The source is 32-bit; the result is 64-bit.
7	The source is 128-bit;

		the result is 64-bit.
	12	The source is 32-bit; the result is 128-bit.
	13	The source is 64-bit; the result is 128-bit.

`source_op1_ptr` `32-bit address`

Address of the first operand. This operand can be a 32-bit, 64-bit, or 128-bit floating-point number depending upon the operation and format. The address is properly aligned, with 32-bit items on a word boundary, and 64-bit and larger items on a double-word boundary.

`source_op2_ptr` `32-bit address`

Address of the second operand. This operand can be a 32-bit, 64-bit, or 128-bit floating-point number depending upon the operation and format. For operations that require only one operand, this field must be ignored.

`result_ptr` `32-bit address`

Points to the result of the operation that resulted in the exception condition. This address can point to a 32-bit, 64-bit, or 128-bit result depending upon the operation and format. You can examine the result and replace it with the desired value.

Note 128-bit floating-point numbers are not yet implemented.

3000 Mode Traps

The 3000 Mode traps include the following conditions:

- 3000 Mode Double Precision Divide By Zero
- 3000 Mode Double Precision Overflow
- 3000 Mode Double Precision Underflow

- 3000 Mode Floating Point Divide By Zero
- 3000 Mode Floating Point Overflow
- 3000 Mode Floating Point Underflow

The record structure for a 3000 Mode trap condition is as follows:

```
 0  |--------------------------------|
    |     instruction                |
 4  |--------------------------------|
    |     offset                     |
 8  |--------------------------------|
    |     space_id                   |
12  |--------------------------------|
    |     error_code                 |
16  |--------------------------------|
    |     result_ptr                 |
    |--------------------------------|
```

The `result_ptr` record field is a 32-bit address that points to the result of the floating-point operation. You can examine the result and replace it with the desired value. The address points to a 64-bit value or a 32-bit value depending upon the type of trap (double-precision or floating-point).

Note An NM arithmetic trap handling routine differs from a CM arithmetic trap handler in its calling sequence [refer to the *Introduction to MPE XL for MPE V Programmers* (30367-90005)] and the method by which the trap routine obtains error information. However, you need supply only the version for the mode in which you invoke intrinsics.

You do not need to modify existing CM applications that use an arithmetic trap handler in order to run them on MPE XL. Likewise, new NM applications need not specify a CM version of their NM trap handling routine. Only those doing mixed-mode programming, who invoke intrinsics in both modes, need to specify and arm trap handling routines in both modes in order to capture all possible arithmetic traps.

Caution A user-defined trap handling procedure cannot perform a `goto` out of that procedure. The state of the process and the program results are not predictable after a non-local `goto`. Performing an ESCAPE (HP Pascal/XL) or completing the trap handling procedure are the only valid ways to return.

Examples

To use the `ARITRAP`, `HPENBLTRAP`, and `XARITRAP` intrinsics to enable/disable or arm/disarm the user-written arithmetic trap handler, you must do the following:

- Declare the intrinsics in your source, using whatever conventions are appropriate to your language.
- Declare the trap handling procedure, using the appropriate formal parameters.

Note Since you can provide only one **plabel** to the `XARITRAP` intrinsic, your arithmetic trap handling procedure must handle all types of armed traps associated with that particular intrinsic.

Example 2-1 is an HP Pascal/XL code excerpt that illustrates how you can handle the IEEE Floating Point Divide By Zero trap condition.

When the program containing this excerpt is executed, it results in a IEEE Divide By Zero exception, and the user-written trap handling routine is invoked. The trap handler replaces the result of the division with the desired value. The output of this program is the value of `maxlongreal`.

