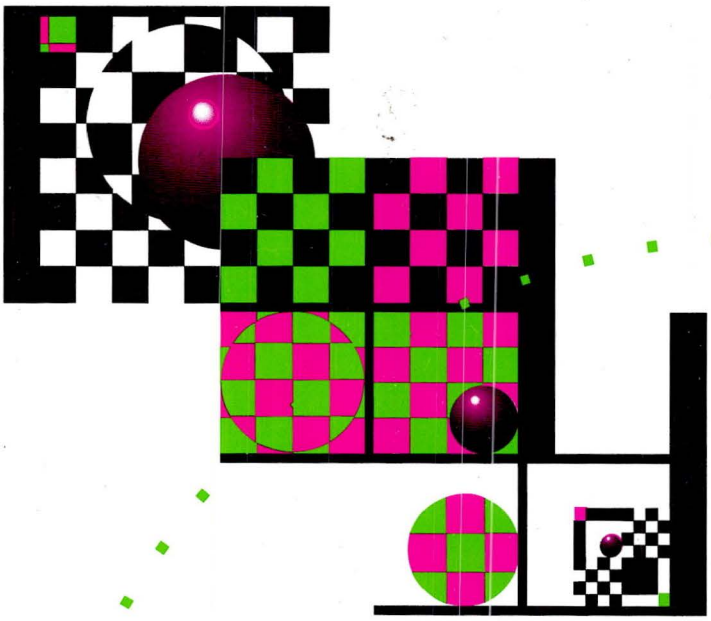


CONVEX

CXdb Reference

Volume 2
Concepts, Windows, and Messages

Third Edition

CONVEX Computer Corporation
3000 Waterview Parkway
P.O. Box 833851
Richardson, TX 75083-3851
United States of America
(214)497-4000

CXdb Reference

Order No. DSW-472

Third Edition
November 1994

CONVEX Press
Richardson, Texas
United States of America

CXdb Reference

Order No. DSW-472

Printed in the United States of America

Revision Information for

CXdb Reference

Edition	Document No.	Description
Third Edition	710-015022-000	Released November, 1994, with CXdb V3.1. This two-volume set replaces the previous two-volume reference set that consisted of the *CONVEX CXdb Reference: Commands and Parameters* (Document No. 710-015430-003) and the *CONVEX CXdb Reference: Concepts and Messages* (Document No. 710-024130-000).
First Edition	710-015430-003	Initial release, November 1992. This document describes commands and command parameters used in CONVEX CXdb V2.0. This document is part of a two-volume set that also includes the *CONVEX CXdb Reference: Concepts and Messages* (Document No. 710-024130-000). Together, these two volumes replace the *CONVEX CXdb Reference* (Document No. 710-015430-002).
Second Edition	710-015430-002	Released December 1991. Documents CONVEX CXdb V1.1.
First Edition	710-015430-001	Initial release, June 1991. Documents CONVEX CXdb V1.0.

Contents

Volume 1: Commands and Parameters

How to use this book

Purpose and audience

The *CXdb Reference* describes each of the CXdb commands and its related parameters. It also describes major concepts such as compiling for CXdb, and it explains how to use the windows of CXdb's graphical user interface.

This manual is intended as a complete reference source for new users as well as users who are already familiar with CXdb. You are not expected to read this entire manual before using CXdb. You should begin by reading the topic, "getting started with CXdb," in the "Concepts" section of this manual. You can then read other topics if you need them.

All the reference topics contained in this manual are also available through the CXdb online help system.

Organization

The *CXdb Reference* is organized into two volumes.

Volume 1 contains the following chapters:

- **Chapter 1, "Commands"** — Contains descriptions, syntax rules and examples for the CXdb commands.
- **Chapter 2, "Parameters"** — Describes major parameters used in CXdb commands.

Volume 2 contains the following chapters:

- **Chapter 3, "Concepts"** — Explains the major concepts involved in using CXdb.
- **Chapter 4, "Windows"** — Describes the windows and dialogs available in CXdb's graphical user interface.
- **Chapter 5, "Messages"** — Lists and explains CXdb messages.

Each volume contains a master index of the *CXdb Reference*.

Notational conventions

This section describes notational conventions used in this book.

Command syntax

The following example illustrates command syntax:

```
(CXdb) command <param1> [, ...] {a | b} [<param2>]
  ①      ②       ③       ④     ⑤       ⑥
```

1. `(CXdb)` is the CXdb command prompt.
2. **`command`** must be typed as it appears.
3. *<param1>* indicates a parameter that you must supplied.
4. The horizontal ellipsis in brackets [**`,`** ...] indicates that you can specify additional parameters of type *<param1>*.
5. You must specify either **`a`** or **`b`**.
6. Brackets [*<param2>*] indicate an optional parameter.

General conventions

- **`Bold constant-width font`** identifies user input in examples.
- *italics*
  - Designate user-supplied variables in a command example (when enclosed in <>).
  - Indicate document titles.
- `Constant-width font` designates input and output, including:
  - Command names and options.
  - System calls.
  - Program statements, command output, and error messages returned.
- Horizontal ellipsis (...) shows repetition of the preceding item(s).
- Vertical ellipsis shows that lines have been left out of an example.

- Words and abbreviations that indicate keyboard keys you press are identified in a distinctive bold type. For example, **RETURN** refers to the carriage return key. Words separated by a hyphen indicate two keys that you must press simultaneously. For example, **CTRL-c** indicates that you must press and hold down the **CTRL** key, and then press the **c** key.
- References to man pages appear in the form `exec(2)`, where the name of the man page is followed by its section number enclosed in parentheses.
- The shell prompt is shown as a percent sign (%).
- Unless otherwise indicated, source code examples are in Fortran. Where there are differences between how CXdb handles C and Fortran, examples in C are also shown.
- Fortran examples are shown in uppercase letters. You can, however, use lowercase.

Notes and cautions

NOTE: A NOTE highlights information that may be of particular interest regarding the software or your files.

Caution

A Caution highlights information that could affect the performance of the software or your system.

Architecture dependencies

The architecture of the computer system can affect some aspects of CXdb and the process you are debugging. These architecture dependencies are indicated in several ways throughout this book.

If the entire reference topic applies to a particular architecture, it is marked by a symbol like the following:

The above symbol means that the reference topic applies to SPP Series machines only.

In other cases, architecture dependencies are indicated by the title of a section or by a notation in parentheses.

Associated documents

For additional information about CXdb, Fortran and C optimizations, or the SPP Series architecture and programming model, refer to:

- *CXdb Quick Reference* (DSW-474) — A brief summary of CXdb windows and commands, including mouse and keyboard shortcuts.
- *CXdb Online Tutorial* (X Windows mode only) — A quick introduction to CXdb, including examples that you can execute while reading the tutorial.
- *Exemplar Programming Guide* (DSW-067) — Describes efficient methods for programming in Convex C and Fortran on Exemplar (also known as SPP Series) computers. Topics covered include the SPP Series programming model, automatic optimizations, basic and advanced manual optimizations, and the Convex Parallel Support Library (CPSlib).
- *Exemplar Architecture* (DSW-014) — Describes the Convex implementation of scalable parallel processing on Exemplar (SPP Series) machines.
- *C Optimization Guide* (DSW-089) — Describes the types of optimizations available in Convex C and shows you how to use optimization directives and options.
- *Fortran Optimization Guide* (DSW-034) — Describes the types of optimizations available in Convex Fortran and shows you how to use optimization directives and options.

Ordering documentation

To order the current edition of this or any other CONVEX document, send requests to:

CONVEX Computer Corporation
Customer Service
P.O. Box 833851
Richardson, TX 75083-3851 USA

Include the order number or the exact title.

In some cases, you might not want the latest edition. To order a specific edition of a document, contact your local CONVEX office or call the Technical Assistance Center.

Technical assistance

If you have questions that are not answered by the documentation, contact the CONVEX Technical Assistance Center (TAC). To contact the TAC, use one of the following phone numbers:

- Within the continental U.S., call 1(800)952-0379.
- From Canada, call 1(800)345-2384
- All other locations, contact the nearest CONVEX office.

The `contact` utility

The TAC recommends using the `contact` utility to report a hardware, software, or documentation problem. The `contact` utility is an interactive program that helps the TAC track reports and route them to the CONVEX personnel most qualified to fix a problem.

After you invoke `contact`, it prompts you for information about the problem. When you finish your report, `contact` mails it to the TAC electronically. The TAC notifies you within 48 hours that your report has been received.

Using `contact` requires:

- UNIX-to-UNIX Communication Protocol (UUCP) connection to the TAC.
- Full path name of the program or utility in question.
- Version number of the program or utility in question.

Refer to the contact(1) man page for complete details.

Concepts

3

This chapter contains reference pages that explain the major concepts associated with CXdb. There is a separate reference page for each concept. The reference pages can contain the following sections:

- **Description** — Text explaining the concept.
- **Examples** — One or more examples illustrating the concept, where applicable.
- **Related Commands** — A list of CXdb commands related to the concept. The commands are described in Chapter 1.
- **Related Concepts** — A list of other concepts related to the concept being described. The related concepts are also described in this chapter.
- **Related Parameters** — A list of command parameters related to the concept being described. CXdb parameters are described in Chapter 2.
- **Related Windows** — A list of CXdb windows that relate to the concept being described. You can use these windows only if you are running CXdb in X Windows mode. The windows are described in Chapter 4.

architecture dependencies

Description

This reference page describes features of CXdb that vary with the architecture of the system where your program is running. The items dependent on system architecture are:

- Command availability
- Command output
- Registers
- Core files
- Windows
- Signals
- Floating point mode
- Local variables and scope paths in Fortran

The specific dependencies are listed below by system architecture.

Command availability

The following CXdb commands are available only on the indicated architecture. Commands not listed here are available on all CONVEX architectures.

C Series only:

```
clear default fixed sched
clear default remotewd
clear fixed sched
clear seq
clear sqs
copy
get
info cregisters
info dynamicobject
info vregisters
load object
put
set default fixed sched
set default fpmode
set default remotewd
set directory
```

```
set evalops fpmode
set fixed sched
set fpmode
set remotewd
set seq
set sqs
```

SPP Series only:

```
info control registers
info floating point registers
info space registers
```

Command output

The following CXdb commands produce different output, depending on the system where your program is running:

```
disassemble
display disassembly
info frame
info frame at
info object map
info psw
info registers
info signal
info threads
info vregisters (C Series only)
```

Registers

The following types of registers are available on the indicated architecture.

C2 and C3 Series:

- Address registers
- Communication registers
- Processor status word (psw)
- Program counter (pc)
- Scalar registers
- Vector registers

C4 Series:

- Address registers
- Communication registers
- Event registers
- Processor status word (psw)
- Program counter (pc)

- Scalar registers
- Vector registers

SPP Series:

- Control registers
- Floating point registers
- General registers
- Instruction queue head and tail pointers
- Processor status word (psw)
- Space registers
- Stack pointer (sp)

Core files

The contents of a core file are strongly dependent on the architecture of the system that generated it. On C Series machines, you can obtain significant debugging information from the core image itself without specifying the name of the executable file that generated the core file. However, on SPP Series machines, you must first specify the name of the associated executable file in order to obtain any relevant debugging information from the core image. You can use the `executable` command to specify the name of the executable file.

Windows

The following CXdb windows are available only on the indicated architecture (in X Windows mode only). Windows not listed here are available on all CONVEX architectures (in X Windows mode only).

C Series only:

- Communication Registers window
- Scalar Registers window
- Vector Registers window

SPP Series only:

- Control Registers window
- Floating Point Registers window
- General Registers window
- Space Registers window

Signals

The following signals are available on the indicated architecture.

C Series:

- SIGHUP (1) — Hangup
- SIGINT (2) — Interrupt
- SIGQUIT (3) — Quit
- SIGILL (4) — Illegal instruction
- SIGTRAP (5) — Trace/breakpoint trap
- SIGIOT (6) — IOT trap
- SIGEMT (7) — EMT trap
- SIGFPE (8) — Floating point exception
- SIGKILL (9) — Killed
- SIGBUS (10) — Bus error
- SIGSEGV (11) — Segmentation violation
- SIGSYS (12) — Bad system call
- SIGPIPE (13) — Broken pipe
- SIGALRM (14) — Alarm clock
- SIGTERM (15) — Terminated
- SIGURG (16) — Urgent I/O condition
- SIGSTOP (17) — Stopped
- SIGTSTP (18) — Stopped (terminal)
- SIGCONT (19) — Continued
- SIGCHLD (20) — Child exited
- SIGTTIN (21) — Stopped (tty input)
- SIGTTOU (22) — Stopped (tty output)
- SIGIO (23) — I/O possible
- SIGXCPU (24) — CPU time limit exceeded
- SIGXFSZ (25) — File size limit exceeded
- SIGVTALRM (26) — Virtual timer expired
- SIGPROF (27) — Profiling timer expired
- SIGWINCH (28) — Window size change
- SIGLOST (29) — Resource lost
- SIGUSR1 (30) — User-defined signal 1
- SIGUSR2 (31) — User-defined signal 2

SPP Series:

- SIGHUP (1) — Floating point exception
- SIGINT (2) — Interrupt
- SIGQUIT (3) — Quit
- SIGILL (4) — Illegal instruction (not reset when caught)
- SIGTRAP (5) — Trace trap (not reset when caught)
- SIGIOT (6) — Process abort signal
- SIGEMT (7) — EMT instruction
- SIGFPE (8) — Floating point exception
- SIGKILL (9) — Kill (cannot be caught or ignored)
- SIGBUS (10) — Bus error
- SIGSEGV (11) — Segmentation violation
- SIGSYS (12) — Bad argument to system call
- SIGPIPE (13) — Write on a pipe with no one to read it
- SIGALRM (14) — Alarm clock
- SIGTERM (15) — Software termination signal from kill
- SIGUSR1 (16) — User defined signal 1
- SIGUSR2 (17) — User defined signal 2
- SIGCHLD (18) — Child process terminated or stopped
- SIGPWR (19) — Power state indication
- SIGVTALRM (20) — Virtual timer alarm
- SIGPROF (21) — Profiling timer alarm
- SIGIO (22) — Asynchronous I/O
- SIGWINCH (23) — Window size change signal
- SIGSTOP (24) — Stop signal (cannot be caught or ignored)
- SIGTSTP (25) — Interactive stop signal
- SIGCONT (26) — Continue if stopped
- SIGTTIN (27) — Read from control terminal attempted by a member of a background process group
- SIGTTOU (28) — Write to control terminal attempted by a member of a background process group
- SIGURG (29) — Urgent condition on I/O channel
- SIGLOST (30) — Remove lock lost (NFS)
- SIGRESERVE (31) — Save for future use
- SIGDIL (32) — DIL signal

Floating point mode

The following floating point modes are available on the indicated architecture.

C Series:

- IEEE
- Native
- Dual

SPP Series:

- IEEE

Local variables and scope paths in Fortran

In Fortran, the local variables for a routine are allocated on the call stack if the routine is reentrant code or in memory if the routine is not reentrant. The local variables maintain their values between calls if they are allocated in memory (not reentrant), but they do not maintain their values if they are allocated on the stack (reentrant).

You can use CXdb scope paths to access local variables that are stored in memory or in the current stack frame. However, to access a variable in a reentrant routine that is not in the current stack frame, you must first use the `frame` command to change to the stack frame that contains the desired routine.

The compiler default settings for reentrant Fortran code are different on C Series and SPP Series machines, as indicated below.

C Series:

- By default, the Fortran compiler does not generate reentrant code.
- Compile with the `-re` option to generate reentrant code, if desired.

SPP Series:

- By default, the Fortran compiler generates reentrant code.
- If desired, compile with the `-nore` option to generate code that is not reentrant.

Related Commands

`core` `debug core` `executable`

Related Concepts

registers scope signals

background execution

Description

CXdb process execution commands can be run in the background with respect to other CXdb commands.

NOTE: This does not place your process in the background in relation to the shell.

With background execution, your process executes while you continue to use CXdb. To run a command in the background, type an ampersand (&) after the command. When a command is run in the background, the CXdb command prompt returns, allowing you to enter other CXdb commands. While a command is running in the background, you cannot use any CXdb commands that require the process to be stopped. The command runs in the background until it completes or until process execution stops. Only one command can run in the background at a time.

The following is a list of process execution commands, all of which can be put in the background:

```
continue
finish
next
next instruction
next over
rerun
run
signal process
signal thread
step
step instruction
step over
```

If one of the above commands is running in the background, you can stop the process with the `stop` command. When the process is stopped (or it terminates), the command in the background is terminated. CXdb displays a message when the command finishes executing.

If you include the ampersand (&) with a command that cannot be run in the background, CXdb ignores the ampersand and displays a warning message.

If you want to stop a CXdb command that is not executing in the background, type CTRL-c in the Command window. This aborts the command and also stops execution of the process.

Examples

The following examples illustrate how to run CXdb commands in the background.

```
(CXdb) run &
Command [#15] backgrounded

Starting process [#0]: docexample
(CXdb)
```

The above command begins execution of a new process and runs the command in the background. Process execution continues until the process is stopped or it terminates. Because the command is running in the background, you can stop process execution with the `stop` command.

```
(CXdb) step statement 1000 &
Command [#17] backgrounded

Stepping process [#0/*] by 1000 statements
Command [#17] completed.
(CXdb)
```

The above command steps the process by 1000 statements and runs the command in the background. In this way, you can enter other CXdb commands while you are waiting for the `step` command to finish executing. Because the process is executing, you cannot enter other commands that require the process to be stopped.

Related Commands

`continue`
`finish`
`next`
`next instruction`
`next over`
`run`
`rerun`
`signal process`
`signal thread`
`step`
`step instruction`
`step over`
`stop`

Related Concepts

stepping

Related Windows

Command window

Description

A breakpoint is a predefined eventpoint, or trap, that enables you to stop execution of a process at key locations in your program. When process execution reaches the location of an enabled breakpoint, the set of actions associated with the breakpoint, called the eventpoint handler, is taken.

There are several different ways to set a breakpoint:

- `break instruction` — Sets the breakpoint at the specified instruction address.
- `break line` — Sets the breakpoint at the first instruction address that maps to the specified line number of a source file.
- `break routine` — Sets the breakpoint at the first executable source unit of the specified routine.
- `break source` — Sets the breakpoint at the starting address of the specified source unit.

For commands that accept an address, any valid source language expression can be used to specify the address.

Other commands allow you to interact with existing breakpoints. These commands are:

- `disable event` — Disable the specified eventpoints.
- `disable eventtype` — Disable all eventpoints of the specified type.
- `enable event` — Enable the specified eventpoints.
- `enable eventtype` — Enable all eventpoints of the specified type.
- `info break` — Display information about all existing breakpoints.
- `info event` — Display information about the specified eventpoints.
- `remove event` — Remove the specified eventpoints.
- `remove eventtype` — Remove all eventpoints of the specified type.
- `set ignore` — Set an ignore count for the specified eventpoints.
- `set handler` — Set a handler for the specified eventpoints.
- `set typehandler` — Set the default handler for all eventpoints of the specified type.

Each breakpoint is assigned a unique number. This number is used in subsequent commands when you want to refer to the breakpoint. Optionally, you can specify a debugger variable to be assigned to the breakpoint. You can then use the debugger variable to refer to the breakpoint in subsequent commands.

All breakpoints have a default handler that prints a message telling you that the breakpoint has been reached. You can specify a different set of actions to take for a particular breakpoint, or change the setting of the default handler itself.

Breakpoints, like all eventpoints, can be enabled or disabled. An enabled breakpoint is reached when process execution reaches its address. A disabled breakpoint is treated as if it does not exist, and therefore can never be reached until it is enabled again. You can prevent breakpoints from being reached without having to remove them by disabling them.

Once a breakpoint is reached, one of two things may occur. If the breakpoint has an ignore count, the counter is incremented by one, and process execution continues. If the breakpoint does not have an ignore count, then the breakpoint is triggered. When a breakpoint is triggered, the commands in its eventpoint handler are executed. If the breakpoint does not have its own eventpoint handler, the default eventpoint handler for breakpoints is used.

The ignore count of an eventpoint is the number of times that eventpoint must be reached before being triggered. A counter keeps track of the number of times an eventpoint has been reached. When the counter matches the ignore count, the ignore count is reset to zero. The *next* time the eventpoint is reached, the eventpoint will be triggered.

Multiple eventpoints can exist at the same address. When process execution reaches an address with mulitple eventpoints, the highest-numbered, enabled eventpoint is reached. If this eventpoint has an ignore count, the counter is updated and the next highest-numbered, enabled eventpoint is reached. This process continues until either an eventpoint is triggered or there are no more eventpoints at the address.

Breakpoints are specific to the existing process. They can be set for specific threads of a process as well. Breakpoints can also be removed.

Examples

The following series of examples creates and manipulates several different breakpoints in one process.

```
(CXdb) break line 10

#0: break line, on [#0/*], Enabled, ignore 0/0
        [0x800053de] EXAMPLE in example.f line 10
```

The above command sets a breakpoint at line 10 in the current source file. The eventpoint number for this breakpoint is 0 and the address of the breakpoint is `800053de` in routine `EXAMPLE` at line 10 in the source file example.f.

```
(CXdb) break routine EXAMPLE

#1: break routine, on [#0/*], Enabled, ignore 0/0
        [0x800053b0] EXAMPLE in example.f line 7
```

The above command sets a breakpoint at the first executable source unit in the routine `EXAMPLE`. The first executable source unit of a routine is usually the first statement of a routine after local variables have been declared, unless there are local initializations.

```
(CXdb) break instruction EXAMPLE

#2: break instruction, on [#0/*], Enabled, ignore 0/0
        [0x800053a0] EXAMPLE in example.f line 1
```

The above command sets a breakpoint at the first instruction address in the routine `EXAMPLE`. The first instruction address of a routine is before the preamble (which manages the stack) of the routine. This address is different than that of the previous example, because breakpoint 1 is at the first source unit of `EXAMPLE`.

```
(CXdb) break source 21

#3: break source, on [#0/*], Enabled, ignore 0/0
        [0x800053ee] EXAMPLE in example.f line 11
```

The above command sets a breakpoint at the starting address of source unit 21. The source unit numbers for a given line can be displayed using the `info line` command.

```
(CXdb) info break
Event   Enabled Ignore  proc/td Address         Where
#0          y    0/0      0/*     [0x800053de] EXAMPLE in example.f line 10
#1          y    0/0      0/*     [0x800053b0] EXAMPLE in example.f line 7
#2          y    0/0      0/*     [0x800053a0] EXAMPLE in example.f line 1
#3          y    0/0      0/*     [0x800053ee] EXAMPLE in example.f line 11
```

The above command displays the status of all the existing breakpoints. All of the breakpoints are initially enabled and do not have an ignore count.

```
(CXdb) run
Starting process [#0]: docexample
Process [#0/0] stopped by Bkpt 2, at [0x800053a0] EXAMPLE in example.f line 1
```

The above example begins process execution without passing any arguments to the process. When execution reaches the address `800053a0`, breakpoint 2 is reached because it is enabled. Because it does not have an ignore count, the breakpoint is triggered. The default handler for breakpoints is executed because the breakpoint did not have its own eventpoint handler. The default handler prints the message that tells you what caused process execution to stop.

```
(CXdb) remove event 1,2
Eventpoint 1 removed
Eventpoint 2 removed
```

The above command removes eventpoints 1 and 2 from the process object. These eventpoint numbers will not be reused during this session with CXdb.

```
(CXdb) break line 10 $Middle

#4: break line, on [#0/*], Enabled, ignore 0/0
        [0x800053de] EXAMPLE in example.f line 10

INFO A65: Eventpoint 0 also set a breakpoint at address 0x800053de.
```

The above command sets another breakpoint at line 10 of the current source file. The debugger variable `$Middle` is created and assigned to this eventpoint. CXdb informs you that two eventpoints now reside at the same address.

```
(CXdb) break line 10 {echo "Breakpoint 5 reached" ; resume;}

#5: break line, on [#0/*], Enabled, ignore 0/0
        [0x800053de] EXAMPLE in example.f line 10
    {
        echo "Breakpoint 5 reached" ;
        resume;
    }

INFO A64: Eventpoint 4 also set a breakpoint at address 0x800053de.
```

The above command sets a third breakpoint at line 10. An eventpoint handler is specified for this eventpoint. The handler prints a message and then resumes execution of the process.

```
(CXdb) continue
Resuming execution of Process [#0/*]
Breakpoint 5 reached
Process [#0/0] stopped by Bkpt 3, at [0x800053ee] EXAMPLE in example.f line 11
```

The above example resumes execution of the process. Breakpoint 5 is triggered because it is the highest-numbered, enabled eventpoint at that location (the other two breakpoints at that location are 0 and 4), and it does not have an ignore count. The eventpoint handler for breakpoint 5 prints the message and then resumes execution of the process. Finally, breakpoint 3 stops the process.

```
(CXdb) disable event 5
Eventpoint 5 disabled
(CXdb) set ignore 2 4
Eventpoint 4 will be ignored 2 times
```

The above commands perform two actions. First, breakpoint 5 is disabled. Second, breakpoint 4 is given an ignore count of 2.

```
(CXdb) run
Process [#0] is already running with pid 12650.
Terminate existing process and restart? y
Starting process [#0]: docexample
Process [#0/0] stopped by Bkpt 0, at [0x800053de] EXAMPLE in example.f line 10
```

When process execution reaches address `800053de`, breakpoint 4 is reached because eventpoint 5 is disabled. Breakpoint 4 has an ignore count, so its counter is incremented by one. Because an eventpoint has still not been triggered, breakpoint 0 is reached. Because it is enabled and does not have an ignore count, breakpoint 0 is triggered.

```
(CXdb) info event *

#0: break line, on [#0/*], Enabled, ignore 0/0
        [0x800053de] EXAMPLE in example.f line 10

#3: break source, on [#0/*], Enabled, ignore 0/0
        [0x800053ee] EXAMPLE in example.f line 11

#4: break line, on [#0/*], Enabled, ignore 1/2
        [0x800053de] EXAMPLE in example.f line 10

#5: break line, on [#0/*], Disabled, ignore 0/0
        [0x800053de] EXAMPLE in example.f line 10
    {
        echo "Breakpoint 5 reached" ;
        resume;
    }
```

The above command displays the status of all eventpoints. The `info event` command displays the eventpoint handler of an eventpoint. In this case, the output indicates that breakpoint 5 has its own handler. The output also indicates that breakpoint 5 is disabled, and breakpoint 4 has been ignored once out of a possible 2 times (`1/2`).

```
(CXdb) enable event 5
Eventpoint 5 enabled
```

The above command enables breakpoint 5. This will cause breakpoint 5 to be triggered the next time address `800053de` is reached because it is the highest-numbered, enabled breakpoint.

```
(CXdb) set ignore 0 4
Eventpoint 4 will be ignored 0 times.
```

The above command resets the ignore count to 0 for breakpoint 4.

For more information about the use of eventpoint handlers, refer to the concepts page on eventpoint handlers.

Related Commands

`break instruction`	`break line`
`break routine`	`break source`
`disable event`	`disable eventtype`
`enable event`	`enable eventtype`
`info break`	`info event`
`info eventtype`	`remove event`
`remove eventtype`	`rerun`
`resume`	`run`
`set default handler`	`set ignore`
`set handler`	`set typehandler`

Related Concepts

eventpoints	eventpoint handlers
tracepoints	watchpoints

Related Parameters

debugger-variable	event-handler
language-expression	line-specifier
process-list	thread-list

Related Windows

Event Point dialog

C language expressions

Description

A C language expression is an expression that follows the syntax rules of C. You can use C language expressions in CXdb commands whenever the current source language of your process is C. The current source language is the language of the source file associated with the current stack frame.

In CXdb, the C language expressions must conform to the rules listed below.

1. Identifiers:

- Restrictions:
  - Identifiers are case sensitive.
  - If a file name appears in the scope path prefix to the identifier, and if the file name contains a special character such as dot (.), precede the special character with a backslash (\).
  - If a dollar sign ($) is part of the identifier name, precede the dollar sign with a backslash (\).
  - An identifier qualified by "`const`" is tracked, and a warning is issued if it is modified.
- Program variables can be either:
  - Unqualified
  - Qualified by a scope path prefix
- CXdb debugger variables:
  - Debugger variables must begin with the CXdb scope prefix `cxdb$` or `$.`
  - Debugger variable names are case sensitive.
  - An assignment to a debugger variable modifies it to have the value, type, and precision of the right-hand side of the assignment.
- Hardware registers:
  - Register names must be qualified with the CXdb scope prefix `cxdb$` or `$.`
  - An assignment to a register modifies its value only; its type and precision remain the same.

2. Constants:

- Restrictions:
  - White space is significant.
  - The default precision for integers is obtained from the setting established by the `set evalopts iprecision` command. The initial setting is 4 bytes (32 bits).
  - The default precision for real numbers is obtained from the setting established by the `set evalopts rprecision` command. The initial setting is 8 bytes (64 bits).
  - The type and precision are determined from the syntactic specification, the default precision, or the resulting value of the constant, in that order.
- Integers without suffixes — the precision is determined by one of the following:
  - The default precision
  - The magnitude of the constant
- Integers with suffixes
  - A suffix of `u` or `U` designates unsigned. The precision is determined from the default or from the magnitude of the constant.
  - A suffix of `l` or `L` designates one of the following, depending on the magnitude of the constant:

    ```
    long int (32 bits)
    unsigned long
    long long
    unsigned long long
    ```

  - A suffix of `ll` or `LL` designates one of the following, depending on the magnitude of the constant:

    ```
    long long int (64 bits)
    unsigned long long
    ```

- Floats:
  - For the significand (the integer component followed by the fractional component), the precision is determined from the default.
  - For a significand followed by an exponent, the precision is determined from the default.
  - For a suffixed significand followed by an exponent, the precision is single precision (32 bits) if the suffix is `f` or `F` and double precision (64 bits) if the suffix is `l` or `L`.

- Characters:
  - The ANSI "wide" character set is not supported and not recognized.
  - The ASCII character set is fully supported.
  - The following character escape sequences are supported:

    `\a` — alert (audible or visual)
    `\b` — backspace
    `\f` — formfeed
    `\n` — newline
    `\r` — carriage return
    `\t` — horizontal tab
    `\v` — vertical tab

  - The following trigraph sequences are accepted:

    `??=` yields `#`
    `??(` yields `[`
    `??)` yields `]`
    `??/` yields `\`
    `??'` yields `^`
    `??<` yields `{`
    `??>` yields `}`
    `??!` yields `|`
    `??-` yields `~`

- Enumeration — Variables that are assigned an enumerated type by the program are internally changed to type `int` by the C compiler. For consistency, CXdb also converts any enumerated type to type `int` within cast expressions.
- String literals — The ANSI "wide" string type is not supported and not recognized.

3. Expressions:

- Postfix operators include:
  - Array subscripting
  - Array slices — An array slice is a subscript expression that contains a slice range. The data type of the slice is "array of ...", where the upper and lower bounds of the resulting array are defined by the slice range. The slice operator is dot dot (`..`). If a slice range expression has been applied to at least one subscript of an array, then the other subscripts default to their entire ranges unless they are explicitly specified as either subscript indexes or slice ranges. For example, if the array definition is `y[10][10]` and the slice specification is `y[2..4]`, then the slice used in the evaluation is `y[2..4][0..9]`. However, if a slice range expression is applied to a pointer designator, then only the specified subscripts are applied to the pointer expression.
  - Function calls
  - Structure and union members:

    Selection
    Remote selection
    Bit fields
  - Postfix increment and decrement operators:

    `++` postfix increment
    `--` postfix decrement
- Unary operators include:
  - Prefix operators:

    `++` prefix increment
    `--` prefix decrement
  - Address and indirection operators:

    `&` address of
    `*` indirection
  - Unary arithmetic operators:

    `+` unary plus
    `-` unary minus
    `~` bitwise complement
    `!` logical negation
  - The `sizeof` operator — Computes the size of an object in bytes.

- Cast operator restrictions:
  - `Struct, union, and enum` data types cannot be defined within a cast expression.
  - Function prototype parameters cannot have their data types defined with the cast expression.
  - Because the C compiler internally converts enumerated types to `int` types, CXdb does not have access to the original enumerated type name. Thus, CXdb cannot obtain the type definition when provided the type name in the cast.
- Multiplicative operators include:

  `*` multiply
  `/` divide
  `%` mod left operand by right
- Additive operators include:

  `+` add
  `-` subtract
- Shift operators include:

  `<<` shift left
  `>>` shift right
- Relational operators include:

  `<` test for less than
  `>` test for greater than
  `<=` test for less than or equal to
  `>=` test for greater than or equal to
- Equality operators include:

  `==` test for equal to
  `!=` test for not equal to
- Bitwise operators include:

  `&` AND
  `^` exclusive OR
  `|` inclusive OR
- Logical operators include:

  `&&` test for AND
  `||` test for OR
- The conditional operator (`?:`):
  - If the test condition is true, evaluate the operand to the left of the colon (`:`).
  - Otherwise, evaluate the operand to the right of the colon.

- Assignment operators include:

  = simple assignment
  *= multiply and assign
  /= divide and assign
  %= mod and assign
  += add and assign
  -= subtract and assign
  &= AND and assign
  ^= exclusive OR and assign
  |= inclusive OR and assign
  <<= shift left and assign
  >>= shift right and assign

- The comma operator (,):
  - Evaluate the left operand, then the right.
  - Return the result of the right operand.

Examples

The following examples illustrate the use of C language expressions with various CXdb commands.

```
(CXdb) break routine 0x80004c32
#1: break routine, on [#0/*], Enabled, ignore 0/0
        [0x80004c32] chapter13C`subxa in chapter13C.c line 45
```

The above command sets a breakpoint at the beginning of the routine that contains the hexadecimal address `80004c32`. The expression `0x80004c32` is C language notation for a hexadecimal address.

```
(CXdb) print ++i
(int) 7
```

In the above example, CXdb increments `i` by 1 and assigns this new data value to the program variable `i`. The command also prints the new value of `i`.

```
(CXdb) print i==1
(int) 0
```

The above command evaluates the relational expression `i==1` and prints the result of the evaluation. The resulting value of `0` indicates the relation `i==1` is false.

```
(CXdb) print/x &i
(int*) 0x8008acd4
```

The above command evaluates the expression `&i` and prints the result in hexadecimal (`/x`) format. The C operator `&` returns the address of the program variable `i`. Thus, `8008acd4` is the hexadecimal address of `i`. (The `0x` in front of the address indicates that it is a hexadecimal number.)

```
(CXdb) print table[3][0..3]
int[1][4]
[3][0..3] :  5 32 117 320
```

The above command prints an array slice, or subset. The slice consists of elements `[3][0]` through `[3][3]` of the array `table`.

```
(CXdb) print info->subject
char[7] "Other1\000"
```

The above command prints the value of a member of a C structure called `info`.

Related Commands

`break instruction`
`break routine`
`copy`
`disassemble`
`evaluate`
`event relation`
`examine`
`fill`
`find memory backward`
`find memory forward`
`goto address`
`info expression`
`info frame at`
`print`
`trace instruction`
`trace routine`
`watch`

Related Concepts

debugger variables
Fortran language expressions
language expressions
process object
scope
source units

Related Parameters

array-slice
debugger-variable
language-expression
string

Related Windows

Source Code window

Description

Cmderr is the list of viewports (destinations) that receive all error and informational messages generated in response to CXdb commands. Cmderr is similar to stderr in the shell.

A viewport for cmderr can be either a file, the CXdb Command window (in X Windows mode only), or stderr (in line mode only). By default, the viewport list for cmderr always contains the CXdb Command window (or stderr in line mode). You cannot delete the Command window (or stderr in line mode) from the viewport list.

The commands for specifying the cmderr viewports are:

- `add cmderr` — Add file names to the viewport list for cmderr.
- `remove cmderr` — Remove file names from the viewport list for cmderr.
- `set cmderr` — Delete all file names from the current viewport list for cmderr, and replace them with the specified list of file names.

CXdb creates the viewport file if it does not exist, and overwrites the file if it does exist. To prevent overwriting of an existing log file, use the `set noclobber` command.

NOTE: To append to an existing log file, use redirection operators.

To display the current noclobber setting and the current list of viewports for cmderr, use the command `info cxdb`.

Examples

The following examples illustrate how to save CXdb messages to a file.

```
(CXdb) set noclobber
```

The above command prevents overwriting of existing files.

```
(CXdb) add cmderr my_err.log
New cmderr: Window #1, my_err.log
```

The above command adds the file my_err.log to the viewport list for cmderr. CXdb creates the file in the console working directory in this case. If this file had already existed, an error would have resulted because the noclobber option was set in the previous example. Note that Window #1 (the Command window) is already on the list because it is the default viewport for cmderr.

Related Commands

`add cmderr`
`clear noclobber`
`info cxdb`
`remove cmderr`
`set cmderr`
`set noclobber`

Related Concepts

cmdlog
cmdout
logging
redirection
viewports

Related Parameters

redirection-operator
viewport

Related Windows

Command window

Description

Cmdlog is the list of viewports (destinations) that receive a log of all input entered on the CXdb command line. This includes input that you enter directly on the command line as well as input read from command files or initialization files. Cmdlog is similar to stdin in the shell.

Initially, the viewport list for cmdlog is empty. The input you enter always displays in the CXdb Command window (or stdin in line mode), regardless of the settings for cmdlog. There is no need to add the Command window (or stdin) to the viewport list for cmdlog. In fact, doing so will cause your input to appear twice on the command line.

The commands for specifying the cmdlog viewports are:

- `add cmdlog` — Add file names to the viewport list for cmdlog.
- `remove cmdlog` — Remove file names from the viewport list for cmdlog.
- `set cmdlog` — Delete all file names from the current viewport list for cmdlog, and replace them with the specified list of file names.

For cmdlog, all of the viewports you specify should be files. CXdb creates the viewport file if it does not exist, and overwrites the file if it does exist. To prevent overwriting of an existing log file, use the `set noclobber` command.

NOTE: To append to an existing log file, use redirection operators.

After defining the viewport list for cmdlog, you can enable and disable logging to those viewports periodically during the debugging session by using the following commands:

- `clear logging` — Disable logging to the cmdlog viewports.
- `set logging` — Enable logging to the cmdlog viewports.

The default for input logging is off (clear).

Examples

The following examples illustrate how to log input to a file.

```
(CXdb) set noclobber
```

The above command prevents overwriting of existing files.

```
(CXdb) add cmdlog my_input.log
New cmdlog: my_input.log
```

The above command adds the file my_input.log to the viewport list for cmdlog. CXdb creates the file in the console working directory in this case. If this file had already existed, an error would have resulted because the noclobber option was set in the previous example.

```
(CXdb) set logging
```

The above command enables logging to all viewports of cmdlog. In this case, any further input to the CXdb command line is now stored in the file my_input.log as well as being echoed on the command line.

Related Commands

`add cmdlog`	`clear echo`
`clear logging`	`clear noclobber`
`info cxdb`	`remove cmdlog`
`set cmdlog`	`set echo`
`set logging`	`set noclobber`

Related Concepts

cmderr	cmdout
logging	viewports

Related Parameters

viewport

Description

Cmdout is the list of viewports (destinations) that receive the normal output generated in response to CXdb commands. Cmdout is similar to stdout in the shell.

A viewport for cmdout can be either a file, the CXdb Command window (in X Windows mode only), or stdout (in line mode only). By default, the viewport list for cmdout always contains the CXdb Command window (or stdout in line mode). You cannot delete the Command window (or stdout in line mode) from the viewport list.

The commands for specifying the cmdout viewports are:

- `add cmdout` — Add file names to the viewport list for cmdout.
- `remove cmdout` — Remove file names from the viewport list for cmdout.
- `set cmdout` — Delete all file names from the current viewport list for cmdout, and replace them with the specified list of file names.

CXdb creates the viewport file if it does not exist, and overwrites the file if it does exist. To prevent overwriting of an existing log file, use the `set noclobber` command.

NOTE: To append to an existing log file, use redirection operators.

To display the current noclobber setting and the current viewports for cmdout, use the command `info cxdb`.

Examples

The following examples illustrate how to save CXdb output to a file.

```
(CXdb) set noclobber
```

The above command prevents overwriting of existing files.

```
(CXdb) add cmdout my_output.log
New cmdout: Window #1, my_output.log
```

The above command adds the file my_output.log to the viewport list for cmdout. CXdb creates the file in the console working directory in this case. If this file had already existed, an error would have resulted because the noclobber option was turned on in the previous example. Note that Window #1 (the Command window) is already on the list because it is the default viewport for cmdout.

Related Commands

`add cmdout`
`clear noclobber`
`info cxdb`
`remove cmdout`
`set cmdout`
`set noclobber`

Related Concepts

cmderr
cmdlog
logging
redirection
viewports

Related Parameters

redirection-operator
viewport

Related Windows

Command window

command files

Description

Command files are files that contain a series of CXdb commands. Any of the CXdb commands may be used in a command file. The commands in a command file adhere to the same syntax rules as commands entered directly on the CXdb command line.

Command files can be created with a standard editor such as `vi` or `emacs`, and they are stored in ASCII format. You can also create a command file by logging your input (cmdlog) to a viewport file and then editing that file.

There are a number of basic uses for command files:

- Defining aliases
- Defining macros
- Defining search paths
- Storing a frequently repeated sequence of commands
- Initializing CXdb

Command files can be invoked from the shell using the `cxdb` command or from within CXdb using the `source` command. There is also a special command file know as an initialization file. Initialization files execute automatically whenever CXdb is invoked.

CXdb reads and executes command files one line at a time. By using the `set echo` and `clear echo` commands, you can control whether or not the lines of the command file display in the Command window as they are executed.

If one of the lines in the command file causes an error, CXdb reports the error and then proceeds to read and execute the next line in the file. CXdb also opens any windows required by each command in the file, and it waits for any interactive input needed from the user.

To continue a command across multiple lines of the command file, use a back-slash (\) at the end of each line you want to continue.

You can add comments to a command file by using a pound sign (#) at the beginning of each comment. CXdb ignores anything between the `#` and the end of the line.

Examples

The following example illustrates a simple command file that defines aliases and macros.

Assume that you have a command file called example.aliases in your console working directory. The file contains the following lines.

```
# Example command file
alias go 'run'
alias se 'step expression'
alias sl 'step loop'
macro p(x) 'print x; @p'
```

Also assume that echoing is enabled (on).

To execute the command file, you could use the `source` command as follows:

```
(CXdb) source example.aliases
(CXdb) alias go 'run'
(CXdb) alias se 'step expression'
(CXdb) alias sl 'step loop'
(CXdb) macro p(x) 'print x; @p'
```

The above `source` command causes CXdb to read and execute the file example.aliases. Because echoing is enabled, CXdb displays each line of the command file as it is executed. Any responses or error messages would also display.

Related Commands

`add cmdlog`
`clear echo`
`clear logging`
`info cxdb`
`remove cmdlog`
`set cmdlog`
`set echo`
`set logging`
`source`

Related Concepts

cmdlog
console working directory
initialization files
logging

Related Parameters

file-name

Compiler-Tools Interface
CTI

Description

The Compiler-Tools Interface (CTI) provides a link between CXdb and the CONVEX Fortran and CONVEX C compilers. The CTI interprets information generated by the compiler and translates that information into a form that CXdb can understand.

Whenever you compile your program with the `-cxdb` option, you are creating CTI data files. The compiler produces these data files, which CXdb uses to debug the program. The CTI places these data files in a subdirectory named .CTI. The CTI creates the .CTI subdirectory underneath the same directory where the compiler places the object (.o) files for your program.

NOTE: Early versions of the CONVEX compilers stored the CTI data files in a subdirectory named .CXdb. The current version of CXdb can also work with the data files in the .CXdb subdirectory.

The CTI places the data files in the .CTI subdirectory during semantic analysis of your source file by the compiler front end. The data files have the same name as your source file, with different extensions added to indicate the type of data they contain. The data files describe the logical structure of your source code.

Using a number of smaller data files allows CXdb to process information incrementally as it is needed. This is in contrast to many other debuggers, which store all debugging data in the executable image and which must process the data all at once.

If you move any of your source files, executable files, or CTI data files after compiling, then you have to give CXdb the new locations of those files. Similarly, if your directory structure changes for any reason, you might have to give CXdb the new paths to your files. The following table shows which CXdb commands to use to specify the new file locations.

If you move:	Specify new location with:
Source files	`add path` or `set path` commands
Executable files	`add default path` or `set default path` commands
CTI data files	`add path` or `set path` commands

For source files

For each source file compiled with the `-cxdb` option, the CTI generates the following types of data files in the .CTI subdirectory:

- Version 8.0 or later of the CONVEX Fortran compiler and version 5.0 or later of the CONVEX C compiler:
  - `.ns` —Name space; provides information about external program symbols and scope blocks.
  - `.fe` —Front end; provides language-specific data and lexical scope information about all program identifiers and source units. The source units reflect the syntax of the source code.
- Prior to version 8.0 of the CONVEX Fortran compiler or version 5.0 of the CONVEX C compiler:
  - `.ns` —Name space; provides information about external program symbols and scope blocks.
  - `.tsi` —Type and scope information; provides language-specific data and lexical scope information about all program identifiers.
  - `.sut` —Source unit table; provides information about individual source units as well as the interrelationships between source units. The source units reflect the syntax of the source code.

For object files

The CTI also generates a number of data files associated with the object files for your program. These data files have the same name as the object file, with different extensions added to indicate the type of data they contain. For each object file, the CTI generates the following types of data files in the .CTI subdirectory:

- Version 8.0 or later of the CONVEX Fortran compiler and version 5.0 or later of the CONVEX C compiler:
  - `.be` —Back end; specifies storage locations and liveness ranges for program variables, synthesized variables, and source units.
  - `.xpt` —Expression table; provides information about all synthesized expressions used by the compiler.
- Prior to version 8.0 of the CONVEX Fortran compiler or version 5.0 of the CONVEX C compiler:
  - `.lrt` —Location range table; specifies the storage locations of each variable for various ranges of the program counter (PC).
  - `.srt` —Source range table; specifies which source units are active over various ranges of the program counter (PC).
  - `.vt` —Variable table; specifies the attributes of all variables, including synthesized variables. It is analogous to the symbol table used by the compiler.

- `.xpt` — Expression table; provides information about all synthesized expressions used by the compiler.

For the executable file

In addition to the above data files, the CTI adds the following information to the executable file produced by the compiler:

- Section table — Indicates which memory ranges within each section of memory (text, data, tdata, bss, and tbss) are used by each object file.
- Source file table — Lists all the source files that were compiled with the `-cxdb` option to produce the executable file.
- Time stamp — Provides a quick check of the CTI data files to ensure that they were all generated during the same compilation.
- Nlist — Lists the names of all the external symbols used by the loader. Although stabs are implemented within the Nlist, CXdb ignores them. However, you can compile your program with both the `-g` and `-cxdb` options specified.

The CTI compresses all of the data files it generates. As a rough estimate, the total storage space used for the CTI data is about two to four times the size of the associated executable image. Also, it takes about one-third longer to build the image. The build time and storage space will vary with different source languages, programming methods, and compiler options.

Because CXdb uses separate data files to hold most of its information, you can store these files separate from the executable image. They do not have to be included with the production version of your program.

Caution

If you use the shell commands `strip` or `ld -s` to reduce the size of the executable file, the section table, source file table, time stamp, and Nlist will all be deleted. Without this information, CXdb cannot do symbolic debugging of your program. Deletion of any of the CTI data files in the .CTI subdirectories also inhibits symbolic debugging with CXdb.

Related Commands

`add path`
`info path`
`remove path`
`set path`

Related Concepts

compiling for CXdb
optimized code
process object
search path

compiling for CXdb

Description

CXdb can debug both C and Fortran programs. To enable full symbolic debugging with CXdb, compile your source code with the `-cxdb` option. For example:

```
% fc -cxdb myprog.f
```

When you compile with the `-cxdb` option, the compiler produces a set of data files that contain all the symbolic debugging information for your program. These data files are accessed and managed by software known as the Compiler-Tools Interface (CTI). Therefore, the files are called CTI data files.

The compiler creates a subdirectory called .CTI and places the CTI data files there. The .CTI subdirectory is created in same directory where the compiler places the object (.o) files for your program. If you use the `-cxdb` option to compile several different programs in the same directory, the .CTI subdirectory will contain the CTI data files for each of those programs.

NOTE: Early versions of the CONVEX compilers stored the CTI data files in a subdirectory named .CXdb. The current version of CXdb can also work with the data files in the .CXdb subdirectory.

When you debug a program with CXdb, the CTI accesses the appropriate CTI data files to provide CXdb with the symbolic information for your program.

If you move any of your source files, executable files, or CTI data files after compiling, then you have to give CXdb the new locations of those files. Similarly, if your directory structure changes for any reason, you might have to give CXdb the new paths to your files. The following table shows which CXdb commands to use to specify the new file locations.

If you move:	Specify new location with:
Source files	`add path` or `set path` commands
Executable files	`add default path` or `set default path` commands
CTI data files	`add path` or `set path` commands

If you have compiled your program *without* the `-cxdb` option, you can still debug it with CXdb. However, in this case you must access program variables and memory locations by their addresses rather than by their symbolic names.

Related Commands

`add default path`
`add path`
`info path`
`set default path`
`set path`

Related Concepts

Compiler-Tools Interface
getting started with CXdb

console working directory

Description

The console working directory is the base directory for all relative path names used in commands that affect CXdb. Relative path names can be directory names or file names.

The console working directory is initially set to the directory from which you invoke CXdb, and the process working directory is initially set to the console working directory.

The commands that affect the console working directory are:

- `cd` — Changes the setting of the console working directory.
- `pwd` — Displays the current setting of the console working directory.

The commands that use the console working directory by default are:

```
add cmderr
add cmdlog
add cmdout
add default path
add path
cd
core
debug core
debug exec
executable
remove cmderr
remove cmdlog
remove cmdout
remove default path
remove path
set cmderr
set cmdlog
set cmdout
set default path
set path
source
```

Examples

The following examples use the console working directory, which is initially set to /mnt/jones in this case.

```
(CXdb) source example3/example_aliases
```

The above command executes the CXdb commands found in the file /mnt/jones/example3/example_aliases. Because the path name to the file is relative, the console working directory is used as the base path.

```
(CXdb) cd example3
```

The above command sets the console working directory to the /mnt/jones/example3 directory. Relative path names now use /mnt/jones/example3 as the base path name.

```
(CXdb) pwd
/mnt/jones/example3
```

The above command displays the current setting of the console working directory.

Related Commands

cd
pwd

Related Concepts

command files
default search path
logging
process object
process working directory
search path

Related Parameters

directory-specifier
file-name

csd debugger

Description

The CONVEX `csd` debugger is a symbolic debugger. It has been replaced by CXdb, but the most frequently used `csd` commands are available in CXdb through aliases. If you are already familiar with `csd`, these aliases can make it easier for you to learn how to use CXdb.

There are two ways to access the `csd` aliases in CXdb:

- From the UNIX command line — By entering the command `cxdb -csd`. Note that using the `-csd` option invokes CXdb in line mode (same as the `-nw` option).
- From the CXdb command line window — By entering the command `csd`.

To incorporates the predefined `csd` aliases automatically each time you invoke CXdb, include the command `source /usr/lib/cxdb/aliases/csd_aliases` in your .cxdbinit file.

With the predefined aliases incorporated, you can type in a `csd` command while using CXdb. If the command has an alias, the alias is substituted, and the equivalent CXdb command is executed. If the command does not have a one-to-one correspondence with a CXdb command, CXdb displays a message indicating that the `csd` command is not aliased and, where possible, suggests a CXdb command with the closest functionality to the `csd` command.

The `csd` commands supported by CXdb aliases are:

csd command	CXdb equivalent
`&`	Use `print loc(`*x*`)` or `print &`*x*.
`?`	`find window backward`
`alias`	Use `alias` command.
`assign`	`evaluate`
`call`	`print`
`catch`	Use `set signal` command.
`cregs`	`info cregisters` (C Series only)
`delete`	`remove event`
`down`	Use `frame` command.
`dump`	`backtrace`
`edit`	`edit`
`file`	Use `list` command.

csd command	CXdb equivalent
`format`	No equivalent.

`format decimal`	`set format byte dec;`
	`set format half dec;`
	`set format word dec;`
	`set format long dec;`
	`set format quad dec`

`format hex`	`set format byte hex;`
	`set format half hex;`
	`set format word hex;`
	`set format long hex;`
	`set format quad hex`

`fpmode ieee`	`set format ieee`
`fpmode native`	`set fpmode native` (C Series only)
`fpmode auto`	`set fpmode dual` (C Series only)
`func`	`info scope`; Use `backtrace` or `display routine` commands for more information.
`help`	`help`
`ignore`	Use `set signal` command.
`list`	Use `list` command.
`mode`	No equivalent.
`mode chained`	`clear seq` (C Series only)
`mode sequential`	`set seq` (C Series only)
`next all`	`next`
`nexti`	`next instruction`
`print`	Use `print` command.
`quit`	`quit`
`rerun`	Use `rerun` command.
`return`	Use `return` command.
`run`	Use `run` command.
`regs`	`info registers`
`set num_elements =`	`set printopts maxarray`
`set precision =`	`set printopts precision 10`
`set deref_aaregs`	Change format in register window.
`set dump_lfmt`	Use `info` commands.
`set dumpvregs`	Use `info vregisters` (C Series only).
`status`	`info event *`
`step`	`step`
`step all`	`step`
`stepi`	`step instruction`
`stop`	No equivalent.
`stop at`	`break line`
`stop in`	`break routine`
`stop if`	`event relation`
`stop threads`	`event spawn; event join`
`stopi at`	`break instruction`

csd command	CXdb equivalent
thread	info process
threads false	remove eventtype spawn, join
threads true	event spawn; event join
trace threads	event spawn {backtrace 1; echo 'thread spawned'; resume;}; event join {backtrace 1; echo 'thread joined'; resume;};
unalias	remove alias
up	Use up alias.
use	set path
vregs	info vregisters (C Series only)
whatis	info expression
when at	event reached line
when in	event reached routine
where	backtrace
whereis	info symbols
which	info symbols

Related Commands

csd
cxdb
gdb
info alias
source

Related Concepts

gdb debugger

debugger variables

Description

You can define debugger variables for use in CXdb commands to take the place of constants or literal values.

The first character of the debugger variable name must be alphabetic. The rest of the name can consist of any number of alphanumeric characters, but it cannot include special characters or white space. Except where otherwise indicated, debugger variable names are case sensitive.

To distinguish a debugger variable from other types of symbols such as program variables, precede the debugger variable name with a dollar sign ($).

Debugger variables are especially useful in command files, initialization files, eventpoint handlers, and macros. A debugger variable can store any of the following types of data:

- A CXdb object number. An object number is a unique identifier that CXdb assigns to a process, an eventpoint, or a window.
- The result of a language expression. The assignment is static, so the debugger variable is not updated if the value of the language expression changes.
- The contents of a register.
- A signal number.

The data type of a debugger variable is the same as the data type of the value assigned to it. Once a value has been assigned to a debugger variable, you can use the variable anywhere a value of that type would be valid. For example, if you assign an eventpoint number to the debugger variable `$E`, you can then use `$E` with any CXdb command that accepts an eventpoint number as an argument.

The data type of a debugger variable is not fixed. If you assign a new value to an existing debugger variable, that variable assumes the data type of the new value.

Commands that use debugger variables

The following commands can assign the values of language expressions, registers, and signal numbers to debugger variables:

```
evaluate
print
```

The following commands create CXdb objects and can assign the object numbers to debugger variables:

```
break instruction
break line
break routine
break source
debug core
debug exec
debug proc
event exec
event join
event modify
event reached instruction
event reached line
event reached routine
event reached source
event relation
event signal
event spawn
trace instruction
trace line
trace routine
trace source
watch
```

Predefined debugger variables

In addition to debugger variables that you define, there are a number of special, predefined debugger variables that enable you to access certain system information such as register values. These debugger variables are dynamic: when your process is running, the debugger variables update automatically to the current values.

For example, there are two predefined debugger variables to represent eventpoints and signals:

- `$self` (or `$SELF`) — The eventpoint number of the last triggered eventpoint.
- `$signal` (or `$SIGNAL`) — The signal number of the last signal sent to the current process.

There are also numerous predefined debugger variables to represent registers. The architecture of your system determines which registers you can access with these predefined debugger variables. Read the "registers" reference page for a list of registers available on your CONVEX system.

Examples

The following examples illustrate how to create and reference debugger variables.

```
(CXdb)  break routine BLD_MATRIX \; $D

#1: break routine, on [#0/*], Enabled, ignore 0/0
        [0x800029bc] BLD_MATRIX in chapter7F.f line 26
```

The above command sets a breakpoint at the routine called `BLD_MATRIX`. The object number for this breakpoint is 1, and this object number is stored in the debugger variable `$D`. The language expression terminator (`\;`) separates the routine name from the debugger variable name.

Once a debugger variable has been created, you can reference it in a subsequent command, as follows:

```
(CXdb) set ignore 3 $D
Event 1 will be ignored 3 times
```

The above command sets an ignore count of 3 for eventpoint 1. The eventpoint is represented in this command by the debugger variable `$D`.

```
(CXdb) evaluate $A=N+M
```

The above command evaluates the language expression `N+M` and assigns the result to the debugger variable `$A`.

```
(CXdb) evaluate $s0=1234
```

The above command stores the value `1234` in scalar register S0 on a C2 Series machine. You can modify the contents of other predefined debugger variables, except `$self`, in a similar way.

Related Commands

`evaluate`
`print`

Related Concepts

architecture dependencies
command files
initialization files
eventpoint handlers
registers

Related Parameters

debugger-variable

default environment

Description

The default environment is the set of environment variables for CXdb.

The default environment is passed to a new process if the process object does not have its own environment. Initially, the default environment is a copy of the environment passed to CXdb when it is invoked. Modifications to the default environment do not affect the environment of an existing process.

You can modify the default environment with the following commands:

- `add default environment` — Adds environment variables to the default environment.
- `clear default environment` — Clears the default environment of all environment variables.
- `info default environment` — Displays the environment variables of the default environment.
- `remove default environment` — Removes environment variables from the default environment.
- `set default environment` — Sets the default environment to be the specified environment variables.

Examples

The following examples illustrate how to use the default environment commands.

```
(CXdb) info default environment
Default environment:
PATH=/usr/bin:/usr/local/bin
SHELL=/bin/csh
TERM=xterm
```

The above command displays the default environment. These are the environment variables passed to CXdb when CXdb was invoked.

If none of the default environment variables are needed, you can clear the default environment.

```
(CXdb) clear default environment
```

The above command clears the default environment of all environment variables.

```
(CXdb) add default environment EDITOR = vi , PAGER = less , LESS = -MQce
```

The above command adds the environment variables `EDITOR`, `PAGER`, and `LESS` to the default environment. Because the variables do not exist in the default environment, they are created.

If a new process is created, and its process object does not have its own environment, it is passed the environment variables of the default environment—in this example, `EDITOR`, `PAGER`, and `LESS`.

If an environment variable is not needed, you can remove it from the default environment.

```
(CXdb) remove default environment EDITOR
```

The above example removes the environment variable `EDITOR` from the default environment. The rest of the variables in the default environment remain unchanged.

```
(CXdb) set default environment INITVAL = "10 20" , ENDVAL = "30 40"
```

The above example clears the default environment first, then adds the variables `INITVAL` and `ENDVAL`. In this example each string must be delimited by quotes because it contains a white space character (a blank).

Related Commands

`add default environment`	`add environment`
`clear default environment`	`clear environment`
`info default environment`	`info environment`
`remove default environment`	`remove environment`
`set default environment`	`set environment`

Related Concepts

environment — process object

Related Parameters

environment-variable — string

default search path

Description

The search path of a process is used to find program source files as well as CXdb compiler-generated data files (CTI files). The default search path is used as the basis for the search path for each new process created by CXdb.

Initially, the default search path is set to the console working directory. Each new process receives a copy of the default search path as the beginning of its search path. Modifications to the default search path only affect new processes. The search path of an existing process is not affected.

The following commands allow you to manipulate the default search path:

- `add default path` — Adds directories to the end of the default search path.
- `info path` — Displays the default search path.
- `remove default path` — Removes directories from the default search path.
- `set default path` — Sets the default search path to the specified directories.

Examples

The following examples illustrate how to use the default search path commands.

```
(CXdb) info path

Default search list:
        .

Process [#0] search list for: docexample
```

The above example displays the default search path and search path for the process. The default search path is the current working directory, represented by a dot (.). The search path initially inherits the default search path, so it to is set to the current working directory.

```
(CXdb) add default path /usr/smith/programs, /doc/cxdb/examples
```

The above example adds two directories to the default search path. The next process that is created will receive the new default search path, which now includes the /usr/smith/programs and /doc/cxdb/examples directories.

```
(CXdb) remove default path /usr/smith/programs
```

The above command removes the /usr/smith/programs directory from the default search path. The other directories in the default search path remain.

```
(CXdb) info path

Default search list:
        .
        /doc/cxdb/examples


Process [#0] search list for: docexample
        .
```

The above example uses the info path command to display the updated default search path.

```
(CXdb) set default path /usr/smith/data, /doc/cxdb/examples
(CXdb) info path

Default search list:
        /usr/smith/data
        /doc/cxdb/examples


Process [#0] search list for: docexample
        .
```

The above command removes all the existing directories from the default search path and sets the default search path to the two listed directories. Now each new process will include the /usr/smith/data and `/doc/cxdb/examples` directories, as well as the process working directory as its search path. The `info path` command displays the new default search path.

You can use the above commands in an initialization file in order to set up a default search path that can be used by all of your processes.

Related Commands

`add default path`
`add path`
`info path`
`remove default path`
`remove path`
`set default path`
`set directory`
`set path`

Related Concepts

console working directory
initialization files
process object
process working directory
search path

Related Parameters

directory-specifier

Description

One of the most frequent debugging activities is the displaying of data. With CXdb, you can display a wide variety of data, including:

- Information about variables, such as data type, size, storage location, and so on
- The contents of program variables, registers, and debugger variables
- The result of evaluating an expression, function, or subroutine
- The contents of memory

The following sections explain how to display these various types of data with CXdb commands.

Displaying information about variables

Rather than just printing the value of a variable, you might want to display additional information, such as the data type, storage location, and size of a variable. CXdb provides the following commands for displaying this information:

- `info args` — Displays the arguments passed to the current routine.
- `info expression` — Displays all the characteristics of a language expression or variable.
- `info locals` — Displays the local variables of the current routine.
- `info symbols` — Displays all symbol names used in the program.
- `info type` — Displays information about type definitions in C programs.

For example, you can display the arguments of the current routine as follows:

```
(CXdb) info args
Process [#0/0]
Frame : 0; [0x8000293c] CHAPTER7 in chapter7F.f line 10
Number of arguments : 1
     1 : ARRAY= INTEGER*4(1:4, 1:4) 0x8008ace8
```

The above example shows that the only argument passed to the current routine is the 4x4 array named `ARRAY`. The starting address of `ARRAY` is `8008ace8`.

You can display all the local variables of the current routine as follows:

```
(CXdb) info locals
Process [#0/0]
Frame : 0; [0x8000293c] CHAPTER7 in chapter7F.f line 10
Number of locals : 3
     1 :  TABLE = INTEGER*4(1:4, 1:4) 0x80077058
     2 :  I = (INTEGER*4) 5
     3 :  J = (INTEGER*4) 5
```

The above example shows that the local variables in the current routine are the integer variables `I` and `J` (both with a current value of 5) and the 4x4 array named `TABLE`.

To display the most detailed information about a variable, use the `info expression` command, as illustrated below:

```
(CXdb) info expression J
object type: Fortran identifier
   location: 0x8007709c
       size: 4 bytes
       type: INTEGER*4
      value: 5
       8 liveness ranges:
              Start      End          Location
         1.  0x800028b8:0x800028be - register s0
         2.  0x800028da:0x800028e2 - register a1
         3.  0x800028e4:0x800028e8 - register a4
         4.  0x800028e2:0x800028fa - register a1
         5.  0x8000290a:0x8000290e - register s0
         6.  0x8000290e:0x80002914 - register s0
         7.  0x8000291a:0x80002920 - register s0
         8.  0x80066000:0x80077000 - 0x8007709c
```

The above example shows that the Fortran variable `J` is a 4-byte integer stored in memory at location `8007709c`, and its current value is 5. The liveness ranges for `J` are also listed. A *liveness range* is a range of memory where the variable is active (alive).

As long as the program counter (PC) is within the bounds of a liveness range, CXdb can retrieve the value of the variable from the indicated storage location. In the above example, if the PC is within the range `80066000` through `80077000`, then CXdb retrieves the value of `J` from memory location `8007709c`. If the PC is outside the bounds of all liveness ranges, then the value of the variable is not available for display.

You can also use the `info expression` command to display information about language expressions, functions, routines, and debugger variables. For example:

```
(CXdb) info expression ISQR
object type: Fortran function
entry point: 0x80002986
return type: INTEGER*4
return size: 4 bytes
  arg count: 0
  prototype:  INTEGER*4 ISQR( INTEGER*4 )

(CXdb) info expression $A0
object type: debugger variable
   writable: yes
   var type: predefined register access
   register: a0 as 32 bits (equivalent to sp)
      value: (INTEGER*4) 0xffffc978
```

The `info expression` commands in the above example display information about the program function `ISQR` and the debugger variable `$A0`.

Printing data

With the `print` command, you can display the values of any of the following data items:

- Program variables
- Language expressions
- Program functions and routines
- Registers
- Debugger variables

For example:

```
(CXdb) print J
(INTEGER*4) 5

(CXdb) print $A0
(INTEGER*4) 0xffffc978

(CXdb) print ISQR(3)
(INTEGER*4) 9
```

In the above example, the `print J` command prints the value of the program variable `J`. The `print $A0` command uses the debugger variable `$A0` to access and print the contents of address register A0. The `print ISQR(3)` command actually executes the program function `ISQR` and prints the value returned by that function.

You can also specify formatting options with the print command, as illustrated below:

```
(CXdb) print/B J
(INTEGER*4) 0000 0000 0000 0000 0000 0000 0000 0101
```

The above example uses the `/B` formatting option to print the value of `J` in binary format.

In addition, you can use the `set printopts` command to set the following print options:

- `maxarray` — The maximum number of array elements printed at one time. The default is 20 elements.
- `padding/nopadding` — Leading zeros to force alignment of integer values. The default is no padding.
- `precision` — The precision used for printing real (floating point) numbers. The default precision is 10.4.

Use the `info formatting` command to display the current settings of the above options.

Examining and searching memory

The `examine` command displays the contents of memory, as illustrated below:

```
(CXdb) examine loc(MATRIX)
Examine Process [#0/0] from 0x800770ac to 0x800770f8
 800770ac:    c1f00000  c141999a  3feb8520  415ccccе
 800770bc:    00000000  c101999a  41ceb852  4283999a
 800770cc:    42e141d4  00000000  418eb852  42ab999a
 800770dc:    4348a0ea  00000000  00000000  42473334
 800770ec:    4330a0ea  43fb6000  00000000  00000000
```

The above example examines the contents of the array named `MATRIX`. The Fortran function `loc()` is used to determine the starting address of the array.

You can also specify the type of memory unit, the display format, and the number of units to display with the `examine` command, as shown below:

```
(CXdb) examine/wf loc(MATRIX):10
Examine Process [#0/0] from 0x800770ac to 0x800770d0
 800770ac:       -7.5000     -3.0250      0.4600      3.4500
 800770bc:  0.0000E+000     -2.0250      6.4600     16.4500
 800770cc:       28.1571  0.0000E+000
```

The above example displays the first 10 (`:10`) words (`/w`) of the array `MATRIX` in floating point (`f`) format.

If you want to search memory for a particular byte pattern, use the `find memory forward` or `find memory backward` commands, as illustrated below:

```
(CXdb) find memory forward c141 loc(MATRIX)
Data found at 0x800770b0
```

The above command searches memory for the byte pattern `c141`, beginning the search at the starting address of `MATRIX`.

If you are using the X Windows interface to CXdb, you can open a Memory Display window to display and search program memory. Refer to the "Memory Display window" topic for more information.

Working with arrays

You can use the `print` command to display an entire array or just a portion of the array (called an array slice). For example:

```
(CXdb) print TABLE
INTEGER*4(1:4, 1:4)
(1..4,1) :  2 4 6 8
(1..4,2) :  5 12 21 32
(1..4,3) :  10 26 54 100
(1..4,4) :  17 48 129 320

(CXdb) print TABLE(1..4,3)
INTEGER*4(1:4, 3:3)
(1..4,3) :  10 26 54 100
```

In the above example, the `print TABLE` command prints all elements of the Fortran array named `TABLE`. The `print TABLE(1..4,3)` command prints only elements (1,3) through (4,3) of the array `TABLE`.

By default, the maximum number of array elements you can print at one time is 20. To change this default, use the `set printopts maxarray` command.

NOTE: An array slice is a temporary copy of the original array elements. CXdb does not save the array slice, so it exists only for the duration of the command that uses it. For more information about array slices, read the "array-slice" reference page.

Using scope paths

Scope paths enable you to access variables that are defined outside the scope of the current routine (or stack frame). For more information about scope paths, read the "scope" reference page.

Related Commands

`examine`
`find memory backward`
`find memory forward`
`info args`
`info expression`
`info locals`
`info symbols`
`info type`
`print`
`set printopts maxarray`
`set printopts nopadding`
`set printopts padding`
`set printopts precision`

Related Concepts

debugger variables
modifying data
registers
synthesized variables
language expressions
optimized code
scope

Related Parameters

array-slice

Description

The environment is the set of variables that define various characteristics for a process. A process does not initially have its own environment. You create an environment for a process the first time you modify one of its environment variables. Any environment variables that you do not modify are copied from the default environment.

The process environment is passed to each new process. If a process does not have its own environment, the default environment of CXdb is passed to each new process.

You can modify the environment of a process with the following commands. (All of these commands except `info environment` create an environment for a process.)

- `add environment` — Adds environment variables.
- `clear environment` — Removes all environment variables.
- `info environment` — Displays the environment variables.
- `remove environment` — Removes environment variables.
- `set environment` — Clears the environment and then adds the specified environment variables.

Examples

The following examples illustrate how to use the environment commands.

```
(CXdb) info environment
Process [#0] environment: (from default environment)
PATH=/usr/bin:/usr/local/bin
SHELL=/bin/csh
TERM=xterm
```

The above command displays the environment variables that will be passed to a new process. Because the current process does not yet have its own environment, the variables displayed are those of the default environment.

```
(CXdb) clear environment
```

The above command creates an environment for the current process. This command also clears the newly created environment of all environment variables. Now that you have created and cleared the environment, you can begin to add the environment variables you need.

```
(CXdb) add environment EDITOR = vi , PAGER = less , LESS = -MQce
```

The above command adds the environment variables `EDITOR`, `PAGER`, and `LESS` to the environment. Because the variables did not exist in the environment, they were created.

A new process will be passed these two environment variables rather than those of the default environment.

```
(CXdb) remove environment EDITOR
```

The above example removes the environment variable `EDITOR` from the environment. The rest of the variables in the environment remain unchanged.

```
(CXdb) set environment INITVAL = "10 20" , ENDVAL = "30 40"
```

The above example clears the environment first, then adds the variables `INITVAL` and `ENDVAL`. In this example, each string must be enclosed in quotes because it contains a white space character (a blank).

Related Commands

`add default environment`	`add environment`
`clear default environment`	`clear environment`
`info default environment`	`info environment`
`remove default environment`	`remove environment`
`set default environment`	`set environment`

Related Concepts

default environment

process object

Related Parameters

environment-variable

string

Description

An eventpoint handler is a collection of CXdb commands to perform when the corresponding event is triggered. The commands are enclosed in curly-braces ({}). Each command inside an eventpoint handler must end with a semi-colon (;).

None of the normal process execution commands are allowed in an eventpoint handler. The following is a list of commands that are *not* allowed:

```
attach
continue
finish
next
next over
next instruction
return
rerun
run
signal process
signal thread
step
step over
step instruction
stop
```

To continue process execution from within a handler, the `resume` command is used. The `resume` command continues the execution of the process by the command that last began execution. Thus, if the eventpoint is triggered after 50 steps of a `step 200` command, when process execution resumes, the remaining 150 steps are taken.

Inside an eventpoint handler, you can use expressions that include function calls. All eventpoints are disabled during a function call from within a handler. After the function call is finished, the eventpoints are enabled once again.

Because the `echo` command does not allocate storage in process memory for the string it echoes, it is the preferred command to use in an eventpoint handler for displaying informative messages.

eventpoint handlers

An eventpoint handler can be given to an eventpoint when it is created, or after, using the `set handler` command. The handler can be removed from the eventpoint using the `clear handler` command.

A handler can be given to a type of eventpoint using the `set typehandler` command. All eventpoints of the type that do not have their own handler use the handler for the type. The handler for a type of eventpoint can be removed using the `clear typehandler` command.

The default handler for eventpoints displays a message indicating the status of the process and what eventpoint was triggered. You can change the default handler using the `set default handler` command. This handler can be removed using the `clear default handler` command.

Two predefined debugger variables are especially useful in eventpoint handlers. They are:

- `$self` — The eventpoint number of the currently executing eventpoint. Using this debugger variable, an eventpoint can display its eventpoint number as a means of identification.
- `$signal` — The signal number of the last signal caught. Using this debugger variable, an eventpoint set to catch signals can display the number of the signal it caught.

Examples

The following examples create eventpoint handlers with the `break line` command.

```
(CXdb) break line 7 {print NUMARGS;}
```

When the breakpoint in the above command is triggered, execution stops, and the value of the variable `NUMARGS` is printed. Execution does not resume.

```
(CXdb) break line 9 {print I; resume;}
```

When the breakpoint in the above command is triggered, execution stops, and the value of the variable `I` is printed. Execution then resumes. Thus, if a `step loop 5` command is running when the breakpoint is triggered, execution resumes allowing the `step loop 5` command to finish.

```
(CXdb) break line 11 {if (NUMARGS==0) {echo "No args";} else {resume;} }
```

When the breakpoint in the above command is triggered, execution stops, and a test is made on the value of the variable `NUMARGS`. If the condition evaluates to TRUE, then the `echo` command is executed, and the eventpoint handler finishes. If the condition evaluates to FALSE, the `echo` command is skipped, and process execution resumes.

The following example creates an eventpoint handler using the `event signal` command.

```
(CXdb) event signal sigFpe {disable event $self ;}
```

The above example uses the predefined debugger variable `$self` to disable this eventpoint after it has been triggered.

```
(CXdb) event signal sigFpe {print FPEoccurred(); echo "Finished." ;}
```

When the eventpoint in the above command is triggered, process execution stops, and the eventpoint handler uses the `print` command to call the `FPEoccurred` routine. CXdb saves the state of the stack, disables all eventpoints, and then begins execution of this routine. When the routine finishes, the eventpoints are enabled, the stack is restored, and control returns to the handler. The handler prints a message and process execution stops. It is important to realize that, although the process stack has been restored, it is still possible that the call to the routine has affected your process due to side effects of the routine's operation. Because of this, be careful when calling routines from inside an eventpoint handler.

```
(CXdb) event signal sigFpe {echo /n "Signal" ; print $signal ; eval
      $signal=0; resume; }
```

When the eventpoint in the above command is triggered, process execution stops, and the handler uses the debugger variable `$signal` to display the signal number of the signal caught. The variable is then set to zero, and process execution is resumed. When the process resumes execution, the signal is sent to the process but is ignored because its value is zero.

The effects of this eventpoint handler can be achieved more simply with the `set signal` command.

Related Commands

`break instruction`
`break line`
`break routine`
`break source`
`event exec`
`event modify`
`event reached instruction`
`event reached line`
`event reached routine`
`event reached source`
`event relation`
`event signal`
`if`
`info event`
`print`
`resume`
`set default handler`
`set handler`
`trace instruction`
`trace line`
`trace routine`
`trace source`
`watch`

Related Concepts

breakpoints
debugger variables
eventpoints
tracepoints
watchpoints

Related Parameters

debugger-variable
event-handler
language-expression
line-specifier

Description

An eventpoint is a trap you create to wait for an event to occur. When the event occurs, the set of actions associated with the eventpoint, called the eventpoint handler, are taken. Eventpoints are the underlying mechanisms used to implement breakpoints, tracepoints, and watchpoints.

You can trap the following types of events:

- Instruction is reached
- Line is reached
- First source unit of a routine is reached
- Source unit is reached
- Address range is modified
- Signal is caught
- Relational expression evaluates to TRUE
- Process exec's
- Thread is spawned
- A thread is joined with another thread

Each eventpoint created is given a unique object number. This object number is used in subsequent commands when you want to refer to this eventpoint. Optionally, you may specify a debugger variable to be assigned to the eventpoint. You may then use the debugger variable to refer to the eventpoint in subsequent commands.

All eventpoints have a default handler that prints a message telling you that the eventpoint has been reached. You may specify a different set of actions to take for a particular eventpoint or all eventpoints of a particular type. You can also change the setting of the default handler itself.

Eventpoints can be enabled or disabled. If an eventpoint is enabled, and its event occurs, it is said to be reached. A disabled eventpoint is treated as if it does not exist, and therefore can never be reached until it is enabled again. The disabling of eventpoints allows you to prevent eventpoints being reached without having to completely remove them from the process object.

Once a eventpoint is reached, one of two things may occur. If the eventpoint has an ignore count, the counter is incremented by one and process execution continues. If the eventpoint does not have an ignore count, then the eventpoint is said to be triggered. When an eventpoint is triggered, the commands in its eventpoint handler are executed. If the eventpoint does not have its own eventpoint handler, nor does its type have a handler, the default handler for eventpoints is used.

Multiple eventpoints can exist at the same address. When process execution reaches an address with multiple eventpoints, the highest-numbered, enabled eventpoint is reached. If this eventpoint has an ignore count, the counter is updated and the next highest-numbered, enabled eventpoint is reached. This process continues until either an eventpoint is triggered or there are no more eventpoints at the address.

The ignore count of an eventpoint is the number of times this eventpoint must be reached before being triggered. A counter keeps track of the number of times an eventpoint has been reached. When the counter matches the ignore count, the ignore count is reset to zero, and the *next* time the eventpoint is reached the eventpoint will be triggered.

Asynchronous eventpoints do not exist at a particular address of your program. They are reached when the event they are waiting for occurs, which can happen at any time during process execution. If an asynchronous eventpoint and an address-based eventpoint are both reached at the same time, the address-based eventpoint is triggered. If two asynchronous eventpoints are reached at the same time, the lowest-numbered eventpoint is triggered.

Eventpoints are specific to the existing process object. They can be set for specific threads of a process as well. Asynchronous eventpoints are specific to the process image, thus, they only exist for the current process. Eventpoints may also be removed.

There are many different ways to set an eventpoint. Eventpoint commands can be categorized by eventpoint types.

The following commands set *reached* eventpoints. These eventpoints are the same as breakpoints. For more information about reached eventpoints, refer to the reference page on breakpoints.

- `event reached instruction` — The eventpoint is placed at the specified address.
- `event reached line` — The eventpoint is placed at the starting address that maps to the specified line number of a source file.
- `event reached routine` — The eventpoint is placed at the first executable source unit of the routine containing the specified address.

- `event reached source` — The eventpoint is placed at the starting address of the specified source unit number of a source file.

The following command sets a *signal* eventpoint.

- `event signal` — The eventpoint waits for the specified signal to be sent to the process.

The following command sets a *modify* eventpoint.

- `event modify` — The eventpoint waits for the specified address range (such as that for a program variable) to change.

The following command sets a *relational* eventpoint.

- `event relation` — The eventpoint waits for the specified relational expression to evaluate to true.

The following command sets an *exec* eventpoint.

- `event exec` — The eventpoint waits for the process to exec.

The following command sets a *join* eventpoint.

- `event join` — The eventpoint waits for a thread to join with another thread.

The following command sets a *spawn* eventpoint.

- `event spawn` — The eventpoint waits for a thread to spawn.

If the command accepts an address, any valid language expression can be used to specify the address.

Commands that allow you to interact with existing eventpoints are described below:

- `clear default handler` — Reset the default handler.
- `clear handler` — Remove the handler for a specified eventpoint.
- `clear typehandler` — Remove the handler for a specified type.
- `disable event` — Disable the specified eventpoints.
- `disable eventtype` — Disable all eventpoints of the specified type.
- `enable event` — Enable the specified eventpoints.
- `enable eventtype` — Enable all eventpoints of the specified type.
- `info event` — Display information about all existing eventpoints.
- `info eventtype` — Display information about the specified eventpoints.
- `remove event` — Remove the specified eventpoints.

- `remove eventtype` — Remove all eventpoints of the specified type.
- `set ignore` — Set an ignore count for the specified eventpoints.
- `set default handler` — Set the default handler for all eventpoints.
- `set handler` — Set a handler for the specified eventpoints.
- `set typehandler` — Set the default handler for all eventpoints of a specific type.

Examples

For examples of how to set, manipulate, and remove eventpoints, refer to the individual reference pages for the above commands in the *CONVEX CXdb Reference* or in the online help topics.

For an overview of how the eventpoint commands function together, refer to the concepts pages on breakpoints, tracepoints, or watchpoints.

Related Commands

`break instruction`
`break line`
`break routine`
`break source`
`clear default handler`
`clear handler`
`clear typehandler`
`disable event`
`disable eventtype`
`enable event`
`enable eventtype`
`event exec`
`event modify`
`event reached instruction`
`event reached line`
`event reached routine`
`event reached source`
`event relation`
`event signal`
`event spawn`
`info break`
`info event`
`info eventtype`
`info trace`
`info watch`
`remove event`
`remove eventtype`
`resume`
`set default handler`
`set ignore`
`set handler`
`set typehandler`
`trace instruction`
`trace line`
`trace routine`
`trace source`
`watch`

Related Concepts

eventpoints
eventpoint handlers
language expressions
tracepoints
watchpoints

Related Parameters

debugger-variable
event-handler
language-expression
line-specifier
process-list
thread-list

Fortran language expressions

Description

A Fortran language expression is an expression that follows the syntax rules of Fortran. You can use Fortran language expressions in CXdb commands whenever the current source language of your process is Fortran. The current source language is the language of the source file associated with the current stack frame.

In CXdb, the Fortran language expressions must conform to the rules listed below:

1. Identifiers:

- Restrictions:
  - Identifiers are not case sensitive.
  - If the identifier name contains a special character (other than alphabetic, underscore, or digits), the name must be preceded by a backslash (\).
- Program variables can be either:
  - Unqualified
  - Qualified by a scope path prefix
- CXdb debugger variables:
  - Debugger variables usually begin with the CXdb scope prefix `cxdb$` or `$.`
  - Debugger variable names are case sensitive.
  - An assignment to a debugger variable either modifies an existing variable or creates a new instance of it. The variable takes on the value, data type, and precision of the right-hand side of the assignment.
  - Language semantics may prohibit certain types of assignments, such as the assignment of an array to a debugger variable.
- Hardware registers:
  - Register names must be qualified with the CXdb scope prefix `cxdb$` or `$.`
  - An assignment to a register modifies its value only; its type and precision remain the same.

2. Constants:

- Restrictions:
  - White space is significant.
  - The default precision for integers is obtained from the setting established by the `set evalopts iprecision` command. The initial setting is 4 bytes (32 bits).
  - The default precision for real numbers is obtained from the setting established by the `set evalopts rprecision` command. The initial setting is 8 bytes (64 bits).
  - The type and precision are determined from the syntactic specification, the default precision, or the resulting value of the constant, in that order.
- Integers:
  - A precision of 4 means 32-bit integers.
  - A precision of 8 means 64-bit integers.
- Real numbers:
  - Basic real numbers can be assigned a precision of either 4 (32 bits) or 8 (64 bits).
  - Real number followed by an exponent:

    `E` suffixed exponent is single precision (32 bits).
    `D` suffixed exponent is double precision (64 bits).
    `Q` suffixed exponent is quad precision (128 bits).
  - Integer followed by an exponent:

    `E` suffixed exponent is single precision (32 bits).
    `D` suffixed exponent is double precision (64 bits).
    `Q` suffixed exponent is quad precision (128 bits).
- Complex numbers:
  - A single-precision complex constant is formed by specifying two single-precision real/integer constants.
  - A double-precision complex constant is formed by specifying either two double-precision real constants or one double precision real constant and one single precision real/integer constant.
  - In double precision, each part of the complex number is 64 bits.
- Strings can be either:
  - Hollerith constants
  - Character constants enclosed in either single quotes (`'`) or double quotes (`"`)
  - Hexadecimal constants. The hexadecimal digits are not case sensitive. To format the constant in standard Fortran notation, enclose the hexadecimal digits in either single or double quotes

and suffix the string with the letter `x` (for example, `'dddd'x`). Alternately, you can use CXdb notation by prefixing the hexadecimal digits with either `0x` or `0X` (for example, `0xdddd`).

- Octal constants. To form an octal constant, enclose the octal digits in either single or double quotes and suffix the letter `o` or `O` (for example, `'777'o`). Octal constants are not case sensitive.

- Logical constants:
  - White space is significant.
  - Logical constants are not case sensitive.
  - To form a logical constant, put a dot (`.`) before and after the word "true" or "false". For example, `.true.`
  - Alternate forms such as `.T.` and `.F.` are not supported.

3. Arithmetic expressions:

- Additive operators:

  \+ (binary)
  \- (binary)
  \+ (unary)
  \- (unary)

- Multiplicative operators (binary):

  *
  /

- Exponential operators (binary):

  **

4. Relational expressions:

- White space is significant.
- Operators are not case sensitive.
- The relational operators (binary) are:

```
.EQ.
.NE.
.LT.
.LE.
.GT.
.GE.
```

5. Logical expressions:
   - White space is significant.
   - Operators are not case sensitive.
   - The logical operators are:

     `.AND.` (binary)
     `.OR.` (binary)
     `.EQV.` (binary)
     `.NEQV.` (binary)
     `.NOT.` (unary)

6. Character expressions:
   - White space is significant.
   - Concatenation is done with the binary concatenation operator `//`.

7. Assignment statements:
   - Assignments can be made to either:
     - Program variables
     - CXdb debugger variables
   - The assignment operator is `=`.

8. Arrays:
   - Refer to the CONVEX *Fortran Language Reference* manual for specifications on array subscript expressions.
   - Array slices:
     - An array slice is a subscript expression that contains a slice range.
     - The data type of the slice is "array of ...", where the upper and lower bounds of the resulting array are defined by the slice range.
     - The syntax for an array slice is `(lower..upper)`, where lower is the first element and upper is the last element, inclusive.
     - Restrictions — All subscripts must be specified in the slice range. For example, if the array definition is `INTEGER Y(10,2)` and the slice specification is `Y(2..4,2)`, then the slice contains data from columns 2, 3, and 4 of row 2 (or rows 2, 3, and 4 of column 2 in the presence of the row-wise compiler directive).

9. Character substring expressions:
   - Refer to the CONVEX *Fortran Language Reference* manual for specifications on character substring expressions.

10. VAX records:
    - Refer to the CONVEX *Fortran Language Reference* manual for the structure and field selection syntax of VAX records.

11. Cray pointers:

- Refer to "Cray compatibility" in the CONVEX *Fortran User's Guide* for specifications on pointers.

12. Intrinsic functions:

- Restrictions:
  - Intrinsic functions cannot be passed as arguments in calls.
  - White space is significant.
  - Data types are relaxed. For example, `IIDNNT` accepts `REAL*4` and `REAL*16` as well as `REAL*8`.
  - Intrinsic function names are not case sensitive.
- Intrinsic functions for converting to integer:

```
INT
INT1
INT2
INT4
INT8
IFIX
IIFIX
JIFIX
KIFIX
```

- Intrinsic functions for converting to real:

```
REAL
DBLE
QEXT
```

- Intrinsic functions for converting to complex:

```
CMPLX
DCMPLX
```

- Intrinsic functions for character conversions:

```
CHAR
ICHAR
```

- Intrinsic functions for string comparisons:

```
LLT
LLE
LGT
LGE
```

- Intrinsic functions for conversion to nearest integer:

```
NINT
ININT
IIDNNT
IIQNNT
JNINT
JIDNNT
JIQNNT
KNINT
KIDNNT
KIQNNT
ANINT
```

- Intrinsic functions for address acquisition:

```
LOC
%LOC
```

- Miscellaneous intrinsic functions:

```
ABS
MOD
```

Examples

The following examples illustrate the use of Fortran language expressions in various CXdb commands.

```
(CXdb) break routine '80005508'x

#1: break routine, on [#0/*], Enabled, ignore 0/0
        [0x80005508] PRINT_ARRAY in example.f line 39
```

The above command sets a breakpoint at the beginning of the routine that contains the hexadecimal address `80005508`. The expression `'80005508'x` is Fortran notation for a hexadecimal address.

```
(CXdb) print I=I+5
(INTEGER*4) 5
```

In the above example, CXdb increments `I` by 5 and assigns this new data value to the program variable `I`. The command also prints the new value of `I`.

```
(CXdb) print I.EQ.1
(LOGICAL*4) .False.
```

The above command evaluates the relational expression `I.EQ.1` and prints the result of the evaluation.

```
(CXdb) print/x loc(PRINT_ARRAY)
(INTEGER*4) 0x80005506
```

The above command evaluates the expression `loc(PRINT_ARRAY)` and prints the result in hexadecimal (`/x`) format. The Fortran function `loc()` returns the address of the program variable `PRINT_ARRAY`. Thus, `8005506` is the hexadecimal address of `PRINT_ARRAY`. (The `0x` in front of the address indicates that it is a hexadecimal number.)

```
(CXdb) print ARRAY(2,1..4)
INTEGER*4(2:2, 1:4)
(2,1..4) :  2 4 6 8
```

The above command prints an array slice, or subset. The slice consists of elements `(2,1)` through `(2,4)` of `ARRAY`.

```
(CXdb) find memory forward 02 loc(ARRAY):40
Data found at 0x8008acef
```

The above command searches for the hexadecimal byte pattern `02` in a region of process memory. The memory region is specified by the language expression `loc(ARRAY):40`. The first part of the expression uses the Fortran function `loc()` to return the starting address of `ARRAY`. The second part of the expression is `40`, and it evaluates to the number of bytes of memory to search.

Related Commands

break instruction	break routine
copy	disassemble
evaluate	event relation
examine	fill
find memory backward	find memory forward
goto address	info expression
info frame at	print
trace instruction	trace routine
watch	

Fortran language expressions

Related Concepts

C language expressions
language expressions
scope
debugger variables
process object
source units

Related Parameters

array-slice
language-expression
debugger-variable
string

gdb debugger

Description

The CONVEX gdb debugger is a symbolic debugger. It has been replaced by CXdb, but the most frequently used `gdb` commands are available in CXdb through aliases. If you are already familiar with `gdb`, these aliases can make it easier for you to learn how to use CXdb.

To incorporate these aliases into CXdb, use the `gdb` command. If you always want to use the gdb aliases in CXdb, you can include the command `source /usr/lib/cxdb/aliases/gdb_aliases` in your `.cxdbinit` file. This incorporates the predefined gdb aliases automatically each time you invoke CXdb.