Example 2-1 Arithmetic Trap Handler

```
(* Declaring record to receive trap information        *)
(* returned by trap mechanism on a trap condition      *)

PROGRAM XAMPL21(output);
```

```
TYPE
   real_ptr   = ^real;
   long_ptr   = ^longreal;

   user_info_rec = record
        instruction  :  integer;
        offset       :  integer;
        space_id     :  integer;
        error_code   :  integer;
        status       :  integer;
        operation    :  integer;
        format       :  integer;
        source1_ptr  :  localanyptr;
        source2_ptr  :  localanyptr;
        result_ptr   :  localanyptr;
   end;  (* record *)

   user_info_ptr = ^user_info_rec;

(* Constants for the trap procedure *)

CONST
   (* mask for trapping all ieee exceptions *)
   ieee_mask = hex('0007C000');

   (* error code for divide by zero *)
   fdiv_zero = hex('00020000');

   (* maximum longreal value *)
   maxlongreal = 1.8E308;

   (* maximum real value *)
   maxreal = 1.8E308;

VAR
   L1, L2, L3  : longreal;
   oldmask     : integer;
   oldplabel   : integer;
```

```
PROCEDURE ARITRAP; intrinsic;
PROCEDURE XARITRAP; intrinsic;
```

Example 2-1 Arithmetic Trap Handler, continued

```
(* Trap handling routine *)

PROCEDURE Trap_Handler(user_info : user_info_ptr);

VAR
   long_res_ptr : long_ptr;
   real_res_ptr : real_ptr;

BEGIN

(* Handle only divide by zero; ignore all others *)

With user_info^ do
   if ( error_code = fdiv_zero ) then
      BEGIN

(* Change the value of the result *)

         if ( format = 0 ) then
            BEGIN
               real_res_ptr := result_ptr;
               real_res_ptr^ := maxreal;
            END
         else if ( format = 1 ) then
                 BEGIN
                    long_res_ptr := result_ptr;
                    long_res_ptr^ := maxlongreal;
                 END;
      END;
END;

(* Main program *)
```

```
BEGIN
   ARITRAP(1);     (* enable all traps *)
   XARITRAP( ieee_mask, baddress( Trap_Handler),
             oldmask, oldplabel);
   (* arm all ieee traps *)

   L1 := 233.0;
   L2 := 0.0;
   L3 := L1/L2;   (* ieee divide by zero *)

(* The following statement is executed upon return from *)
(* the trap handler; the value of L3 is maxlongreal     *)

   writeln(L3);

END.

(* end example 2-1 *)
```

3

MPE XL Code-related Traps

Code-related traps result from illegal or erroneous coding. The following are the kinds of traps that can result:

- Data memory protection trap (caused by nil pointer referencing, illegal alignment, or accessing a data area with improper access rights)
- Illegal instruction trap
- Instruction memory protection trap
- Break instruction trap
- Privileged operation trap
- Privileged register trap
- Invalid data pointer (caused when an illegal data virtual address is specified)
- Invalid code pointer (caused when the target of a branch is not a valid virtual address)

Code-related traps are permanently enabled.

Whenever a code-related trap condition occurs, the MPE XL trap subsystem takes one of the following actions:

1. If you have executed an HP Pascal XL TRY statement, control is passed to the RECOVER block by doing an ESCAPE. However, if a trap is caused because the program is branching to an invalid address, then no ESCAPE is executed, and the program aborts with an error message.
2. If you have not executed an HP Pascal XL TRY statement, an error message is output and the process aborts.

4

MPE XL CONTROL-Y Traps

If you are running a program in an interactive session, you can arm a special trap that transfers control in the program to a trap handling procedure whenever a subsystem break signal is entered from the session terminal. On most terminals, you transmit the subsystem break signal by pressing the CTRL Y key. To do this, press the Y key while holding down the CTRL key. For this reason, subsystem break traps are commonly called CONTROL-Y traps.

Note You can change the key that causes such a subsystem break. Refer to the discussion of **controlcode** 25 of the `FCONTROL` intrinsic in the *MPE XL Intrinsics Reference Manual* (32650-90028).

There are two MPE XL intrinsics that are used in handling CONTROL-Y traps:

- `XCONTRAP`
- `RESETCONTROL`

The `XCONTRAP` intrinsic arms or disamrs the user-written CONTROL-Y trap handling procedure. When a process is initiated, it has no CONTROL-Y trap handler. Only one process in a session can receive a CONTROL-Y trap at any one time. The process that called `XCONTRAP` most recently receives the next CONTROL-Y trap. Once a process has received a CONTROL-Y trap, it cannot receive another until it calls the `RESETCONTROL` intrinsic. Only processes running in a session can arm CONTROL-Y traps. The trap handler can be any procedure in the program or in the libraries to which the program is bound. The CONTROL-Y trap handler has no parameters.