After incorporating the aliases, you can type a gdb command on the CXdb command line. If the command has an alias, the alias is expanded, and the equivalent CXdb command is executed. If there is no one-to-one correspondence with a CXdb command, CXdb displays a message indicating that the gdb command is not aliased and, where possible, suggests a CXdb command that is similar to the gdb command.

The gdb commands supported by CXdb aliases are:

gdb command	CXdb equivalent
`add-file`	Use `load object` command.
`b`	`break routine`
`commands`	Use an eventpoint handler.
`condition`	Use `if` command within an eventpoint handler.
`core-file`	`core`
`define`	Use `alias` command.
`delete`	`remove event`
`directory`	`add path`
`disable breakpoints`	`disable event`
`document`	No equivalent.
`down`	`frame -1`
`dump-me`	Send `kill` command to CXdb from the shell.
`enable breakpoints`	`enable event`
`exec-file`	`executable`
`forward-search`	`find source`
`handle`	`set signal`
`ignore`	Use `set ignore` command.
`info address`	`info expression`

gdb command	CXdb equivalent
`info comm-registers`	`info cregisters` (C Series only)
`info directories`	`info process`
`info display`	`info event`
`info files`	`info process`
`info functions`	`info symbols`
`info methods`	No equivalent.
`info sources`	`info objectmap`
`info types`	`info type`
`info variables`	`info symbols`
`jump`	Use `goto line` or `goto address` command.
`list`	Use `list` command.
`output`	No equivalent.
`printf`	No equivalent.
`printsyms`	`info symbols >`
`ptype`	`info type`
`reverse-search`	`find source`
`search`	No equivalent.
`set args`	Specify arguments with `run` or `rerun` command.
`set array-max`	`set printopts maxarray`
`set base`	Use `set format` and `examine`, or `print` with format options.
`set compile off`	No equivalent.
`set compile on`	No equivalent.
`set compiled-breakpoints`	No equivalent.
`set debug-flag`	No equivalent.
`set editing off`	No equivalent.
`set editing on`	No equivalent.
`set history expansion off`	No equivalent.
`set history expansion on`	No equivalent.
`set history file`	`add cmdlog`
`set history size`	No equivalent.
`set history write off`	`clear logging`
`set history write on`	`set logging`
`set parallel fixed`	`set fixed sched` (C Series only)
`set parallel off`	No equivalent.
`set parallel on`	`clear fixed sched`
`set pipeline off`	`set seq` (C Series only)
`set pipeline on`	`clear seq` (C Series only)
`set prettyprint`	No equivalent.
`set unionprint`	No equivalent.
`set prompt`	Done in .Xdefaults file.
`set screensize`	No equivalent.
`set verbose off`	No equivalent.

gdb command	CXdb equivalent
`set verbose on`	No equivalent.
`symbol-file`	Done automatically as needed.
`tbreak`	Use an eventpoint handler.
`term-status`	No equivalent.
`thread`	`info threads`
`tty`	Process interface window created automatically.
`undisplay`	No equivalent.
`unset environment`	`remove environment`
`until`	`finish loop`
`up`	`frame +1`

Related Commands

`csd`	`gdb`
`info alias`	`source`

Related Concepts

csd debugger

getting started with CXdb

Description

CXdb is a symbolic debugger for C and Fortran programs. It can debug programs at any of the optimization levels available with the CONVEX compilers.

You can invoke CXdb in either of two modes:

- X Windows mode
- Line mode

In X Windows mode, you can use CXdb's mouse-driven interface, or you can use the keyboard to enter commands in the CXdb Command window. Line mode is for keyboard input only.

Using CXdb in X Windows mode

1. From the shell, issue a command to compile your program with the `-cxdb` flag. For example:

   ```
   % fc -cxdb myprog.f
   ```

2. From the shell, invoke CXdb in X Windows mode. (This opens the CXdb Command window.) For example:

   ```
   % cxdb &
   ```

NOTE: If your DISPLAY environment variable is not set, CXdb will come up in line mode instead of X Windows mode.

3. In the CXdb Command window, use the `debug exec` command to specify the executable file for your program. (When you do this, CXdb creates a process object to store the process state information. CXdb also opens the Source Code window to display your source code.) For example:

   ```
   (CXdb) debug exec a.out
   ```

4. If the source files and CTI data files for your program are not in the same directory as the executable file, use the `add path` command to specify their locations.

5. Set an eventpoint near the beginning of your program. For example:

   ```
   (CXdb) break line 10
   ```

6. Run your program, and allow it to execute until it reaches the first eventpoint. With the `run` command, you can also pass arguments to your program or redirect its input and output. In the example below, arguments `arg1` and `arg2` are passed to the program, and program output is redirected to the file prog_output. (Note that the backslash is required before the shell redirection operator to distinguish it from a CXdb redirection operator.)

   ```
   (CXdb) run arg1 arg2 \> prog_output
   ```

7. Continue to debug your program by displaying variables (`print` command), stepping execution (`step` command), looking at the assembly language instructions (`disassemble` command), and so on. You can also open other CXdb windows to view the assembly language code, source code, memory, stack frames, registers, and online help.

8. To exit CXdb, use the `quit` command.

Using CXdb in line mode

1. From the shell, issue a command to compile your program with the `-cxdb` flag. For example:

   ```
   % fc -cxdb myprog.f
   ```

2. From the shell, invoke CXdb in line mode. For example:

   ```
   % cxdb -nw
   ```

NOTE: The `-nw` option overrides your DISPLAY environment variable.

3. Use the `debug exec` command to specify the executable file for your program. (When you do this, CXdb creates a process object to store the process state information.) For example:

   ```
   (CXdb) debug exec a.out
   ```

4. If the source files and CTI data files for your program are not in the same directory as the executable file, use the `add path` command to specify their locations.

5. Set an eventpoint near the beginning of your program. For example:

   ```
   (CXdb) break line 10
   ```

6. Run your program, and allow it to execute until it reaches the first eventpoint. With the `run` command, you can also pass arguments to your program or redirect its input and output. In the example below, arguments `arg1` and `arg2` are passed to the program, and program output is redirected to the file prog_output. (Note that the backslash is required before the shell redirection operator to distinguish it from a CXdb redirection operator.)

```
(CXdb) run arg1 arg2 \> prog_output
```

7. Continue to debug your program by looking at the source code (`list` command), displaying variables (`print` command), stepping execution (`step` command), and so on.
8. To exit CXdb, use the `quit` command.

Related Commands

`break instruction`
`break line`
`break routine`
`break source`
`cxdb`
`disassemble`
`help`
`print`
`quit`
`run`
`step`

Related Concepts

compiling for CXdb
process object
redirection
search path

Related Parameters

redirection-operator

Related Windows

Command window
Source Code window

Description

Initialization files are command files that CXdb executes automatically whenever it is invoked. Any of the CXdb commands may appear in an initialization file. The commands in an initialization file adhere to the same syntax rules as commands entered directly in the Command window.

The primary uses for initialization files are:

- Setting defaults for CXdb and any processes it creates
- Setting search paths
- Defining aliases
- Defining macros
- Executing a standard sequence of start-up commands

When first invoked, CXdb reads the initialization file and executes the commands in it one at a time. By using the `set echo` and `clear echo` commands, you can control whether or not each line of the initialization file appears in the Command window as it is executed.

If one of the lines in the initialization file causes an error, CXdb reports the error and proceeds to read and execute the next line in the file. CXdb also opens any windows required by each command in the file, and it waits for any interactive input needed from the user.

Initialization files can be created with a standard editor such as `vi` or `emacs`, and stored in ASCII format. Every initialization file must be named .cxdbinit. These files can reside in any of the following directories:

- /usr/lib/cxdb
- Your home directory
- Your console working directory

There can be a different .cxdbinit file in each of the above directories. When you invoke CXdb, it first executes the .cxdbinit file in /usr/lib/cxdb. It then searches for a .cxdbinit file in your home directory and executes that file, if it exists. Finally, if your console working directory is different from your home directory, CXdb searches for the .cxdbinit file in the console working directory and executes that file if it exists.

The commands in a later .cxdbinit file take precedence over the commands in an earlier initialization file. However, if a parameter is not explicitly changed by the new initialization file, then it retains its previous settings.

To continue a command across multiple lines of the initialization file, use a backslash (\) at the end of each continued line.

You can add comments to an initialization file by using a pound sign (#) at the beginning of each comment. CXdb ignores anything between the # and the end of the line.

Examples

The following example illustrates several uses of initialization files.

Assume there is a .cxdbinit file in the directory /usr/lib/cxdb, and that file contains the following lines:

```
alias p print
alias se 'step expression'
alias sl 'step loop'
set default step statement
```

Also assume there is a .cxdbinit file in the console working directory, and that file contains the following lines:

```
debug exec a.out
break routine l$_MAIN__     #Break at main routine
run
disassemble
alias s print
set default step loop
```

When CXdb is invoked, the following output appears in the Command window:

```
(CXdb)
Default source file: example.f
Default source language: Fortran

Process [#0] created

#0: break routine, on [#0/*], Enabled, ignore 0/0
        [0x800053b0] EXAMPLE in example.f line 7
Starting process [#0]: docexample
Process [#0/0] stopped by Bkpt 0, at [0x800053b0] EXAMPLE in example.f line 7
Disassemble Process [#0/0] from 0x800053a0 to 0x800054fc
   0x800053a0  EXAMPLE:         sub.w   #0,a0
   0x800053a2  EXAMPLE+(0x2):   mov     psw,a5
          .
          .
          .
```

First CXdb executes the file /usr/lib/cxdb/.cxdbinit, which establishes some aliases and sets the default step granularity to statement. Next CXdb executes the .cxdbinit file in the console working directory, and that file starts a process, displays the assembly language code, defines an alias, and sets the default stepping granularity.

Both .cxdbinit files contain the `set default step` command. The `set default step` command in the second .cxdbinit file overrides the `set default step` command in the first .cxdbinit file. So, the default stepping granularity is set to loop in this case.

Both files also define an alias for the `print` command. The two alias names are different, so there is no conflict in this case. Therefore, the second name is also added to the list of aliases. Now both `p` and `s` are valid aliases for the `print` command.

Related Commands

`add cmdlog`
`clear echo`
`clear logging`
`info cxdb`
`remove cmdlog`
`set cmdlog`
`set echo`
`set logging`
`source`

Related Concepts

cmdlog
command files
console working directory
logging

Related Parameters

file-name

Related Windows

Command window

language expressions

Description

A language expression is any expression that can be evaluated in the current source language. The current source language is the language of the source file associated with the current stack frame.

A language expression can contain any valid combination of the following:

- Literal values
- Character strings
- Operators
- Program identifiers (including their scope paths, if necessary)
- Debugger variables

The exact syntax and meaning of language expressions depend on the source language used to construct them. CXdb supports all language expressions that are valid in either Fortran or C. For details on language specifics, refer to the concepts pages for "Fortran language expressions" and "C language expressions" in this manual.

In addition to the standard C and Fortran language expressions, CXdb also supports the following extensions:

- Array slices — Subsets of an array
- Memory region specifiers — Two different formats for specifying a region of memory:

  <starting-address>`..`*<ending-address>*
  <starting-address>`:`*<unit-count>*

CXdb evaluates a language expression according to the rules of the current source language. The resulting value of the language expression is then used in the CXdb command that contains the expression. CXdb commands use the resulting value of a language expression in one of two ways:

- As a data value
- As an address

For example, the `break routine` command uses the resulting value of a language expression as an address, but the `print` command uses it as a data value.

When another parameter follows the language expression on the CXdb command line, you can terminate the language expression with a backslash-semicolon (\;) to distinguish it from the next parameter.

Examples

The following examples illustrate the use of language expressions as data values and as addresses. Where there are differences between Fortran and C syntax, examples of both are shown.

```
(CXdb)  break routine PRINT_ARRAY \; $BreakA

#0: break routine, on [#0/*], Enabled, ignore 0/0
        [0x80005508] PRINT_ARRAY in example.f line 39
```

The above command sets a breakpoint at the beginning of the routine called `PRINT_ARRAY`. In this case, CXdb interprets the language expression `PRINT_ARRAY` to be the starting address of the routine `PRINT_ARRAY`. The debugger variable `$BreakA` stores the number of this breakpoint, which is 0. The language expression terminator (`\;`) is required to separate the language expression `PRINT_ARRAY` from the debugger variable `$BreakA`.

```
(CXdb) break routine '800029bc'x
Process [#0/0] stopped by Bkpt 1, at [0x800029bc] BLD_MATRIX in
      chapter7F.f line 26
```

The above command sets a breakpoint at the beginning of the routine that contains the hexadecimal address `8000139a`. The expression `'800029bc'x` is Fortran notation for a hexadecimal address.

```
(CXdb) break routine 0x80004c32
#2: break routine, on [#0/*], Enabled, ignore 0/0
       [0x80004c32] chapter13C`subxa in chapter13C.c line 45
```

The above command sets a breakpoint at the beginning of the routine that contains the hexadecimal address `80004c32`. The expression `0x80004c32` is C-language notation for a hexadecimal address.

```
(CXdb) print I+2
(INTEGER*4) 2
```

In the above example, CXdb evaluates the language expression `I+2` and prints the resulting data value.

```
(CXdb) print J=I+2
(INTEGER*4) 2
```

In the above example, CXdb evaluates the language expression `J+2` and assigns this new data value to the program variable `J`. The command also prints the new value of `J`.

```
(CXdb) print/x loc(J)
(INTEGER*4) 0x800772a4
```

The above command evaluates the expression `loc(J)` and prints the result in hexadecimal (`/x`) format. The Fortran function `loc()` returns the address of the program variable `F`. Thus, `800772a4` is the hexadecimal address of `J`. (The `0x` in front of the address indicates that it is a hexadecimal number.)

```
(CXdb) print/x &info->subject
(char*) 0xffffca28
```

The above command evaluates the expression `&info->subject` and prints the result in hexadecimal (`/x`) format. The C operator `&` returns the address of the program variable `info->subject`. Thus, `8000c268` is the hexadecimal address of `info->subject`. (The `0x` in front of the address indicates that it is a hexadecimal number.)

```
(CXdb) print "This is a test"
(CHARACTER*15) 'This is a test.'
```

The above command prints a literal string of characters.

```
(CXdb) print $B=1
(INTEGER*4) 1
(CXdb) print $B=$B+5
(INTEGER*4) 6
```

The first command above initializes the debugger variable `$B` to a value of 1 and prints the result. The second command increments `$B` by 5 and prints that result.

```
(CXdb) print ARRAY(1,1..4)
INTEGER*4(1:1, 1:4)
(1,1..4) :  0 2 3 4
```

The above command prints an array slice, or subset. The slice consists of elements `(1,1)` through `(1,4)` of `ARRAY`. The subscripts in this example are in Fortran notation.

```
(CXdb)  print table[0][0..3]
int[1][4]
[0][0..3] :  2 6 12 20
```

The above command prints an array slice, or subset. The slice consists of elements `[0][0]` through `[0][3]` of `table`. The subscripts in this example are in C notation.

```
(CXdb) find memory forward ff '80002096'x..'80002196'x
Data found at 0x800020c0
```

The above command searches for the hexadecimal byte pattern `ff` in a region of process memory. The memory region is specified by the language expression `'80002096'x..80002196'x`, where `80002096` is the starting address of the region and `80002196` is the ending address. This example uses Fortran notation for the address expression. The same address range specified in C syntax is `0x80002096..0x80002196`.

```
(CXdb) find memory forward 09 loc(ARRAY):I+64
Data found at 0x8008ad13
```

The above command searches for the hexadecimal byte pattern `09` in a region of process memory. The memory region is specified by the language expression `loc(ARRAY):I+64`. The first part of the expression uses the Fortran function `loc()` to return the starting address of `ARRAY`. (The equivalent function for returning the address of a variable in C syntax is `&`.) The second part of the expression is `I+64`, and it evaluates to the number of bytes of memory to search.

Related Commands

`break instruction`
`break routine`
`copy`
`disassemble`
`evaluate`
`event relation`
`examine`
`fill`
`find memory backward`
`find memory forward`
`goto address`
`info expression`
`info frame at`
`print`
`trace instruction`
`trace routine`
`watch`

Related Concepts

C language expressions
debugger variables
Fortran language expressions
process object
scope
source units

Related Parameters

array-slice
debugger-variable
language-expression
string

Related Windows

Memory Display window
Source Code window

Description

Line mode allows you to use CXdb interactively without the graphical user interface. Line mode is designed primarily for use on windowless terminals such as a VT-100, but you can also use it on X terminals.

At your shell prompt, enter the following command to invoke CXdb in line mode:

```
cxdb -nw
```

NOTE: The `-nw` option overrides your DISPLAY environment variable.

In line mode, echoing of input from command files and initialization files is disabled by default. To enable echoing of this input, use the `set echo` command.

In line mode, process output appears intermixed with CXdb output on your screen (stdout), unless you redirect the output. By default, the output is paged using the `less` utility.

NOTE: If your PAGER environment variable is set to anything other than `less`, paging of output will not work properly.

Editing the command line

CXdb line mode provides command line editing functions similar to UNIX command line editing. You can display the available editing functions by entering **META?** or **ESC-?** on the CXdb command line. The editing functions are as follows:

Function	Key sequence
backward character	**CTRL-b**
backward word	**ESC-b**
beginning of line	**CTRL-a**
capitalize forward word	**ESC-c**
complete current command	**TAB**
delete backward character	**CTRL-h**
delete backward character	**DEL**
delete backward word	**ESC-h**
delete forward character	**CTRL-d**
delete forward word	**ESC-d**
display key bindings	**META-?** or **ESC ?**
end of line	**CTRL-e**

erase line	CTRL-g
erase screen	ESC-g
execute current command	RETURN
forward character	CTRL-f
forward word	ESC-f
kill to end of line	CTRL-k
list current command choices	ESC-TAB
lower case work	ESC-l
next command	CTRL-n
previous command	CTRL-p
redraw screen	CTRL-l
redraw screen	CTRL-r
transpose characters	CTRL-t
transpose words	ESC-t
upper case word	ESC-u

Source code context in line mode

In line mode, whenever the process is stopped or the stack frame is changed, CXdb displays 5 lines of source code surrounding the line of source code at the point of execution. A > symbol to the right of the line number indicates the line containing the current point of execution.

As shown in the following example, lines of source code context are displayed when your process is stopped by a breakpoint or an eventpoint of type spawn, join, signal, or reached. The > symbol in the output indicates that line 9 contains the current point of execution.

Source code context is also displayed when the process stops after executing commands affect process execution such as `step`, `stepi`, `next`, `finish`, or `goto`.

```
(CXdb) break line 9

#0: break line, on [#0/*], Enabled, ignore 0/0
 [0x800053d4] EXAMPLE in example.f line 9
(CXdb) run
Starting process [#0]: /doc/cxdb/examples/docexample
 EXAMPLE PROGRAM STARTED
Process [#0/0] stopped by Bkpt 0, at [0x800053d4] EXAMPLE in
      example.f line 9
 7   : PRINT *, "EXAMPLE PROGRAM STARTED"
 8   :
 9 >: DO I=1,4
 10  : DO J=1,4
 11  : ARRAY(I,J)=I*J
```

When you change the current stack frame by executing the `frame` command or the aliases `up` and `down`, CXdb also outputs lines of source code context, as shown in the following example.

```
(CXdb) frame +2
    2 : 0x800054ca  in EXAMPLE()  (example.f line 26)
  24  :          CALL CHAPTER13F(ARRAY)
  25  :          CALL chapter13c(ARRAY)
  26 >:  CALL CHAPTER15
  27  :       ELSE
  28  :          CALL EXAMPLE_INPUT
```

Commands not available in line mode

The following commands are *not* available in line mode:

`display disassembly`—Use the `disassemble` command.
`display examine`—Use the `examine` command.
`display routine`–Use `list routine` *<language_expression>*.
`display source`—Use the `list` command.
`display stack`—Use the `info frame` command.
`recall`–Press **CTRL-p** to backwards and **CTRL-n** to go forward through the CXdb command history.

Most of these commands open additional CXdb windows for displaying information in X Windows mode.

Related Commands

`cxdb`

Related Concepts

command files
initialization files
redirection

Related Windows

Command window

Description

Logging is the recording of activity that takes place during a debugging session. You can control three major aspects of logging with CXdb:

- Type of information logged
- Overwrite protection on log files
- Logging and echoing of input

Type of information logged

There are three types of information you can log, and each type has a particular name:

- cmderr — Error messages and informational messages generated by CXdb in response to commands. (Equivalent to stderr in the shell.)
- cmdlog — User entries on the CXdb command line. (Equivalent to stdin in the shell.)
- cmdout — Normal output generated by CXdb in response to commands. (Equivalent to stdout in the shell.)

You can log any of the above information by directing it to a viewport. A viewport is either a file, the CXdb Command window (in X Windows mode only), or stderr and stdout (in line mode only).

You can direct cmderr, cmdlog, and cmdout to any number of viewports at the same time by specifying a separate list of viewports for each of these three types of information. Use the following commands to modify the viewport lists:

- `add cmderr` — Add file names to the list of cmderr viewports.
- `add cmdlog` — Add file names to the list of cmdlog viewports.
- `add cmdout` — Add file names to the list of cmdout viewports.
- `remove cmderr` — Remove file names from the list of cmderr viewports.
- `remove cmdlog` — Remove file names from the list of cmdlog viewports.
- `remove cmdout` — Remove file names from the list of cmdout viewports.

- `set cmderr` — Delete all file names from the current list of cmderr viewports, and replace them with a new list of file names.
- `set cmdlog` — Delete all file names from the current list of cmdlog viewports, and replace them with a new list of file names.
- `set cmdout` — Delete all file names from the current list of cmdout viewports, and replace them with a new list of file names.

By default, the viewport list for cmderr and cmdout always contains the CXdb Command window (or stderr and stdout in line mode). You cannot delete the Command window (or stderr and stdout in line mode) from the viewport lists. There is no default viewport for cmdlog because your entries on the CXdb command line are always automatically echoed there.

The viewports for cmderr and cmdout apply to all commands executed by CXdb, regardless of whether the commands are entered directly or read from a command file.

Overwrite protection for log files

If you specify a viewport file that does not exist, CXdb creates it. If a specified file already exists, CXdb attempts to overwrite the contents of that file. You can use the following commands to control whether or not CXdb overwrites an existing log file:

- `clear noclobber` — Allow overwriting of existing log files.
- `set noclobber` — Respond with an error message if attempting to overwrite an existing log file.

The default for noclobber is off (clear).

NOTE: To append to an existing log file, use redirection operators.

Logging and echoing of input

After defining the viewport list for cmdlog (input), you can enable and disable input logging periodically during the debugging session. This is useful if there is certain input that you do not want to record in the log file. To enable or disable logging of all input to cmdlog, use the following commands:

- `clear logging` — Disable logging to the cmdlog viewports.
- `set logging` — Enable logging to the cmdlog viewports.

The default for input logging is off (clear).

If your CXdb input is coming from command files or initialization files, you can also control whether or not that input is echoed to the cmdlog viewports by using the following commands:

- `clear echo` — Do not echo the input from command files and initialization files to the viewports of cmdlog.
- `set echo` — Echo the input from command files and initialization files to the viewports of cmdlog.

The default for echo is off (clear).

Displaying logging information

The `info cxdb` command displays the current settings for echo, logging, and noclobber, as well as the current viewport lists for cmderr, cmdlog, and cmdout.

Other methods of logging

In addition to viewports, you can use redirection operators to log CXdb output and error messages. For each individual command, redirection operators can override the viewport lists for cmdout and cmderr. The redirection operators enable you to specify a special viewport list that applies only to the one command with which it appears. Redirection operators also allow you to append to existing log files in addition to overwriting them.

Examples

The following examples illustrate some typical situations for logging.

Logging everything to the same file

To keep a record of an entire debugging session, you might want to log all input, output, and error messages to a single file. The following example shows how to log everything to a file called my_session.

```
(CXdb) set noclobber
(CXdb) add cmderr my_session
New cmderr: Window #[1], my_session
(CXdb) add cmdlog my_session
New cmdlog: my_session
(CXdb) add cmdout my_session
New cmdout: Window #[1], my_session
(CXdb) set logging
```

In the above example, the `set noclobber` command enables the noclobber option and prevents overwriting of existing files. The `add cmderr`, `add cmdlog`, and `add cmdout` commands specify the file my_session as a viewport to receive all input, output, and error messages. Finally, the `set logging` command enables logging of input (cmdlog) to my_session.

The `info cxdb` command shows the new settings of all the logging options, as illustrated below.

```
(CXdb) info cxdb

Current CXdb state:

     ENVIRONMENT:
             pid: 7750
             cwd: /doc/cxdb/examples
   command modes: echo off, logging on, noclobber on
          cmdout: Window #1, my_session
          cmderr: Window #1, my_session
          cmdlog: my_session
       evalopts: fpmode = dual, iprecision = 4, rprecision = 4
           shell: tcsh

PROCESS DEFAULTS:
fixed scheduling: Off
       step size: statement
   process shell: csh
          fpmode: dual
     memory size: (none)
  memory formats: byte=(none), halfword=(none), word=(none)
                  longword=(none), quadword=(none)
     search path:
         .


       PROCESSES:
```

Logging input only

In some cases, you might want to log just the input that you enter on the CXdb command line. This is a useful way of recording your session as well as a way of creating a command file that you can execute later to repeat your steps.

```
(CXdb) set noclobber
(CXdb) add cmdlog my_input
New cmdlog: my_input
(CXdb) set logging
```

In the above example, the `set noclobber` command enables the noclobber option and prevents overwriting of existing files. The `add cmdlog` command specifies the file my_input as a viewport to receive all input entered on the CXdb command line. This includes input entered directly by you as well as input read in from command files. Finally, the `set logging` command enables logging of input (cmdlog) to my_input.

After logging your input to a file, you can edit that file and use the `source` command to execute it as a command file if you ever need to repeat your steps. For example:

```
(CXdb) set echo
(CXdb) source my_input

(CXdb) debug exec para

Default source file: para.f
Default source language: Fortran

Process [#0] created
(CXdb) break line 20

#0: break line, on [#0/*], Enabled, ignore 0/0
        [0x80001654] SUB1 in para.f line 20
(CXdb) run
Starting process [#0]: para
Process [#0/0] stopped by Bkpt 0, at [0x80001654] SUB1 in para.f
      line 20
...
```

In the above example, the `set echo` command enables echoing of input read from command files. Next, the `source` command executes the command file called my_input. The CXdb commands stored in my_input then begin to appear on the screen as they execute.

Using redirection operators

You can use redirection operators on individual CXdb commands to override the general viewports. For example:

```
(CXdb) info process > p_state
```

In the above example, the output from the `info process` command is redirected to the file p_state. The redirection affects only this one command. Note that the output from this command does not appear on the screen because it has been redirected to p_state only.

Related Commands

`add cmderr`
`add cmdlog`
`add cmdout`
`clear logging`
`clear noclobber`
`info cxdb`
`remove cmderr`
`remove cmdlog`
`remove cmdout`
`set cmderr`
`set cmdlog`
`set cmdout`
`set logging`
`set noclobber`

Related Concepts

cmderr
cmdlog
cmdout
redirection
viewports

Related Parameters

redirection-operator
viewport

Related Windows

Command window

Description

With CXdb commands, you can modify the contents of:

- Program variables
- Registers
- Memory regions
- Debugger variables

The process you are debugging must be stopped before you can modify a data value. The following sections explain how to modify data values with CXdb commands.

Modifying variables and registers

The `print` and `evaluate` commands allow you to assign a new value to a variable or register. You can use any valid language expression to make the assignment. The `evaluate` command just assigns the new value to the specified variable or register, but the `print` command also displays the new value after making the assignment. The following example illustrates some of these points:

```
(CXdb) evaluate J=J+1
(CXdb) print J=J+1
(INTEGER*4) 2
(CXdb) print $S1=2
(INTEGER*8) 2
(CXdb) print $SaveVal=ISQR(3)+$S1
(INTEGER*8) 11
```

In the above example, the `evaluate J=J+1` command increments program variable `J` by 1, but it does not display the result. The `print J=J+1` command also increments `J` by 1, and it displays the result. The `print $S1=2` command assigns the integer value 2 to scalar register S1. Finally, the `print $SaveVal=ISQR(3)+$S1` command executes the program function `ISQR` with an argument of 3, adds the result of that evaluation to the value in register S1, and stores the sum in the newly created debugger variable `$SaveVal`.

In addition to the `print` and `evaluate` commands, you can use any CXdb command that accepts a language expression as a parameter to modify data values. For example:

```
(CXdb) info expression $Count=I+J

object type: Fortran expression result
       size: 4 bytes
       type: INTEGER*4
      value: 3
```

The above example creates the debugger variable `$Count` and uses it to store the value of the language expression `I+J`.

Modifying memory (C Series only)

To modify the contents of a memory region on a C Series machine, use the `copy` or `fill` commands. The `copy` command copies one region of memory into another, while the `fill` command fills a memory region with the value resulting from a language expression. For example:

```
(CXdb) copy loc(MATRIX):20 \; loc(MATRIX(1,5,1))
(CXdb) fill loc(MATRIX):5 \; 0
```

In the above example, the `copy` command copies the first 20 *bytes* of the array `MATRIX` into memory starting at `MATRIX(1,5,1)`. The Fortran function `loc()` provides the source and destination addresses in this case. The language expression terminator (`\;`) separates the two expressions for the source and destination addresses.

The `fill` command in the above example fills the first 5 elements of `MATRIX` with the value 0. The Fortran function `loc()` provides the starting address for the fill in this case.

Note that the `copy` command always operates on bytes of memory, but the memory unit size for the `fill` command is determined by the item specified as the starting address. In the above example, the memory unit for the `fill` command is one element of the array `MATRIX` because the starting address for the fill is specified as `loc(MATRIX)`.

Caution

The `copy` and `fill` commands do not ask for confirmation before writing to process memory. Use these commands carefully, or you could accidentally write over areas of memory that you did not intend to affect.

Related Commands

`copy`	`evaluate`
`examine`	`fill`
`info expression`	`print`

Related Concepts

debugger variables	displaying data
language expressions	registers

Description

CONVEX compilers generate highly optimized object code that maximizes use of the architectural features of your machine. Many optimizations are performed automatically when you compile your source code with one of the command line options for optimization (`-O3`, `-O2`, etc.). You can also optimize a program yourself by inserting compiler directives (for example, `FORCE_PARALLEL`) in your source code.

CXdb can perform source-level debugging of any of the optimizations generated on CONVEX machines. Source units are the underlying mechanisms that enable this type of debugging.

Source units and optimized code

Source units are elements that represent the syntax and structure of your source code. The types of source units are:

- **Expression** — Any combination of constants, operators, and operands that is valid in the source language of your program.
- **Statement** — One or more expressions that form a single complete instruction in the source language.
- **Block** — The set of statements that make up the body of a routine, loop, or conditional construct. Block source units are not the same as basic blocks.
- **Loop** — A special type of statement that encloses a block source unit and causes it to execute repeatedly.
- **Routine** — A main routine, subroutine, or function.

When you compile your program with the `-cxdb` option, the compiler generates a set of Compiler-Tools Interface (CTI) data files that contain information about all the source units of your program. These CTI data files provide a mapping between the source units in your source code and the machine instructions in the object code generated by the compiler. CXdb then uses this mapping information to perform symbolic debugging of your program.

Different levels of optimization produce different mappings between source units and machine instructions, even when the source code is the same. The mapping also depends on the size (granularity) of the source units. Sometimes one source unit can map to one machine instruction, but it is more likely that one source unit maps to several machine instructions and that one machine instruction maps to several source units. The highlighting of source units in the Source Code window reflects all of these types of mappings.

Synthesized variables

If you compile your program at optimization level `-O1` or higher, the compiler generates synthesized variables that are more efficient implementations of your program variables. For example, the compiler might replace your array subscript (a loop induction variable) with a pointer to the array. CXdb can track these synthesized variables and display them for you.

The `info expression` command lists all the synthesized variables related to a single program variable. For example:

```
(CXdb) info expression J
object type: Fortran identifier
   location: <none>
       size: 4 bytes
       type: INTEGER*4
      value: 1
       used to create 6 synthesized variable(s):
         1. <INDV>     ?i0 = J+((-1*N)-1)
         2. <INDV>     ?i1 = loc(A)+((4*M)*(J-1))
         3. <INDV>     ?i2 = loc(B)+((4*M)*(J-1))
         4. <INDV>     ?i5 = (-4+?i1)+(4*(I-1))
         5. <INDV>     ?i6 = ?i2+(4*(I-1))
         6. <SEXP>     ?c7 = ?i1+(-1*(4*(1+(-1*M))))
```

The above example shows that there are 6 synthesized variables derived from the program variable `J`. The names of the synthesized variables always start with a special character, such as a question mark (`?`). The `info expression` command also lists the equation used to derive each synthesized variable along with the reason for the synthesis (such as `INDV` for induction variable).

Notice that the storage location for `J` in the above example is `<none>`. This is because `J` has been replaced by synthesized variables, so `J` is not stored or accessed directly.

When you want to print or display a program variable that has been synthesized, CXdb calculates the current value of a program variable from the equations for the associated synthesized variable. Sometimes CXdb must try to solve several simultaneous equations that do not have a unique solution. When this happens, CXdb issues a message indicating that either the program variable you requested is not currently available or the variable has more than one possible current value.

For more information about synthesized variables, read the reference page on "synthesized variables."

Hints for debugging optimized code

Although the specifics of each case are different, there are some general guidelines for debugging optimized code that can help in most cases:

1. Compile and debug your program at the lowest available optimization level first. After fixing all the bugs at this level, recompile your program at the next highest optimization level and debug it again. Continue this process of recompiling and debugging until your program performs properly at the desired optimization level.
2. Use the `set step expression` command to set the stepping granularity to expression. The stepping granularity controls the level of source unit highlighting in the Source Code window, and expressions are the finest granularity of source unit. Therefore, setting the granularity to expression gives the most detailed highlighting of source units.
3. Use the `info line` command to display the source units at a given line of source code. This command lists the address ranges of machine instructions that map to each source unit. Such information can help with setting eventpoints, stepping the process, and tracking its execution.
4. Use the `info expression` command rather than the `print` command to display program variables. The `info expression` command shows the liveness ranges and storage locations for each variable. It also lists any synthesized variables that are derived from the specified program variable.
5. Use the Assembly Code window (or the `disassemble` command) to display the assembly language code associated with the current point of execution. This shows which machine instructions map to the current source unit. It also shows which instruction will execute next, thus revealing the execution order of the optimized code.

6. At parts of the program that are of critical interest to you, use the `step instruction` command to continue execution of the process. Stepping by one instruction at a time helps ensure that you do not miss or overstep the critical points in the program.

Setting eventpoints in optimized code

Eventpoints such as `break line` and `trace routine` are based on a location or address in the source code. CXdb sets these eventpoints at the machine instruction that maps to the source code location you specify. Because optimizations can change the mapping of machine instructions to source code, the eventpoint markers displayed in the Source Code window do not always appear where you expect to see them. For example:

```
(CXdb) break routine LEVEL_O2
#1: break routine, on [#0/*], Enabled, ignore 0/0
        [0x80004328] LEVEL_O2 in chapter15.f line 88

(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] stopped by Bkpt 1, at [0x80004328] LEVEL_O2 in
       chapter15.f line 88
```

```
 84          SUBROUTINE LEVEL_O2(M,N,A,B,X)
 85          REAL A(M,N), B(M,N)
 86
 87          DO J=1,N
 88 B           DO I=1,M
 89                TEMP = 3.0 * B(I,J)
 90                A(I,J) = TEMP/(2.0*X)
 91                B(I,J) = 2.0 * TEMP
 92             ENDDO
 93          ENDDO
 94
 95          RETURN
 96          END
```

In the above example, `LEVEL_O2` is a Fortran subroutine that has been compiled at optimization level `-O2`. The `break routine` command sets a breakpoint at the first executable source unit of `LEVEL_O2`. You might expect this breakpoint to appear at the outer `DO` loop on line 87, but instead it appears at the inner `DO` loop on line 88 because of the optimizations that have occurred.

In some cases, the same eventpoint can appear at several different addresses even though you specified only one location for it. This occurs when optimizations cause one source unit to map to several different ranges of machine instructions, or when one machine instruction maps to several different source units.

Stepping through optimized code

Stepping follows the execution order of the machine instructions for your process, but optimizations can change the order of those instructions as well as the mapping between machine instructions and source units. Because of this, it generally is not possible to predict where execution will stop if you are stepping a process by source units or source code lines. To make your results more predictable, use a more careful and systematic stepping approach, such as the following:

1. Step by large source units such as blocks, loops, or routines until the current point of execution is relatively near the program location that is of interest to you.
2. If in doubt about which stepping granularity to use or how many steps to take at once, use a smaller granularity and take fewer steps at one time.
3. Near critical points of your program, use the `step instruction` command until you reach the precise location that is of interest to you.
4. For loop iterations, use the `step block` or `next block` commands rather than stepping by loop.
5. While stepping the process, examine the assembly language code and source code frequently to determine where execution has stopped and which instruction will execute next.

Debugging multiple threads

If you compiled your program at optimization level `-O3`, it can generate multiple threads that execute in parallel. (A thread is a sequence of instructions that executes on a single CPU.) The following hints can help you debug programs that have been optimized in this way:

1. Before debugging with CXdb on CONVEX C-Series machines, enable fixed scheduling with the `set fixed sched` command.
2. Use the `event spawn` and `event join` commands to detect the creation and termination of threads.
3. Use the Thread Activity window and the navigation hints area of the Source Code window to locate the most interesting threads in your program.
4. Use thread lists to apply CXdb commands to specific threads. For example, the command `:t2\step instruction` steps thread 2 by one machine instruction, but other threads are not affected.

For more information on threads, read the "threads" reference page.

Related documentation

For more information about specific types of optimizations, refer to the following documents:

- CONVEX Fortran Optimization Guide
- CONVEX C Optimization Guide
- CONVEX Exemplar Programming Guide

Examples

The following example illustrates some typical steps for debugging optimized code. Assume the example program is running on a C Series machine.

```
(CXdb) debug exec docexample
Default source file: example.f
Default source language: Fortran

Process [#0] created

(CXdb) set step expression

(CXdb) break routine LEVEL_O2
#0: break routine, on [#0/*], Enabled, ignore 0/0
        [0x80004328] LEVEL_O2 in chapter15.f line 88

(CXdb) run
Starting process [#0]: docexample
Process [#0/0] stopped by Bkpt 0, at [0x80004328] LEVEL_O2 in chapter15.f line 88

(CXdb) info line 90
     Id    Address Boundaries     Start       End      Kind
 1. (204)  80004380:80004386    90 x 28    90 x 30  <EXPR>  2.0
 2. (205)  80004336:8000433a    90 x 32    90 x 32  <EXPR>  X
 3. (203)  80004380:80004386    90 x 28    90 x 32  <EXPR>  2.0*X
           80004336:8000433a
 4. (201)  80004390:80004392    90 x 22    90 x 25  <EXPR>  TEMP
           8000437a:80004380
 5. (202)  80004380:80004386    90 x 27    90 x 33  <EXPR>  (2.0*X)
           80004336:8000433a
 6. (198)        0:       0    90 x 15    90 x 15  <EXPR>  I
 7. (199)        0:       0    90 x 17    90 x 17  <EXPR>  J
 8. (200)  80004390:80004394    90 x 22    90 x 33  <EXPR>  TEMP/(2.0*X)
           8000437a:80004386
           80004336:8000433a
 9. (197)  80004390:80004398    90 x 13    90 x 33  <STMT>  A(I,J) = TEMP/(2.0*X)
           8000437a:80004386
           80004336:8000433a
```

```
(CXdb) break source 197
#1: break source, on [#0/*], Enabled, ignore 0/0
        [0x80004336] LEVEL_O2 in chapter15.f line 90
        [0x8000437a] LEVEL_O2 in chapter15.f line 91
        [0x80004390] LEVEL_O2 in chapter15.f line 91

(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] stopped by Bkpt 1, at [0x80004336] LEVEL_O2 in chapter15.f line 90

(CXdb) info expression X
object type: Fortran identifier
   location: @(@($ap+16)) <0x800044f8>
       size: 4 bytes
       type: REAL*4
      value:     1.3000
       used to create 2 synthesized variable(s):
         1. <SEXP>    ?cd = ((2*X))
         2. <SEXP>   ?c19 = ((2*X))
       2 liveness ranges:
              Start      End         Location
         1.  0x8000433a:0x80004362 - register s3
         2.  0x80004328:0x800043d0 - @(@($ap+16))

(CXdb) disassemble $pc:3
Disassemble Process [#0/0] from 0x80004336 for 3 machine instructions
   0x80004336  LEVEL_O2+(0x12):        ld.w    @16(ap),s3              ;  X
   0x8000433a  LEVEL_O2+(0x16):        ld.w    12(ap),a5               ;  B
   0x8000433e  LEVEL_O2+(0x1a):        ld.w    8(ap),a1                ;  A

(CXdb) step instruction
Stepping process [#0/*] by 1 instruction
Process [#0/0] stopped stepping at [0x8000433a] LEVEL_O2 in chapter15.f line 88

(CXdb) info expression X
object type: Fortran identifier
   location: register s3
             @(@($ap+16)) <0x800044f8>
       size: 4 bytes
       type: REAL*4
      value:     1.3000
       used to create 2 synthesized variable(s):
         1. <SEXP>    ?cd = ((2*X))
         2. <SEXP>   ?c19 = ((2*X))
       2 liveness ranges:
              Start      End         Location
         1.  0x8000433a:0x80004362 - register s3
         2.  0x80004328:0x800043d0 - @(@($ap+16))
```

In the above example, the `debug exec` command begins the debugging session by specifying the executable file docexample. The `set step expression` command sets the stepping granularity to expression so that the Source Code window shows the most detailed highlighting of source units. The `break routine` command sets a breakpoint at the beginning of the `LEVEL_O2` routine, which has been compiled at optimization level `-O2`. The `run` command then starts execution of the program, which stops at the breakpoint in routine `LEVEL_O2`.

The `info line` command displays information about all the source units associated with line 90 of the source code. The `break source` command sets a breakpoint at source unit 197. Notice that this breakpoint appears at three locations because source unit 197 maps to three address ranges in the executable object code. The `continue` command continues process execution until it reaches the first of these three breakpoints.

The `info expression` command displays information about the program variable `X`, including the synthesized variables derived from `X`. Notice that the current value of `X` is stored in memory at location `800044f8`. The `disassemble` command reveals that the next instruction loads the value of `X` into scalar register S3. The `step instruction` command executes this instruction, and the final `info expression` command shows that the value of `X` is now stored in register S3 as well as memory location `800044f8`.

Related Commands

`clear default fixed sched`	`clear fixed sched`
`event join`	`event spawn`
`info expression`	`info line`
`info threads`	`set default fixed sched`
`set fixed sched`	`set threads`
`signal thread`	

Related Concepts

synthesized variables

threads

Related Parameters

thread-list

process object

Description

A process object contains all of the information about the process you are currently debugging in CXdb. The process object is made up of the following three parts:

- Image — The image being debugged. An image can be thought of as a snapshot of a process. You can debug three different types of images:
  - Process image — The image of a running process.
  - Core image — The image of a dead process found in a core file.
  - Executable image — The image from an executable file that is used to create a new process.
- CTI information — The information used to map symbols from the source code to the image. The CTI includes:
  - Executable file — The basis for the rest of the CTI information.
  - CTI data files generated by the compiler, as specified in the executable file.
  - Source files — The source code of the program, as specified in the executable file and CTI data files.
- Process settings — The settings that control various aspects of a running process. The available process settings are:
  - Process working directory in which to run the process
  - Environment in which to run the process
  - Search path for source files and compiler-generated data files
  - Eventpoints created in the process and their handlers
  - Display formats
  - Memory formats
  - Process shell in which the process executes
  - Settings of seq and sqs bits (C Series only)
  - Remote working directory for remote debugging (C Series only)
  - Floating point mode (C Series only)

Creating a process object

To debug a process in CXdb, you must create a process object, and have an image to debug. To symbolically debug a process, you must also specify an executable file as the basis for the CTI information. You can create a process object by using any one of the following `debug` commands:

- `debug core` — Specifies a core image and, if the `executable` flag is used, specifies an executable file as the basis for the CTI information.
- `debug exec` — Specifies an executable file as the basis for the CTI information and also creates an executable image from that file.
- `debug proc` — Specifies a process image from a process that is already running, and if the `executable` flag is used, specifies an executable file as the basis for the CTI information.

Specifying new CTI information for the process object

Once a process object has been created, you can change the basis for the CTI information or change the image being debugged. The following commands enable you to change the image or CTI information in the process object:

- `executable` — Specifies an executable file as the basis for the CTI information. Any CTI information currently in the process object is replaced with the information from the new executable file and its corresponding CTI data files and source files.
- `run` — Creates a new process from the executable image. The image from this new process becomes the image being debugged, replacing an existing core or process image.
- `attach` — Attaches to a running process. The image from this process becomes the image being debugged, replacing an existing core or process image.
- `core` — Retrieves the image of a dead process from a core file. This image becomes the image being debugged, replacing an existing core or process image.
- `kill process` — Removes a core or process image from the process object. If an executable file has been specified, the executable image from the executable file once again becomes the image being debugged.

Before a process image is replaced with either a new process image or a core image, CXdb asks you to confirm the action. This ensures that a process image, which cannot be retrieved, is not lost accidentally.

If the process you are debugging exits, its image is removed from the process object. If an executable file has been specified (through either the `debug exec` or `executable` command), the executable image from the executable file once again becomes the image being debugged.

Examples

The following series of examples create and then modify the process object in CXdb.

```
(CXdb) debug exec docexample

Default source file: example.f
Default source language: Fortran

Process [#0] created
```

This first example creates a process object. The executable file docexample provides the basis for the CTI information in the process object. CXdb uses the docexample file to find the appropriate CTI data files and source files. The image being debugged at this point is the executable image created from the docexample file.