The `RESETCONTROL` intrinsic resets the CONTROL-Y trap for a process so the process can accept another subsystem break signal. The process must have previously armed a CONTROL-Y trap handler with the `XCONTRAP` intrinsic. After your CONTROL-Y trap handler has been invoked, you should call `RESETCONTROL` when you are ready to receive another subsystem break signal. `RESETCONTROL` can be called from within the CONTROL-Y trap handler, or from any other procedure.

Note Once your process has received a subsystem break signal, only a call to `RESETCONTROL` allows it to receive another such signal. Calling `XCONTRAP` again does not have this effect.

Discussions of these intrinsics follow.

XCONTRAP

Arms or disarms the user-written CONTROL-Y trap handling procedure.

Syntax

The `XCONTRAP` intrinsic is called as follows:

`XCONTRAP(`**plabel,oldplabel**`);`

Parameters

plabel **32-bit signed integer passed by value (required)**

The plabel of your CONTROL-Y trap handling procedure. This plabel can be either an NM or a CM plabel. If this value is 0, `XCONTRAP` disarms CONTROL-Y traps for this process.

How you obtain the external plabel of your NM trap handling procedure depends on your programming language. In HP Pascal/XL, for example, you can obtain the plabel by using the `waddress` function. Supply the name of your CONTROL-Y trap handler as an argument to `waddress`.

To pass a CM plabel, set it up as follows:

1. Obtain the 16-bit external CM plabel of your CM CONTROL-Y trap handler. One way to do this is by using the `LOADPROC` intrinsic.
2. Pass this 16-bit plabel in the following 32-bit format:

Bits	**Setting**
(0:16)	16-bit external CM plabel.
(16:13)	Reserved. Set to zero.
(29:1)	Set to 1.
(30:1)	Set to 0.
(31:1)	Set to 1.

oldplabel **32-bit signed integer passed by reference (required)**

Returns the plabel of your process' previous CONTROL-Y trap handler. This plabel can be either a CM or NM plabel, as described above. If no plabel was previously configured, **oldplabel** returns 0.

Condition Codes

CCE	Request granted. Trap armed.
CCG	Request granted. Trap disarmed.
CCL	Request denied because of an illegal **plabel**, or because `XCONTRAP` was called from a job.

RESET CONTROL

Allows a process to accept another CONTROL-Y signal.

Syntax

The `RESETCONTROL` intrinsic is called as follows:

```
RESETCONTROL;
```

Parameters

None.

Condition Codes

CCE	Request granted.
CCG	Not returned by this intrinsic.
CCL	Request denied because the trap procedure was not invoked.

Handling CONTROL-Y Traps

When more than one process is currently running within your process tree structure, the CONTROL-Y signal interrupts the last process to arm the trap.

When a process is interrupted by a CONTROL-Y signal, the following occur:

1. The input/output transactions pending between the process and the terminal are completed and flagged as though all were completed successfully.
2. Control is transferred to the trap procedure. This procedure executes at the same execution level (either privileged or nonprivileged) as the interrupted user program.
3. Control returns from the trap procedure to the interrupted program or procedure.

a. If the interrupted program or procedure was waiting for completion of input/output (reading from or writing to the terminal) when the CONTROL-Y signal was received, the program continues execution immediately after the `FREAD` or `FWRITE` call. These intrinsics will indicate successful completion.

b. If the CONTROL-Y signal was received during reading, the number of characters typed in before this signal is returned to you as the value of `FREAD`. The "carriage" position is unchanged.

If you send another CONTROL-Y signal, it is ignored unless you issued a call to the `RESETCONTROL` intrinsic at some point prior to the signal.

CONTROL-Y trap handlers differ from other trap handlers in that a process cannot arm a CM and an NM trap handler simultaneously. If the last call to `XCONTRAP` armed an NM trap handler, then the next CONTROL-Y trap invokes this procedure. If the program was running in CM at the time the CONTROL-Y trap occurred, the system actually switches to NM to enter the trap handler. The converse is also true.

When called in NM, `XCONTRAP` can arm either a CM or an NM trap handler. The old plabel returned can be either CM or NM also.

When called in CM, `XCONTRAP` can accept only CM plabels and returns only CM plabels. If `XCONTRAP` is called to configure a CM trap handler and the process' previous CONTROL-Y trap handler was an NM procedure, `XCONTRAP` returns 0 as the **oldplabel** value. This occurs because NM plabels are 32-bits, while a call to `XCONTRAP` in CM can return only a 16-bit plabel value.

To use the `XCONTRAP` and `RESETCONTROL` intrinsics to enable/disable and arm/disarm user-written CONTROL-Y trap handler, you must do the following:

- Declare the intrinsics in your source, using whatever conventions are appropriate to your language.
- Declare the trap handling procedure, with no parameters and no functional return.

Caution A user-defined trap handling procedure cannot perform a `goto` out of that procedure. The state of the process and the program results are not predictable after a non-local `goto`.

Performing an ESCAPE (HP Pascal/XL) or completing the trap handling procedure are the only valid ways to return.

5

MPE XL Software Library Traps

The software library trap reacts to errors that occur during execution of procedures from the compiler libraries. You can arm or disarm the user-written software library trap handing procedure by calling the `XLIBTRAP` intrinsic.

When a program begins execution, the user-written subsequently library trap hander is disarmed automatically. If armed by the `XLIBTRAP` intrinsic, and subsequently activated by an error, the user-written software library trap handler executes.

This trap handler, in turn, returns to your program four words containing the stack marker created when the library procedure was called by your program. In addition, the trap handling procedure returns an integer representing the error number. Although you define the trap handling procedure, it must conform to the special format discussed in the *HP 3000 Compiler Library Reference* (30000-90028).

Note Upon exiting most trap handling procedures, control returns to the instruction following the one that activated the trap. In the case of the library trap, however, you can specify that the process be aborted when control exists from the trap handling procedure.

A discussion of the `XLIBTRAP` intrinsic follows.

XLIBTRAP

Arms or disarms the user-written software library trap handling procedure.

Syntax

The `XLIBTRAP` intrinsic is called as follows:

`XLIBTRAP(`**plabel,oldplabel**`);`

Parameters

plabel **32-bit signed integer passed by value (required)**

The address of your trap handling procedure. If the value of **plabel** is 0, the trap handler is disarmed.

oldplabel **32-bit signed integer passed by reference (required)**

Returns the plabel of your process' previous software library trap handler. If no plabel was previously configured, **oldplabel** returns 0.

Condition Codes

CCE	Request granted. Trap armed.
CCG	Request granted. Trap disarmed.
CCL	Request denied because of an illegal **plabel**.

Handling Software Library Traps

The subsequent paragraphs discuss issues relevant to the task of handling software library traps, as follows:

- Parameters of the library trap handler
- Escape from the library trap handler
- Failing to arm the library trap handler
- Calling the `XLIBTRAP` intrinsic

Parameters

The user-written software library trap handler has the following three parameters:

- Information-pointer
- Error-code
- Abort flag

Information-pointer

The `information-pointer` parameter is a pointer to a record that contains information about the procedure that invoked the library routine. This record can contain additional information specific to a particular library and error condition. for example, the HP Pascal/XL run-time library passes this information. The HP FORTRAN 77/XL library, on the other hand, passes the address of the result value, which you can change. For more information on their relevance of this record in your programming language, refer to the appropriate language reference manual.

On MPE V/E-based systems, the parameter `USERSTACK` is passed to the library trap handler. `USERSTACK` is a pointer to the base of the stack marker that is placed on the stack when your program called the compiler library. On MPE XL-based systems, the `information-pointer` parameter is the functional equivalent of `USERSTACK`. You define the record pointed to by this parameter as follows:

0 31

```
|-----------------------------------|
| SPACE ID of procedure             |
|-----------------------------------|
| OFFSET of procedure               |
|-----------------------------------|
| SP (stack pointer) of procedure   |
|-----------------------------------|
| DP (data pointer) of procedure    |
|-----------------------------------|
```

These fields describe the environment of the procedure that called the library routine.