A Source Code window opens, showing the main routine of the program. At this point you can look at disassembled code for the executable image, and print global and static symbols.

```
(CXdb) attach 20860
Attaching Process [#0] to pid 20860
Process [#0/0] stopped by attach at [0x80001500] CHAPTER4 in chapter4.f line 10
```

The above command attaches CXdb to the process whose process ID (`pid`) is `20860`. CXdb attaches to the process and then stops it. The process is now under CXdb's control. The process image for this process becomes the image being debugged. However, the CTI information is still based on the executable file docexample. The Source Code window updates to reflect the new image in the process object.

```
(CXdb) run &
Command [#21] backgrounded

Process [#0] is already running with pid 20860.
Terminate existing process and restart? y
Starting process [#0]: docexample
Command [#21] completed
```

The `run &` command in the above example creates a new process from the executable image of docexample. Process `20860` is terminated and its process image is replaced by the process image from the new process. The ampersand (`&`) is used to place the `run` command in the background.

```
(CXdb) executable para
Default source file: para.f
Default source language: Fortran
```

The above example replaces the CTI information in the process object with the information in the new executable file, para, and its associated data files and source files. An executable image from the para file is created and replaces the existing executable image.

The Source Code window is deleted and a new Source Code window opens for the new CTI information. However, the image being debugged is still that created by the `run` command in the previous example. Typically, you would *not* want to continue debugging a process using another executable file's CTI information.

```
(CXdb) break routine SUB1
#0: break routine, on [#0/*], Enabled, ignore 0/0
        [0x800014dc] SUB1 in para.f line 16

(CXdb) run
Starting process [#0]: para
Process [#0/0] stopped by Bkpt 0, at [0x800014dc] SUB1 in para.f line 16
```

The `break routine` command in the above example sets a breakpoint at the beginning of the `SUB1` routine. Note that the breakpoint is placed according to the para executable file, not the docexample process currently running. The `run` command replaces the existing process image with the process image from a new process. The new process is created from the executable image from the executable file para. The Source Code window updates to reflect the new process image.

```
(CXdb) info process

status of process [#0]:

      executable: para
       arguments: (none)
fixed scheduling: off
          pshell: csh
    image status: created pid 10322, state = stopped
     working dir: /doc/cxdb/examples
    default step: statement
default language: Fortran
         threads: 2 [active: 0]
  current thread: 0

 thread 0 status: stopped at [0x800014dc] SUB1 in para.f line 16

source file search path:
        .
```

The above command displays the current settings of the process object.

The following examples manipulate the process settings of the process object.

```
(CXdb) add environment PAGER = less
```

The above command creates an environment for the process object based on the default environment and then adds the environment variable PAGER to the newly created environment.

```
(CXdb) set directory $PROG2
(CXdb) add path /mnt/jones/program2/source
```

The above two commands set up the process working directory and search path for the process object. The process working directory is the directory from which the process will run. The search path is used to find the source files of the program.

Related Commands

`add environment`
`add path`
`attach`
`clear default remotewd`
`clear environment`
`clear fixed sched`
`clear seq`
`clear sqs`
`core`
`debug core`
`debug exec`
`debug proc`
`detach`
`executable`
`info environment`
`info path`
`info process`
`info threads`
`kill process`
`remove environment`
`remove path`
`rerun`
`run`
`set default remotewd`
`set directory`
`set environment`
`set format`
`set fpmode`
`set memory`
`set path`
`set pshell`
`set remotewd`

Related Concepts

breakpoints
Compiler-Tools Interface
environment
eventpoints
eventpoint handlers
remote debugging
search path
tracepoints
watchpoints

Related Parameters

process-list
thread-list

Related Windows

Source Code window

C Series only

process working directory

Description

The process working directory is the directory from which CXdb runs your program.

The following commands work with the process working directory:

- `set directory` — Sets the process working directory.
- `info process` — Displays the current setting of the process working directory.

All relative path names in your program use the process working directory as the base path.

The process working directory is initially set to reflect the current console working directory. Thus, you can change the console working directory with the `cd` command, and the process working directory will change as well. Once you set the process working directory using the `set directory` command, the process working directory will no longer reflect changes to the console working directory.

After each modification to the process working directory, the new directory is added to the search path of the process object if it is not already in the search path.

Examples

The following commands use the process working directory.

```
(CXdb) set directory /doc/cxdb/examples
```

This command sets the process working directory to be the /doc/cxdb/examples directory.

```
(CXdb) info process

status of process [#0]:

      executable: docexample
       arguments: (none)
fixed scheduling: off
          pshell: csh
    image status: executable image from file docexample
     working dir: /doc/cxdb/examples
    default step: statement
default language: Fortran

source file search path:
        .
```

The above command displays information about the current process object. The setting of the process working directory is shown to be /doc/cxdb/examples. The process will use this directory to locate any files that it tries to access.

Related Commands

`add default path`
`add path`
`info cxdb`
`info process`
`remove default path`
`remove path`
`set directory`
`set default path`
`set path`

Related Concepts

console working directory
default search path
search path

Related Parameters

directory-specifier

Description

Redirection is the ability to direct input, output, and error messages to a location (usually a file) other than your terminal. With CXdb, you can:

- Redirect the input and output of the process you are debugging
- Redirect CXdb output and messages
- Log CXdb input, output, and messages

Redirecting process I/O

You can redirect the input and output of the process you are debugging by using a shell redirection operator with the CXdb `run` command. Use a backslash (\) before the shell redirection operator to distinguish it from a CXdb redirection operator. For example:

```
(CXdb) run \< prog_input \> prog_output
```

The above command starts execution of your program. Input to the program comes from the file prog_input, and output from the program is directed to the file prog_output.

Redirecting CXdb output and messages

CXdb has its own set of redirection operators that are different than the shell redirection operators.

NOTE: CXdb redirection operators apply only to the particular command in which they appear.

The operators for redirecting CXdb command *output* are:

Operator	Meaning
>	Overwrite existing output file, unless overwrite protection is enabled (with the `set noclobber` command).
>>	Append new output to an existing file. An error occurs if the specified file does not exist.
>!	Overwrite existing output file, regardless of overwrite protection.
>>!	Append new output to an existing file. Create the file if it does not exist.

The following example illustrates the use of these operators. In this example, the output from the `info process` command is appended to the file process_status.

```
(CXdb) info process >> process_status
```

In the above example, the CXdb output is sent only to the file process_status. The output does not display on the screen because the Command window (or stdout in line mode) was not included as one of the destinations for the redirection. To display the output on your screen at the same time it is redirected to a file, specify window number 1 (or stdout in line mode) in the redirection list, as follows:

```
(CXdb) info process >> 1, process_status
```

The operators for redirecting CXdb error *messages* are:

Operator	Meaning
`>&`	Overwrite existing message output file, unless overwrite protection is enabled (`set noclobber` command).
`>>&`	Append new message output to an existing file. An error occurs if the specified file does not exist.
`>&!`	Overwrite existing message output file, regardless of overwrite protection.
`>>&!`	Append new message output to an existing file. Create the file if it does not exist.

The following example illustrates the use of these operators. In this example, any error messages from the `source` command are appended to the file startup_errors.

```
(CXdb) source startup_file >>& startup_errors
```

Logging CXdb input, output, and messages

While CXdb redirection operators affect only the current command, logging lets you define a redirection operation for the entire debugging session. Logging also allows you to record (log) the input you enter in CXdb as well as the output and error messages. For more information on logging, refer to the "logging" topic.

Related Commands	`clear noclobber`	`set noclobber`
Related Parameters	redirection-operator	viewport
Related Concepts	logging	viewports

Description

CXdb provides predefined debugger variables that enable you to access the registers on your machine. Different types of machines have different types of registers. The following sections list the registers that are available on each type of CONVEX computer. The registers are listed as the debugger variable names that you can use to access them.

C2 and C3 Series only

You can access the following registers on C2 and C3 Series machines:

- `$a0` to `$a7` (or `$A0` to `$A7`) — The 32-bit address registers A0 to A7.
- `$ap` (or `$AP`) — The argument pointer, AP (same as A6).
- `$c0` to `$c63` — The 32-bit communication registers from ring 4, regardless of which set is allocated.
- `$C0` to `$C63` — The 64-bit communication registers from ring 4, regardless of which set is allocated.
- `$cl0` to `$cl63` (or `$CL0` to `$CL63`) — The lock bits for the communication registers (one bit per register).
- `$cir` (or `$CIR`)— The 3-bit communication index register, CIR. You can access this register only for an active process that is stopped.
- `$fp` (or `$FP`) — The frame pointer, FP (same as A7).
- `$pc` (or `$PC`) — The program counter, PC.
- `$psw` (or `$PSW`) — The processor status word, PSW.
- `$s0` to `$s7` — The 32-bit scalar registers S0 to S7.
- `$S0` to `$S7` — The 64-bit scalar registers S0 to S7.
- `$sp` (or `$SP`) — The stack pointer, SP (same as A0).
- `$v0` to `$v7` — The 32-bit vector registers V0 to V7. Each vector register can contain up to 128 elements (numbered 0 to 127), and each element is 32 bits long. To access a particular element, use array notation. For example, to access the 123rd element of vector register V4, use the notation `$v4(123)` in Fortran or `$v4[122]` in C.

- `$V0` to `$V7` — The 64-bit vector registers V0 to V7. Each vector register can contain up to 128 elements (numbered 0 to 127), and each element is 64 bits long. To access a particular element, use array notation. For example, to access the 123rd element of vector register V4, use the notation `$V4(123)` in Fortran or `$V4[122]` in C.
- `$vl` (or `$VL`) — The vector length register, VL.
- `$vml` (or `$VML`) — The lower half of the vector merge register, VM.
- `$vmu` (or `$VMU`) — The upper half of the vector merge register, VM.
- `$vs` (or `$VS`) — The vector stride register, VS.

C4 Series only

You can access the following registers on C4 Series machines:

- `$a0` to `$a31` (or `$A0` to `$A31`) — The 32-bit address registers A0 to A31.
- `$ap` (or `$AP`) — The argument pointer, AP (same as A6).
- `$c0` to `$c63` — The 32-bit communication registers from ring 4, regardless of which set is allocated.
- `$C0` to `$C63` — The 64-bit communication registers from ring 4, regardless of which set is allocated.
- `$cl0` to `$cl63` (or `$CL0` to `$CL63`) — The lock bits for the communication registers (one bit per register).
- `$cir` (or `$CIR`) — The 3-bit communication index register, CIR. You can access this register only for an active process that is stopped. The CIR register is read-only.
- `$ss0` and `$ss1` — Two 32-bit scalar stride registers, SS0 and SS1. The `ld0` and `ld1` instructions use these registers to permit explicit cache prefetching under software control.
- `$evcnt` (or `$EVCNT`) — The 64-bit hardware event counter register, EVCNT.
- `$evsel` (or `$EVSEL`) — The 32-bit hardware event selector register, EVSEL.
- `$fp` (or `$FP`) — The frame pointer, FP (same as A7).
- `$pc` (or `$PC`) — The program counter, PC.
- `$psw` (or `$PSW`) — The processor status word, PSW.
- `$s0` to `$s31` — The 32-bit scalar registers S0 to S31.
- `$S0` to `$S31` — The 64-bit scalar registers S0 to S31.

- `$sp` (or `$SP`) — The stack pointer, SP (same as A0).
- `$v0` to `$v15` — The 32-bit vector registers V0 to V15. Each vector register can contain up to 128 elements (numbered 0 to 127), and each element is 32 bits long. To access a particular element, use array notation. For example, to access the 123rd element of vector register V4, use the notation `$v4(123)` in Fortran or `$v4[122]` in C.
- `$V0` to `$V15` — The 64-bit vector registers V0 to V7. Each vector register can contain up to 128 elements (numbered 0 to 127), and each element is 64 bits long. To access a particular element, use array notation. For example, to access the 123rd element of vector register V4, use the notation `$V4(123)` in Fortran or `$V4[122]` in C.
- `$vf` (or `$VF`) — The vector first register, VF, specifies the first element of a vector register be either read or written by a vector instruction.
- `$vl` (or `$VL`) — The vector length register, VL.
- `$vml` (or `$VML`) — The lower half of the vector merge register, VM.
- `$vmu` (or `$VMU`) — The upper half of the vector merge register, VM.
- `$vs` (or `$VS`) — The vector stride register, VS.

SPP Series only

You can access the following registers on SPP Series machines:

- `$arg0` to `$arg3` (or `$ARG0` to `$ARG3`)— The argument registers arg0 to arg3 (same as r26 to r23, respectively).
- `$cr0` to `$cr31` (or `$CR0` to `$CR31`) — The control registers cr0 to cr31. You can access only cr0, cr8 to cr10, cr12, cr13, cr15, cr19 to cr22, and cr24 to cr26. The other control registers are reserved for system use. All control registers are read-only.
- `$dp` (or `$DP`) — The data pointer, dp (same as r27).
- `$farg0` to `$farg3` (or `$FARG0` to `$FARG3`) — The floating point argument registers farg0 to farg3 (same as fr4 to fr7, respectively).
- `$fr0` to `$fr31` (or `$FR0` to `$FR31`) — The 64-bit floating point registers fr0 to fr31. Registers fr0 through fr3 are read-only.
- `$fr0l` to `$fr31l` (or `$FR0L` to `$FR31L`) — The upper (left) half (32 bits) of the floating point registers fr0 to fr31. Registers fr0 through fr3 are read-only.
- `$fr0r` to `$fr31r` (or `$FR0R` to `$FR31R`) — The lower (right) half (32 bits) of the floating point registers fr0 to fr31. Registers fr0 through fr3 are read-only.

- `$fret` (or `$FRET`) — The floating point return register, fret (same as fr4).
- `$iioq_head` (or `$IIOQ_HEAD`) — The pointer to the head of the instruction queue (same as pc).
- `$iioq_tail` (or `$IIOQ_TAIL`) — The pointer to the tail of the instruction queue.
- `$mrp` (or `$MRP`) — The millicode return pointer, mrp (same as r31).
- `$pc` (or `$PC`) — The program counter, pc.
- `$psw` (or `$PSW`) — The processor status word, psw (same as cr22).
- `$r0` to `$r31` (or `$R0` to `$R31`) — The 32-bit general registers r0 to r31.
- `$ret0` and `$ret1` (or `$RET0` to `$RET1`) — The return value registers ret0 and ret1 (same as r28 and r29, respectively).
- `$rp` (or `$RP`)— The return pointer, rp (same as r2).
- `$sarg` (or `$SARG`) — The space argument register sarg (same as sr1).
- `$sl` (or `$SL`) — The static link register, sl (same as r29).
- `$sp` (or `$SP`) — The stack pointer, sp (same as r30).
- `$sr0` to `$sr7` (or `$SR0` to `$SR7`) — The space registers sr0 to sr7.
- `$sret` (or `$SRET`) — The space return register, sret (same as sr1).

Displaying register contents

If you are using CXdb in X Windows mode, you can open special register windows to display the contents of registers. In both line mode and X Windows mode, you can use the following commands to display the registers:

- `info control registers` — Displays the contents of all accessible control registers.
- `info cregisters` — Displays the contents of all accessible communication registers.
- `info expression` — Displays the contents of any accessible register.
- `info floating point registers` — Displays the contents of all accessible floating point registers.
- `info psw` — Displays the contents of the processor status word.

- `info registers` — Displays the contents of the program counter, the processor status word, the stack pointer, and all accessible address registers, scalar registers, and general registers.
- `info space registers` — Displays the contents of all accessible space registers.
- `info vregisters` — Displays the contents of all accessible vector registers.
- `print` — Displays the contents of any accessible register.

Modifying register contents

You can assign a new value to a register with any CXdb command that accepts a language expression as a parameter. For example:

```
(CXdb) evaluate $s3=22
```

The above command assigns a value of 22 to scalar register S3.

Caution

Modifying registers such as the program counter (pc) can drastically alter the execution of your program. In addition, if you change the value of the pc (iioq_head) on an SPP Series machine, you will also probably need to change the value of iioq_tail.

Examples

The following examples illustrate how to use CXdb commands to access the registers.

```
(CXdb) info registers
Process [#0/0]
pc : 0x80002ae2
psw: 0x03909480
fp : 0xffffc968
ap : 0x80002d70
a5 : 0x00000258
a4 : 0x00000040
a3 : 0x8008ad38
a2 : 0x0000026c
a1 : 0x000001f4
sp : 0xffffc958
s7 : 0x3030303000000000
s6 : 0x6c756520000023cf
s5 : 0x0000000000000001
s4 : 0x5445523720535441
s3 : 0x494e45200000001e
s2 : 0x2053554200000000
s1 : 0x5441525400000004
s0 : 0x2020202000000001
```

The above example uses the `info registers` command to display the address and scalar registers on a C2 machine.

```
(CXdb) print $vl
(INTEGER*4) 128
```

The above command prints the contents of the vector length register.

```
(CXdb) examine $pc
Examine Process [#0/0] from 0x80002ae2 to 0x80002b2e
 80002ae2:    32b00008  3638fff4  11880001  36408007
 80002af2:    72a83240  800772a8  3239fff4  4d810640
 80002b02:    80002c6e  32408007  72a43241  800772a8
 80002b12:    50c5510a  168a0003  32438007  72a02a41
 80002b22:    800772b8  50aa1482  ffff5988  5c515998
```

The above command displays the contents of a memory region that begins at the address specified by the program counter (pc).

```
(CXdb) evaluate $s3=22
```

The above command assigns a new value of 22 to scalar register S3.

Related Commands

`info control registers`
`info cregisters`
`info expression`
`info floating point registers`
`info psw`
`info registers`
`info space registers`
`info vregisters`
`print`

Related Concepts

debugger variables
displaying data
language expressions
modifying data

Related Windows

Communication Registers window
Control Registers window
Floating Point Registers window
Processor Status Word window
Scalar Registers window
Space Registers window
Vector Registers window

C Series only

remote debugging

Description

CXdb enables you to debug an executable file, a core file, or a process residing on a remote host (a CONVEX C Series machine other than the one on which CXdb is currently running). Once you specify the remote executable file, core file, or process, debugging proceeds normally.

Requirements for remote debugging

To debug a process or file on a remote host, your system must meet the following requirements:

- Both the local and remote machines must have CXdb (V1.2 or later) installed.
- You must have an account capable of running CXdb on both machines.
- All source files and CTI data files for your program must reside on the local host. (If necessary, you can copy them from the remote host.) The associated executable file, the running process, and any generated core files or checkpoint files can reside on either the local or remote host.
- Your .rhosts file on the local machine must contain the name of the remote host. You can find the remote host name and its Internet address in the /etc/hosts file.
- You must be able to execute a remote shell from the local host.

Initiating remote debugging

The following commands allow you to specify a remote process, executable file, or core file to debug:

- `debug exec` — Creates a process object with an image of the executable file you specify.
- `executable` — Specifies a different executable file to debug.
- `debug proc` — Creates a process object with an image of the process you specify. CXdb stops the process immediately.
- `attach` — Attaches to a different process to debug. CXdb stops the process immediately.
- `debug core` — Creates a process object with an image of the core file you specify.
- `core` — Specifies a different core file to debug.

To debug a remote file or process with any of the above commands, specify the remote host name (or its Internet address) followed by a colon (:) followed by the desired file name or process ID. For example:

```
(CXdb) debug exec pixel:/usr/smith/programs/docexample

Default source file: example.f
Default source language: Fortran

Process [#0] created
```

The above command initiates debugging of the executable file docexample. This file resides in the directory /usr/smith/programs on the remote host pixel.

If you specify a remote executable file or core file using a relative path name, CXdb searches for the file in the following order:

- Remote working directory — The relative path for executable files and core files on the remote host. You can set this path by using the `set remotewd` command.
- Default remote working directory — The default setting for the remote working directory. You can set this default directory by using the `set default remotewd` command.
- Remote console working directory — A remote directory with the same name as the local console working directory (the directory from which you invoked CXdb). You can set the *local* console working directory by executing the `cd` command from within CXdb.
- Your home directory on the remote host — Your login directory on the remote host.

The `info process` command displays the remote working directory and indicates whether or not the executable file resides on a remote host.

All other aspects of remote debugging are the same as local debugging. You can use the full CXdb command set to debug a remote process or file.

Caution

Debugging of processes or files on a *remote* C4 Series machine is not supported at this time. Results are unpredictable if you attempt to debug on a remote C4 host. However, you can debug normally with a C4 Series machine as a *local* host.

Examples

The following series of examples debugs a remote executable file and process.

```
(CXdb) set default remotewd /usr/smith/programs
```

The above command sets the default remote working directory. If you now specify a remote file with a relative path name, CXdb uses the /usr/smith/programs directory as a base path name. You can override this directory by specifying a remote directory for the process object (using the `set remotewd` command) or by clearing the default remote working directory (using the `clear default remotewd` command).

```
(CXdb) debug exec pixel:docexample

Default source file: example.f
Default source language: Fortran

Process [#0] created
```

The above example creates a process object. The executable file docexample provides the basis for the CTI information in the process object. The executable file is located on the remote host named pixel. Because an absolute path was not given, CXdb searches for the file in the default remote working directory (/usr/smith/programs). CXdb uses the docexample file to find the appropriate CTI data files and source files on the *local* host.

The image being debugged at this point is the executable image created from the docexample file.

```
(CXdb) attach pixel:18324
process on pixel
Attaching Process [#0] to pid 18324
Process [#0/0] stopped by attach at [0x80001500] CHAPTER4 in chapter4.f line 10
```

The above command attaches CXdb to the process whose process ID (`pid`) is `18324` on the remote host named pixel. The process was created previously from the remote executable file docexample. CXdb attaches to the process and then stops it. This remote process can now be debugged as if it were a local process.

The image for this process is now being debugged. The CTI information is still derived from the executable file docexample. The Source Code window updates to reflect the new image in the process object.

```
(CXdb) run

Process [#0] is already running with pid 18324.
Terminate existing process and restart? y

Starting process [#0]: docexample
```

The `run` command creates a new process from the executable image of docexample. The old process image that was attached in the previous example is replaced by the process image from the new process. The new process is running on the remote host pixel because docexample, the executable file used to create the process, is located on pixel.

```
(CXdb) info process
status of process [#0]:

      executable: docexample
       arguments: (none)
fixed scheduling: off
          pshell: csh
    image status: created pid 23001, state = stopped
     remote host: pixel
     working dir: /usr/smith/programs
    default step: statement
default language: Fortran
         threads: 1
  current thread: 0

 thread 0 status: stopped at [0x8000149a] CHAPTER4 in chapter4.f line 6

source file search path:
```

The above command displays the current status of the process and the process object. The remote host is pixel, and the remote working directory is /usr/smith/programs. The executable file docexample resides in the remote working directory.

Related Commands

`add path`	`attach`
`clear default remotewd`	`core`
`debug core`	`debug exec`
`debug proc`	`detach`
`info path`	`info process`
`kill process`	`remove path`
`rerun`	`run`
`set default remotewd`	`set path`
`set remotewd`	

Related Concepts

background execution	Compiler-Tools Interface
process object	process working directory
search path	

saving data

Description

After running your program for a while, you might want to save the data values and run the program again using the same data. To do this, you can use the following commands:

- `put` — Saves specified data (memory regions) to a binary file.
- `get` — Retrieves data from a binary file (saved with the `put` command) and restores it to program memory.

When you use the `put` command to save data to a file, CXdb creates the specified file, if it does not already exist. The file is created in the console working directory unless you specify a different path name. If the save file already exists, the `put` command overwrites it. (To prevent overwriting of a save file, use the `set noclobber` command.)

The following sections describe how to use the `put` and `get` commands to save and restore various types of program data.

Saving and restoring memory regions

The `put` and `get` commands can save and restore any accessible region of program memory. You can specify the memory region in one of two ways:

- An address range
- A starting address and a byte count

For example, you can use an address range with the `put` command as illustrated below:

```
(CXdb) put saveMem 0x80004700..0x80004707
```

The above command saves 8 bytes of memory to the binary file saveMem. The saved memory region ranges from address `80004700` to `80004707`.

You can also use a byte count to specify a memory region, as follows:

```
(CXdb) get saveMem 0x80004680:8
```

The above command retrieves the 8 bytes (`:8`) from the saveMem file and restores them to program memory starting at address `80004680.`

Saving and restoring variables

To save and restore variables, specify the storage addresses of the variables, as illustrated below:

```
(CXdb) put saveVars loc(I) \; loc(J) \; loc(K)
```

The above command saves the contents of program variables `I, J,` and `K` in the binary file saveVars. The Fortran function `loc()` provides the storage addresses of the variables. Note that the language expression terminator (`\;`) is required to separate multiple address expressions in the list.

To restore these variable values to their original locations, you can use the `get` command without specifying any addresses. For example:

```
(CXdb) get saveVars
```

The above command restores the values of `I, J,` and `K` to their original locations.

If you want to restore the values to different locations, or if you want to restore only some of the saved values, you must specify the new storage addresses with the `get` command. For example:

```
(CXdb) get saveVars loc(I) \; loc(N)
```

The above command restores the original value of `I` to the current location of `I,` and the original value of `J` to the current location of `N.` The `get` command retrieves the data values in the order they were saved in the file, without skipping any values in between. However, you can skip values at the end of the file. Thus, the original value of `K` is not restored in this case.

NOTE: The data type and storage size of the restored variable must match those of the original saved variable. If they differ, the data value might not restore properly.

Saving and restoring arrays

You can use the `put` and `get` commands to save and restore an entire array or parts of an array. For example:

```
(CXdb) put saveArrays SLICE \; MATRIX
```

The above command saves the two arrays `SLICE` and `MATRIX` to the file saveArrays.

If you want to save part of an array, you can specify the address range of the array elements that you want to save. For example:

```
(CXdb) put saveSome loc(TABLE(1,1))..loc(TABLE(4,1))
```

The above command saves elements `TABLE(1,1)` to *but not including* `TABLE(4,1)`. The Fortran function `loc()` provides the addresses of these array elements.

You can also specify part of an array using a byte count, as illustrated below:

```
(CXdb) put saveSome loc(TABLE(1,1)):16
```

The above command saves 16 bytes of the array `TABLE`, starting at the address of element `TABLE(1,1)`.

NOTE: The `put` and `get` commands save and restore array elements in the same order that the elements are stored in memory. The storage order of array elements depends on the source language of your program.

Related Commands

`get` `put`

Related Concepts

displaying data
modifying data
language expressions

Related Parameters

language-expression

Description

The scope of a program identifier determines where that identifier is visible. There are two concepts involving scope in CXdb:

- **Scope path** — A scope path is a complete path to the declaration of a program identifier. Scope paths enable you to reference program identifiers that are not currently visible, such as static variables, shadowed variables, and global variables. Scope paths also enable you to reference variables in other namespaces, such as loader symbols or debugger variables.
- **Current scope** — The current scope is used to find identifiers that do not have a complete scope path. The current scope is determined by the current program counter (PC). Usually the PC is in the current stack frame, but by using the `frame` command you can change the current frame, thus also changing the current scope.

A scope path can be used to access several different namespaces. Namespaces are the storage allocations given to the different types of language identifiers CXdb can recognize. The possible namespaces are:

- Debugger variables (Scope path is `cxdb$` or `$`.)
- Loader symbols (Scope path is `l$`, or `asm$`.)
- Fortran (Scope path is explained below.)
- C (Scope path is explained below.)

Because the scope rules are different for Fortran and C, so are their scope paths.

In Fortran programs

In Fortran, scope paths enable you to reference common block identifiers from any point in the program. This lets you view the data of a common block from different perspectives.

The syntax for the scope path in Fortran is:

[£$][*<routine-name>*`\`]*<identifier>*

- *<routine-name>* — The routine in which the identifier is declared. If the routine name is omitted, the current scope is used to find the identifier.

In Fortran, the local variables for a routine are allocated on the call stack if the routine is reentrant code or in memory if the routine is not reentrant. The local variables maintain their values between calls if they are allocated in memory (not reentrant), but they do not maintain their values if they are allocated on the stack (reentrant).

You can use CXdb scope paths to access local variables that are stored in memory or in the current stack frame. However, to access a variable in a reentrant routine that is not in the current stack frame, you must first use the `frame` command to change to the stack frame that contains the desired routine.

The compiler default settings for reentrant Fortran code are different on C Series and SPP Series machines, as indicated below.

C Series:

- By default, the Fortran compiler does not generate reentrant code.
- Compile with the `-re` option to generate reentrant code, if desired.

SPP Series:

- By default, the Fortran compiler generates reentrant code.
- If desired, compile with the `-nore` option to generate code that is not reentrant.

In C programs

In C, scope paths enable you to reference static identifiers and identifiers that are shadowed. An identifier is shadowed when another identifier with the same name is declared in an inner block of the same scope.

The syntax for the scope path in C is:

[**c$**][<*file-name*>`'`][<*routine-name*>`'`][<*block-list*>`'`]<*identifier*>

- <*file-name*> — The name of the file in which the identifier is declared. If this is omitted, the current source file is used.
- <*routine-name*> — The name of the routine in which the identifier is declared. If this is omitted, the current routine is used or, if a *file-name* has been specified, the global declarations of that file.
- <*block-list*> — A list of block numbers specifying the block in which the identifier is declared. A block is anything inside of braces. Unless you have compiled your program using the `-pcc` option, block numbers are not assigned to blocks in which variables are not declared. There can be one or more blocks in a block list, corresponding to the nesting of the blocks.

The first block, inside the routine, is block 1. Subsequent blocks have incremental block numbers. Each new level of blocks begins again at 1. The following C pseudocode uses identifiers named block to help demonstrate block numbers.

```
            main()
  |----------{
  |              static float block = 0;
  |       |------{
  |       |          static float block = 1;
  |       |   |-----{
  |       |   1         static float block = 1.1;
  |       1   |-----}
  |       |   |-----{
  |       |   2         static float block = 1.2;
  |       |   |-----}
main      |------}
  |               {   /* block ignored unless -pcc option used */
  |               }
  |       |------{
  |       2          static float block = 2;
  |       |------}
  |       |------{
  |       3          static float block = 3;
  |       |------}
  |               execution_stopped_here();
  |----------}
```

Assuming execution was stopped at the end of the routine, the following scope paths print the different identifiers named block:

```
print block - 0
print `1`block - 1
print `1`1`block - 1.1
print `1`2`block - 1.2
print `2`block - 2
print `3`block - 3
```

The current scope from the end of the routine is the main routine. Thus, no scope path is needed to reach the identifier with a value of zero.

By starting a scope path with a backquote, you are asking CXdb to find the identifier within the current routine. In the above commands, the current scope was the main routine. Thus, all of the scope paths that start with a backquote begin with an implied `c$prog`main`.

Examples

The following examples demonstrate the use of scope paths; first for Fortran and then for C.

In Fortran

Consider the following Fortran source code.

```
1       SUBROUTINE CHAPTER13F(ARRAY)
2       INTEGER ARRAY(4,4), I
3       CHARACTER*12 INFO_BLOCK
4       COMMON /BLK/ INFO_BLOCK
5
6       PRINT *, "SUBROUTINE CHAPTER13F BEGINNING"
7       CALL SUB13D
8       OPEN (7, FILE="data")
9      PRINT *, "TEST NAME    TEST-1  TEST-2  TEST-3  AVERAGE"
10     PRINT *, "---------    ------  ------  ------  -------"
11      DO I=1,3
12         CALL SUB13A
13         CALL SUB13B
14         CALL SUB13C
15      ENDDO
16      CLOSE (7)
17      PRINT *
18      PRINT *, "SUBROUTINE CHAPTER13F FINISHING"
19      PRINT *
20      END
21
22      SUBROUTINE SUB13B
23      CHARACTER*6 NAME
24      INTEGER*1 NUM(6), I
25      REAL AVG
26      COMMON /BLK/ NAME, NUM, AVG
27
28      SUM = 0
29      DO I=1,6,2
30          SUM = SUM + (NUM(I)-48)*10 + (NUM(I+1) - 48)
31      ENDDO
32      AVG = SUM / 3
33      END
```

Assume that program execution has been stopped at the call to the function `SUB13B` (line 13) during the first iterations of the `DO` loop. Also assume that the source code is *not* reentrant, so variables are allocated storage in memory rather than on the stack. The current scope is based from frame 0, the current point of execution.

```
(CXdb) info scope
Process [#0/0], frame 0 scope: f$CHAPTER13F
(CXdb) print INFO_BLOCK
(CHARACTER*12) 'CONVEX999897'
(CXdb) print I
(INTEGER*4) 1
(CXdb) print SUB13B`NAME
(CHARACTER*6) 'CONVEX'
(CXdb) print SUB13B`NUM
INTEGER*1(1:6)
(1..3)= 57
(4)=56
(5)=57
(6)=55
(CXdb) print SUB13B`I
(INTEGER*1) 0
```

The above example prints out the values of the identifiers. Because INFO_BLOCK is in the same COMMON block as NAME and NUM, they represent the same identifier location. Thus, they can be referenced in the subroutine because they share the same storage as INFO_BLOCK. The variable I, on the other hand, has a different storage location and a different value in each routine.

For the next series of examples, assume that process execution is stopped just before the end of SUB13B (line 33).

```
(CXdb) info scope
Process [#0/0], frame 0 scope: f$SUB13B
(CXdb) print NAME
(CHARACTER*6) 'CONVEX'
(CXdb) print NUM
INTEGER*1(1:6)
(1..3)= 57
(4)=56
(5)=57
(6)=55
(CXdb) print CHAPTER13F`INFO_BLOCK
(CHARACTER*12) 'CONVEX999897'
```

The above examples print out the values of the various identifiers. Again, because NAME and NUM represent the same memory storage as INFO_BLOCK, they can all be referenced from the subroutine.

In C

The next series of examples use the following C source code.

```
1       ansi_block(i)
2       int i;
3       {
4          i = 0 ;
5
6          if (i == 0)
7          {
8             int i=1 ;
9
10            if (i == 1)
11            {
12                int i=11 ;
13            }
14
15            if (i == 1)
16            {
17                int i=12 ;
18            }
19         }
20        printf("\nThe routine ansi_block is finishing.\n");
21      }
```

CXdb scope paths allow you to access the different identifiers named `i`. Assume that execution has stopped after executing the first instruction of line 17.

```
(CXdb) info scope
Process [#0/0], frame 0 scope: c$ansi_block`ansi_block`1`2
(CXdb) print i
(int) 12
(CXdb) print ansi_block`i
(int) 0
(CXdb) print ansi_block`1`i
(int) 1
(CXdb) print ansi_block`1`1`i
(int) 11
(CXdb) print ansi_block`1`2`i
(int) 12
```

The above examples reference the different identifiers named `i`. The current scope is found using the `info scope` command.

The current scope is shown to be the block in which `i` is set to 12. The first command does not use a scope path, so `i` is found in the current scope.

Now assume that execution has continued to just before the `printf` statement at line 20.

```
(CXdb) info scope
Process [#0/0], frame 0 scope: c$ansi_block`ansi_block
(CXdb) print i
(int) 0
(CXdb) print ansi_block`1`2`i
(int) 12
(CXdb) print ansi_block`1`i
(int) 1
(CXdb) print ansi_block`1`1`i
(int) 11
```

The above example again print the values of the identifiers named `i`. The current scope is the main routine, `ansi_block`.

Related Commands

`evaluate`	`frame`
`info scope`	`print`

Related Concepts

displaying data	source units

search path

Description

The search path is a list of directories CXdb searches when it is looking for either source files or the compiler-generated CTI data files. Initially, the search path for each process object is set to the default search path. If an executable is specified for the process object, CXdb adds the directories stored in the executable to the search path. The executable has absolute paths to the directories where the source code is located.

NOTE: If you have compiled your source code using a version of the CONVEX Fortran compiler later than V7.0, or the CONVEX C compiler later than V4.3, you may not need to specify a search path. If the source code has not changed location since compilation time, CXdb can find the source and CTI files from the information in the executable. If the source code has changed location, or if different source files or CTI data files are desired, you may need to update the search path.

The following commands manipulate the search path of a process object:

- `add path` — Adds directories to the search path.
- `info path` — Displays the directories in the search path.
- `remove path` — Removes directories from the search path.
- `set path` — Sets the search path to the specified directories.

Examples

The following examples use the search path commands.

```
(CXdb) info path
Default search list:
        .

Process [#0] search list for: docexample
        .
```

The above example displays the default search path and the search path of the process object. The search path contains the current working directory, represented by a dot (.).

```
(CXdb) add path /mnt/jones/libraries , /mnt/jones/math/libraries
```

The above example adds two directories to the search path. The next time CXdb is searching for a source file it will use the new search path, which now includes the /mnt/jones/libraries and /mnt/jones/math/libraries directories.

```
(CXdb) remove path /mnt/jones/math/libraries
```

The above command removes the /mnt/jones/math/libraries directory from the search path. The other directories in the search path remain in the search path.

```
(CXdb) info path
Default search list:
        .

Process [#0] search list for: docexample
        .
        /mnt/jones/libraries
```

The above command displays the updated search path, after the `add path` and `remove path` commands.

```
(CXdb) set path /mnt/jones/project2 , /mnt/jones/project2/source
(CXdb) info path
Default search list:
        .

Process [#0] search list for: docexample
        /mnt/jones/project2
        /mnt/jones/project2/source
```

The above command removes all the existing directories from the search path and sets it to the two listed directories. The `info path` command displays the new search path for the process object.

Related Commands

`add default path`	`add path`
`attach`	`cd`
`core`	`cxdb`
`debug core`	`debug exec`
`debug proc`	`executable`
`info cxdb`	`info path`
`remove default path`	`remove path`
`set default path`	`set path`

Related Concepts

command files	Compiler-Tools Interface
console working directory	default search path
process object	process working directory
remote debugging	

Related Parameters

directory-specifier

Description

Signals sent to an executing process can be controlled through several CXdb commands. Eventpoints can be set to watch for a particular signal. For each different type of signal, you can set the actions to be taken when CXdb catches the signal. You can also specify which signals the process receives.

CXdb catches all signals sent to the process before the process ever receives them, unless the signal was sent by CXdb itself. When CXdb catches a signal, the signal number for that signal is stored in the debugger variable `$signal`.

If an eventpoint is triggered by a signal, the commands of the eventpoint handler are executed. If an eventpoint is not triggered, the actions CXdb takes after a signal is caught are as follows:

- **Stop** —Stop process execution. If `stop` is not set, process execution is not stopped.
- **Print** — Print a message telling which signal was caught. If `print` is not set, no message is printed.
- **Pass** — Pass the value of the predefined debugger variable `$signal` to the process when process execution resumes. By modifying the value stored in `$signal`, you can change which signal is passed to the process. If `pass` is not set, no signal is passed to the process.

The above actions are independent of one another, and can be set or cleared for each signal.

The following commands work with signals:

- `info signal` — Displays the current actions for a signal or all signals.
- `event signal` — Sets an eventpoint for a signal.
- `set signal` — Sets the actions for a signal.
- `signal process` — Sends a signal to the specified process.
- `signal thread` — Sends a signal to the specified thread.

Some of the signals are different on C Series machines and SPP Series machines. The following sections list the signals for each type of architecture.

C Series only

The signals for C Series machines are listed below. The signal number is shown in parentheses. Signal 0 is used to indicate that no signal should be sent to the process. Signal names are *not* case sensitive.

- `SIGHUP` (1) — Hangup
- `SIGINT` (2) — Interrupt
- `SIGQUIT` (3) — Quit
- `SIGILL` (4) — Illegal instruction
- `SIGTRAP` (5) — Trace/breakpoint trap
- `SIGIOT` (6) — IOT trap
- `SIGEMT` (7) — EMT trap
- `SIGFPE` (8) — Floating point exception
- `SIGKILL` (9) — Killed
- `SIGBUS` (10) — Bus error
- `SIGSEGV` (11) — Segmentation violation
- `SIGSYS` (12) — Bad system call
- `SIGPIPE` (13) — Broken pipe
- `SIGALRM` (14) — Alarm clock
- `SIGTERM` (15) — Terminated
- `SIGURG` (16) — Urgent I/O condition
- `SIGSTOP` (17) — Stopped
- `SIGTSTP` (18) — Stopped (terminal)
- `SIGCONT` (19) — Continued
- `SIGCHLD` (20) — Child exited
- `SIGTTIN` (21) — Stopped (tty input)
- `SIGTTOU` (22) — Stopped (tty output)
- `SIGIO` (23) — I/O possible
- `SIGXCPU` (24) — CPU time limit exceeded
- `SIGXFSZ` (25) — File size limit exceeded
- `SIGVTALRM` (26) — Virtual timer expired
- `SIGPROF` (27) — Profiling timer expired
- `SIGWINCH` (28) — Window size change
- `SIGLOST` (29) — Resource lost
- `SIGUSR1` (30) — User-defined signal 1
- `SIGUSR2` (31) — User-defined signal 2

SPP Series only

The signals for SPP Series machines are listed below. The signal number is shown in parentheses. Signal 0 is used to indicate that no signal should be sent to the process. Signal names are *not* case sensitive.

- SIGHUP (1) — Floating point exception
- SIGINT (2) — Interrupt
- SIGQUIT (3) — Quit
- SIGILL (4) — Illegal instruction (not reset when caught)
- SIGTRAP (5) — Trace trap (not reset when caught)
- SIGIOT (6) — Process abort signal
- SIGEMT (7) — EMT instruction
- SIGFPE (8) — Floating point exception
- SIGKILL (9) — Kill (cannot be caught or ignored)
- SIGBUS (10) — Bus error
- SIGSEGV (11) — Segmentation violation
- SIGSYS (12) — Bad argument to system call
- SIGPIPE (13) — Write on a pipe with no one to read it
- SIGALRM (14) — Alarm clock
- SIGTERM (15) — Software termination signal from kill
- SIGUSR1 (16) — User defined signal 1
- SIGUSR2 (17) — User defined signal 2
- SIGCHLD (18) — Child process terminated or stopped
- SIGPWR (19) — Power state indication
- SIGVTALRM (20) — Virtual timer alarm
- SIGPROF (21) — Profiling timer alarm
- SIGIO (22) — Asynchronous I/O
- SIGWINCH (23) — Window size change signal
- SIGSTOP (24) — Stop signal (cannot be caught or ignored)
- SIGTSTP (25) — Interactive stop signal
- SIGCONT (26) — Continue if stopped
- SIGTTIN (27) — Read from control terminal attempted by a member of a background process group
- SIGTTOU (28) — Write to control terminal attempted by a member of a background process group
- SIGURG (29) — Urgent condition on I/O channel
- SIGLOST (30) — Remove lock lost (NFS)
- SIGRESERVE (31) — Save for future use
- SIGDIL (32) — DIL signal

Examples

The following commands control the effect of the SIGINT signal on the process. For these examples, assume that the process is currently stopped.

```
(CXdb) info signal SIGINT

The current signal actions are:

Signal                          Signal
number   Stop   Pass   Print    name
------   ----   ----   -----    ----------------------
2         Yes    Yes    Yes     Interrupt
```

The above command displays the current settings for the actions to take when CXdb catches the SIGINT signal. Initially, CXdb will stop the process, pass the signal, and print a message when the signal is caught.

```
(CXdb) event signal SIGINT {eval $signal = 0; resume;}

#1: signal 2 on [#0], Enabled, ignore 0/0
  {
      eval $signal=0;
      resume;
  }
```

The above command sets an eventpoint watching for the SIGINT signal. The eventpoint handler sets the debugger variable $signal to zero and then resumes execution. Because the value of $signal is zero, when execution resumes the signal is not sent to the process. In effect, this handler causes the SIGINT signal to be ignored.

```
(CXdb) set signal INT nostop, print, nopass
```

The above command changes the default settings for the SIGINT signal. When CXdb catches the signal, a message is printed saying the signal has been caught. Because the stop action is not set, process execution continues. However, the signal is not passed to the process. This has the same effect as the previously set eventpoint; the SIGINT signal is ignored.

```
(CXdb) signal process 2
Resuming execution of Process [#0] with signal 2

process [#0] terminated with signal 2 (Interrupt)
```

The above command causes process execution to continue with the SIGINT signal (signal number 2) immediately being sent to the process. Because this signal is generated by CXdb, the signal is not caught by CXdb.

Related Commands

`info signal`
`set signal`
`signal thread`
`event signal`
`signal process`

Related Parameters

process-list
thread-list
signal-specifier

source units

Description

Source units are syntactic units of source code. There are five granularities of source units:

- Expression — Any valid combination of constants, operators, and operands.
- Statement — A combination of expressions that constitutes a complete instruction in the source language.
- Block — The statements that make up the body of a routine, a loop, or a conditional construct.
- Loop — A special type of statement that encloses a block.
- Routine — A main routine, subroutine, or function.

When you compile your source code with the `-cxdb` option, the compiler generates information about each of the source units in your code. It also assigns a unique identification number to each source unit. The compiler then stores this source unit information in Compiler-Tools Interface (CTI) data files that are associated with the executable file for your program. To list this information for all source units on a given line of source code, use the `info line` command. To list information for a given source unit, use the command `info sourceunit` *<source-unit>*, where *<source-unit>* is the unique ID number of a source unit.

Source units let you control the granularity used in analyzing your source code. You can specify source units for stepping as well as for setting breakpoints, tracepoints, and other eventpoints. Source units are essential for debugging optimized code, because optimized code often does not execute in the same sequence as the original source statements.

Examples

The following two examples present the same section of source code written in both Fortran and C. This will help you compare source units in the two languages.

First consider the following section of Fortran source code:

```
1       SUBROUTINE PRIME(N)
2       INTEGER A(1000)
3
4       DO J = 2, N
5         IF (A(J) .EQ. 1)
6           K = J*2
7           DO WHILE ((K .LE. N).AND.(K .LE. 1000))
8             A(K) = 0
9             K = K + J
10          ENDDO
11        ENDIF
12      ENDDO
13      RETURN
14      END
```

In the above code, there is only one routine source unit, and `PRIME` is the name of the routine represented by this source unit. This source unit consists of everything from the `SUBROUTINE` statement on line 1 up to and including the `END` statement on line 14.