For more information on the `information-pointer` record, refer to the appropriate language reference manual.

Error-code

The `error-code` parameter is a number indicating the type of compiler library error. This parameter is a 32-bit integer by reference that has the following format:

```
 0               15 16              31
|-----------------+-----------------|
| error number    | subsystem number |
|-----------------+-----------------|
```

For a complete listing of the error number values, refer to Appendix A. The subsystem number depends on your programming language.

Abort-flag

Your library trap handler returns a value in the `abort-flag` parameter, a 32-bit integer by reference.

```
 0                                31
|-----------------------------------|
| Abort Flag                        |
|-----------------------------------|
```

If `abort-flag` is set to 1 before the library trap handler completes execution,the compiler liberary aborts your program with the standard error message (just as if no trap handling procedure had been executed). If `abort-flag` is set to 0, the compiler library does not abort your program, and no error message is printed. Processing continues with the result as modified by your trap handling procedure.

Escaping the Library Trap Handler

The library trap handler can execute an ESCAPE if you have previously executed an HP Pascal/XL TRY statement. Execution resumes with your RECOVER block. An ESCAPE from the library trap handler is possible, because all frames on the stack conform to the stack-unwinding conventions.

Unarmed Library Trap

If you do not arm the library trap handler, the compiler library takes a default action. This default action may be different for different languages. For example, the default action for an HP Pascal/XL library is to do an ESCAPE.

Calling XLIBTRAP

The following is an HP Pascal/XL example of how you can arm a library trap handler. Let `My_Library_Trap_Handler` be the name of the user-written library trap handler. It is declared as follows:

```
Procedure My_Library_Trap_Handler
                    ( VAR information_rec : Pstmrk;
                      VAR errorcode : integer;
                      VAR abortflag : integer);
```

To arm the library trap handler, call the `XLIBTRAP` intrinsic as follows:

```
XLIBTRAP (baddress (My_Library_Trap_Handler),
          old_handler);
```

Note An NM library trap handling routine differs from a CM library trap handler in its calling sequence [refer to the *Introduction to MPE XL for MPE V Programmers* (30367-90005)] and the method by which the trap routine obtains error information. However, you need supply only the version for the mode in which you invoke intrinsics.

You do not need to modify existing CM applications that use a library trap handler in order to run them on MPE XL. Likewise, new NM applications need not specify a CM version of their NM trap handling routine. Only those doing mixed-mode programming, who invoke intrinsics in both modes, need to specify and arm trap handling routines in both modes in order to capture all possible library traps.

Caution A user-defined trap handling procedure cannot perform a `goto` out of that procedure. The state of the process and the program results are not predictable after a non-local `goto`. Performing an ESCAPE (HP Pascal/XL) or completing the trap handling procedure are the only valid ways to return.

Examples

To use the `XLIBTRAP` intrinsic to arm/disarm the user-written library trap handler, you must do the following:

- Declare the intrinsic in your source, using whatever conventions are appropriate to your language.
- Declare the trap handling procedure, using the appropriate formal parameters.

Example 5-1 is an HP Pascal/XL code excerpt that describes the algorithm for implementing user handling of library traps:

Example 5-1 Software Library Trap Handler

```
(* The following is an example of using the XLIBTRAP
   intrinsic to catch Pascal library errors.  In this
   example, the user trap handler ignores the file
   close errors and aborts the program for other
   library errors.                                    *)

(* Declare the following record as the information
   record.                                            *)
```

```
TYPE
   Pstmrk = record
      user_SpaceID,
      user_Offset,
      user_StackPointer,
      user_DataPointer : integer;
   end;  (* record *)

VAR
   Paserr_CloseError : integer;
   oldplabel : integer;

PROCEDURE XLIBTRAP; intrinsic;
```