There are two loop source units in the above example: a `DO` loop and a `DO WHILE` loop. They consist of the following:

- `DO` loop — Lines 4 through 12, inclusive.
- `DO WHILE` loop — Lines 7 through 10, inclusive.

There are four block source units in the above example. The first is a routine block, the second is a `DO` block, the third is an `IF` block, and the fourth is a `DO WHILE` block. These blocks consist of the following statements:

- Routine block — Lines 2 through 14, inclusive.
- `DO` block — Lines 5 through 11, inclusive.
- `IF` block — Lines 6 through 10, inclusive.
- `DO WHILE` block — Lines 8 and 9.

The statement source units in the above example are:

- Line 4 — `DO J = 2, N`
- Line 4 — `J = 2`
- Line 5 — `IF (A(J) .EQ. 1) THEN`
- Line 6 — `K = J*2`
- Line 7 — `DO WHILE((K .LE. N).AND.(K .LE. 1000))`
- Line 8 — `A(K) = 0`
- Line 9 — `K = K + J`
- Line 13 — `RETURN`
- Line 14 — `END`

Notice that a statement must contain some executable code in order to be counted as a source unit. Thus, statements such as `SUBROUTINE`, `INTEGER`, `ENDIF`, and `ENDDO` are not considered to be source units. Neither are comment lines or blank lines.

Also notice that the `DO` and `DO WHILE` statements are both statement source units and loop source units at the same time.

The expression source units in the above example are:

- Line 4:
  ```
  N
  2
  ```
- Line 5:
  ```
  A(J) .EQ. 1
  A(J)
  J
  1
  ```
- Line 6:
  ```
  J*2
  J
  2
  ```

- Line 7:

```
(K .LE. N).and.(K .LE. 1000)
(K .LE. 1000)
K .LE. 1000
(K .LE. N)
K .LE. N
1000
K (first occurrence)
N
K (second occurrence)
```

- Line 8:

```
K
0
```

- Line 9:

```
K + J
K
J
```

An expression can be as small as a single variable or as large as a complete statement. Notice that different appearances of the same expression count as different source units. For example, the variable `K` appears twice in line 7 above, so it counts as two different expression source units for that line. As with the other types of source units, the source language determines what constitutes a valid expression.

Consider the same section of code written in C:

```
1       prime(n)
2       int n;
3       {
4         int j,k;
5
6         for (j=2; j<=n; j++)
7         {
8           if (a[j]==1)
9           {
10            k = j*2;
11            while ((k<=n) && (k<=1000))
12            {
13              a[k] = 0;
14              k = k+j;
15            }
16          }
17        }
18      }
```

In the above example, there is one routine source unit. It consists of lines 3 through 18, inclusive.

There are two loop source units in the above example: a `for` loop and a `while` loop. They consist of the following:

- `for` loop — Lines 6 through 17, inclusive.
- `while` loop — Lines 11 through 15, inclusive.

There are four block source units in the above example. The first is a routine block, the second is a `for` block, the third is an `if` block, and the fourth is a `while` block. These blocks consist of the following statements:

- Routine block — Lines 3 through 18, inclusive.
- `for` block — Lines 7 through 17, inclusive.
- `if` block — Lines 9 through 16, inclusive.
- `while` block — Lines 12 through 15, inclusive.

The statement source units in the above example are:

- Line 6 — `for (j=2; j<=n; j++)`
- Line 8 — `if (a[j]==1)`
- Line 10 — `k = j*2`
- Line 11 — `while ((k<=n) && (k<=1000))`
- Line 13 — `a[k] = 0`
- Line 14 — `k = k+j`

The expression source units in the above example are:

- Line 6:
  `j<=n`
  `j++`
  `j=2`
  `j` (first occurrence)
  `2`
  `j` (second occurrence)
  `n`
  `j` (third occurrence)
- Line 8:
  `a[j]==1`
  `a[j]`
  `a`
  `j`
  `1`
- Line 10:
  `k = j*2`
  `j*2`
  `k`
  `2`
  `j`
- Line 11:
  `(k<=n) && (k<=1000)`
  `(k<=1000)`
  `k<=1000`
  `(k<=n)`
  `k<=n`
  `1000`
  `k` (first occurrence)
  `n`
  `k` (second occurrence)

- Line 13:
  ```
  a[k] = 0
  a[k]
  a
  k
  0
  ```
- Line 14:
  ```
  k = k+j
  k+j
  k (first occurrence)
  k (second occurrence)
  j
  ```

Related Commands

`break source`	`clear step`
`event reached source`	`info line`
`info sourceunit`	`next`
`next over`	`set default step`
`set step`	`step`
`step over`	`trace source`

Related Concepts

breakpoints	eventpoints
stepping	tracepoints

Related Parameters

granularity	source-unit

Related Windows

Source Code window

Description

Stepping is the incremental execution of a program. CXdb provides a sophisticated set of stepping commands that allows you to control not only the number of steps to execute but also the size of each step.

In the broadest sense, stepping can be done in one of two ways: by machine instructions or by source units. The commands for stepping by machine instruction are:

- `next instruction` — Execute the specified number of machine instructions. Do not count instructions within a called routine as part of the specified number.
- `step instruction` — Execute the specified number of machine instructions. Count instructions within a called routine as part of the specified number.

The commands for stepping by source units allow you to specify the source unit granularity, or step size, as well as the number of steps. These commands are:

- `next` — Continue executing the process until it reaches the next source unit of the specified granularity. Do not count the source units within a called routine.
- `step` — Continue executing the process until it reaches the next source unit of the specified granularity. Count the source units within a called routine.
- `finish` — Finish executing the innermost active source unit of the specified granularity. Stop execution at the next source unit of default granularity.
- `next over` — Complete execution of the current source unit of the specified granularity. Stop execution at the next source unit of default granularity. Do not count the source units within a called routine.
- `step over` — Complete execution of the current source unit of the specified granularity. Stop execution at the next source unit of default granularity. Count the source units within a called routine.

Step size

The source unit granularities are:

- Routine
- Loop
- Block
- Statement
- Expression

If you do not specify the granularity for a particular stepping command, CXdb uses the default granularity. Initially the default granularity is statement, but you can modify this with the following commands:

- `set default step` — Set the default granularity (or step size) for all new process objects.
- `set step` — Set the default granularity (or step size) for an existing process object.

As a general rule, you will probably want to use a larger stepping granularity when the point of execution is far away from a problem area of the program and a finer granularity as the point of execution approaches the problem area.

The *current source unit* starts at the address indicated by the current value of the program counter (PC). Several source units of different granularities might all start at the same location. Therefore, all of these source units can be current at the same time. That is why it is important for you to specify the granularity of the particular current source unit you want.

The *innermost active source* unit is the one of specified granularity whose address range (or extent) includes the current value of the PC. If you start at the location indicated by the current PC and look backward through the code, the innermost active source unit is the first one of the specified granularity you encounter. In other words, it is the innermost source unit of specified granularity that encloses or contains the location indicated by the current PC.

Some examples might help to clarify the difference between the current source unit and the innermost active one. Consider the following pseudocode:

```
start Loop 1
    Statement 5
    start Loop 2
        Statement 10
        Statement 11
    end Loop 2
    Statement 15
end Loop 1
```

If the PC is pointing to Statement 11 in the above pseudocode, the innermost active loop is Loop 2, and the innermost active block is the block that includes Statement 10 and Statement 11. However, the current source unit is Statement 11 because the PC is pointing to it. Even though Loop 2 is active, it is not current because the PC does not point directly at the beginning of Loop 2.

With the PC pointing at Statement 11, Loop 2 and its included block are considered active because they contain the location indicated by the current value of the PC. Loop 1 is also active, but it is not the *innermost* active loop in this case.

If the PC is pointing to Statement 5 in the above pseudocode, then Loop 1 is the innermost active loop and Statement 5 is the current source unit. If the PC is pointing directly at the start of Loop 2, then Loop 2 is both the current loop source unit and the innermost active loop source unit.

Order of execution

One final and most important point: stepping is a dynamic activity. It proceeds according to the order of execution of the object code, not according to the order of the source code. When CXdb is searching for the particular source unit you want to step to, it checks each source unit individually before executing it. Therefore, if conditional constructs in the source code cause execution to skip over the source unit you want, or if optimization has eliminated the desired source unit, then stepping cannot take you to that source unit.

Examples

The stepping examples shown below relate to the following Fortran source code, which has been compiled at optimization level -no:

```
1       SUBROUTINE CHAPTER5(ARRAY)
2       INTEGER ARRAY(4,4)
3
4       PRINT *, "SUBROUTINE CHAPTER5 STARTING"
5       DO I = 1, 10
6         PRINT 99, "I = ", I
7         CALL SUB5A(I)
8         PRINT *, "Subroutine SUB5A has returned."
9       ENDDO
10      PRINT *, "The loop for M is next."
11      DO J = 1, 4
12        DO M = 1, 4
13          ARRAY(J,M) = J**M
14          PRINT 99, "J = ", J, "M = ", M, "ARRAY(J,M) = ", ARRAY(J,M)
15        ENDDO
16      ENDDO
17   99 FORMAT (A,I2,3X,A,I2,5X,A,I4)
18      PRINT *, "SUBROUTINE CHAPTER5 FINISHING"
19      END
20
21      SUBROUTINE SUB5A(N)
22      PRINT 98, "Subroutine SUB5A has started. The value of N is ", N
23      DO K = 1, N
24        PRINT 98, "K = ", K
25        IF (K .LE. 5) THEN
26          DO L = 1, N
27            PRINT 98, "L = ", L
28          ENDDO
29          PRINT 98, "The loop for L is done, with L = ", L
30        ENDIF
31      ENDDO
32      PRINT 98, "Subroutine SUB5A is done. The value of K is ", K
33      RETURN
34   98 FORMAT (A,I2)
35      END
```

Assume that the default stepping granularity is statement. Also assume that the process is stopped, and the program counter (PC) is pointing to the beginning of line 4.

```
(CXdb) step
Stepping process [#0/*] by 1 statement
Process [#0/0] stopped stepping at [0x800017ec] CHAPTER5 in chapter5.f line 5
```

Because statement is the default granularity, the above command steps the current process by one statement. The PC now points to the beginning of line 5, which is the beginning of a `DO` loop.

```
(CXdb) step 2
Stepping process [#0/*] by 2 statements
Process [#0/0] stopped stepping at [0x80001836] CHAPTER5 in chapter5.f line 7
```

The above command steps the current process by two statements. The PC points to the beginning of line 7, which is a call to a subroutine.

```
(CXdb) next
Nexting process [#0/*] by 1 statement
Process [#0/0] stopped nexting at [0x80001846] CHAPTER5 in chapter5.f line 8
```

The above command again steps the process by one statement. However, before the command executed, the PC was at the beginning of line 7, which is a subroutine call. The `next` command ignores all source units in a called subroutine. Therefore, the process continues to execute until it reaches the next statement after returning from the subroutine. When the process stops, the PC is pointing to the beginning of line 8.

```
(CXdb) step loop 2
Stepping process [#0/*] by 2 loops
Process [#0/0] stopped stepping at [0x80001af8] SUB5A in chapter5.f line 26
```

The above command steps the process by two loops. Because the step command looks at source units in called routines, this command continues execution of the process until it reaches the second `DO` loop in subroutine `SUB5A`. When the process stops, the PC is pointing to the beginning of line 26.

```
(CXdb) step over loop
Stepping process [#0/*] by 1 statement  outside current loop
Process [#0/0] stopped stepping at [0x80001b76] SUB5A in chapter5.f line 29
```

The above command completes execution of the current loop that begins on line 26. Since the default granularity is statement, execution stops at the next statement after the loop. The PC now points to the beginning of line 29.

```
(CXdb) finish loop
Finishing innermost loop in Process [#0/*]
Process [#0/0] stopped nexting at [0x80001bd8] SUB5A in chapter5.f line 32
```

The above command completes execution of the innermost active loop. When the PC is pointing to line 29, the innermost active loop is the DO loop that begins on line 23. The above command completes execution of this entire loop and stops the process at the first default source unit (statement) after the loop. Therefore, when the process stops, the PC points to the beginning of line 32.

Related Commands

finish	info cxdb
info line	info process
info sourceunit	next
next instruction	next over
set default step	set step
step	step instruction
step over	

Related Concepts

process object

source units

Related Parameters

granularity

Related Windows

Source Code window

synthesized variables

Description

Synthesized variables are generated by the CONVEX Fortran or CONVEX C compiler at optimization level `-O1` or higher. Synthesized variables enhance the performance of a program in two major ways:

- By replacing a program variable with a more efficient construct. For example, a synthesized variable can be used as a pointer to a particular array element. This pointer can replace a loop induction variable that acts as an index to an array element.
- By providing runtime support for the program. For example, synthesized variables can be used to maintain register spill areas in memory.

To generate a synthesized variable, the compiler performs transformations based on mathematical equations. CXdb can solve these equations to determine the current value of the synthesized variable as well as the current value of the program variable that is replaced by the synthesized variable. The `info expression` command displays the equations used to derive the synthesized variables.

The `info expression` command also lists the reason for the use of each synthesized variable. The reasons are abbreviated to acronyms, which are defined as follows:

- `ALTE` — Alternate entry point of a loop that executes conditionally.
- `BITB` — Bit bucket storage.
- `BOOT` — Storage of a value removed from a scalar register.
- `BSS` — Starting address of BSS memory region.
- `CMIN` — Index used to find the minimum element of an array by pattern matching.
- `CREG` — Communication register storage.
- `CTMP` — Call temporary, used for temporary storage of arguments that are passed by value to a subroutine.
- `DATA` — Starting address of DATA memory region.
- `DEAD` — Loop induction variable whose current value is impossible to calculate. No synthesized variable is reported in this case.

- DEXP — Expression that has been distributed over several optimized loops.
- FLNK — Forward link to a constant value that is propagated to the distributends of a vectorized loop.
- INDV — Loop induction variable that has undergone strength reduction.
- ISTR — Inner strip counter of a strip-mined loop.
- MLXS — Subscript of a multidimensional array that has been transposed into a one-dimensional array during vectorization.
- OSTR — Outer strip counter of a strip-mined loop.
- PBKE — Loop-invariant expression.
- PBKU — Loop-invariant constant.
- REXP — Reduced subexpression that is hoisted out of a loop.
- SEXP — Subscript expansion that folds the subscripts of a multi-dimensional array into one subscript.
- SINK — Sink variable that replaces the original induction variable when the loop iteration count is too small to warrant strip mining.
- SPLL — Pointer to the spill area for scalar registers.
- STML — Strip mine length.
- TBSS — Starting address of TBSS memory region.
- TDATA — Starting address of TDATA memory region.
- TEXT — Starting address of TEXT memory region.
- TPTR — Temporary pointer, used for temporary storage of arguments that are passed by reference to a subroutine.
- TRIP — Trip count used to replace a more complex loop iteration quantity.
- UREX — Expression from an unrolled loop.
- URIV — Induction variable of an unrolled loop.
- UTRP — Trip counter of an unrolled loop.
- VBOT — Storage of a value removed from a vector register.
- VMSK — Vector mask storage.
- VSPL — Pointer to the spill area for vector registers.
- ZMSK — Zero mask storage.

You can use a synthesized variable in any *<language-expression>*, in the same way you would use a program variable. However, in most cases you will want to display only the current value of the synthesized variable by using the `print` command.

Examples

The following examples illustrate how to display and reference synthesized variables.

```
(CXdb) info expression I

object type: Fortran identifier
   location: <none>
       size: 4 bytes
       type: INTEGER*4
      value: 3
       used to create 2 synthesized variable(s):
         1. <INDV>     ?i5 = (-4+?i1)+(4*(I-1))
         2. <INDV>     ?i6 = ?i2+(4*(I-1))
```

The above command displays information about the program variable `I`. The response indicates that the current value of `I` is 3. This value is not stored (location = <none>) because the synthesized variables `?i5` and `?i6` replace the use of `I`. The reason for the replacement is `INDV`, which means the induction variable `I` has undergone strength reduction. There are two synthesized variables because `I` is used as an index to two different arrays. The equations used to generate the synthesized variables `?i5` and `?i6` are also shown.

In the source code, `I` is a loop induction variable that is used as an index to reference specific elements of an array. In the object file, `?i6` serves as a pointer to the array elements. The compiler replaces `I` with `?i6` because it is more efficient to increment the pointer than it is to increment `I` and recalculate the address of the desired array element on each iteration of the loop.

For purposes of the `info expression` command, CXdb calculates the current value of `I` by solving for it in the equation shown for `?i6`.

```
(CXdb) info expression \?i6

object type: Fortran identifier
   location: register a3
       size: 4 bytes
       type: INTEGER*4
      value: -2146955280
       Reason: Loop induction variable
       created from 1 equation(s):
         1. <INDV>  ?i2+(4*(I-1))
       2 liveness ranges:
              Start      End          Location
         1.  0x800042d8:0x800042dc - register a3
         2.  0x800042dc:0x800042fc - register a3
```

The above command displays information about the synthesized variable `?i6`. The response shows the equation that the compiler uses to generate `?i6`. It also shows the liveness ranges and corresponding storage locations for the variable. The reason for generating `?i6` is that it replaces a loop induction variable.

```
(CXdb) print/x \?i6
(INTEGER*4) 0x80080ff0
```

The above command prints the current value of the synthesized variable `?i6` in hexadecimal format. Because `?i6` is a pointer to an array in this case, the current value of `?i6` is the starting address of the next array element to be accessed.

Related Commands

`evaluate` `info expression` `print`

Related Concepts

debugger variables, language expressions

Related Parameters

language-expression, synthesized-variable

Description

A thread is a sequence of machine instructions that executes on a single processor (CPU). If your program is running on a machine that has only one CPU, the entire program executes sequentially as a single thread. However, most CONVEX machines have multiple CPUs. Programs running on these machines can contain some sections of code that execute as multiple threads running in parallel and other sections of code that execute sequentially as a single thread.

Multiple threads executing in parallel (simultaneously) can reduce the overall time to completion of your program. As one thread finishes executing all its instructions, it terminates by performing a join operation. When all the threads have joined, the process can continue to execute sequentially as a single thread.

If your CONVEX machine has multiple CPUs, there are several ways to create multiple threads in your program:

- Compile your source code with optimization option `-O3`. This generates `spawn` instructions in the assembly language code.
- Include compiler directives (such as `FORCE_PARALLEL`) in your source code to spawn threads explicitly in specified sections of code. When you compile with the `-O3` option, these directives generate `spawn` instructions in the assembly language code.
- Use assembly language instructions (such as `spawn` or `pfork`) in your source code to create threads at specific locations

The `spawn` instruction can generate as many threads as there are CPUs available on your machine. Each of these spawned threads executes the same instruction stream using the same data, so this is called *symmetric* parallel processing.

The `pfork` instruction generates only one additional thread, and this child thread can execute a different instruction stream using different data than the parent thread. This is known as *asymmetric* parallel processing.

NOTE: Differences in CPU availability and resource contentions can cause the threads to behave differently each time a multithreaded process is executed. This type of behavior is called *non-determinism*, and it is normal on machines with multiple CPUs.

The general steps for debugging a multithreaded process are:

1. If you are running your program on a C Series machine, enable fixed scheduling for the process.
2. Set eventpoints to catch the spawning and joining of threads.
3. Run the process.
4. Display the thread information.
5. Use CXdb commands on individual threads as needed.

These steps are described in more detail below.

Enable fixed scheduling (C Series only)

Fixed scheduling reserves all the CPUs of the machine for each timeslice that your process executes. Fixed scheduling does not guarantee that your process will generate multiple threads, but it does provide the most stable environment for multiple threads by minimizing non-determinism. Fixed scheduling is available on C Series machines only.

Before running your process with CXdb, use the `set fixed sched` command to enable fixed scheduling, as shown below.

```
(CXdb) set fixed sched
```

If you have not yet created a CXdb process object, you can use the `set default fixed sched` command to enable fixed scheduling for the entire CXdb session. Any processes you create after that will then have fixed scheduling enabled by default.

Set eventpoints for spawn and join

You can use the `event spawn` command to catch the spawning of a new thread and the `event join` command to detect when a thread joins. Whenever one of these eventpoints is triggered, it stops all threads of the process. The following example shows how to set these eventpoints.

```
(CXdb) event spawn
#0: spawn, on [#0], Enabled, ignore 0/0
(CXdb) event join
#1: join, on [#0], Enabled, ignore 0/0
```

The `event spawn` eventpoint detects execution of a `spawn` instruction in the assembly code, and `event join` detects a `join` instruction. If you have used a `pfork`, `cfork`, or `wfork` instruction in the assembly code to create or terminate a thread, then you can use the `break instruction` command to detect execution of those instructions.

Run the process

The `run` command starts execution of the process.

```
(CXdb) run
Starting process [#0]: para
Process [#0/1] thread spawned
Process [#0/0] stopped at [0x8000166e] SUB1 in para.f line 22
Process [#0/1] stopped by Eventpoint 0, at [0x8000162c] SUB1 in
      para.f line 21
```

When the process spawns a new thread (thread 1 in this case), it triggers eventpoint 0. The eventpoint, in turn, stops all threads of the process, as indicated by the response to the `run` command.

Display thread information

The `info threads` command displays information about all threads of the process, as illustrated in the following example.

```
(CXdb) info threads

Status of process [#0] threads:

  thread count: 2
active threads: 0,1
current thread: 1

     Thread 0: stopped at [0x8000166e] SUB1 in para.f line 22
               by general process stop

     Thread 1: stopped at [0x8000162c] SUB1 in para.f line 21
               by thread creation trap
```

In addition to listing the current state of each individual thread, the `info threads` command also displays the following summary information:

- **thread count**—The maximum number of threads that the process can have active at any one time, but not necessarily the number of threads that are currently active. The thread count is equal to the number of CPUs on your machine.

- **active threads**—The ID numbers of the threads that currently exist.
- **current thread**—The default thread used in CXdb commands when you do not specify a different thread. Generally, the current thread is the one that caused process execution to stop.

In the above example, there are two active threads, 0 and 1. The thread count is 2, so this is also the maximum number of threads that can exist for this process. Thread 1 is the current thread because it stopped the process by triggering the spawn eventpoint (thread creation trap) when the thread was spawned.

If you are using CXdb in X Windows mode, you can also display thread information by assigning specific threads to certain windows. For example, you can assign a Source Code window to track the execution of thread 0 only or a Assembly Code window to monitor only thread 1. The "Threads dialog" reference page describes how to assign threads to a window.

Use commands on individual threads

Many of the CXdb commands have a thread option that lets you apply the command only to specified threads, rather than to the entire process. For example, you can set a breakpoint on a particular thread or step the execution of a single thread without affecting the other threads of the process.

You can apply a command to particular threads by specifying a thread list in front of the command, as illustrated in the following example.

```
(CXdb) :t1 step block

Stepping process [#0/1] by 1 block
Process [#0/0] stopped at [0x8000166e] SUB1 in para.f line 22
Process [#0/1] stopped stepping at [0x80001654] SUB1 in para.f
       line 20
(CXdb) :t0 print J
(INTEGER*4) 497
(CXdb) :t1 print J
(INTEGER*4) 498
(CXdb) :t0,1 step block

Stepping process [#0/0,1] by 1 block
Process [#0/0] stopped at [0x8000166e] SUB1 in para.f line 22
Process [#0/1] stopped stepping at [0x8000166c] SUB1 in para.f
       line 22
(CXdb) :t0 print I
(INTEGER*4) 897
(CXdb) :t1 print I
(INTEGER*4) 1
```

The above example shows that the syntax for a thread list is `:t` followed by the thread ID numbers. The command `:t1 step block` steps thread 1 by one block, while `:t0,1 step block` steps both threads 0 and 1 by one block. The print commands show that the loop induction variables `I` and `J` have different values in the two threads. This is because the threads are at different points of execution when they stop.

Examples

The following example illustrates how to begin a typical debugging session for a multithreaded process on a C Series machine. The steps are the same on an SPP Series machine, except that you would skip the `set fixed sched` command.

```
(CXdb) debug exec para

Default source file: para.f
Default source language: Fortran

Process [#0] created
(CXdb) set fixed sched
(CXdb) event spawn
#0: spawn, on [#0], Enabled, ignore 0/0
(CXdb) event join
#1: join, on [#0], Enabled, ignore 0/0
(CXdb) run
Starting process [#0]: para
Process [#0/1] thread spawned
Process [#0/0] stopped at [0x8000166e] SUB1 in para.f line 22
Process [#0/1] stopped by Eventpoint 0, at [0x8000162c] SUB1 in
      para.f line 21
(CXdb) :t0,1 step block

Stepping process [#0/0,1] by 1 block
Process [#0/0] stopped at [0x8000166e] SUB1 in para.f line 22
Process [#0/1] stopped stepping at [0x8000166c] SUB1 in para.f
      line 22
(CXdb) :t0 print I
(INTEGER*4) 897
(CXdb) :t1 print I
(INTEGER*4) 1
(CXdb) info threads

Status of process [#0] threads:

  thread count: 2
active threads: 0,1
current thread: 1

     Thread 0: stopped at [0x8000166e] SUB1 in para.f line 22
               by general process stop

     Thread 1: stopped at [0x8000166c] SUB1 in para.f line 22
               by breakpoint
```

The major steps in the above example are:

1. `debug exec para` creates a process object using the image of the executable file para.
2. `set fixed sched` enables fixed scheduling (C Series only) for the process object created in step 1.
3. `event spawn` and `event joint` set eventpoints to trap the spawning and joining of threads.
4. `run` starts execution of the process. When the process spawns thread 1, the eventpoint from step 3 stops execution of all threads.
5. `:t0,1 step block` steps execution of threads 0 and 1 by one block.
6. `:t0 print I` prints the value of variable `I` for thread 0, and `:t1 print I` prints the value of variable `I` for thread 1.
7. `info threads` displays the current status of all threads.

Related Commands

`clear default fixed sched`
`clear fixed sched`
`event join`
`event spawn`
`info threads`
`set default fixed sched`
`set fixed sched`
`set threads`
`signal thread`

Related Concepts

process object

Related Parameters

thread-list

Related Windows

Assembly Code window
Source Code window
Thread Activity window
Threads dialog

tracepoints

Description

A tracepoint is a predefined eventpoint, or trap, that you place in your executable code. Tracepoints allow you to trace the execution of your process through key locations in your program. When process execution reaches the location of an enabled tracepoint, the set of actions associated with the tracepoint, called the eventpoint handler, is taken.

Each tracepoint has a unique object number. This object number is used in subsequent commands when you want to refer to this tracepoint. Optionally, you can specify a debugger variable to be assigned to the tracepoint. You can then use the debugger variable to refer to the tracepoint in subsequent commands.

All tracepoints have a default handler that prints a message telling you that the tracepoint has been reached and then resumes process execution. You may specify a different set of actions to take for a particular tracepoint, or change the setting of the default handler itself.

Tracepoints, like all eventpoints, can be enabled or disabled. When process execution reaches the address of an enabled tracepoint, the tracepoint is said to be reached. A disabled tracepoint is treated as if it does not exist, and therefore can never be reached unless it is enabled again. By disabling a tracepoint, you can prevent it from being reached without having to remove it completely.

Once a tracepoint is reached, one of two actions can occur. If the tracepoint has an ignore count, the counter is incremented by one and process execution continues. If the tracepoint does not have an ignore count, the eventpoint is said to be triggered. When a tracepoint is triggered, the commands in its eventpoint handler are executed. If the tracepoint does not have its own eventpoint handler, the default eventpoint handler for tracepoints is used.

Multiple eventpoints can exist at the same address. When process execution reaches an address with multiple eventpoints, the highest-numbered, enabled eventpoint is reached. If this eventpoint has an ignore count, the counter is updated, and the next highest-numbered, enabled eventpoint is reached. This process continues until either an eventpoint is triggered or there are no more eventpoints at the address.

The ignore count of an eventpoint is the number of times the eventpoint must be reached before being triggered. A counter keeps track of the number of times an eventpoint has been reached. When the counter matches the ignore count, the ignore count is reset to zero, and the *next* time the eventpoint is reached the eventpoint is triggered.

Tracepoints are specific to the existing process object. They can be set for specific threads of a process as well. Tracepoints can also be removed.

There are several different ways to set a tracepoint:

- `trace instruction` — Sets the tracepoint at the specified address.
- `trace line` — Sets the tracepoint at the starting address that maps to the specified line number of a source file.
- `trace routine` — Sets the tracepoint at the first executable source unit of the routine containing the specified address.
- `trace source` — Sets the tracepoint at the starting address of the specified source unit number of a source file.

For commands that accept an address, any valid language expression may be used to specify the address.

Several commands allow you to interact with existing tracepoints. These commands are described below:

- `disable event` — Disable the specified eventpoints.
- `disable eventtype` — Disable all eventpoints of the specified type.
- `enable event` — Enable the specified eventpoints.
- `enable eventtype` — Enable all eventpoints of the specified type.
- `info trace` — Display information about all existing tracepoints.
- `info event` — Display information about the specified eventpoints.
- `remove event` — Remove the specified eventpoints.
- `remove eventtype` — Remove all eventpoints of the specified type.
- `set ignore` — Set an ignore count for the specified eventpoints.
- `set handler` — Set a handler for the specified eventpoints.
- `set typehandler` — Set the default handler for all eventpoints of the specified type.

Examples

The following series of examples create and manipulate several different tracepoints in one process object.

```
(CXdb) trace line 10

#0: trace line, on [#0/*], Enabled, ignore 0/0
        [0x800053de] EXAMPLE in example.f line 10
```

The above command sets a tracepoint at line 10 in the current source file. The eventpoint number for this tracepoint is 0, and the address of the tracepoint is `800053de` in routine `EXAMPLE` at line 10 in the source file example.f.

```
(CXdb) trace routine EXAMPLE

#1: trace routine, on [#0/*], Enabled, ignore 0/0
        [0x800053b0] EXAMPLE in example.f line 7
```

The above command sets a tracepoint at the first executable source unit in the routine `EXAMPLE`. The first executable source unit of a routine is usually the first statement of a routine after local variables have been declared, unless there are local initializations.

```
(CXdb) trace instruction EXAMPLE

#2: trace instruction, on [#0/*], Enabled, ignore 0/0
        [0x800053a0] EXAMPLE in example.f line 1
```

The above command sets a tracepoint at the first address of `EXAMPLE`. The first address of a routine begins the preamble of the routine. (The preamble manages the stack for that routine.) This address is different than that of the previous example, because tracepoint 1 is at the first source unit of `EXAMPLE`.

```
(CXdb) trace source 21

#3: trace source, on [#0/*], Enabled, ignore 0/0
        [0x800053ee] EXAMPLE in example.f line 11
```

The above command sets a tracepoint at the starting address of source unit 21. The source unit numbers of a given line may be found using the `info line` command.

```
(CXdb) info trace
Event   Enabled Ignore  proc/td Address         Where
#0          y    0/0      0/*     [0x800053de] EXAMPLE in example.f line 10
#1          y    0/0      0/*     [0x800053b0] EXAMPLE in example.f line 7
#2          y    0/0      0/*     [0x800053a0] EXAMPLE in example.f line 1
#3          y    0/0      0/*     [0x800053ee] EXAMPLE in example.f line 11
```

The above command displays the status of all the existing tracepoints. All of the tracepoints are initially enabled and do not have an ignore count.

```
(CXdb) run
Process [#0/0] hit Tracepoint 2 at [0x800053a0] EXAMPLE in example.f line 1
Process [#0/0] hit Tracepoint 1 at [0x800053b0] EXAMPLE in example.f line 7
Process [#0/0] hit Tracepoint 0 at [0x800053de] EXAMPLE in example.f line 10
Process [#0/0] hit Tracepoint 3 at [0x800053ee] EXAMPLE in example.f line 11
   ...
```

The above example begins process execution without passing any arguments to the process. When execution reaches the address of `800053a0`, tracepoint 2 is reached because it is enabled. Because it does not have an ignore count, the tracepoint is triggered. Because the tracepoint did not have its own eventpoint handler, the default handler for tracepoints is executed, which prints a message and then resumes execution. Each of the tracepoints is triggered in turn. For this example, assume that the program runs to completion.

```
(CXdb) remove event 1,2
Eventpoint 1 removed
Eventpoint 2 removed
```

The above command removes eventpoints 1 and 2 from the process object. These eventpoint numbers will not be reused during this session of CXdb.

```
(CXdb) trace line 10 $Middle

#4: trace line, on [#0/*], Enabled, ignore 0/0
        [0x800053de] EXAMPLE in example.f line 10

INFO 104: Eventpoint 0 also has a breakpoint at address
      0x800053de.
```

The above command sets another tracepoint at line 10 of the current source file. The debugger variable `$Middle` is created and assigned to this eventpoint. CXdb informs you that two eventpoints now reside at the same address.

```
(CXdb) trace line 10 {echo "Tracepoint 5 reached" ;}

#5: trace line, on [#0/*], Enabled, ignore 0/0
        [0x800053de] EXAMPLE in example.f line 10
    {
        echo "Tracepoint 5 reached" ;
    }

INFO 104: Eventpoint 4 also has a breakpoint at address
      0x800053de.
```

The above command sets a third tracepoint at line 10. An eventpoint handler is given to this eventpoint. The handler prints a message but does *not* resume execution of the process.

```
(CXdb) run
Starting process [#0]: docexample
Tracepoint 5 reached
```

The above example restarts execution of the process. Tracepoint 5 is triggered because it is the highest-numbered, enabled eventpoint at that location (the other two tracepoints at that location are 0 and 4), and it does not have an ignore count. The eventpoint handler for tracepoint 5 is used instead of the default handler for tracepoints. The handler for tracepoint 5 prints a message but does not resume process execution.

```
(CXdb) disable event 5
Eventpoint 5 disabled
(CXdb) set ignore 2 4
Eventpoint 4 will be ignored 2 times
```

The above example does two things. First, tracepoint 5 is disabled. Second, tracepoint 4 is given an ignore count of 2.

```
(CXdb) info event *

#0: trace line, on [#0/*], Enabled, ignore 0/0
        [0x800053de] EXAMPLE in example.f line 10

#3: trace source, on [#0/*], Enabled, ignore 0/0
        [0x800053ee] EXAMPLE in example.f line 11

#4: trace line, on [#0/*], Enabled, ignore 0/2
        [0x800053de] EXAMPLE in example.f line 10

#5: trace line, on [#0/*], Disabled, ignore 0/0
        [0x800053de] EXAMPLE in example.f line 10
    {
        echo "Tracepoint 5 reached" ;
    }
```

The above command displays the status of all eventpoints. Using the `info event` command, you can display the eventpoint handler of an eventpoint. From the output of this command, you can also see that tracepoint 5 is disabled and that tracepoint 4 has an ignore count of 2. (Tracepoints 1 and 2 were removed in a previous example.)

```
(CXdb) run
Process [#0] is already running with pid 16139.
Terminate existing process and restart? y
Starting process [#0]: docexample
Process [#0/0] hit Tracepoint 0 at [0x800053de] EXAMPLE in example.f line 10
Process [#0/0] hit Tracepoint 3 at [0x800053ee] EXAMPLE in example.f line 11
    ...
```

In the above example, the `run` command causes CXdb to display a warning message indicating that the current process still exists. If you answer yes, CXdb kills the current process and begins execution of a new process. When process execution reaches address `800053de`, tracepoint 4 is reached because eventpoint 5 is disabled. Tracepoint 4 has an ignore count, so its counter is incremented by one. Because an eventpoint has still not been triggered, tracepoint 0 is reached. Because it is enabled and does not have an ignore count, tracepoint 0 is triggered and the default handler executes. Then process execution continues.

```
(CXdb) enable event 5
Eventpoint 5 enabled
```

The above command enables tracepoint 5. This causes tracepoint 5 to be triggered the next time address `800053de` is reached because it is the highest-numbered, enabled tracepoint.

```
(CXdb) set ignore 0 4
Eventpoint 4 will be ignored 0 times.
```

The above command resets the ignore count to 0 for tracepoint 4.

For more information about the use of eventpoint handlers, see the concepts page on eventpoint handlers.

Related Commands

`disable event`	`disable eventtype`
`enable event`	`enable eventtype`
`info event`	`info eventtype`
`info trace`	`remove event`
`remove eventtype`	`resume`
`set default handler`	`set ignore`
`set handler`	`set typehandler`
`trace instruction`	`trace line`
`trace routine`	`trace source`

Related Concepts

breakpoints	eventpoints
eventpoint handlers	watchpoints

Related Parameters

debugger-variable	event-handler
language-expression	line-specifier
process-list	thread-list

viewports

Description

Viewports are destinations for CXdb input, output, and error messages. A viewport can be either a file, the CXdb Command window (in X Windows mode only), or stderr and stdout (in line mode only).

There are three types of viewports, each of which is capable of receiving a different type of information. The types are:

- cmderr — CXdb error messages and informational messages generated in response to commands.
- cmdlog — User entries on the CXdb command line.
- cmdout — Normal CXdb output generated in response to commands.

CXdb maintains separate viewport lists for cmderr, cmdlog, and cmdout. Each viewport in the list receives a copy of the designated information for its given type. For example, all the viewports for cmdout receive a copy of the output simultaneously.

The commands for modifying the viewport lists are:

- `add cmderr` — Add file names to the current list of cmderr viewports.
- `add cmdlog` — Add file names to the current list of cmdlog viewports.
- `add cmdout` — Add file names to the current list of cmdout viewports.
- `remove cmderr` — Remove file names from the current list of cmderr viewports.
- `remove cmdlog` — Remove file names from the current list of cmdlog viewports.
- `remove cmdout` — Remove file names from the current list of cmdout viewports.
- `set cmderr` — Remove all file names from the current cmderr viewport list, and replace them with a new list of file names.
- `set cmdlog` — Remove all file names from the current cmdlog viewport list, and replace them with a new list of file names.
- `set cmdout` — Remove all file names from the current cmdout viewport list, and replace them with a new list of file names.

The default viewport for cmderr is the Command window (Window #1) in X Windows mode or stderr in line mode. The default viewport for cmdout is the Command window (Window #1) in X Windows mode or stdout in line mode. There is no default viewport for cmdlog because your command input is automatically echoed on the CXdb command line.

For logging, all of the viewports you specify will be files. If the specified file does not exist, CXdb creates it. If the specified file already exists, then you can use the following commands to control whether or not CXdb writes (either overwrites or appends) to the existing file:

- `clear noclobber` — Allow writing (either overwriting or appending) to existing files.
- `set noclobber` — Respond with an error message if the specified file already exists.

The default for noclobber is clear (off).

For cmderr and cmdout, you can override the viewport list and redirect the response of an individual command by using redirection operators with that command. Each redirection operator allows you to specify a viewport list that applies only to the command with which it appears.

For cmdlog, you can enable and disable the viewports with the commands `set logging` and `clear logging`, respectively. The default is logging disabled (clear).

The command `info cxdb` displays the current settings of cmderr, cmdlog, cmdout, log, and noclobber.

Examples

The following examples illustrate how to modify and use viewport lists. These examples were generated in X Windows mode.

```
(CXdb) add cmdout cxdb.info
New cmdout: Window #1, cxdb.info
```

The above command adds a file to the viewport list for cmdout. The response indicates that the new viewports for cmdout are Window #1 (the Command window) and the file cxdb.info.

```
(CXdb) add cmderr cxdb.info, cxdb.err
New cmderr: Window #1, cxdb.info, cxdb.err
```

The above command adds two new files to the viewport list for cmderr. The response indicates that the new viewports for cmderr are Window #1 (the Command window), the file cxdb.info, and the file cxdb.err.

```
(CXdb) remove cmderr cxdb.info
New cmderr: Window #1, cxdb.err
```

The above command removes a file from the viewport list for cmderr. The response indicates that the new viewports for cmderr are Window #1 (the Command window) and the file cxdb.err.

```
(CXdb) set cmdout output_data
New cmdout: Window #1, output_data
```

The above command deletes the current viewport list for cmdout and replaces it with the file name output_data. Notice that the Command window always remains on the viewport list, so there is no need to specify it again.

With each individual CXdb command, you can override the viewport lists by using redirection operators on the command line, as shown in the following example of the `info process` command:

```
(CXdb) info process > process_status
```

The above example redirects the output of the `info process` command to the file called process_status instead of to the viewports for cmdout. Only the output of this one command is redirected to process_status. The output of any other commands is not affected by the redirection in this case, and neither are the viewports for cmderr and cmdlog.

You can also keep a log of the commands you use during the debugging session, as shown in the following example:

```
(CXdb) add cmdlog debug_script
New cmdlog: debug_script
(CXdb) set logging
```

The above response indicates that the new viewport for cmdlog is the file debug_script. The `set logging` command enables logging to this file. This type of log file is a good way to keep track of your actions during a debugging session. If you ever need to retrace your steps, you can use the `source` command to execute this log file as a command file.

Notice that the above response does not list Window #1 (the Command window) as one of the viewports for cmdlog. This is because everything you type in the Command window is automatically echoed there. If you add Window #1 to cmdlog, then everything you type will appear twice in the Command window.

Related Commands

`add cmderr`	`add cmdlog`
`add cmdout`	`clear logging`
`clear noclobber`	`info cxdb`
`remove cmderr`	`remove cmdlog`
`remove cmdout`	`set cmderr`
`set cmdlog`	`set cmdout`
`set logging`	`set noclobber`

Related Concepts

cmderr	cmdlog
cmdout	logging
redirection	viewports

Related Parameters

redirection-operator	viewport

Related Windows

Command window

watchpoints

Description

A watchpoint is a predefined eventpoint, or trap, that you set to watch a region of process memory. When the value stored in the watched address region changes, process execution stops, and the set of actions associated with the watchpoint, called the eventpoint handler, is taken.

Each watchpoint has a unique object number. You can use this object number in subsequent commands if you want to refer to the watchpoint. Optionally, you can specify a debugger variable to be assigned to the watchpoint. You can then use the debugger variable to refer to the watchpoint in subsequent commands.

All watchpoints have a default handler that displays a message telling you that the watchpoint's region has been modified. You may specify a different set of actions to take for a particular watchpoint, or change the setting of the default handler itself.

Watchpoints, like all eventpoints, can be enabled or disabled. When there is a change in the address region watched by an enabled watchpoint, the watchpoint is said to be reached. A disabled watchpoint is treated as if it does not exist, and therefore can never be reached unless it is enabled again. The disabling of watchpoints allows you to prevent them from being reached without having to completely remove them from the process object.

Once a watchpoint is reached, one of two things may occur. If the watchpoint has an ignore count, the counter is incremented by one and process execution continues. If the watchpoint does not have an ignore count, then it is said to be triggered. When a watchpoint is triggered, the commands in its eventpoint handler are executed. If the watchpoint does not have its own eventpoint handler, the default eventpoint handler for watchpoints is used.

Multiple eventpoints can exist at the same address. When multiple eventpoints can be reached, the highest-numbered, enabled eventpoint is reached. If this eventpoint has an ignore count, the counter is updated and the next highest-numbered, enabled eventpoint is reached. This process continues until either an eventpoint is triggered or there are no more eventpoints to be reached.

The ignore count of an eventpoint is the number of times this eventpoint must be reached before being triggered. A counter keeps track of the number of times an eventpoint has been reached. When the counter matches the ignore count, the ignore count is reset to zero, and the *next* time the eventpoint is reached the eventpoint will be triggered.

Watchpoints are specific to the existing process object. They can be set for specific threads of a process as well. Watchpoints can also be removed.

The `watch` command creates a watchpoint. A process image must exist before you can create a watchpoint.

Several commands allow you to interact with existing watchpoints. These commands are described below:

- `disable event` — Disable the specified eventpoints.
- `disable eventtype` — Disable all eventpoints of the specified type.
- `enable event` — Enable the specified eventpoints.
- `enable eventtype` — Enable all eventpoints of the specified type.
- `info watch` — Display information about all existing watchpoints.
- `info event` — Display information about the specified eventpoints.
- `remove event` — Remove the specified eventpoints.
- `remove eventtype` — Remove all eventpoints of the specified type.
- `set ignore` — Set an ignore count for the specified eventpoints.
- `set handler` — Set a handler for the specified eventpoints.
- `set typehandler` — Set the default handler for all eventpoints of the specified type.

Examples

The following series of examples creates and manipulates several different watchpoints in one process object. The following Fortran routine is used throughout the examples:

```
1       SUBROUTINE CHAPTER7(ARRAY)
2       INTEGER ARRAY(4,4), TABLE(4,4)3
3
4       PRINT *, "SUBROUTINE CHAPTER7 STARTING"
5       DO I = 1, 4
6         DO J = 1, 4
7           TABLE(I,J) = ARRAY(I,J) + I*ISQR(J)
8         ENDDO
9       ENDDO
10      CALL BLD_MATRIX(4,4,5,TABLE)
11      PRINT *, "SUBROUTINE CHAPTER7 FINISHING"
12      RETURN
13      END
14
15
16      FUNCTION ISQR(N)
17      ISQR = N*N
18      RETURN
19      END
```

For the following examples, assume that process execution has stopped at the beginning of the routine.

```
(CXdb) watch loc(I)

#1: watch 0x80077098..0x8007709b, on [#0/0], Enabled, ignore 0/0
```

The above command creates a watchpoint that monitors any changes to the Fortran variable `I`. The response from CXdb shows the current settings for the created watchpoint. The output is the same as if an `info event` command had been issued for this eventpoint.

Because the variable is a four-byte integer, the address range is from `80077098` to `8007709b`. The eventpoint number is 1, it is currently enabled, and it does not have an ignore count.

```
(CXdb) info expression TABLE

object type: Fortran array
orientation: column
     bounds: INTEGER*4(1:4, 1:4)
  base type: INTEGER*4
  base size: 4 bytes
 total size: 64 bytes
  base addr: 0x80077058

(CXdb) watch '80077058'x :64

#2: watch 0x80077058..0x80077097, on [#0/0], Enabled, ignore 0/0
```

The above example shows another way to set a watchpoint. First the address of the array `TABLE` is obtained using the `info expression` command. Second, a watchpoint is created to watch the address range starting with the hexadecimal address of `80077058` and extending for 64 bytes. The syntax for specifying a hexadecimal address is Fortran specific.

```
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] memory region 0x80077098..0x8007709b modified
Process [#0/0] stopped by Watchpoint 1, at [0x800028b4] CHAPTER7 in chapter7F.f line 6
```

The above example resumes execution of the current process. The process is stopped when the address region watched by watchpoint 1 changes. The default handler for watchpoints, which displays the above messages, is executed. Process execution does not resume.

```
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] memory region 0x80077058..0x80077097 modified
Process [#0/0] stopped by Watchpoint 2, at [0x800028be] CHAPTER7 in chapter7F.f line 7

(CXdb) print TABLE
INTEGER*4(1:4, 1:4)
(1..4,1) :  2 0 0 0
(1..4,2) :  0 0 0 0
(1..4,3) :  0 0 0 0
(1..4,4) :  0 0 0 0

(CXdb) set ignore 1 2
Event 2 will be ignored 1 time
```

The above example does several things. First, process execution is continued. The process is stopped by watchpoint 2 because the contents of the array `TABLE` have changed. Second, the elements of `TABLE` are printed. Finally, watchpoint 2 is given an ignore count of 1. The next time the watchpoint is reached, it will not be triggered. When it is reached a second time, it is triggered.

```
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] memory region 0x80077058..0x80077097 modified
Process [#0/0] stopped by Watchpoint 2, at [0x800028be] CHAPTER7 in chapter7F.f line 7

(CXdb) print TABLE
INTEGER*4(1:4, 1:4)
(1..4,1) :  2 0 0 0
(1..4,2) :  5 0 0 0
(1..4,3) :  10 0 0 0
(1..4,4) :  0 0 0 0

(CXdb) disable event 1
Eventpoint 1 disabled
```

The above example resumes process execution. Watchpoint 2 eventually stops the process. The `print` command shows that the watchpoint was ignored once, but it triggered when the value of element `TABLE(1,3)` changed. Finally, watchpoint 1 is disabled. This watchpoint can no longer be reached.

```
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] memory region 0x80077058..0x80077097 modified
Process [#0/0] stopped by Watchpoint 2, at [0x800028b4] CHAPTER7 in chapter7F.f line 6
```

The above command resumes process execution. The process is stopped by watchpoint 2 when the value of the variable `TABLE(1,4)` changes.

```
(CXdb) info watch
Event   Enabled Ignore  proc/td         Region
#1          n   0/0     0/0     0x80077098      0x8007709b
#2          y   0/0     0/0     0x80077058      0x80077097
```

The above command displays information about all existing watchpoints. The eventpoint number, enabled status, ignore count, process and thread number, and region of memory being watched is displayed for each watchpoint.

```
(CXdb) enable event 1
Eventpoint 1 enabled
(CXdb) remove event 2
Eventpoint 2 removed
```

The above example performs two actions. First, watchpoint 1 is enabled again, so it can be reached. Second, watchpoint 2 is completely removed from the process. It can not be brought back.

```
(CXdb)  watch loc(TABLE) \; {print TABLE; disable event $self; resume;}

#3: watch 0x80077058..0x80077097, on [#0/0], Enabled, ignore 0/0
    {
        print TABLE;
        disable event $self;
        resume;
    }
```

The above example creates a new watchpoint that watches the array TABLE. In addition, an eventpoint handler is defined for this watchpoint. When the watchpoint is triggered, the elements of TABLE are printed, and then the watchpoint disables itself by using the predefined debugger variable $self. Finally, process execution is resumed. This allows the other watchpoint, watchpoint 1, to be triggered the next time through the loop.

```
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] memory region 0x80077098..0x8007709b modified
Process [#0/0] stopped by Watchpoint 1, at [0x800028be] CHAPTER7 in chapter7F.f line 7

(CXdb) continue
Resuming execution of Process [#0/*]
INTEGER*4(1:4, 1:4)
(1..4,1) :  2 4 0 0
(1..4,2) :  5 0 0 0
(1..4,3) :  10 0 0 0
(1..4,4) :  17 0 0 0

Eventpoint 3 disabled
Process [#0/0] memory region 0x80077098..0x8007709b modified
Process [#0/0] stopped by Watchpoint 1, at [0x800028b4] CHAPTER7 in chapter7F.f line 6
```

The above commands continues process execution. First, watchpoint 1 is triggered when I is incremented. Then the second continue command triggers watchpoint 3. The eventpoint handler for watchpoint 3 executes, causing the elements of TABLE to be printed, the watchpoint to be disabled, and process execution to resume.

The next time through the loop, watchpoint 1 is triggered again because it is now the highest-numbered, enabled eventpoint to be reached. Watchpoint 1 stops process execution.

Related Commands

```
disable event                disable eventtype
enable event                 enable eventtype
event modify                 info event
info eventtype               info watch
remove event                 remove eventtype
set default handler          set handler
set ignore                   set typehandler
watch
```

Related Concepts

breakpoints
eventpoints
process object
debugger variables
eventpoint handlers
tracepoints

Description

The default X resource settings (Xdefaults) for the X Windows interface of CXdb are specified in the file /usr/lib/X11/app-defaults/Cxdb.

To modify these resource settings, copy the default specifications into the file that contains your own X resource specifications (usually .Xdefaults or .Xresources). Then modify the specifications to suit your needs.

After modifying the resource specifications, you must enter the following command in your xterm window:

xrdb -merge *~/resource_file*

resource_file is the name of the file that contains your resource specifications (usually .Xdefaults or .Xresources).

NOTE: If any of these resource specifications conflict with the settings used by your window manager, the window manager may override your CXdb resource specifications.

The Xdefaults in the /usr/lib/X11/app-defaults/Cxdb file are organized into the following categories:

- Generic application resources
- Window geometry and sizing resources
- Auto update flags
- Resources specific to the Command window
- Keyboard translations

Generic application resources — These Xdefaults define basic properties for all CXdb windows:

```
Cxdb*keyboardFocusPolicy:          pointer
Cxdb*genericRows:                  24
Cxdb*genericColumns:               80
Cxdb.autoCreate:                   True
```

Specific geometry and size-related resources — These Xdefaults determine the size and placement of individual CXdb windows:

```
Cxdb.Command Window.geometry:              +0+0
Cxdb.Source Code Window.geometry:          +0-0
Cxdb.Stack Trace Window.geometry:          +0-0
Cxdb.Assembly Code Window.geometry:        -0+0
Cxdb.Memory Display Window.geometry:       -0+0
Cxdb.processWindowGeometry:                -0-0
```

```
Cxdb.Scalar Registers.geometry:          -0-0
Cxdb.Vector Registers.geometry:          -0-0
Cxdb.Communication Registers.geometry: -0-0
Cxdb.Space Registers.geometry:           -0-0
Cxdb.Floating Point Registers.geometry: -0-0
Cxdb.Control Registers.geometry:         -0-0
Cxdb.Help Window.geometry:                -0+0
Cxdb.Help Window.?.width:              700
Cxdb.Help Window.?.height:             400
Cxdb.Help Window*XcRichText.width:     620
Cxdb*commandRows:                      20
Cxdb*commandColumns:                   80
Cxdb*sourceRows:                       20
Cxdb*sourceColumns:                    80
Cxdb*stackRows:                        24
Cxdb*stackColumns:                     80
Cxdb*disassemblyRows:                  24
Cxdb*disassemblyColumns:               80
Cxdb*examineRows:                      24
Cxdb*examineColumns:                   80
```

Auto update flags — These Xdefaults determine whether or not the named window will update automatically during a debugging session.

```
Cxdb.sourceAutoUpdate:                 True
Cxdb.stackAutoUpdate:                  True
Cxdb.disassemblyAutoUpdate:            True
Cxdb.examineAutoUpdate:                True
Cxdb.pswAutoUpdate:                    True
Cxdb.scalarAutoUpdate:                 True
Cxdb.communicationAutoUpdate:          True
Cxdb.vectorAutoUpdate:                 True
```

Other Command window resources — These Xdefaults determine attributes of the Command window, such as the primary and secondary prompt strings, and whether or not the command buttons and the command menu bar are displayed. These resources are:

```
Cxdb*commandPrompt:                    (CXdb)
Cxdb*secondaryPrompt:                  (cxdb)
Cxdb*commandButtons:                   True
Cxdb*commandMenu:                      True
Cxdb*commandMenuImmediate:             False
Cxdb*maxCommandWindowLines:            1000
Cxdb*monitorCommandWindowLength:       False
```

Keyboard translations — These Xdefaults define keyboard functions you can use in various windows. The following translation tables apply to text fields in pop-up dialog boxes, text regions of windows other than the Command window or pop-up dialog boxes, the Command window, the Source Code window, the Assembly Code window, and the Stack Trace, respectively:

```
Cxdb*XmTextField*translations:  #override \
                !Ctrl<Key>a:    beginning-of-line() \n\
                !Ctrl<Key>b:    backward-character() \n\
                !Ctrl<Key>d:    delete-next-character() \n\
                !Ctrl<Key>e:    end-of-line() \n\
                !Ctrl<Key>f:    forward-character() \n\
                !Ctrl<Key>h:    delete-previous-character() \n\
                !Ctrl<Key>k:    delete-to-end-of-line() \n\
                !Meta<Key>b:    backward-word() \n\
                !Meta<Key>f:    forward-word()

Cxdb*XmText*translations:       #override \
                !Ctrl<Key>a:    beginning-of-line() \n\
                !Ctrl<Key>b:    backward-character() \n\
                !Ctrl<Key>d:    delete-next-character() \n\
                !Ctrl<Key>e:    end-of-line() \n\
                !Ctrl<Key>f:    forward-character() \n\
                !Ctrl<Key>h:    delete-previous-character() \n\
                !Ctrl<Key>k:    kill-to-end-of-line() \n\
                !Ctrl<Key>l:    redraw-display() \n\
                !Ctrl<Key>r:    redraw-display() \n\
                !Ctrl<Key>u:    beginning-of-line()kill-to-end-of-line()\n\
                !Ctrl<Key>v:    next-page() \n\
                !Ctrl<Key>y:    unkill() \n\
                !Meta<Key>b:    backward-word() \n\
                !Meta<Key>d:    kill-next-word() \n\
                !Meta<Key>f:    forward-word() \n\
                !Meta<Key>h:    kill-previous-word() \n\
                !Meta<Key>v:    previous-page()

Cxdb*CommandText*translations:  #override \
                !Meta<Key>osfDelete: kill-previous-word() \n\
                !<Key>osfDelete: delete-previous-character() \n\
                !<Key>Tab:      complete() \n\
                !Ctrl<Key>i:    complete() \n\
                !Ctrl<Key>n:    down-history() \n\
                !Ctrl<Key>p:    up-history() \n\
                !Ctrl<Key>a:    beginning-of-command-line() \n\
                !Ctrl<Key>b:    backward-character() \n\
                !Ctrl<Key>d:    delete-next-character() \n\
                !Ctrl<Key>e:    end-of-command-line() \n\
                !Ctrl<Key>f:    forward-character() \n\
                !Ctrl<Key>h:    delete-previous-character() \n\
                !Ctrl<Key>k:    kill-to-end-of-line() \n\
```

```
                !Ctrl<Key>l:    redraw-display() \n\
                !Ctrl<Key>r:    redraw-display() \n\
                !Ctrl<Key>u:    beginning-of-command-line()kill-to-end-of-line()
                !Ctrl<Key>v:    next-page() \n\
                !Ctrl<Key>y:    unkill() \n\
                !Meta<Key>b:    backward-word() \n\
                !Meta<Key>d:    kill-next-word() \n\
                !Meta<Key>f:    forward-word() \n\
                !Meta<Key>h:    kill-previous-word() \n\
                !Meta<Key>v:    previous-page()

Cxdb.Source Code Window*SourceForm*XmText*translations: #override \
                Ctrl<Key>v:             next-page() \n\
                Meta<Key>v:             previous-page() \n\
                <Key>c:                 submit-command(continue) \n\
                <Key>n:                 submit-command(next) \n\
                <Key>s:                 submit-command(step) \n\

Cxdb.Assembly Code Window*textareaSW*XmText*translations:      #override \
                <Key>c:                 submit-command(continue) \n\
                <Key>s:                 submit-command(step instruction) \n\
                <Key>n:                 submit-command(next instruction)

Cxdb.Stack Trace Window*XmText*translations:    #override \
                <Key>0:                 submit-command(frame 0) \n\
                <Key>1:                 submit-command(frame 1) \n\
                <Key>2:                 submit-command(frame 2) \n\
                <Key>3:                 submit-command(frame 3) \n\
                <Key>4:                 submit-command(frame 4) \n\
                <Key>5:                 submit-command(frame 5) \n\
                <Key>6:                 submit-command(frame 6) \n\
                <Key>7:                 submit-command(frame 7) \n\
                <Key>8:                 submit-command(frame 8) \n\
                <Key>9:                 submit-command(frame 9) \n\
                <Key>t:                 submit-command(frame 0) \n\
                <Key>u:                 submit-command(frame +1) \n\
                <Key>d:                 submit-command(frame -1)
```

Examples

The following example sets several CXdb Xdefaults.

Assume that the following lines have been added to the .Xdefaults file of your home directory.

```
Cxdb.Command Window.geometry:   700x900+0+0
Cxdb.examineRows:               60
Cxdb*fontList:                  fixed
Cxdb.disassemblyAutoUpdate:     False
```

Adding the above lines to your ~/.Xdefaults file sets the Command window geometry, sets the Memory Display window height to 60 rows, specifies a "fixed" character font for all CXdb windows, and disables automatic screen updating in the Assembly Code window.

Related Commands

`clear autocreate` `set autocreate`

Windows

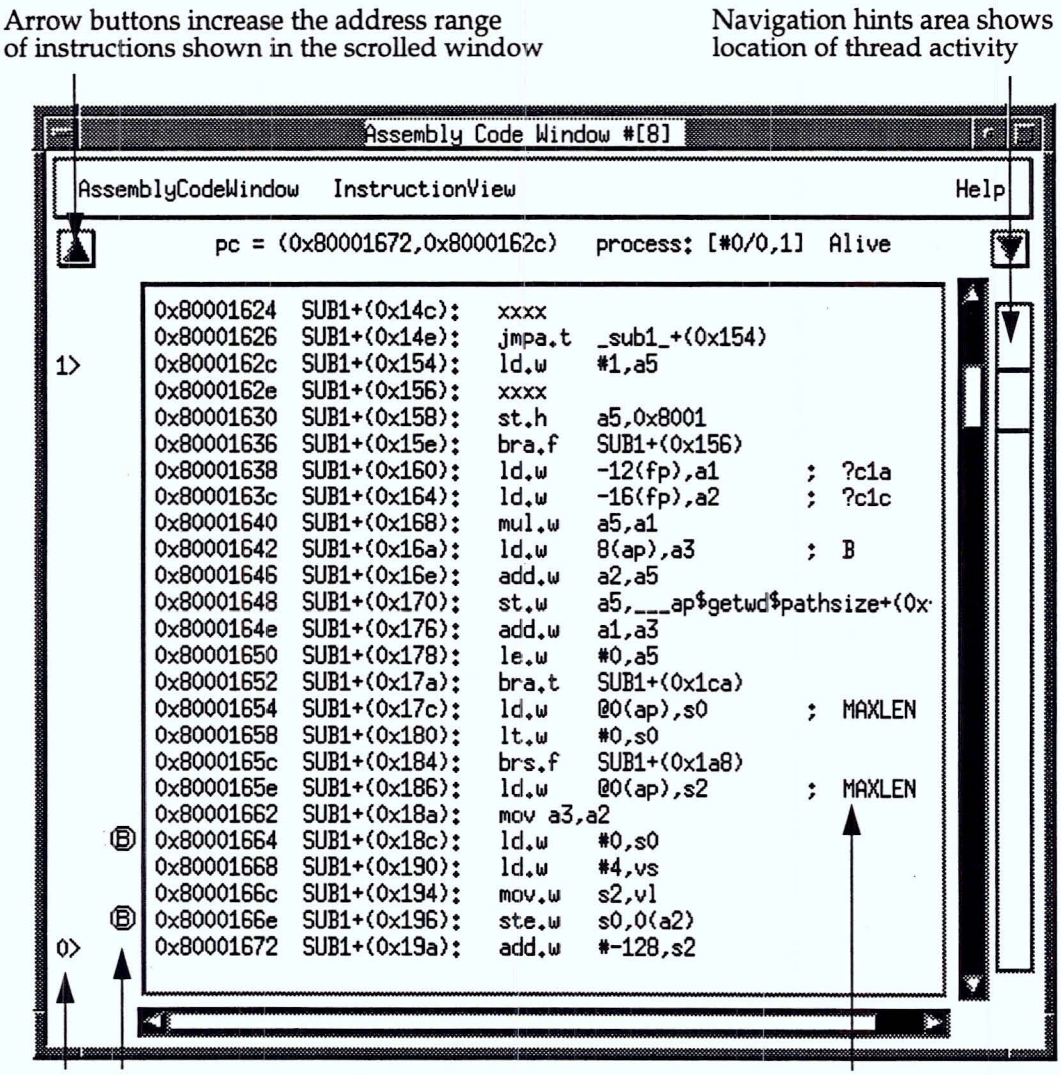
4

This chapter contains reference pages that explain how to use the CXdb windows. These windows are available only if you are running CXdb in X Windows mode.

There is a separate reference page for each window, dialog, and main menu. The reference pages can contain the following sections:

- **Description** — Text and figures explaining the purpose and operation of the window.
- **Menus** — Descriptions of the menus found in the window.
- **Context** — A description of how to open the window.
- **Related Commands** — A list of CXdb commands related to the window. The commands are described in Chapter 1.
- **Related Concepts** — A list of concepts related to the window. The concepts are described in Chapter 3.
- **Related Parameters** — A list of command parameters related to the window. CXdb parameters are described in Chapter 2.
- **Related Menus** — A list of menus that are related to the current window being described. The related menus are also described in this chapter.
- **Related Windows** — A list of other CXdb windows that are related to the current window being described. The related windows are also described in this chapter.

Assembly Code window



Description

The Assembly Code window displays assembly language instructions that are generated from your source code.

Assembly Code window

When the Assembly Code window is initially opened, it displays the machine instructions starting at the current program counter (pc). By default, information for all threads will be displayed. To control which threads are visible, refer to the section "Controlling thread visibility in the Assembly Code window."

The disassembled code output and markers shown in the Assembly Code window include the following:

- Hexadecimal address of each instruction
- Relative address based on the start of the routine
- Assembly language instructions
- Associated program or compiler-generated variable, if there is one
- Thread markers indicating the number and location of active threads within the address range shown in the window
- Eventpoint markers indicating the location of eventpoints with respect to machine instructions. Click with the left mouse button on these markers to view information about and manipulate eventpoints.
- Navigation hints bar indicating the current location of active threads

You can use the arrow buttons to increase the address range for the instructions shown in the scrolled window:

- Clicking the **UP** arrow button extends the range upwards.
- Clicking the **DOWN** arrow button extends the address range downwards.

Use the scroll bars to scroll the window vertically and horizontally with the mouse.

You can open any number of Assembly Code windows during a debugging session, and each can display a different section of assembly language code.

Use the Assembly Code window to display machine instructions in memory; to view the data portion of your program, use the Memory Display window.

Changing the area of memory to view

You can display assembly code that is not near the current point of execution using the following procedure:

1. Select the New address item from the InstructionView menu. This brings up the New Address dialog.
2. Enter a symbolic or absolute address in the Memory Address field. You can specify the address using any language expression in the syntax of the current source language that evaluates to a valid program address. For example, you can enter a routine name or an expression such as `LOC(ARRAY(1,4))` in Fortran or `&var1` in C.
3. Click OK. CXdb tries to locate the specified memory address and, if successful, closes the dialog. If not successful, error messages are displayed in the Command window.

To return the Assembly Code window display to the current point of execution, select the Point of execution item from the InstructionView menu.

Controlling thread visibility in the Assembly Code window

Each visible thread in the Assembly Code window is indicated by a > marker and is preceded by a thread number. This marker indicates the current point of execution for that thread and points to the next instruction to be executed if the process continues normally. When the Assembly Code window is first opened, all active threads are displayed.

You can associate the Assembly Code window with a specific thread or group of threads using the following procedure:

1. Select the threads item from the AssemblyCodeWindow menu. This opens a Threads dialog.
2. Toggle the selector buttons in the threads dialog to select and/or deselect visible threads. You can also click on All to select all threads or click on None to deselect all threads.
3. Click OK to close the dialog and apply the changes.

Creating eventpoints in the Assembly Code window

You can use the CXdb Command window break and trace buttons to set a breakpoint or tracepoint on a machine instruction displayed in the Assembly Code window. Use the following procedure:

1. In the CXdb Command window, click the left mouse button on the break or trace button. The `Composition selection: line number/instruction/source unit` prompt at the bottom of the Command window is highlighted.
2. Move the mouse cursor into the Assembly Code window. The mouse cursor changes to a pencil shape.
3. Position the mouse cursor over the machine instruction where you wish to create a breakpoint or tracepoint, and click the left mouse button.
4. The appropriate `break instruction` or `trace instruction` command is executed, and the breakpoint or tracepoint marker is displayed beside the machine instruction in the left margin.

Manipulating eventpoints in the Assembly Code window

Use the following procedure to disable, enable, or remove an eventpoint using the eventpoint markers in the Assembly Code window:

1. Position the mouse cursor over the eventpoint marker of the eventpoint you wish to manipulate, and click the left mouse button.
2. Click the left mouse button on the enable, disable, or remove button to perform the appropriate action.
3. Click OK to close the dialog.

Menus

Name	Description
AssemblyCodeWindow	Contains the following items:
	Close — Closes the window.
	Auto update — Enables or disables automatic updating of window contents.
	threads — Brings up a Threads dialog where you can associate the Assembly Code window with a particular thread.

InstructionView	Contains the following items: **Point of execution** — Returns the Assembly Code window display to the current point of execution. **New address** — Brings up a New Address dialog where you can specify a starting address for viewing a different area of memory.
Help	Contains items for invoking the CXdb Help system. Refer to the "Help menu" reference page for more information.

Context

The Assembly Code window appears when you:

- Select Create window, then select the Assembly Code item from the CXdbWindows menu in either the Command window or the Source Code window.
- Execute the `display disassembly` command from the `(CXdb)` prompt in the Command window.

Related Windows

Command window
Event Point dialog
Memory Display window
New Address dialog
Threads dialog

Related Menus

Help menu

Related Commands

`disassemble`
`display disassembly`

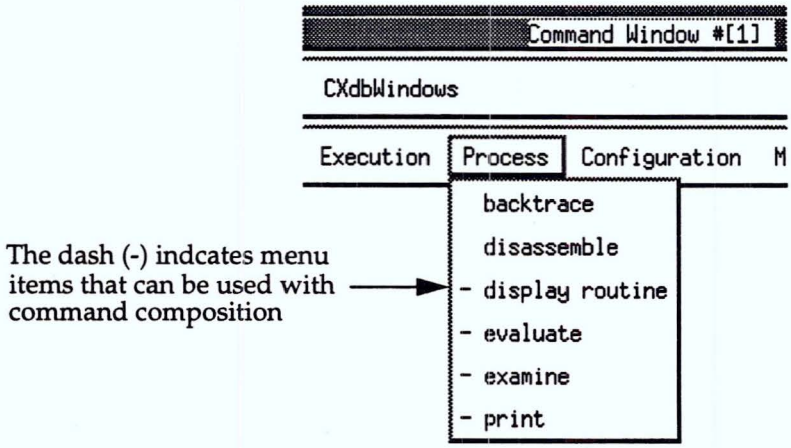
Description

You can use CXdb's *command composition* feature to compose and execute commands using the mouse with menus or the break and trace command buttons. With command composition, you do not have to manually type in commands and parameters.

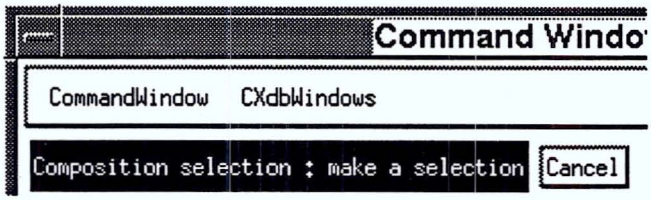
Using command composition with menus

Perform the following steps to use command composition with items on menus in the Command window:

1. Select a menu item with the mouse.



The cursor changes to a pencil shape, and CXdb displays a message similar to the following in the upper left corner of the Command window, just below the menubar. CXdb also automatically inserts the command you selected on the command line.

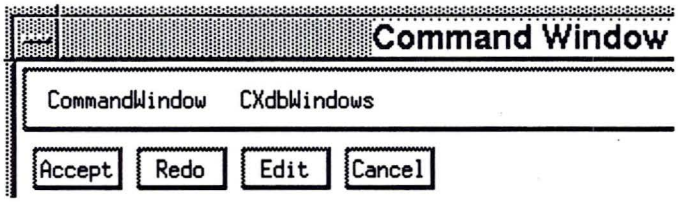


2. Use the mouse to select or highlight the appropriate item or text in the Command window, Assembly Code window, or Source Code

window, depending on what type of parameter you need to supply to the command.

For example, if you are composing a `display routine` command, you can highlight the text showing the routine name. If you are composing a `remove event` command, you can highlight the eventpoint number of the eventpoint you wanted to remove.

When you release the mouse, CXdb completes the command line with the selected item or text, then displays the following set of buttons:

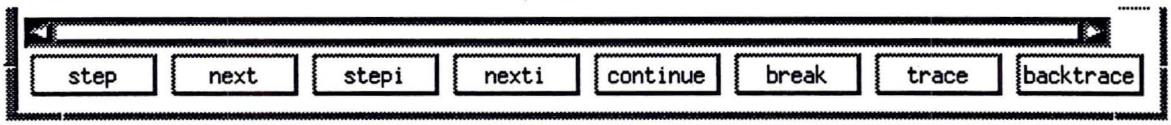


3. Select the appropriate action:
   - Press **RETURN** or click Accept to execute the command.
   - Click Redo to change your selection. CXdb erases the parameter supplied by the command composition, the cursor changes to a pencil shape, and you are prompted to make another selection.
   - Edit the command line manually.
   - Click Cancel to erase all of the text on the command line and remove the command composition prompt.

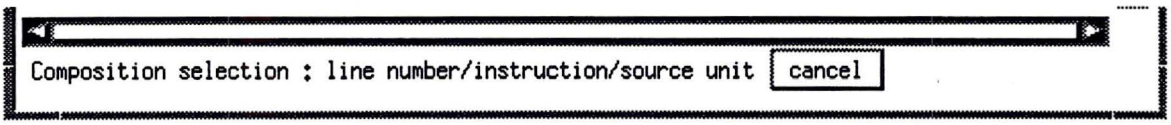
Using command composition with the break and trace buttons

Perform the following steps to use command composition with the break and trace buttons in the Command window:

1. Click the break or trace command button in the Command window.



The cursor changes to a pencil shape, CXdb inserts the word "break" or "trace" on the command line, and then replaces the row of command buttons with the following message:



2. At this point you can:
   - Use the mouse to highlight a line number, source unit, or instruction in the Source Code window or Assembly Code window.
   - Click cancel to cancel the operation.

   When you release the mouse button, CXdb completes the command line with the selected item or text and executes the command.

Related Windows

Assembly Code window
Command window
Source Code window

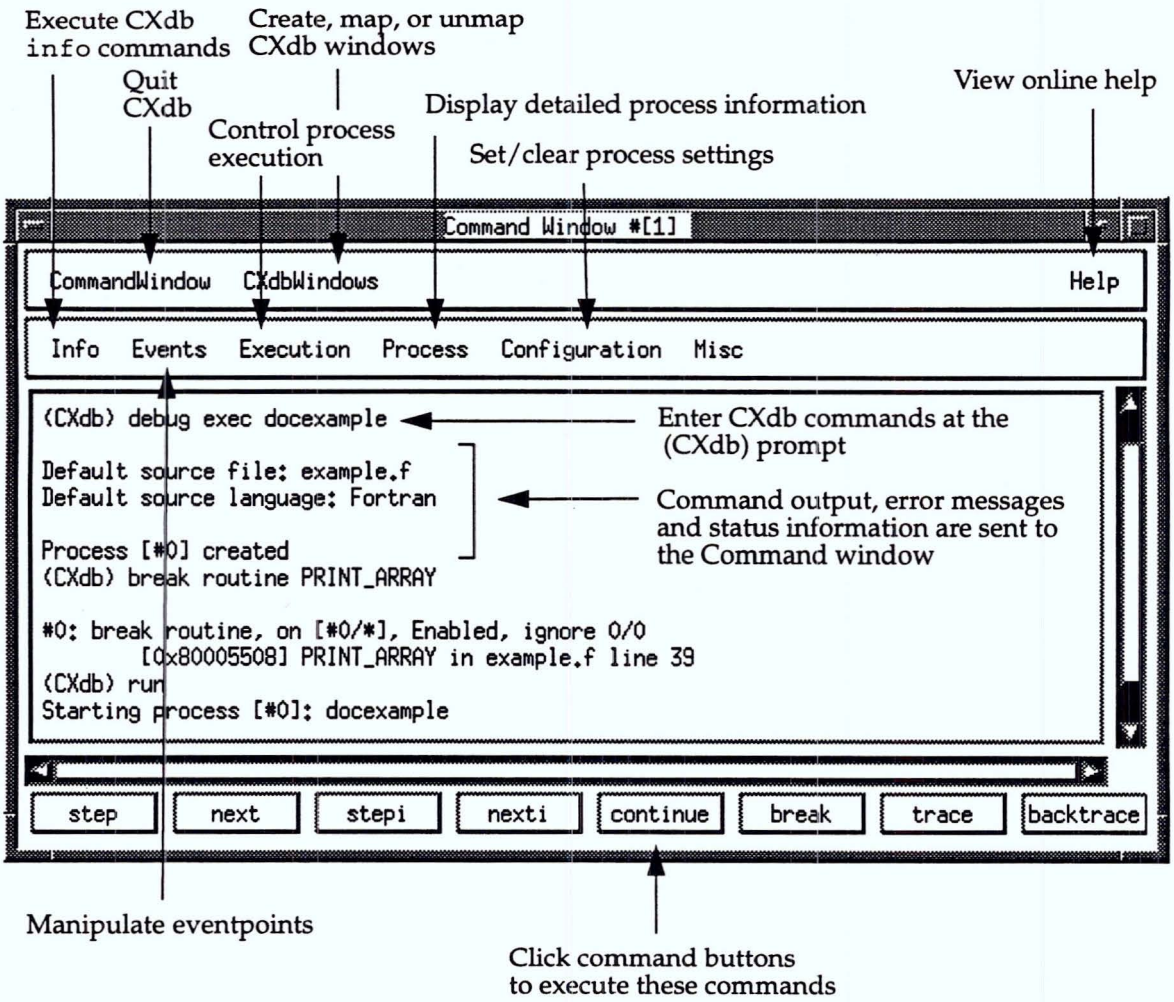
Related Menus

Events menu
Execution menu
Info menu
Process menu

Related Concepts

mouse and keyboard shortcuts

Command window



Description

The CXdb Command window is the primary way to communicate with CXdb. In the Command window, you can:

- Load a program or core file to debug using the `debug exec`, `debug proc`, `debug core`, or `executable` commands

- Execute CXdb commands by:
  - Entering them at the `(CXdb)` prompt
  - Clicking on command buttons
  - Selecting them from menus
- Receive output from CXdb commands
- Receive error messages or status information about CXdb commands
- Open and close other windows that display additional information about your process.
- Review previous commands and retrieve them from the command history
- Edit and repeat commands

Executing CXdb commands

You can execute CXdb commands by entering them on the command line, selecting them from menus, or clicking on command buttons:

- **Command line** — The `(CXdb)` prompt is where you enter commands manually.
- **Command menus** — The Info, Events, Execution, Process, Configuration, and Misc menus on the command menubar contain items for executing CXdb commands. Refer to the "Menus" section of this reference page for more information.
- **Command buttons** — These buttons immediately execute the command indicated by the button name. To activate a button, click on it with the mouse. Refer to the "Buttons" section of this reference topic for more information.

Several features of CXdb provide shortcuts for entering commands:

- **Command abbreviations** — You can abbreviate CXdb commands. For example, the command `info default environment` can be abbreviated to `in d e`. The shortest unique abbreviation for each command is shown in the upper right corner of the first reference page for each command, immediately below the command name.
- **Command completion** — This feature makes it easy to enter long variable names or command names. With completion, you can type only the first few letters of a command or variable name, then press **TAB**. CXdb fills in as much of the command as it can, up to the point where it determines the command is no longer unique.

  For example, `bre` **TAB** produces the completion "break." CXdb cannot complete the command any further, because the next word in the command could be "line," "instruction," "routine," or "source."

- **Command history** — CXdb stores the last 100 commands you entered in a history buffer. At the command line you can:
  - Press **CTRL-p** to go to the previous entry in the command history.
  - Press **CTRL-n** to go to the next entry in the command history.
  - Use the `recall` command to retrieve a command from the history list and execute it again.
  - Use the `info history` command to display the history list.
- **Command aliases** — Many CXdb commands have default aliases. You can also create custom aliases using the `alias` command. An alias can represent the first part of a command line, a complete command, or multiple commands. Use the `info alias` command to display current definitions of all aliases.
- **Command macros** — A macro can substitute for part of a command, a complete command, or multiple commands. Macros can accept parameters, and the parameters can have default values. Use the `macro` command to create a macro; use the `info macro` command to display macro definitions. For more information on creating CXdb command macros, refer to the reference page for the `macro` command.

Menus

Name	Items
CommandWindow	Contains items for quitting CXdb, enabling or disabling automatic creation of Source Code windows, and limiting the number of lines retained in the Command window text scrolling area.
CXdbWindows	Contains items for opening and closing other windows that display information about the current process.
Help	Contains items for invoking the CXdb Help system and for displaying the Product Information dialog.
Info	Contains items for executing CXdb `info` commands that display information about the current process, CXdb environment and default settings, specific eventpoints, eventpoint types, and registers.
Events	Contains items for executing CXdb commands that create, modify, disable, and clear handlers for specific eventpoints and eventpoint types.

Execution	Contains items for executing CXdb commands that control the execution of your program.
Process	Contains items for executing the `backtrace`, `disassemble`, `display routine`, `echo`, `evaluate`, `examine`, and `print` commands.
Configuration	Contains items for setting and clearing global (default) process settings and process settings for the current process.
Misc	Contains items for opening a Help window, opening an editor window, opening a shell window, displaying the name of the console working directory, quitting CXdb, and re-executing a command.

Buttons

Name	Action
step	Executes the `step` command, which continues execution of your process until it reaches the next source unit of the default granularity.
next	Executes the `next` command, which continues execution of your process until it reaches the next source unit of the default granularity, ignoring subroutine calls.
stepi	Executes the `step instruction` command, which steps your process by one machine instruction.
nexti	Executes the `next instruction` command, which continues execution of your process by one machine instruction, ignoring subroutine calls.
continue	Executes the `continue` command, which continues execution of a stopped process or stopped threads of a process until the process terminates or is stopped by an event.
break	Enables you to set a breakpoint in the Source Code window or Assembly Code window with the mouse.

When you click on the break button, the cursor shape changes to a pencil. You can then move the cursor into the Source Code window and click on a source unit or line number to set a breakpoint. CXdb executes the appropriate `break source` or `break line` command. You can also move the pencil cursor into the Assembly Code window to select a machine instruction. CXdb then executes the appropriate `break instruction` command.

trace	Allows you to set a tracepoint in the Source Code or Assembly Code window with the mouse. When you click on the break button, the cursor shape changes to a pencil. You can then move the cursor into the Source Code window and click on a source unit or line number to set a tracepoint. CXdb executes the appropriate `trace source` or `trace line` command. You can also move the pencil cursor into the Assembly Code window to select a machine instruction. CXdb then executes the appropriate `trace instruction` command.
backtrace	Executes the `backtrace` command, which displays summary information about the frames of the process stack.

Context

The CXdb Command window appears automatically when you invoke CXdb from the shell with the `cxdb` command, assuming your DISPLAY environment variable is set appropriately.

Related Menus

CommandWindow menu
Configuration menu
Execution menu
Info menu
Process menu
CXdbWindows menu
Events menu
Help menu
Misc menu

Related Windows

Assembly Code window
Processor Status Word window
Stack Trace window
Memory Display window
Source Code window

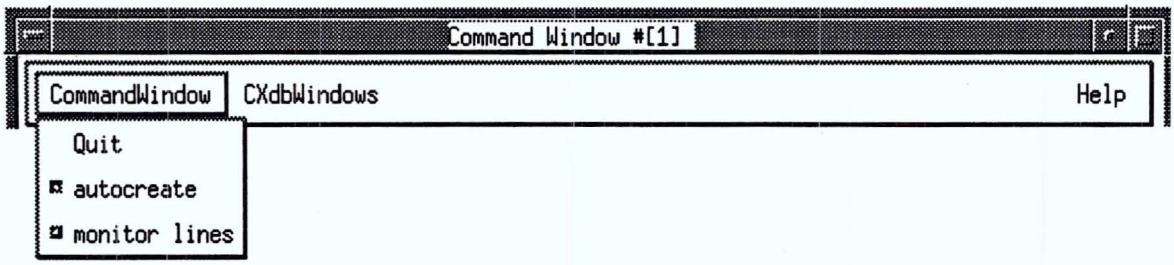
Related Concepts

command composition

Related Commands

`alias`
`backtrace`
`break line`
`break instruction`
`break source`
`continue`
`cxdb`
`info alias`
`info history`
`info macro`
`macro`
`next`
`next instruction`
`recall`
`set autocreate`
`step`
`step instruction`
`trace instruction`
`trace line`
`trace source`

CommandWindow menu



Description

The CommandWindow menu contains items for quitting CXdb, for controlling automatic creation of Source Code windows, and for limiting the number of lines retained in the Command window text scrolling area.

Menu Items

Item	Action
Quit	Quits CXdb and closes any windows that you have opened. If a process is still running when you quit, CXdb asks you to confirm whether you want to kill the process. The default is to kill the process, so if you press **RETURN** without responding, CXdb kills the process.
autocreate	This toggle button enables and disables automatic creation of Source Code windows when an executable file is specified. By default autocreation is enabled.
monitor lines	This toggle button enables and disables monitoring of the number of lines retained in the Command window text scrolling area. When monitor lines is enabled, the number of lines retained in the Command window's scrolling text area is limited. The number of lines retained is controlled by the X resource `CXdb*maxCommandWindowLines`. The default is 1000 lines.

You should enable monitor lines for long debugging sessions.

When monitor lines is disabled, all text displayed in the Command window is retained in the text scrolling area. You may notice performance degradation when the number of lines retained exceeds 1000.

Context

The CommandWindow menu appears when you select CommandWindow from the menubar in the CXdb Command window.

Related Windows

Command window

Related Concepts

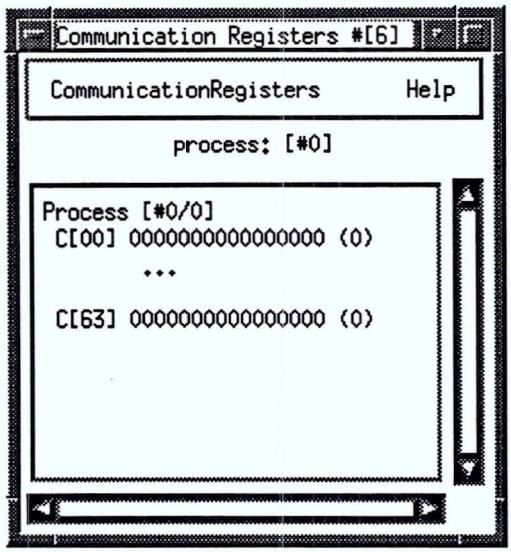
Xdefaults

Related Commands

`clear autocreate` `set autocreate` `quit`

C Series only

Communication Registers window



Description

The Communication Registers window displays the contents of the communication registers for the specified process. By default, the contents are displayed in hexadecimal format.

The value in parentheses at the end of each line is the lock bit for that register. Ellipsis (. . .) indicates that all of the intervening communication registers have the same contents.

CONVEX SPP Series systems do not use communication registers. Other CONVEX computer models use different configurations for the communication registers. For more information about these registers, refer to the "registers" reference topic and to the *CONVEX Architecture Reference Manual (C Series)*.

Communication Registers window

Menus

Name	Description
CommunicationRegisters	Contains the following items: **Close** — Closes the Communication Registers window. **Auto update** — Enables and disables automatic updating of the information displayed in the Communication Registers window. **Change format** — Opens a Format dialog where you can specify a different display format for the values displayed in the Communication Registers window. Refer to the "Format dialog" reference topic for more information.
Help	Contains items for invoking the CXdb Help system. Refer to the "Help menu" reference topic for more information.

Context

The Communication Registers window appears when you select Create window, then select the Communication Registers item from the CXdbWindows menu in either the Command Window or the Source Code window.

Related Windows

Format dialog

Related Menus

Help menu

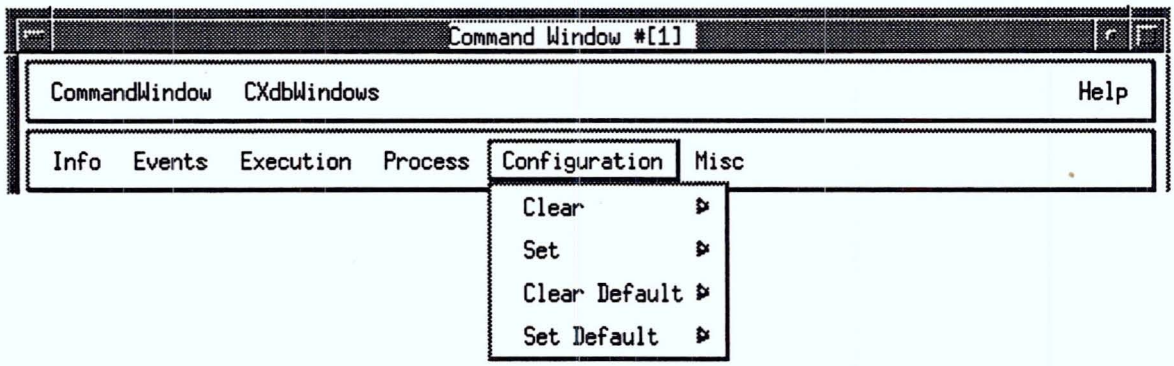
Related Commands

`info cregisters`

Related Concepts

architecture dependencies registers

Configuration menu



Description

The Configuration menu enables you to execute CXdb commands for setting and clearing process settings by selecting them from submenus rather than by entering them at the command prompt.

Menu items containing the word "default" affect default (global) process settings. Default settings are passed to new processes that do not have explicit settings. Menu items without the word "default" only affect process settings for the current process.

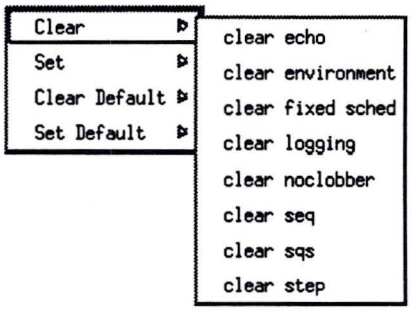
Note that changes made to default process settings have no effect on the setting of a process that has already been created, except in the case of the `clear step` command.

For more detailed information on a specific command, refer to the corresponding online help topic or reference page for that command.

Commands executed from the Configuration menu and submenus are executed using default parameters. To specify additional or different parameters, you must enter the commands from the keyboard.

Menu Items

Item	Action
Clear	Brings up a submenu with items for executing CXdb commands that remove or disable settings for the current process object:



clear echo — Disables echoing of input from initialization files and command files.

clear environment — Removes all environment variables from the process environment.

clear fixed sched (C Series only) — Disables fixed scheduling for the current process.

clear logging — Disables logging of CXdb input.

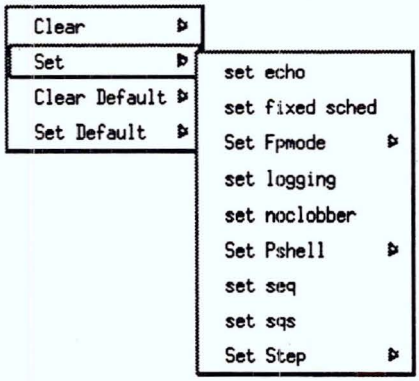
clear noclobber — Disables the noclobber option, allowing overwriting of existing log files.

clear seq (C Series only) — Clears the sequential mode (SEQ) bit of the processor status word (PSW) register.

clear sqs (C Series only) — Clears the sequential store enable (SQS) bit of the processor status word (PSW) register.

Set

Brings up a submenu with the following items for executing CXdb commands that enable settings for the current process object:



set echo — Enables echoing of input from command files.

set fixed sched (C Series only) — Enables fixed scheduling for the current process.

Set Fpmode (C Series only) — Brings up a submenu with items for setting the floating-point mode of the current process to either IEEE or native. On SPP Series systems, the only available floating-point mode is IEEE.

set logging — Enables logging of CXdb input.

set noclobber — Enables the noclobber option. When noclobber is enabled, CXdb responds with an error message if you attempt to overwrite an existing log file.