Example 5-1 Software Library Trap Handler, continued

```
PROCEDURE My_Library_Trap_Handler
                     ( Var info_rec   : Pstmrk;
                       Var errorcode : integer;
                       Var abortflag : integer);

BEGIN

(* ignore file close errors *)

   if ( errorcode = Paserr_CloseError ) then
      BEGIN
      writeln ('File close error, continue execution');
      abortflag := 0;   (* no abort *)
      END
   else
      abortflag := 1;   (* abort *)
END;  (* My_Library_Trap_Handler *)

BEGIN

(* arming the user-written software library trap handler *)
```

```
   XLIBTRAP( baddress(My_Library_Trap_Handler),
             oldplabel);

(* remainder of program *)

END.
```

6

MPE XL Software System Traps

A software system trap is a software-detected error that occurs while system code (typically a system intrinsic) is executing. These traps result in the calling process being aborted. You can intercept system traps by supplying and arming a system trap handling routine. This trap handler can obtain information about the error and prevent the process from being aborted.

Some intrinsics are designed to abort the calling process when certain errors are detected. Typical errors are:

- Illegal access. You attempt to access an intrinsic for which you do not have access capability.
- Illegal parameters. You pass intrinsic parameters that are not defined for your environment.
- Illegal environment. You did not pass a required intrinsic parameter.
- Resource violation. The resource you requested is either illegal or outside the constraints imposed by MPE XL.

The `XSYSTRAP` intrinsic arms (or subsequently disarms) the user-written system trap handling procedure.

When a program begins execution, the user-written system trap handler is disarmed automatically. If armed by the `XSYSTRAP` intrinsic, and subsequently activated by an error, the uwer-written system trap handler executes.

A discussion of the `XSYSTRAP` intrinsic follows.

XSYSTRAP

Arms or disarms the user-written system trap handling procedure.

Syntax

The `XSYSTRAP` intrinsic is called as follows:

`XSYSTRAP(`**plabel,oldplabel**`);`

Parameters

plabel — **32-bit signed integer passed by value (required)**

The address of your trap handling procedure. If the value of **plabel** is 0, the software trap handler is disarmed.

oldplabel — **32-bit signed integer passed by reference (required)**

Returns the plabel of your process' previous software system trap handler. If no plabel was previously configured, **oldplabel** returns 0.

Condition Codes

CCE	Request granted. Trap armed.
CCG	Request granted. Trap disarmed.
CCL	Request denied because of an illegal **plabel**.

Handling Software System Traps

Information relating to the system trap is passed to the system trap handling routine through its four parameters:

trapcode **32-bit signed integer passed by value**

A constant value that is useful when the same trap handler is used to handle different types of system traps.

intrinsicnum **32-bit signed integer passed by value**

A unique intrinsic identifier (ID).

See Appendix C for a complete list of intrinsic numbers.

intrinsicerr **32-bit signed integer passed by value**

A number that identifies the error detected by the intrinsic. If the error is greater than 20, the error is specific to a given intrinsic. Otherwise, the number identifies a general intrinsic error.

parmnum **32-bit signed integer passed by value**

A number that identifies the error-causing parameter that was passed to the intrinsic. If the error is not caused by a parameter, then zero is passed as the **parmnum** value.

Note

An NM system trap handling routine differs from a CM system trap handler in its calling sequence [refer to the *Introduction to MPE XL for MPE V Programmers* (30367-90005)] and the method by which the trap routine obtains error information. However, you need supply only the version for the mode in which you invoke intrinsics.

You do not need to modify existing CM applications that use a system trap handler in order to run them on MPE XL. Likewise, new NM applications need not specify a CM version of their NM trap handling routine. Only those doing mixed-mode programming, who invoke intrinsics in both modes, need to specify and arm trap handling routines in both modes in order to capture all possible system traps.

Caution A user-defined trap handling procedure cannot perform a `goto` out of that procedure. The state of the process and the program results are not predictable after a non-local `goto`. Performing an ESCAPE (HP Pascal/XL) or completing the trap handling procedure are the only valid ways to return.

Examples

To use the `XSYSTRAP` intrinsic to arm/disarm the user-written system trap handler, you must do the following:

- Declare the intrinsic in your source, using whatever conventions are appropriate to your language.
- Declare the trap handling procedure, using the appropriate formal parameters.