Set Pshell — Brings up a submenu with items for selecting either csh or sh as the current process shell.

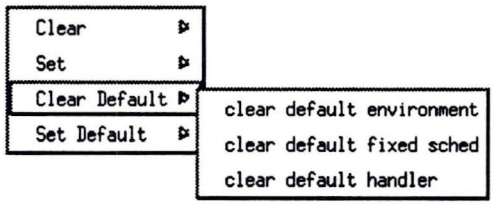
set seq (C Series only) — Sets the sequential mode (SEQ) bit of the processor status word (PSW) register.

set sqs (C Series only) — Sets the sequential store enable (SQS) bit of the process status word (PSW) register.

Set Step — Brings up a submenu with items for setting the step size for all threads of the current process. You can select a step size of routine, block, loop, statement, or expression.

Clear Default

Brings up a submenu with items for executing CXdb commands that disable or remove global process settings:



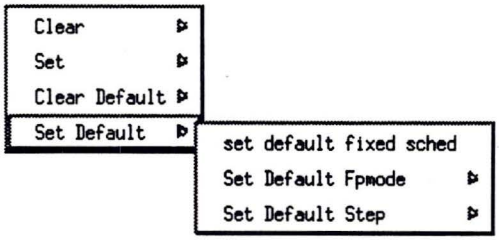
clear default environment — Removes all environment variables from the default environment.

clear default fixed sched (C Series only) — Disables fixed scheduling in the default process settings.

clear default handler — Removes the default handler for all eventpoints. The default handler returns to its initial setting, which displays a message.

Set Default (C Series only)

Brings up a submenu with the following items:



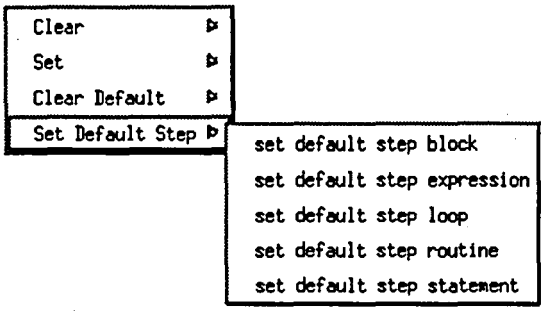
set default fixed sched — Enables fixed scheduling in the default process settings.

Set Default Fpmode — Brings up a submenu with items for setting the default mode for floating-point operations to IEE, native, or dual.

Set Default Step — Brings up a submenu with items for setting the default step size for all threads of the current process. You can select a step size of routine, block, loop, statement, or expression.

Set Default Step (SPP Series only)

Brings up a submenu with items for setting the default step size for all threads of the current process:



You can select a default step size of block, expression, loop, routine, or statement.

Context

The Configuration menu appears when you select Configuration from the command menubar in the CXdb Command window.

Related Windows

Command window

Related Commands

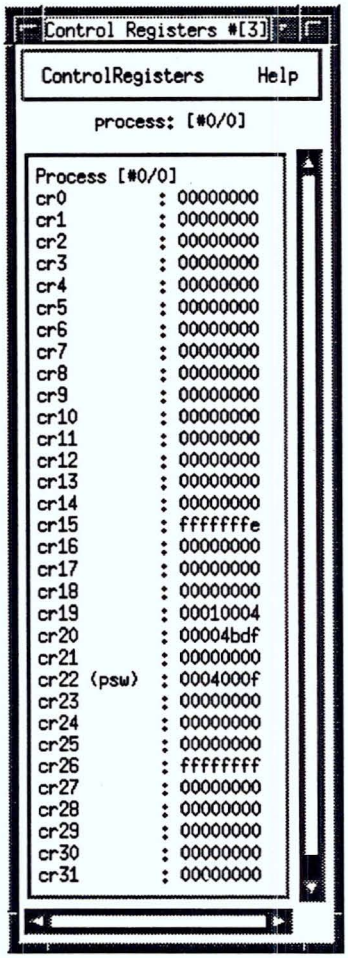
```
clear default environment
clear default fixed sched
clear default handler
clear echo
clear environment
clear fixed sched
clear logging
clear noclobber
clear seq
clear sqs
clear step
set default fixed sched
set default fpmode
set default step
set echo
set fixed sched
set fpmode
set logging
set noclobber
set pshell
set seq
set sqs
set step
```

Related Concepts

default environment
logging
process object
stepping

SPP Series only

Control Registers window



Description

The Control Registers window displays the contents of the control registers for the current thread of the specified process. By default, the display is in hexadecimal format. There are 32 control registers, designated as cr0 through cr31. Register cr22 is the processor status word (PSW).

You can specify a different display format for the window, enable and disable automatic updating, and specify which threads are visible in the window.

Control Registers window

For more information about these registers, refer to the *PA-RISC 1.1 Architecture and Instruction Set Reference Manual* and the *PA-RISC Procedure Calling Conventions Reference Manual*, both available from Hewlett-Packard.

Menus

Name	Description
ControlRegistersWindow	Contains the following items:
	Close — Closes the window.
	Auto update — Enables and disables automatic screen updating.
	Change format — Brings up a Format dialog where you can specify a different display format for the values displayed in the Control Registers window.
	threads — Brings up a Threads dialog where you can control which thread is visible in the Control Registers window.
Help	Contains items for invoking the CXdb Help system. Refer to the "Help menu" reference page or online help topic for more information.

Context

The Control Registers window (SPP Series systems only) appears when you select Create window, then select the Control Registers item from the CXdbWindows menu in either the Command window or the Source Code window.

Related Windows

Command window
Source Code window
Format dialog
Threads dialog

Related Menus

Help menu

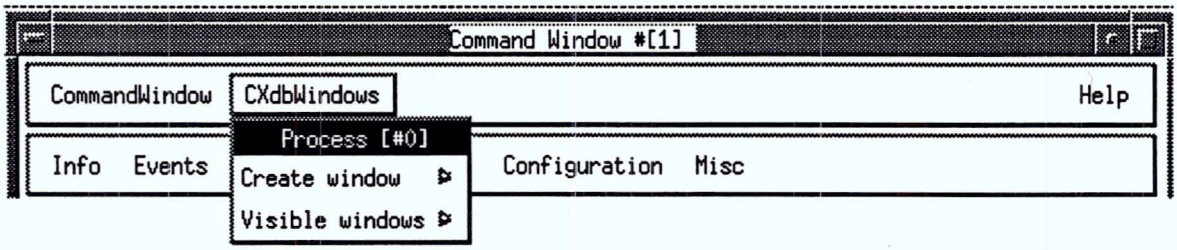
Related Commands

`info control registers`

Related Concepts

registers

CXdbWindows menu



Description

The CXdbWindows menu contains items for creating windows for viewing several types of information about the current process. You can show or hide individual windows, or all windows (except the CXdb Command window).

Using the Visible windows menu items to show or hide CXdb windows saves time spent in window creation and makes it easier to manage or conserve space on your display screen.

You can also use the Visible windows submenu to quickly view a list of open CXdb windows that indicates their type, window number, and whether they are currently visible (mapped) on your display.

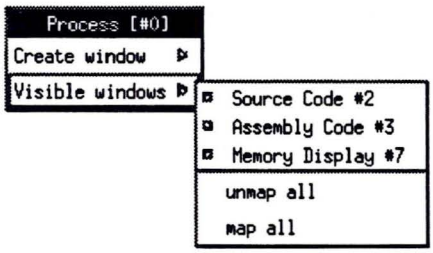
Menu Items

Item	Action
Create window	Brings up a submenu with items for creating CXdb windows. Types of windows you can create are Source Code, Stack Trace, Assembly Code, Memory Display, Processor Status Word, Thread Activity window, and register windows. The types of register windows you can create vary between C Series and SPP Series systems. On C Series systems, the register windows you can create are Scalar Registers, Vector Registers, Processor Status Word, and Communication Registers.

On SPP Series systems, the register windows you can create are General Registers, Floating Point registers, Control Registers, Space Registers, and Processor Status Word.

Visible windows

Brings up a submenu with toggle buttons and items that enable you to control which CXdb windows are visible on your display:



Toggle buttons — Enable you to show or hide individual CXdb windows.

Filled-in toggle buttons indicate windows that are currently visible (mapped) on your display.

The name and number of each open CXdb window are listed to the right of the toggle buttons.

unmap all — Hides all open CXdb windows (except the Command window).

map all — Makes all open CXdb windows visible.

Context

The CXdbWindows menu appears when you select CXdbWindows from the menubar in the CXdb Command window or the Source Code window.

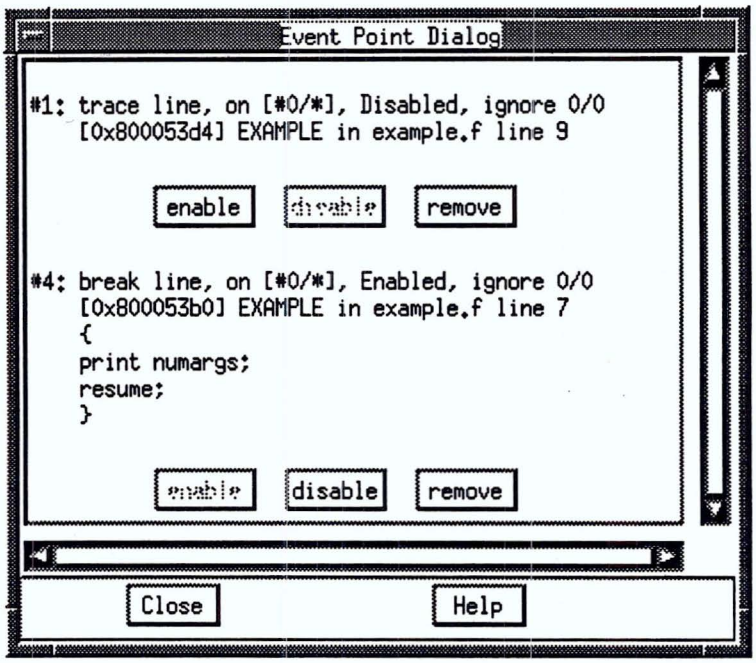
Related Windows

Assembly Code window
Command window
Communication Registers window
Control Registers window
Floating Point Registers window
General Registers window
Memory Display window
Processor Status Word window
Scalar Registers window
Source Code window
Space Registers window
Stack Trace window
Thread Activity window
Vector Registers window

Related Commands

```
display disassembly
display examine
display routine
display source
display stack
```

Event Point dialog



Description

With the Event Point dialog, you can enable, disable, or remove selected eventpoints using the mouse in the Source Code or Assembly Code windows.

The information displayed about each eventpoint includes the eventpoint number, eventpoint type, process number, thread number, enabled state, ignore count setting, hexadecimal address where the eventpoint is located, and the corresponding source code location. If an eventpoint handler has been defined, it is also shown.

To open an Event Point dialog, click on the eventpoint marker of the eventpoint you wish to manipulate:

- In the Source Code window, the Event Point dialog that appears will include a scrollable list of all eventpoints that are located within the current source unit.

Event Point dialog

- In the Assembly Code window, the Event Point dialog that appears only lists eventpoints located at the specified address. Multiple eventpoints are only listed if they are set at the same address.

Choices that are not available are stippled out.

Buttons

Name	Action
enable	Executes the `enable event` command on the indicated eventpoint. An eventpoint that is enabled is triggered when it is reached, unless it has an ignore count.
disable	Executes the `disable event` command on the indicated eventpoint. The eventpoint is not triggered when it is reached. It remains disabled until it is enabled or removed from the process object.
remove	Executes the `remove event` command on the indicated eventpoint. Removed events cannot be restored, and can no longer be referenced in CXdb commands.
Close	Closes the Event Point dialog.
Help	Brings up the CXdb Help window and displays the help topic for the Event Point dialog.

Context

The Event Point dialog appears when you click the left mouse button on an eventpoint marker in either the Source Code window or the Assembly Code window.

Related Windows

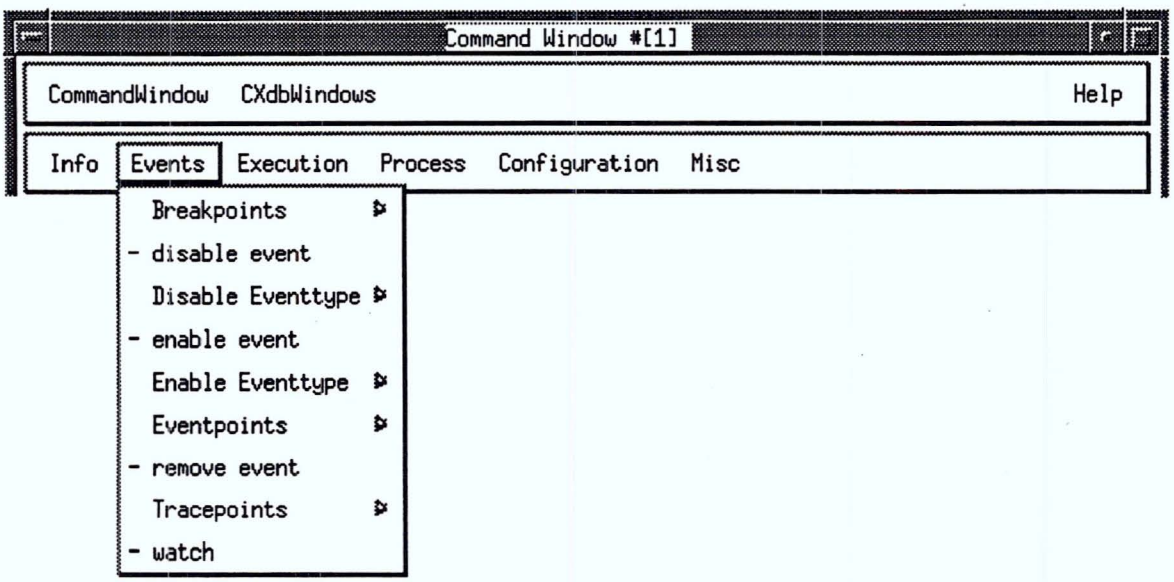
Assembly Code window

Source Code window

Related Commands

```
disable event
enable event
info event
remove eventtype
set ignore
disable eventtype
enable eventtype
remove event
set handler
```

Events menu



Description

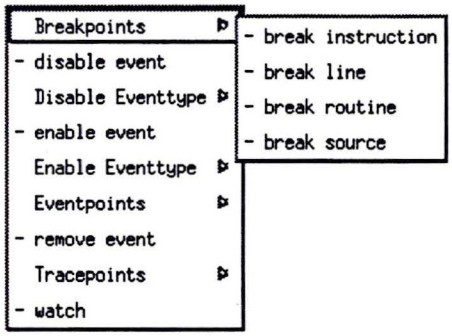
The Events menu enables you to execute CXdb commands for creating and manipulating eventpoints and for clearing eventpoint handlers by selecting commands from submenus rather than by entering them at the command prompt. For more detailed information on a specific command, refer to the reference page for that command.

Items on the Events menu and submenus preceded by a dash (–) require you to supply an additional parameter. You can do this using one of two methods:

- By typing it on the command line, then pressing **RETURN**
- By using command composition (refer to the "command composition" reference topic for more information)

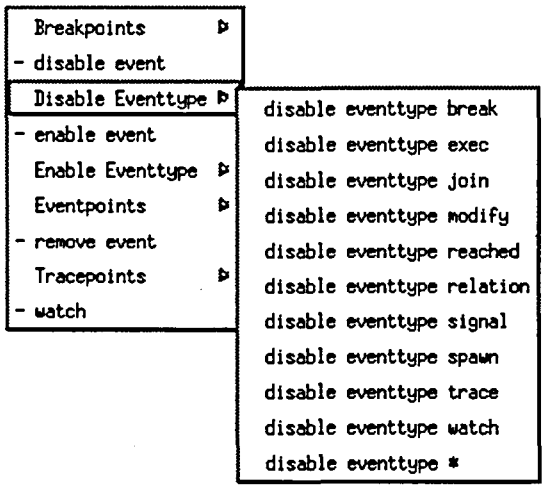
Items on submenus not preceded with a dash (–) are executed using default parameters. To specify additional or different parameters, you must enter the commands from the keyboard.

Menu items

Name	Action
Breakpoints	Brings up the following submenu:  You can create a breakpoint at an instruction, a line, a routine, or a source unit. The dash (–) preceding each of these menu items indicates that you must specify an additional parameter, either on the command line or using command composition. For example, if you select – break line, you must enter a *<line-specifier>* such as `example.f:7` or click on a line number in the Source Code window. For information about parameters, refer to the reference page for the corresponding `break` command.
– disable event	Executes the `disable event` command. You must provide the *<event-specifier>* parameter. You can do this by clicking on an eventpoint marker in the Source Code or Assembly Code window, entering an eventpoint number at the command line, entering a CXdb debugger variable that you have assigned to an eventpoint, or entering `*` to disable all eventpoints in the current process.

Disable Eventtype

Brings up a submenu containing items for executing CXdb commands for disabling all eventpoints of a selected type:



Eventpoint types you can select are break, exec, join, modify, reached, relation, signal, spawn, trace, and watch.

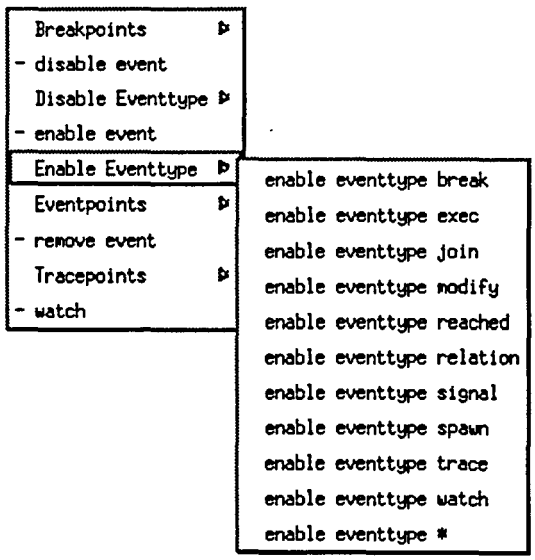
Selecting disable eventtype * disables all eventpoints for the current process.

– enable event

Executes the `enable event` command. You must provide the *<event-specifier>* parameter. You can do this by clicking on an eventpoint marker in the Source Code or Assembly Code window, entering an eventpoint number on the command line, entering a CXdb debugger variable that you have assigned to an eventpoint, or entering * to enable all eventpoints in the current process.

Enable Eventtype

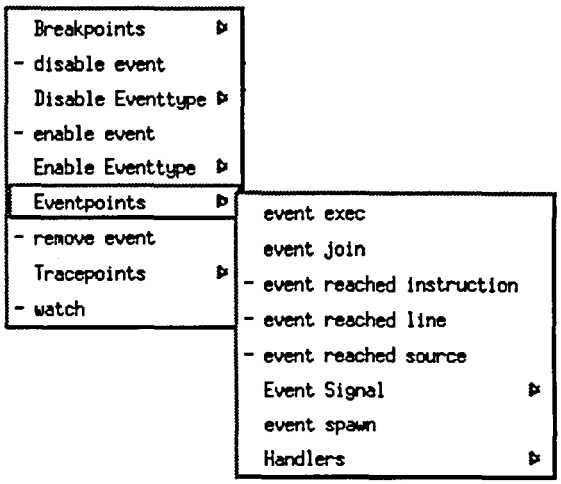
Brings up a submenu containing items for executing CXdb commands for enabling all eventpoints of a selected type.



Eventpoint types you can select are break, exec, join, modify, reached, relation, signal, spawn, trace, and watch. Selecting enable eventtype * disables all eventpoints for the current process.

Eventpoints

Brings up the following submenu:



event exec (C Series only) — Executes the `event exec` command to watch for an `exec(2)` system call by the process.

event join — Sets an eventpoint to trap a thread joining.

– event reached instruction (C Series only) — Sets an eventpoint at an instruction. You must supply a *<language-expression>* parameter that evaluates to a valid instruction address. You can enter the parameter on the command line or use command composition.

– event reached line (C Series only) — Sets an eventpoint at a line. You must supply a *<line-specifier>* parameter, which can be an integer or a file name and a line number in the format *<file-name>* **:** *<integer>*. You can enter the parameter on the command line or use command composition.

– event reached source (C Series only) — Sets an eventpoint at a source unit. You must supply a *<source-unit>* parameter, either by entering the source unit number on the command line or by clicking on a source unit with the mouse in the Source Code window.

Event Signal — Brings up a submenu with items for executing `event signal` commands. This enables you to create eventpoints to trap specific signals. Signals on this menu are listed by their full name. The signals listed differ between C Series systems and SPP Series systems. Refer to the "signals" reference topic for more information.

event spawn — Sets an eventpoint to trap a thread spawning.

Handlers — Contains menu items for restoring the default handler for all eventpoints to its original setting (clear default handler), clearing handlers defined for specific types of eventpoints (Clear Typehandler), or clearing a handler for a specific eventpoint (– clear handler). You must supply an *<event-specifier>* parameter with – clear handler.

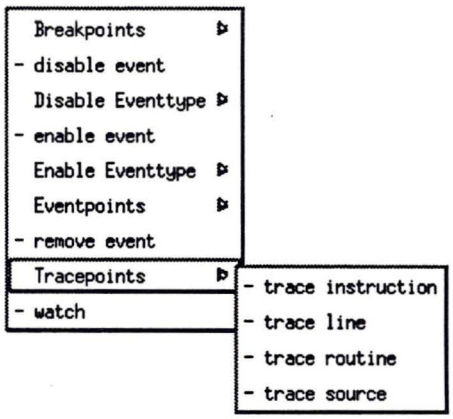
– remove event

Executes the `remove event` command, which removes the specified eventpoint. Events that have been removed can no longer be referenced in CXdb commands.

You must supply the *<event-specifier>* parameter. You can do this by clicking on an eventpoint marker in the Source Code or Assembly Code window, entering an eventpoint number, entering a CXdb debugger variable that you have assigned to an eventpoint, or entering * to remove all eventpoints in the current process.

Tracepoints

Brings up the following submenu:



You can create a tracepoint at an instruction, a line, a routine, or a source unit. When the executing process reaches the tracepoint, CXdb informs you of the event, but execution does not stop.

The dash (–) preceding each of these menu items indicates that you must specify an additional parameter, either on the command line or using command composition. For information about parameters, refer to the reference page for the corresponding `trace` command.

– watch

Executes the CXdb `watch` command. You must specify a starting address, an address range, or a starting address and a number of bytes to monitor. Refer to the reference page or online help topic for the `watch` command for more information.

Context

To access the Events menu, select Events from the command menubar in the CXdb Command window.

Related Windows

Command window

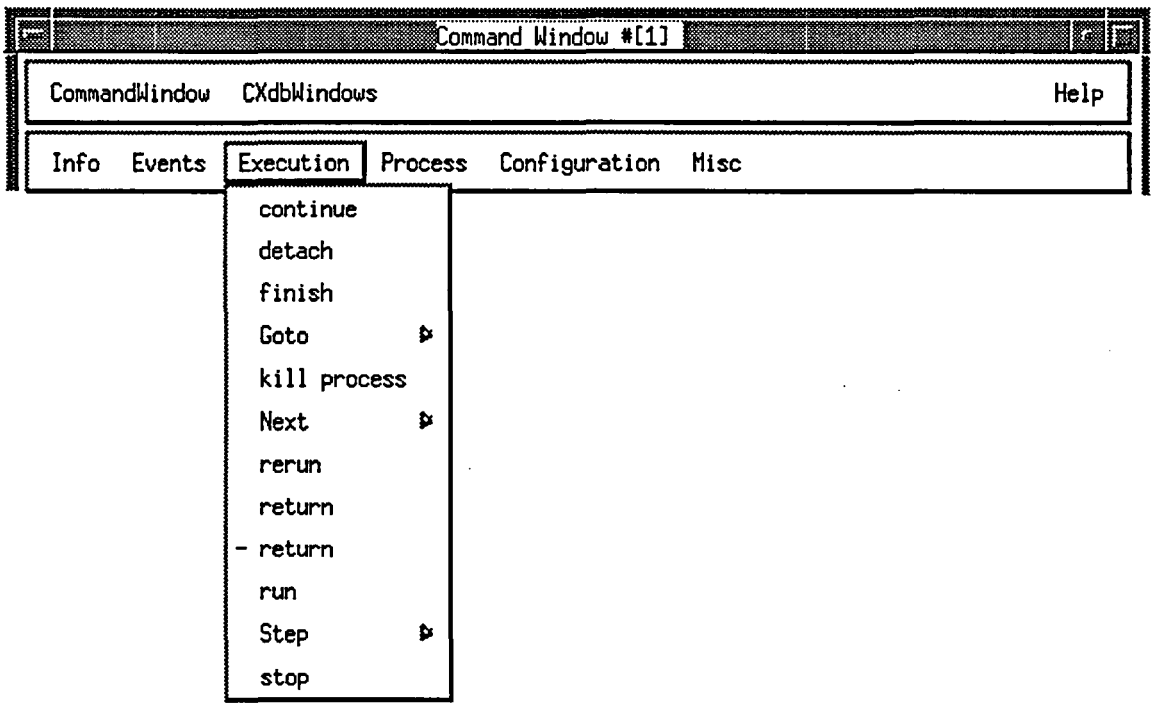
Related Commands

```
break instruction
break line
break routine
break source
clear default handler
clear handler
clear typehandler
disable event
disable eventtype
enable event
enable eventtype
event exec
event join
event modify
event reached instruction
event reached line
event reached source
event relation
event signal
event spawn
trace instruction
trace line
trace routine
trace source
watch
```

Related Concepts

breakpoints
command composition
eventpoint handlers
eventpoints
signals
source units
tracepoints
watchpoints

Execution menu



Description

The Execution menu enables you to execute CXdb commands for controlling the execution of your process by selecting them from menus rather than by typing them at the command line. Refer to the command reference pages in *CONVEX CXdb Commands and Parameters*, for more information on specific commands.

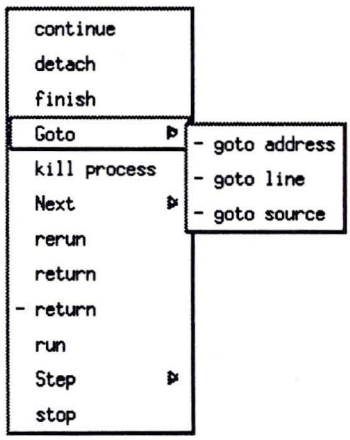
Items on the Execution menu and submenus preceded by a dash (–) require you to specify an additional parameter. You can do this using one of two methods:

- By typing it on the command line, then pressing **RETURN**
- By using command composition in the Source Code window. Refer to the "command composition" reference page or online help topic for more information.

Items on submenus not preceded with a dash (–) are executed using default parameters. To specify additional or different parameters, you must enter the commands from the keyboard.

Menu Items

Item	Action
continue	Executes the `continue` command, which continues execution of a stopped process. Images from core files cannot be continued.
detach	Executes the `detach` command, which detaches CXdb from a process. A process must be stopped in order to detach CXdb from it.
finish	Executes the `finish` command, which completes execution of (steps out of) the innermost source unit of the default stepping granularity for the current process.
Goto (C Series only)	Brings up a submenu with items for executing CXdb `goto` commands.



CAUTION:

The `goto address`, `goto line`, and `goto source` commands drastically change the order of execution for your program. Using these commands can lead to unpredictable results, particularly with optimized code.

– goto address — Executes the `goto address` command, which sets the program counter (PC) to the specified address. You must supply a *<language-expression>* parameter either by typing it at the command line, or selecting it

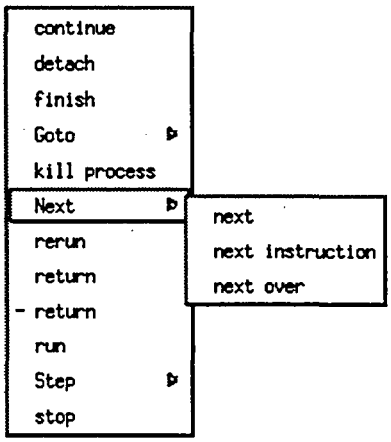
with the mouse in the Source Code window or Assembly Code window. The expression must evaluate to an address in the default language of the specified process. The address specified must also be the beginning of a machine instruction.

– **goto line** — Executes the `goto line` command, which sets the program counter (PC) to the starting address of the specified line of source code. You must supply a *<line-specifier>* parameter. This can include a source file name as well as the line number in the format [*<file-name>*] : *<integer>*.

– **goto source** — Executes the `goto source` command, which sets the program counter (PC) to the starting address of the specified source unit. You much supply a *<source-unit>* parameter. This can include a source file name as well as an integer that is a valid source unit number in the format [*<file-name>*] : *<integer>*.

kill process — Executes the `kill process` command, which terminates the current process.

Next — Brings up a submenu with items for executing `next` commands:



next — Executes the `next` command, which steps to the next source unit, ignoring subroutine calls. CXdb uses the default granularity of the current process as the step size.

next instruction — Executes the `next instruction` command, which steps to the next machine instruction, ignoring subroutine calls.

next over — Executes the `next over` command, which steps from the current source unit of specified granularity to the next source unit of default granularity, ignoring subroutine calls. CXdb uses the default granularity of the specified process as the step size.

rerun — Executes the `rerun` command, which kills the current process, then creates and executes a new process using the previous argument list.

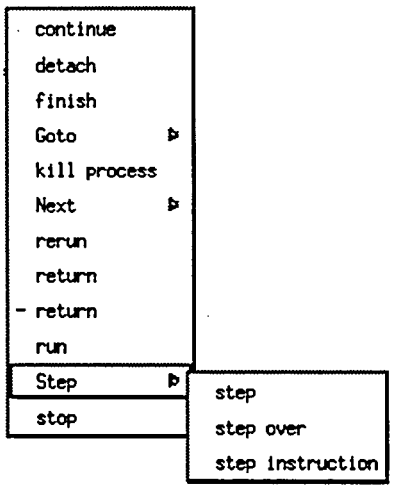
return — Executes the `return` command without parameters, which forces a return to the calling routine. CXdb prompts you to specify whether or not you want the function to return immediately. The default is yes (`y`).

– return — Executes the `return` command, which returns the value you specify to the calling routine. You must specify a *<language-expression>* parameter which can be any language expression that evaluates to a valid return value.

run — Executes the `run` command, which creates a new process from the executable image of the process object and then begins execution of that process. No arguments are passed to the process shell.

Step

Brings up a submenu with items for executing stepping commands:



step — Steps the process until it reaches the next source unit of the specified granularity. CXdb uses the default stepping granularity of the current process as the step size.

step over — Steps from the current source unit of specified granularity to the next source unit of default granularity. CXdb uses the default stepping granularity of the current process as the step size.

step instruction — Steps the process by one machine instruction.

stop

Executes the `stop` command, which stops execution of a process controlled by a command that is running in the background. The `stop` command stops all threads in the process.

Context

The Execution menu appears when you select Execution from the command menubar in the CXdb Command window.

Related Windows

Command window

Source Code window

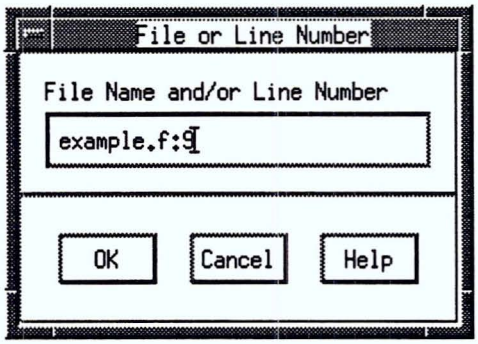
Related Commands

`continue`	`detach`
`finish`	`goto address`
`goto line`	`goto source`
`info process`	`kill process`
`next`	`next instruction`
`next over`	`rerun`
`return`	`run`
`step`	`step instruction`
`step over`	`stop`

Related Concepts

command composition
source units
stepping

File or Line Number dialog



Description

The File or Line Number dialog enables you to specify a different source code file or new location in the current source code file to display in the Source Code window.

Fields

Name	Description
File Name/Line Number	Valid values for this field are as follows: **File name** — Absolute or relative path name of a source file. **Line number** — A valid line number in the current source file. **File name and line number** — A file name and line number specified using the format *<file-name>*: *<integer>*.

Buttons

Name	Action
OK	Closes the dialog. If a valid file and/or line number is specified, the Source Code window displays the source code at that location.
Cancel	Closes the dialog without applying changes.
Help	Invokes the CXdb Help system and displays the help topic for the File or Line Number dialog.

File or Line Number dialog

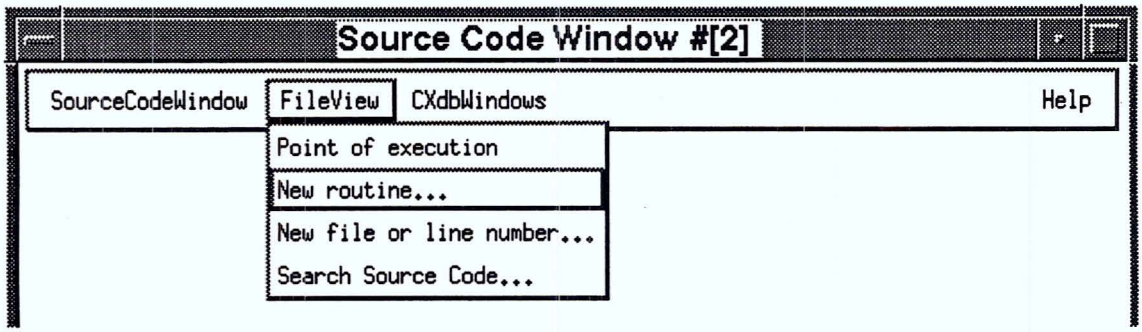
Context

The File or Line Number dialog appears when you select the New file or line number item from the FileView menu in the Source Code window.

Related Windows

Source Code window

FileView menu



Description

The FileView menu enables you to specify another file or routine to view in the Source Code window, to display the source code at the current point of execution, or to search for a text string in the Source Code window.,

Menu Items

Item	Action
Point of execution	Displays the source code at the current point of execution.
New routine	Brings up the Routine Name dialog, where you can specify a different routine in your source code to display in the Source Code window.
New file or line number	Brings up the File or Line Number dialog where you can specify a different file or line number in the current source file to display in the Source Code window.
Search Source Code	Brings up the Search Source Code dialog where you can specify a text string to search for in the file that is currently displayed in the Source Code window.

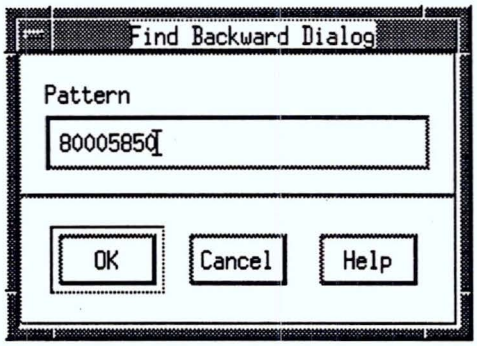
Context

The FileView menu appears when you select FileView from the main menubar of the Source Code window.

Related Windows

File or Line Number dialog
Routine Name dialog
Search Source Code dialog
Source Code window

Find Backward dialog



Description

The Find Backward dialog enables you to search backward (from highest address to lowest address) in memory for a hexadecimal pattern. The address range searched is the area of memory shown in the Memory Display window.

Fields

Name	Valid values
Pattern	The pattern to search for. It is expressed as a hexadecimal number. The specified byte pattern must contain an even number of hexadecimal digits.

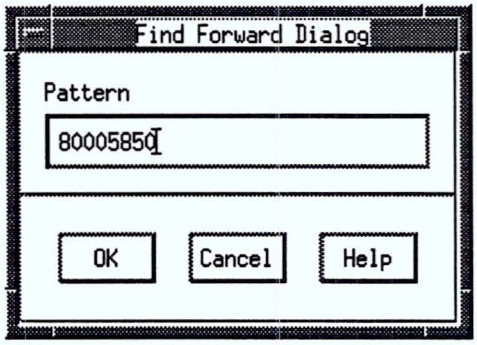
Buttons

Name	Action
OK	Closes the dialog and searches memory backward for the specified pattern. The results of the search—either the first match or a "Data not found..." message—are displayed in the Command window.
Cancel	Closes the dialog without executing a search.
Help	Invokes the CXdb Help system and displays the help topic for the Find Backward dialog.

Find Backward dialog

Context	The Find Backward dialog appears when you select the Find backward item from the DataView menu in the Memory Display window.	
Related Windows	Find Forward dialog	Memory Display window
Related Commands	`find memory backward`	`find memory forward`

Find Forward dialog



Description

The Find Forward dialog enables you to search forward (from lowest address to highest address) in memory for a hexadecimal pattern. The address range searched is the area of memory shown in the Memory Display window.

Fields

Name	Description
Pattern	The pattern to search for. It is expressed as a hexadecimal number. The specified byte pattern must contain an even number of hexadecimal digits.

Buttons

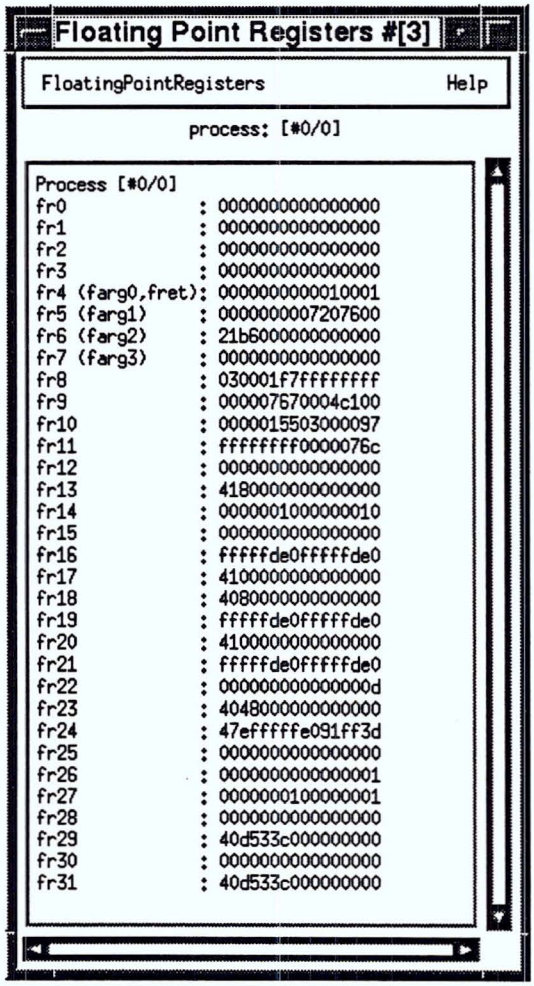
Name	Action
OK	Closes the dialog and searches memory forward for the specified pattern. The results of the search—either the first match or a "Data not found..." message—are displayed in the Command window.
Cancel	Closes the dialog without executing a search.
Help	Invokes the CXdb Help system and displays the Find Forward dialog help topic.

Find Forward dialog

Context	The Find Forward dialog appears when you select the Find forward item from the DataView menu in the Memory Display window.	
Related Windows	Find Backward dialog	Memory Display window
Related Commands	`find memory backward`	`find memory forward`

SPP Series only

Floating Point Registers window



Description

The Floating Point Registers window displays the contents of the floating point registers for the specified process. The contents are displayed in hexadecimal format by default. The floating point mode on SPP Series machines is IEEE.

Floating Point Registers window

There are 32 floating point registers, designated as fr0 through fr31. Registers fr4 to fr7 are the floating point argument registers (farg0 to farg3), and register fr4 is also called the floating point return register (fret).

For more information about the floating point registers, refer to the *PA-RISC 1.1 Architecture and Instruction Set Reference Manual* and the *PA-RISC Procedure Calling Conventions Reference Manual*, both available from Hewlett-Packard.

Menus

Name	Description
FloatingPointRegistersWindow	
	Contains the following items:
	Close — Closes the window.
	Auto update — Enables and disables automatic screen updating.
	Change format — Brings up a Format dialog where you can specify a different display format for the values displayed in the Floating Point Registers window.
	threads — Brings up a Threads dialog where you can control which thread is visible in the Floating Point Registers window.
Help	Contains items for invoking the CXdb Help system. Refer to the "Help menu" reference page or online help topic for more information.

Context

The Floating Point Registers window (SPP Series systems only) appears when you select Create window, then select the Floating Point Registers item from the CXdbWindows menu in either the Command window or the Source Code window.

Related Windows

Command window
Source Code window
Format dialog
Threads dialog

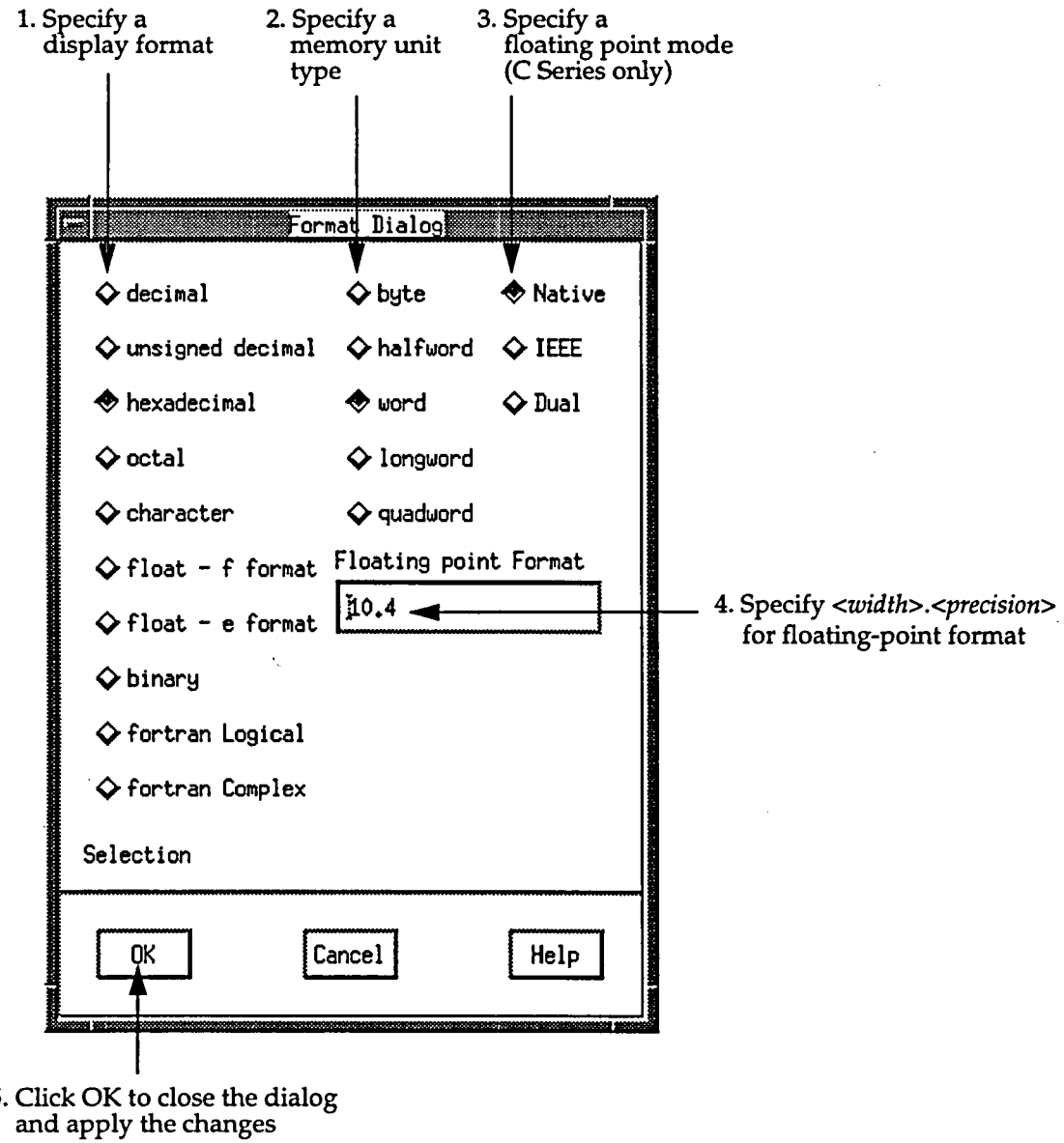
Related Commands

`info floating point registers`

Related Concepts

registers

Format dialog



Format dialog

Description

The Format dialog enables you to specify the display format for the values displayed in the Memory Display window and register windows. The display format determines the format used to display a segment of memory of a particular size, called a memory unit. Each memory unit can have a display format associated with it.

The size and number of selections displayed in the Format dialog vary depending on whether you are changing the display format of the Memory Display window (long form) or a register window (short form).

Specifying a format

The dialog box has three columns of radio buttons:

- The column on the left lists available formats.
- The center column lists available memory units.
- The column on the right lists floating point modes (the floating point mode determines the method used by the process when it performs a floating point calculation). On SPP Series systems, this column is not displayed because IEEE is the only floating point mode available.

Not all display formats are available for the various memory units. The memory unit types and their available formats are as follows:

- byte (8 bits)—Binary, character, decimal, Fortran logical, hexadecimal, octal, and unsigned decimal.
- halfword (16 bits)—Binary, decimal, Fortran logical, hexadecimal, octal, and unsigned decimal.
- word (32 bits)—Binary, decimal, floating point, scientific notation, Fortran logical, hexadecimal, octal, and unsigned decimal.
- longword (64 bits)—Binary, decimal, floating point, scientific notation, Fortran complex, Fortran logical, hexadecimal, octal, and unsigned decimal.
- quadword (128 bits)—Binary, floating point, scientific notation, Fortran complex, Fortran logical, hexadecimal, and octal.