Note Since you can provide only one **plabel** to the `XSYSTRAP` intrinsic, your software system trap handling procedure must handle all types of software system traps.

Example 6-1 is an HP Pascal/XL program that illustrates the declaration and arming of a system trap handler, as well as an erroneous intrinsic call that would invoke the declared trap handler.

Example 6-1 Software System Trap Handler

```
Program foo (input,output);

procedure XSYSTRAP; intrinsic;
procedure activate; intrinsic;
procedure quit; intrinsic;

procedure sys_trap_procedure (trap_code    : integer;
                              intrin_num   : integer;
                              intrin_error : integer;
                              intrin_parm  : integer);
```

```
{ This is the trap handling routine that will be invoked }
{ once a system trap occurs.                             }

begin

   writeln (output, 'Now in the trap handling routine.');
   writeln (output);
   writeln (output, 'Trap code        = ', trap_code);
   writeln (output, 'Intrinsic number = ', intrin_num);
   writeln (output, 'Error            = ', intrin_error);
   writeln (output, 'Parameter        = ', intrin_parm);

end;  {sys_trap_procedure}

procedure arm_systrap_routine;

{ This routine calls XSYSTRAP with the plabel of the   }
{ preceding system trap handling routine.              }

var
   plabel    : integer;
   oldplabel : integer;

begin

  plabel := waddress (sys_trap_procedure);
  XSYSTRAP (plabel, oldplabel);   { pass the offset }

end;   { arm_systrap_procedure }
```

Example 6-1 Software System Trap Handler, continued

```
begin

{ This is the program's outer block.  First set up   }
{ sys_trap_procedure to be the system trap handling  }
{ routine.  Then attempt to activate the CI process, }
{ which should result in a system trap.              }
```

```
{ When sys_trap_procedure exits, control should      }
{ return to the statement following the call to the  }
{ ACTIVATE intrinsic, which is a writeln statement.  }

arm_systrap_routine;
activate (0,0);       {invalid ACTIVATE call }
writeln (output, 'Now back in the program outer block.');

end.   { foo }
```

A

MPE XL Trap Subsystem Escape Codes

The escape codes from the trap library are 32-bit integer values that are read in two 16-bit fields:

```
          +---------------+---------------+
          |  Error Number |   Subsystem   |
          +---------------+---------------+
     Bit   0            15 16            31
```

Bits (0:16) comprise *ecode.errnum*. Bits (16:16) comprise *ecode.subsys*. The subsystem number for the MPE XL trap subsystem is 200. The values of *ecode.errnum* are listed in Table A-1:

Table A-1. Trap Library Escape Codes (Page 1)

Error Number	Escapecode in Hex	Meaning
31	0x001f00c8	HP 3000 Mode Floating Point Divide by Zero
29	0x001d00c8	HP 3000 Mode Floating Point Underflow
28	0x001c00c8	HP 3000 Mode Floating Point Overflow
24	0x001800c8	HP 3000 Mode Double Precision Divide by Zero
25	0x001900c8	HP 3000 Mode Double Precision Underflow
26	0x001a00c8	HP 3000 Mode Double Precision Overflow
30	0x001e00c8	Integer Divide by Zero
27	0x001b00c8	Integer Overflow
23	0x001700c8	Decimal Overflow
22	0x001600c8	Invalid ASCII Digit
21	0x001500c8	Invalid Decimal Digit
18	0x001200c8	Decimal Divide by Zero
17	0x001100c8	IEEE Floating Point Inexact Result
16	0x001000c8	IEEE Floating Point Underflow

Table A-1 Trap Library Escape Codes (Page 2)

Error Number	Escapecode in Hex	Meaning
15	0x000f00c8	IEEE Floating Point Overflow
14	0x000e00c8	IEEE Floating Point Divide by Zero
13	0x000d00c8	IEEE Floating Point Invalid Operation
12	0x000c00c8	Range Errors
11	0x000b00c8	Software-detected Nil Pointer Reference
10	0x000a00c8	Software-detected Misaligned Pointer or Conversion of Long Pointer into Short Pointer Error
09	0x000900c8	Unimplemented Condition Trap
08	0x000800c8	Paragraph Stack Overflow
51	0x003300c8	Illegal Instruction
52	0x003400c8	Data Memory Protection Trap, Hardware-detected Nil or Misaligned Pointer
53	0x003500c8	Illegal Code or Data Virtual Address

Note There are four escape codes that are associated with illegal pointers. If you want to catch code errors due to illegal pointers, you must check for all four escape codes:

```
0x000a00c8
0x000b00c8
0x003400c8
0x003500c8
```

B

Intrinsic Numbers

One of the parameters passed to a user-defined system trap handling routine is the number of the intrinsic that caused the trap. The following lists the system intrinsics and their associated numbers. In some instances, no number is assigned. This occurs when the intrinsic cannot cause an abort.

Intrinsic Name	Intrinsic Number
ABORTSESS	196
ACTIVATE	104
ADJUSTUSLF	83
ALMANAC	406
ALTDSEG	134
ARITRAP	51
ASCII	63
BEGINLOG	216
BINARY	62
CALENDAR	43
CATCLOSE	417
CATOPEN	415
CATREAD	416
CAUSEBREAK	56
CLEANUSL	88
CLOCK	44
CLOSELOG	212
COMMAND	68
CREATE	100
CREATEPROCESS	101
CTRANSLATE	61
DASCII	75
DATELINE	89
DBINARY	74
DEBUG	99
DLSIZE	135
DMOVIN	132
DMOVOUT	133
ENDLOG	217
EXPANDUSLF	84
FATHER	109
FCHECK	10
FCLOSE	8
FCONTROL	13
FDELETE	309
FDEVICECONTROL	310
FERRMSG	307

Intrinsic Name	Intrinsic Number
FINDJCW	86
FINTEXIT	23
FINTSTATE	24
FLABELINFO	25
FLOCK	15
FLUSHLOG	213
FMTCALENDAR	90
FMTCLOCK	91
FMTDATE	92
FOPEN	1
FPARSE	312
FPOINT	6
FREAD	2
FREADBACKWARD	39
FREADDIR	7
FREADLABEL	19
FREADSEEK	12
FREEDSEG	131
FREELOCRIN	31
FRELATE	18
FRENAME	17
FSETMODE	14
FSPACE	5
FUNLOCK	16
FUPDATE	4
FWRITE	3
FWRITEDIR	8
FWRITELABEL	20
GENMESSAGE	No number assigned.
GETDSEG	130
GETINFO	87
GETJCW	73
GETLOCRIN	30
GETORIGIN	105
GETPRIORITY	120
GETPRIVMODE	200
GETPROCID	112

Intrinsic Name	Intrinsic Number
HPSETCCODE	No number assigned.
HPSETDUMP	No number assigned.
HPSWITCHTOCM	No number assigned.
HPUNLOADCMPROCEDURE	No number assigned.
INITUSLF	82
IODONTWAIT	37
IOWAIT	22
JOBINFO	180
KILL	102
LOADPROC	80
LOCKGLORIN	34
LOCKLOCRIN	32
LOCRINOWNER	36
LOGINFO	215
LOGSTATUS	214
MAIL	106
MYCOMMAND	71
NLAPPEND	412
NLCOLLATE	402
NLCONVCLOCK	409
NLCONVCUSTDATE	408
NLFMTCALENDAR	413
NLFMTCLOCK	410
NLFMTCUSTDATE	407
NLFMTDATE	414
NLGETLANG	411
NLINFO	400
NLKEYCOMPARE	405
NLREPCHAR	403
NLSCANMOVE	401
NLTRANSLATE	404
OPENLOG	210
PAUSE	45
PRINT	65
PRINTFILEINFO	21
PRINTOP	66
PRINTOPREPLY	67

Intrinsic Name	Intrinsic Number
TIMER	40
SUSPEND	103
TERMINATE	60
UNLOADPROC	81
UNLOCKGLORIN	35
UNLOCKLOCRIN	33
WHO	69
WRITELOG	211
XARITRAP	50
XCONTRAP	54
XLIBTRAP	52
XSYSTRAP	53
ZSIZE	136