To specify a format:

1. Select an appropriate display format from the leftmost column of radio buttons.
2. Select an appropriate memory unit from the center column of radio buttons.

3. Select a floating-point mode (native, IEEE, or dual) from the rightmost column of radio buttons (C Series only). On SPP Series systems, IEEE is the only available floating point mode.
4. If you have specified a float format (either scientific or floating point), specify the precision used to display floating point numbers in the Floating point Format field. The format is *<width>.<precision>*. Refer to the "Fields" section below for more information.
5. Click OK to close the dialog and apply the changes. The Memory Display or register window display updates to the new format.

Buttons

Name	Action
OK	Closes the dialog and applies the changes.
Cancel	Closes the dialog without making any changes.
Help	Invokes the CXdb Help system and displays the help topic for the Format dialog.

Fields

Name	Valid values
Floating point Format	Specify the floating point format as *<width>.<precision>*. The *<width>* is the maximum number of characters to display (including the decimal point) and the *<precision>* is the maximum number of digits to display to the right of the decimal point. Both *<width>* and *<precision>* must be positive integers. The default is `10.4`.

Context

The Format dialog appears when you select the Change format option from either the DataView menu in the Memory Display window or the leftmost menu of any register window (except the Processor Status Word window).

Format dialog

Related Windows

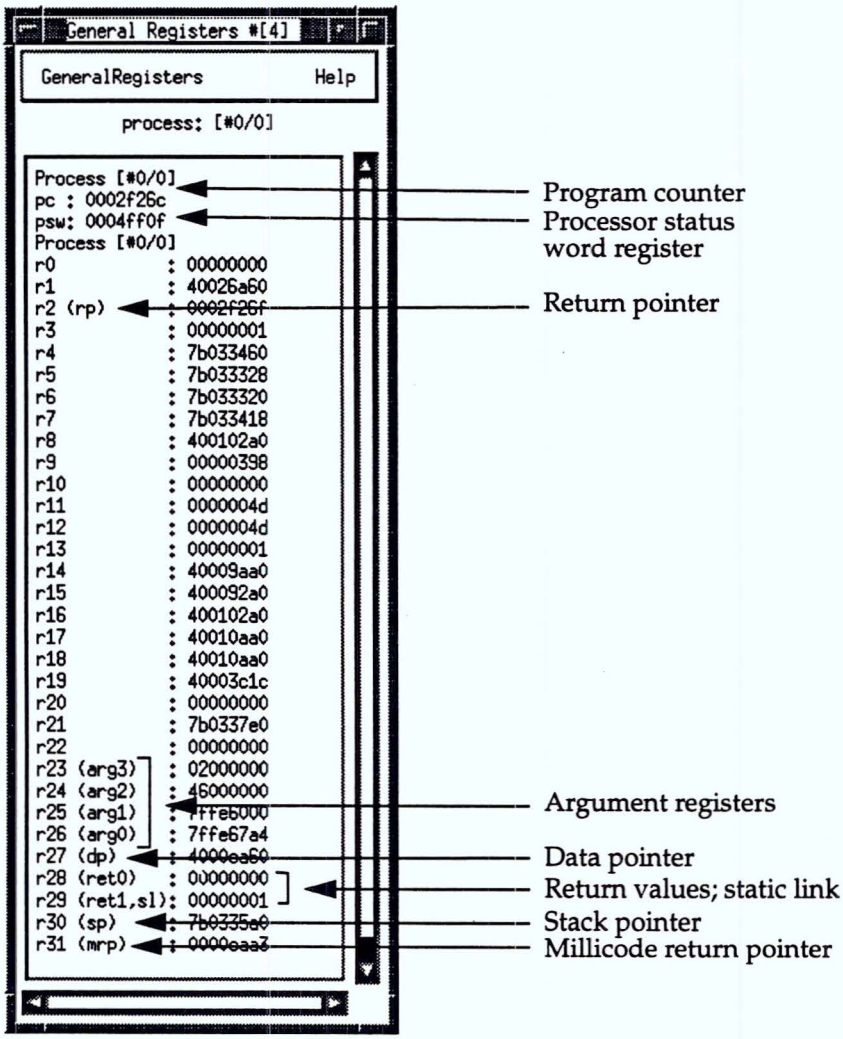
Control Registers window
Communication Registers window
Floating Point Registers window
General Registers window
Memory Display window
Scalar Registers window
Space Registers window
Vector Registers window

Related Commands

`set format`
`set default fpmode`
`set evalopts fpmode`
`set evalopts iprecision`
`set evalopts rprecision`
`set fpmode`

SPP Series only

General Registers window



Description

The General Registers window displays the contents of the general registers for the current thread of the specified process. By default, the display is in hexadecimal format.

You can specify a different display format for the window, enable and disable automatic updating, and specify which threads are visible in the window.

The registers displayed are:

- Program counter (pc)
- Processor status word (psw)
- General registers (r0 to r31). Alternate names for some of these registers are as follows:
  - arg0 to arg3 (same as r26 to r23) — Argument registers
  - dp (same as r27) — Data pointer
  - mrp (same as r31) — Millicode return pointer
  - ret0 and ret1 (same as r28 and r29) — Return values
  - rp (same as r2) — Return pointer
  - sl (same as r29) — Static link
  - sp (same as r30) — Stack pointer

For more information about these registers, refer to the *Exemplar Architecture* manual.

Menus

Name	Description
GeneralRegisters	Contains the following items: **Close** — Closes the window. **Auto update** — Enables and disables automatic screen updating. **Change format** — Brings up a Format dialog where you can specify a different display format for the values displayed in the General Registers window. **threads** — Brings up a Threads dialog where you can specify which thread is visible in the General Registers window.
Help	Invokes the CXdb Help system and displays the CXdb Help window. Refer to the "Help menu" reference page or online help topic for more information.

Context

The General Registers window appears when you select Create window, then select the General Registers item from the CXdbWindows menu in either the Command window or the Source Code window.

Related Windows

Format dialog

Threads dialog

Related Menus

Help menu

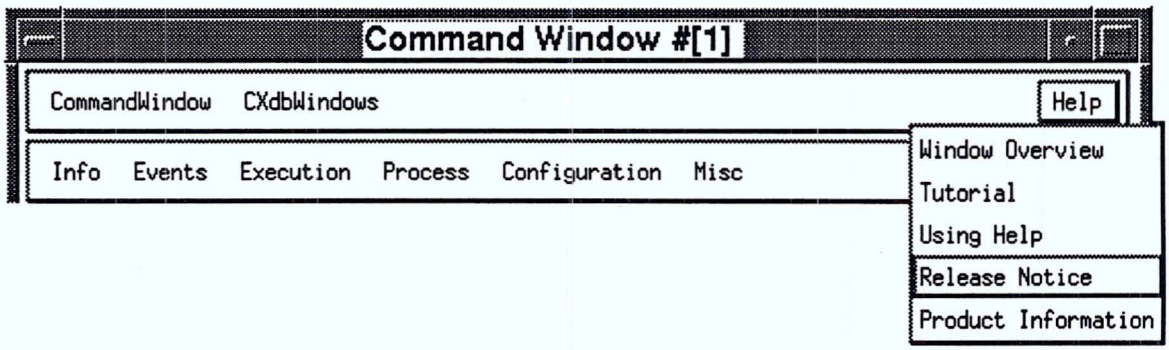
Related Commands

`info registers`

Related Concepts

registers

Help menu



Description

The Help menu contains items for providing context-sensitive help on CXdb windows, accessing the online tutorial, displaying instructions for using the Help system, displaying the release notice for the version of CXdb you are running, and displaying production information.

Selecting the Window Overview, Tutorial, Release Notice, or Using Help items invokes the CXdb Help system. If there is not an open Help window, CXdb creates one and displays the appropriate Help topic. Otherwise, the Help system simply displays the new topic in the open Help window.

Menu Items

Item	Action
Window Overview	Displays the help topic in the CXdb Help window for the window or dialog from which you invoked the Help system.
Tutorial	Displays the initial page of the CXdb online Tutorial in the Help window.
Using Help	Displays the initial page of the CXdb online Tutorial in the Help window.
Release Notice	Displays the release notice in the Help window for the version of CXdb you are running.
Product Information	Brings up a Product Information dialog. Refer to the "Product Information dialog" reference page or online help topic for more information.

Help menu

Context

The Help menu appears when you click on Help in any of the primary CXdb windows or register windows.

Related Windows

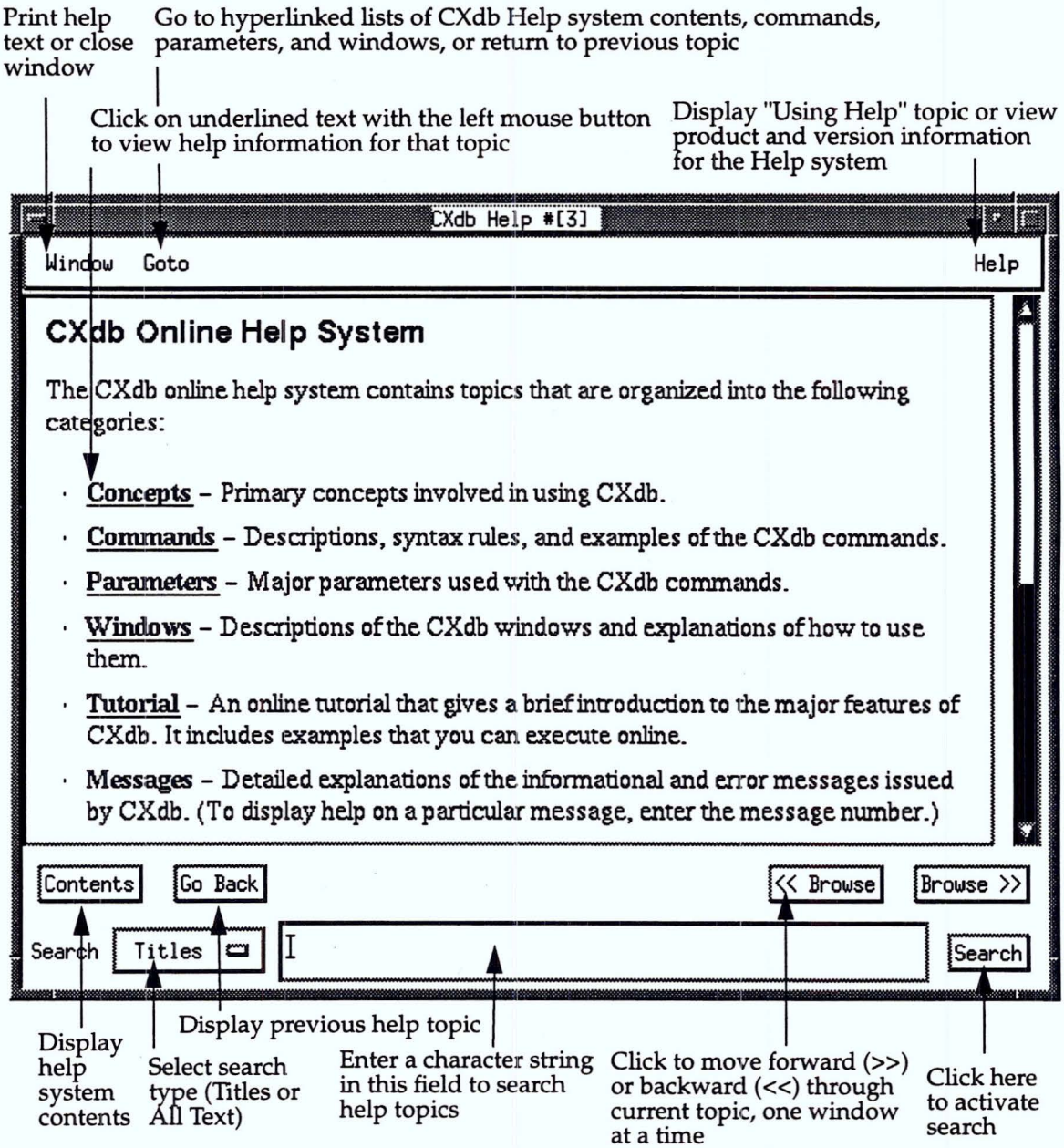
Help window

Product Information dialog

Related Commands

`help`

Help window


Print help text or close window
Go to hyperlinked lists of CXdb Help system contents, commands, parameters, and windows, or return to previous topic
Click on underlined text with the left mouse button to view help information for that topic
Display "Using Help" topic or view product and version information for the Help system
CXdb Help #[3]
Window
Goto
Help
CXdb Online Help System
The CXdb online help system contains topics that are organized into the following categories:
Concepts – Primary concepts involved in using CXdb.
Commands – Descriptions, syntax rules, and examples of the CXdb commands.
Parameters – Major parameters used with the CXdb commands.
Windows – Descriptions of the CXdb windows and explanations of how to use them.
Tutorial – An online tutorial that gives a brief introduction to the major features of CXdb. It includes examples that you can execute online.
Messages – Detailed explanations of the informational and error messages issued by CXdb. (To display help on a particular message, enter the message number.)
Contents
Go Back
<< Browse
Browse >>
Search
Titles
Search
Display help system contents
Select search type (Titles or All Text)
Display previous help topic
Enter a character string in this field to search help topics
Click to move forward (>>) or backward (<<) through current topic, one window at a time
Click here to activate search

Help window

Description

Using the CXdb Help window, you can navigate the CXdb online help system, print help text, search help topics (by title or full text), display instructions for using the Help system, and display version information for the help system.

Menus

Name	Items
Window	Contains the following items:
	Print Text—Pipes a formatted ASCII text version (without graphics) of the current help topic to the `lpr` command. The `lpr` command looks for the printer specified in your PRINTER environment variable. If this variable is not set, nothing is printed.
	Close—Closes the Help window.
Goto	Contains the following items:
	Previous Topic—Displays the previous topic stored in the help history.
	Contents page—Displays the CXdb Help system contents page, where you can select the category of help you want to view using hyperlinks.
	Commands list—Displays a hyperlinked list of help topics for all CXdb commands.
	Parameters list—Displays a hyperlinked list of help topics for all parameters used with CXdb commands.
	Concepts list—Displays a hyperlinked list of help topics for CXdb concepts.
	Windows list—Displays a hyperlinked list of help topics for CXdb windows, menus, and dialogs.
Help	Contains the following items:
	On Help—Displays instructions for using the CXdb Help system.
	On Version—Displays a Product Information dialog that identifies the version of the CXdb Help system (Convex Interactive Documentation Toolkit) you are using.

Buttons

Name	Action
Contents	Displays a hyperlinked table of contents for the online help system.
Go Back	Displays the previous topic stored in the help history. The help history contains all the topics you viewed during the current session.
Browse >>	Lets you move forward through the current help topic, one window length at a time. This button is deactivated when you reach the end of a topic.
<< Browse	Lets you move backward through the current help topic, one window length at a time. This button is deactivated when you reach the beginning of a topic.
Search	Searches for the character string you entered in the Search field.
Titles/All Text	Lets you select whether to search only the topic titles or the full text of all topics.

Context

The Help window appears when you:

- Execute the `help` command in X Windows mode
- Select the Window Overview, Using Help, or Tutorial item from the Help menu in the upper right corner of any CXdb window
- Click the Help button on any CXdb dialog

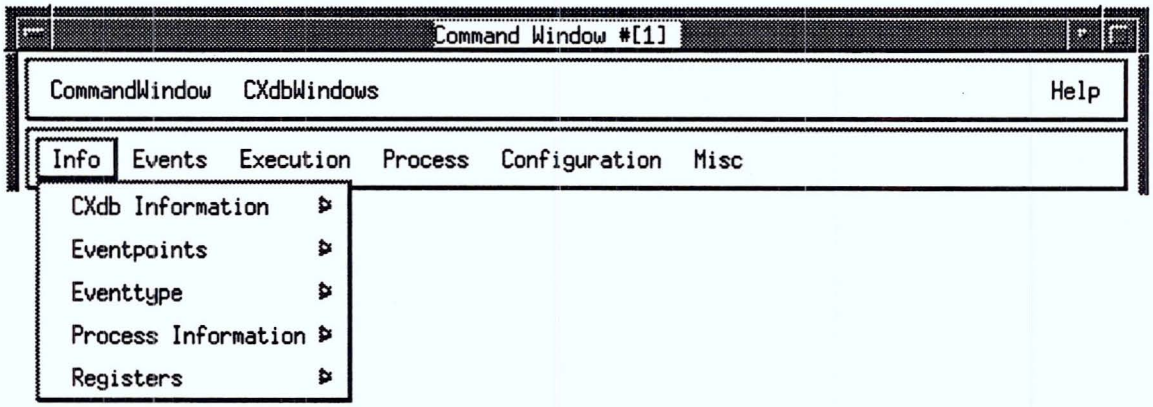
Related Menus

Help menu

Related Commands

`help`

Info menu



Description

The Info menu enables you to execute CXdb `info` commands by selecting them from menus rather than by entering them at the command prompt. For more detailed information on a specific command, refer to the reference page or online help topic for that command.

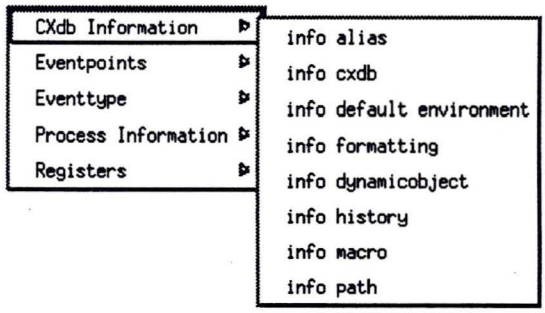
Items on submenus preceded by a dash (-) require you to specify an additional parameter. You can do this using one of two methods:

- Type the parameter on the command line, then press **RETURN.**
- Use command composition. Refer to the "command composition" reference page or online help topic for more information.

Items on Info submenus not preceded with a dash (-) are executed using default parameters. To specify additional or different parameters, you must enter the commands from the keyboard.

Menu items

Name	Action
CXdb Information	Brings up the following submenu with items for executing CXdb `info` commands that display information about the current debugging session:



info alias — Displays information about currently defined aliases

info cxdb — Displays information about the status of the current CXdb session

info default environment — Displays all default environment variables.

info formatting — Displays settings for memory display formats.

info dynamicobject — Displays memory settings for dynamically loaded objects.

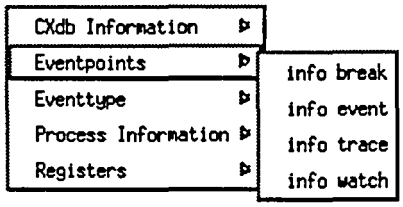
info history — Displays the CXdb command history.

info macro — Displays macro definitions.

info path — Displays the search path CXdb uses to find the source code for the current executable.

Eventpoints

Brings up a submenu containing items for executing CXdb `info` commands that display information about eventpoints:



info break — Displays all existing breakpoints.

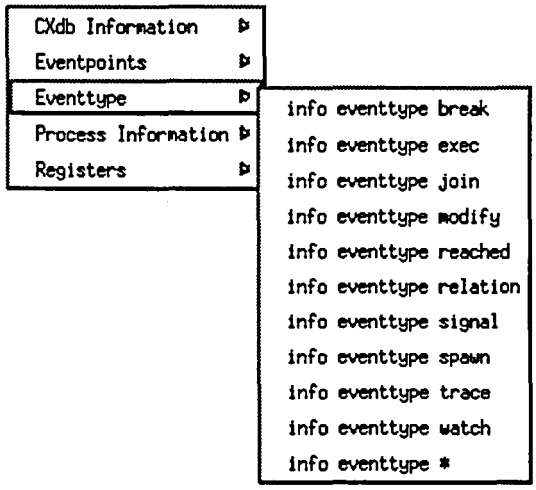
info event — Displays all existing eventpoints.

info trace — Displays all existing tracepoints.

info watch — Displays all existing watchpoints.

Eventtype

Brings up a submenu containing items for executing CXdb `info eventtype` commands:



info eventtype break — Displays all existing breakpoints.

info eventtype exec — Displays all existing exec eventpoints.

info eventtype join — Displays all existing join eventpoints.

info eventtype modify — Displays all existing modify eventpoints.

info eventtype reached — Displays all existing event reached eventpoints.

info eventtype relation — Displays all existing relation eventpoints.

info eventtype signal — Displays all existing signal eventpoints.

info eventtype spawn — Displays all existing spawn eventpoints.

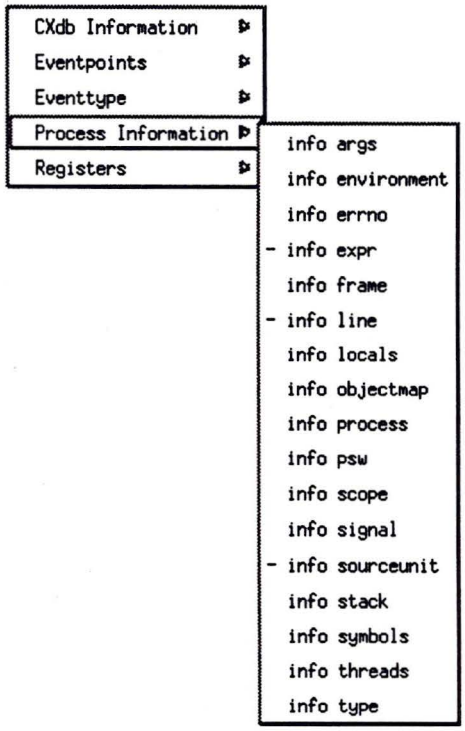
info eventtype trace — Displays all existing tracepoints.

info eventtype watch — Displays all existing watch points.

info eventtype * — Displays all existing eventpoints.

Process Information

Brings up a submenu containing items for executing CXdb `info` commands that display detailed information about the current proces.



info args — Displays arguments of the current routine.

info environment — Displays all process environment variables.

info errno — Displays the system error message received by the process.

- info expr — Displays the characteristics of the specified language expression. You must supply a <*language-expression*> parameter, either by typing it on the command line or by using command composition.

info frame — Displays information about the current stack frame.

- info line — Displays the source units for the specified line. You must supply a <line-number> parameter, by either typing it on the command line or clicking on a line number in the Source Code window and pressing **RETURN**.

info locals — Displays the local variables of the current routine.

info objectmap — Displays the object map.

info process — Displays the status of the process.

info psw — Displays the contents of the processor status word (PSW) register.

info scope — Displays the current scope path.

info signal — Displays signal names, numbers, and settings for signal actions.

- info sourceunit — Displays the specified source unit.You must specify a <*source-unit*> number parameter, either by typing it on the command line or using command composition.

info stack — Displays information about the process stack.

info symbols — Displays program symbols.

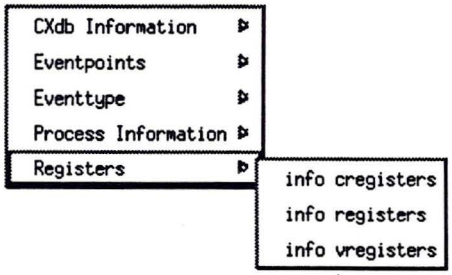
info threads — Displays information about the threads of the current process.

info type — Displays type definitions.

Registers

Brings up a submenu containing items for executing CXdb `info` commands that display the contents of registers. The menu items available vary between C Series and SPP Series systems:

On C Series systems:

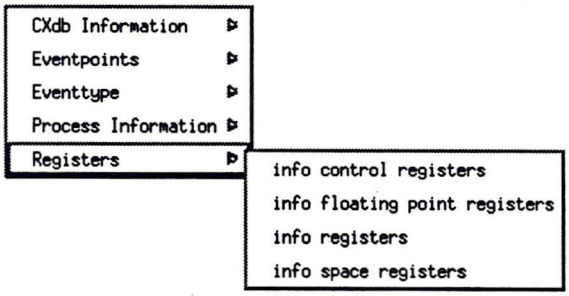


info cregisters — Displays contents of communications registers.

info registers — Displays contents of the PC (program counter), PSW (processor status word register), address registers, and scalar registers.

info vregisters — Displays contents of vector registers.

On SPP Series systems:



info control registers — Displays contents of control registers.

info floating point registers — Displays contents of floating point registers.

info registers — Displays contents of the PC, PSW, and general registers.

info space registers — Displays contents of space registers.

Context

To access the Info menu, select the Info menu item from the command menubar in the CXdb Command window.

Related Windows

Command window

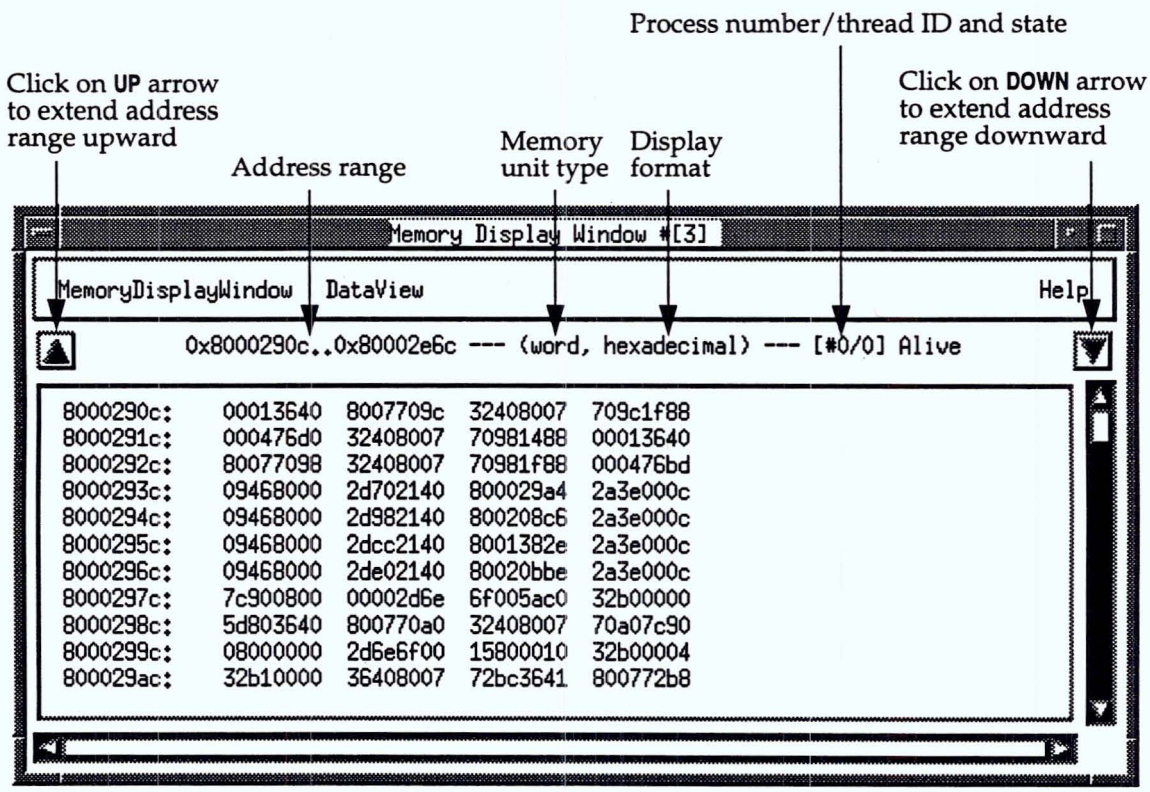
Related Commands

```
info alias                  info args
info break                  info control registers
info cregisters             info cxdb
info default environment    info dynamicobject
info environment            info errno
info event                  info eventtype
info expression             info floating point registers
info frame                  info formatting
info history                info line
info locals                 info macro
info objectmap              info path
info process                info psw
info registers              info scope
info signal                 info sourceunit
info alias                  info args
info space registers        info stack
info symbols                info threads
info trace                  info type
info vregisters             info watch
```

Related Concepts

breakpoints
command composition
eventpoints
registers
scope
signals
source units
threads
tracepoints
watchpoints

Memory Display window



Description

The Memory Display window displays the contents of memory. When a Memory Display window first opens, it displays the area of memory starting at the current program counter (PC) in the currently specified display format. The default memory unit size is word; the default format is hexadecimal. By default, automatic updating is enabled. This means that the display updates automatically whenever the program modifies the area of memory being displayed.

In the Memory Display window, you can:

- Specify a new address to examine using any language expression that evaluates to a valid program address
- Search the window forward or backward for a specified byte pattern

- Select which threads of your process are visible in the memory contents window
- Change the format of the values displayed in the window

You can open multiple Memory Display windows during a debugging session, each one displaying a different section of memory.

Generally, the Memory Display window is for examining the data portion of memory. To view machine instructions, use the Assembly Code window.

You can use the arrow buttons to increase the address range for the instructions shown in the scrolled window:

- Clicking the **UP** arrow button extends the range upwards.
- Clicking the **DOWN** arrow button extends the address range downwards.

Menus

Name	Description
MemoryDisplayWindow	Contains the following options:
	Close — Closes the window.
	Auto update — Enables or disables automatic updating of window contents.
	threads — Pops up a Threads dialog where you can associate the Memory Display window with a particular thread.
DataView	Contains the following options:
	New address — Brings up a New Address dialog where you can specify a starting address for viewing a different area of memory.
	Find forward — Brings up a Find Forward dialog where you can specify a byte pattern to search for.
	Find backward — Brings up a Find Backward dialog where you can specify a byte pattern to search for.
	Change format — Brings up a Format dialog where you can specify a different display format for the values displayed in the Memory Display window.

Help	Contains items for invoking the CXdb Help system. Refer to the "Help menu" reference page or online help topic for more information.

Context

The Memory Display window appears when you:

- Select Create window, then select the Memory Display item from the CXdbWindows menu in either the Command window or the Source Code window.
- Enter the command `display examine` *<address-expression>* at the `(CXdb)` prompt in the Command window. The *<address-expression>* can be any language expression that evaluates to an address in the syntax of the current source language.

Related Windows

Assembly Code window
Find Backward dialog
New Address dialog
Format dialog
Find Forward dialog

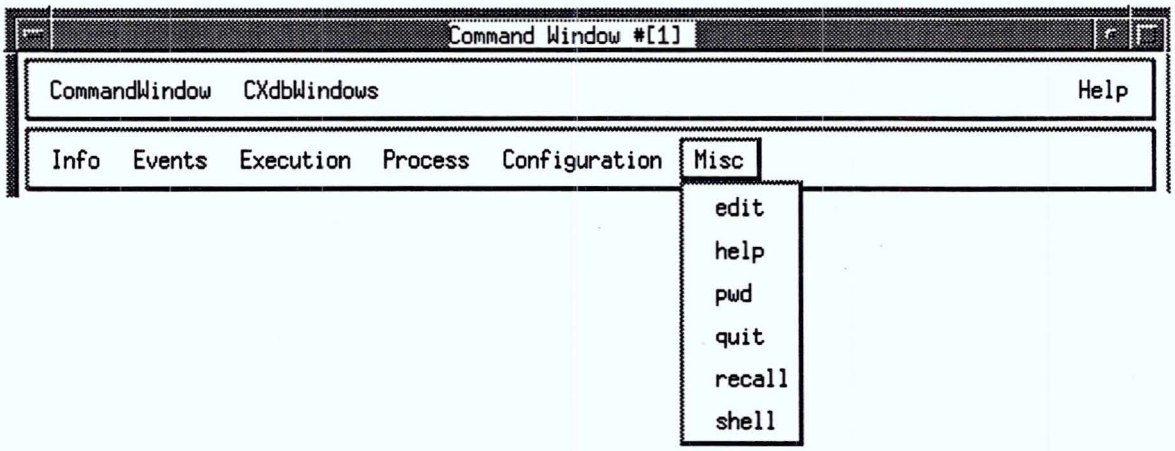
Related Menus

Help menu

Related Commands

`display examine`
`find memory backward`
`print`
`examine`
`find memory forward`
`set format`

Misc menu



Description

The Misc menu enables you to execute several CXdb commands by selecting them from submenus rather than by entering them at the command prompt.

The items on the Misc menu are executed using default parameters. To specify additional or different parameters, you must enter the commands from the keyboard.

For more detailed information on a specific command, refer to the reference page or online help topic for that command.

Menu Items

Item	Action
edit (C Series only)	Executes the `edit` command without parameters, which opens an editor window and invokes your default editor.
help	Executes the `help` command without parameters, which invokes the CXdb Help system and displays the "Using Help" topic in the Help window.
pwd	Executes the `pwd` command, which displays the name of the console working directory.

Misc menu

quit	Executes the `quit` command, which automatically closes all files and windows that it opened and exits CXdb.
recall	Executes the `recall` command, which retrieves the previous command in the command history and automatically executes it again. CXdb does not ask for confirmation before executing the retrieved command.
shell	Opens an interactive shell window of the current shell type. Multiple shells can be open at once.

Context

The Misc menu appears when you select Misc from the command menubar in the CXdb Command window.

Related Windows

Command window

Related Commands

`edit`
`help`
`pwd`
`quit`
`recall`
`shell`

mouse and keyboard shortcuts

Description

CXdb provides several mouse and keyboard shortcuts for composing and executing commands.

Shortcuts for composing, editing, and executing commands in all modes include key bindings for command-line editing; command abbreviations, aliases and macros; command completion; CXdb command files; and the ability to display and scroll entries in the command history.

Shortcuts available in specific CXdb windows include the use of the mouse to manipulate eventpoints in the Source Code and Assembly Code windows, default key bindings for executing CXdb commands, command buttons, command composition, and the actions popup menu in the Source Code window.

Command-line shortcuts available in all modes

CXdb provides the following shortcuts for composing, editing, and executing commands on the command line (these are available in both X Window mode and line mode):

- **Command-line editing** — The following key bindings for command-line editing are available in both X Windows mode and line mode, assuming that you have not modified the default key bindings:

Key	Function
CTRL-a	Go to the beginning of the line
CTRL-b	Back one character
META-b	Back one word
CTRL-d	Delete next character
METa-d	Delete next word
CTRL-e	End of line
CTRL-f	Forward one character
META-f	Forward one word
CTRL-h	Delete previous character
CTRL-k	Kill to end of line
CTRL-l	Redraw the screen

CTRL-p	View previous entry in the command history
CTRL-n	View next entry in the command history
TAB	Complete current command

In line mode, there are additional key bindings. Enter **META-?** or **ESC-?** to list all available key bindings. In X Windows mode, you can also use the **LEFT ARROW** and **RIGHT ARROW** keys in the Command window to move left or right one character, respectively.

- **Command abbreviations** — You can abbreviate CXdb commands. For example, the command `info default environment` can be abbreviated to `in d e`. The shortest unique abbreviation for each command is shown in the upper right corner of the first reference page for each command, immediately below the command name.

- **Command completion** — This feature makes it easy to enter long variable names or command names. With completion, you can type only the first few letters of a command or variable name, then press **TAB**. CXdb fills in as much of the command as it can, up to the point where it determines the command or variable name is no longer unique.

  For example, `bre` **TAB** produces the completion "break." CXdb cannot complete the command any further, because the next word in the command could be "line," "instruction," "routine," or "source."

- **Command history** — CXdb stores the last 100 commands you entered in a history buffer. At the CXdb command line you can:
  - Use the `info history` command to display the history list.
  - Press **CTRL-p** to go to the previous entry in the command history.
  - Press **CTRL-n** to go to the next entry in the command history.

  To execute a command in the command history, use **CTRL-p** and **CTRL-n** to move up and down in the command history to locate the desired command, then press **RETURN**. You can also use the `recall` command to execute the most recent command that matches a given substring.

- **Command aliases** — Many CXdb commands have default aliases. You can also create custom aliases using the `alias` command. An alias can represent the first part of a command line, a complete command, or multiple commands. Use the `info alias` command to display current definitions of all aliases, both default and user-defined.

- **Command macros** — A macro can substitute for part of a command, a complete command, or multiple commands. Macros can accept parameters, and the parameters can have default values. Use the `macro` command to create a macro; use the `info macro` command to display macro definitions. For more information on creating CXdb command macros, refer to the reference page for the `macro` command.
- **Command files** — You can create a command file to store frequently used sequences of CXdb commands. To execute the command file, use the command `source` *<command_file>*. Refer to the "command files" reference topic or the reference page for the `source` command for additional information and examples.

Command window shortcuts

In addition to the command-line shortcuts described in the previous section, you can use the following shortcuts in the Command window:

- **Command buttons and menus** — Enable you to select and execute CXdb commands without having to type them at the command line. Refer to the "Command window" reference topic for more information.
- **Command composition** — Enables you to compose and execute commands using the mouse instead of the keyboard. Refer to the "command composition" reference topic for more information.
- **xterm copy and paste** — You can use the mouse to copy sections of text from any xterm window or CXdb window and paste it on the CXdb command line.
- **Command-line editing shortcuts** — In addition to the key binding listed in the previous section, the following key bindings for command-line editing are available in the Command window, assuming that you have not modified the default key bindings:

Key	Function
CTRL-v	Scroll down one page
Meta-v	Scroll up one page
CTRL-y	Yank last kill
CTRL-u	Beginning of line, kill line
CTRL-i	Complete current command

Source Code window shortcuts

Using the mouse, mouse and key combinations, or the actions popup menu in the Source Code window you can:

- Execute the `continue`, `step`, and `next` commands.
- Select (highlight) text.
- Select (highlight) source units.
- Disable, enable, or remove eventpoints.
- Print the value of a variable or expression.
- Execute the `info expression`, `info line`, and `info source` commands.
- Set a breakpoint at a line number, source unit, or instruction.
- Set a tracepoint at a line number, source unit, or instruction.
- Run to a location.
- Print the value referenced by a C pointer variable (equivalent to `print *(`*ptr*`)`).

Mouse button and key bindings and the actions they perform in the Source Code window (assuming you have not modified the default mouse button/key bindings) are as follows:

Key/button	Function
c	Executes the `continue` command.
n	Executes the `next` command
s	Executes the `step` command using the default granularity.
Left button (in text area)	Selects (highlight) text.
Left button (on eventpoint markers)	Enable, disable, or remove eventpoints using the Event Point dialog. Refer to the "Source Code window" reference topic for more information.
Right button	Brings up an actions popup menu and highlights the source unit at the location of the mouse pointer. When you release the mouse button, the command (action) you selected uses the highlighted source unit as its argument. The actions menu contains the following items:

print — If the highlighted source unit is a valid language-expression, performs the `print` command on the highlighted source unit.

print * (print indirect) — If the highlighted source unit is a C pointer variable, selecting this item prints the value referenced by the pointer (equivalent to `print *(ptr)`).

breakpoint — Sets a breakpoint at the highlighted line number or source unit.

tracepoint — Sets a tracepoint at the highlighted line number or source unit.

info expression — If the highlighted source unit is a valid language-expression, performs the `info expression` command on the highlighted source unit.

info source — Performs the `info source` command on the highlighted source unit.

info line — Performs the `info line` command for the first (or only) line containing the highlighted source unit.

run to — Sets a breakpoint at the highlighted line number or source unit, continues execution, stops execution at the breakpoint, and then removes the breakpoint.

CTRL-v	Scroll down one page
META-v	Scroll up one page

Assembly Code window shortcuts

The following shortcuts are available in the Assembly Code window (assuming you have not modified the default key bindings):

Key	Function
c	Execute the `continue` command.
n	Execute the `next instruction` command.
s	Execute the `step instruction` command
Left button	Click the left mouse button on any eventpoint marker to bring up an Event Point dialog where you can enable, disable, or remove eventpoints. Refer to the "Assembly Code window" reference topic for more information.

Right button	Click the right mouse button and drag the mouse to highlight text. When you release the mouse button, CXdb executes the `print` command on the selected text.

Stack Trace window shortcuts

The following shortcuts are available in the Stack Trace window (assuming you have not modified the default mouse button/key bindings):

Key	Function
0 through **9**	Changes the current frame to the number pressed.
d	Moves up the stack one frame at a time (equivalent to the `frame +1` command)
t	Makes the top frame the current frame (equivalent to the `frame 0` command)
u	Moves down the stack one frame at a time (equivalent to the `frame -1` command).
Left button	Click anywhere on the line describing a stack frame in the Stack Trace window to open a Stack Frame Description dialog displaying detailed information about that frame.

Thread Activity window shortcuts

The following shortcuts are available in the Thread Activity window (assuming you have not modified the default button/key bindings).

Key	Function
Left button	Click on bar in thread activity graph to display source code for the corresponding routine or file.
SHIFT-Left button	Click on bar in thread activity graph to toggle between displaying threads per file in program or threads per routine in file.
Right button	Click on bar in thread activity graph to pop up an actions menu. Refer to the "Thread Activity window" reference topic for more information.

Related Commands

`alias`
`continue`
`frame`
`info alias`
`info history`
`info expression`
`info line`
`info sourceunit`
`macro`
`print`
`source`

Related Parameters

language-expression
source-unit

Related Windows

Assembly Code window
Command window
Source Code window
Stack Trace window
Thread Activity window

Related Concepts

breakpoints
C language expressions
command composition
command files
Fortran language expressions
language expressions
line mode
source units
stepping
tracepoints
Xdefaults

New Address dialog

Description

The New Address dialog enables you to specify a starting address for viewing:

- Assembly code in the Assembly Code window
- Areas of memory in the Memory Display window

The address that you specify can be either symbolic or absolute.

Fields

Name	Description
Memory Address	Enter an absolute or symbolic address. You can specify the address using any language expression in the syntax of the current source language that evaluates to a valid program address. Example of valid entries in this field are `0x800053d8` or `LOC(SUB1)`

Buttons

Name	Action
OK	Tries to locate the specified memory address and, if successful, closes the dialog. If not successful, error messages are displayed in the Command window, and you must either specify a different address or click Cancel to close the dialog.

New Address dialog

Cancel	Closes the dialog without changing the addresses you are viewing.
Help	Displays the "New Address dialog" help topic in the CXdb Help window.

Context

The New Address dialog appears when you select the New address option from either the DataView menu in the Memory Display window or the InstructionView menu in the Assembly Code window.

Related Windows

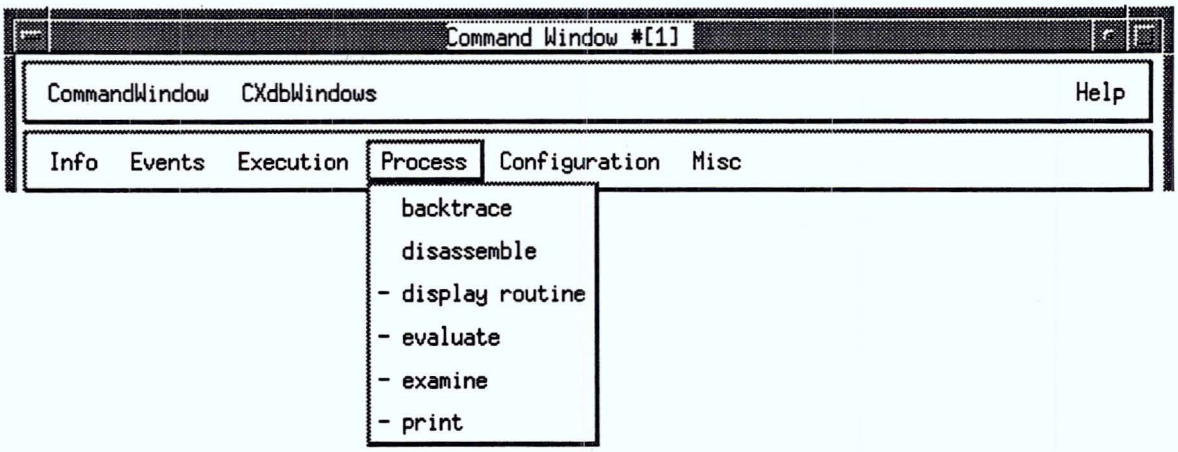
Assembly Code window

Memory Display window

Related Commands

`disassemble`
`display disassembly`
`display examine`
`examine`

Process menu



Description

The Process menu enables you to execute CXdb commands for displaying more detailed information about the current process by selecting them from submenus rather than by entering them at the command prompt. For more detailed information on a specific command, refer to the corresponding reference page in *CONVEX CXdb Commands and Parameters*.

Items on the Process menu and submenus preceded by a dash (–) require you to supply an additional parameter. You can do this using one of two methods:

- By typing it on the command line, then pressing **RETURN.**
- By using command composition

Items on submenus not preceded with a dash (–) are executed using default parameters. To specify additional or different parameters, you must enter the commands from the keyboard.

Menu items

Item	Action
backtrace	Executes the `backtrace` command, which displays a list of the frames currently on the stack.

disassemble	Executes the `disassemble` command. This displays the disassembled code for all threads of the current process. The disassembly begins at the starting address of the routine indicated by the current stack frame.
– display routine	Executes the `display routine` command. You must supply a *<language-expression>* parameter that evaluates to a valid address within the bounds of the process. Executing this command opens a new Source Code window that displays the source code of the named routine or the routine that contains the specified address. The new window is not associated with any threads of the current process.
– evaluate	Executes the `evaluate` command, which evaluates the specified language expression You must supply a *<language-expression>* parameter, which can be any expression that is valid in the current source language. The main purpose of the `evaluate` command is to assign values to debugger variables and to change the values of process variables.
– examine	Executes the `examine` command, which displays a specified region of memory. You must supply an address range or a starting address and the number of memory units to display. You can specify any *<language-expression>* that evaluates to an address.
– print	Executes the `print` command, which evaluates a language expression and prints the result. You must specify a *<language-expression>* parameter. This can be any expression that is valid in the current source language.

Context

The Process menu appears when you select Process from the command menubar in the Command window.

Related Windows

Command window

Related Commands

`backtrace`
`disassemble`
`display routine`
`evaluate`
`examine`
`print`

Processor Status Word window

Description

The Processor Status Word window displays a bit-by-bit description of the contents of the Processor Status Word register for the current stack frame.You can specify which thread of your process is visible in the window and control automatic updating.

The detailed breakdown of the PSW output is shown in three-column format, as follows:

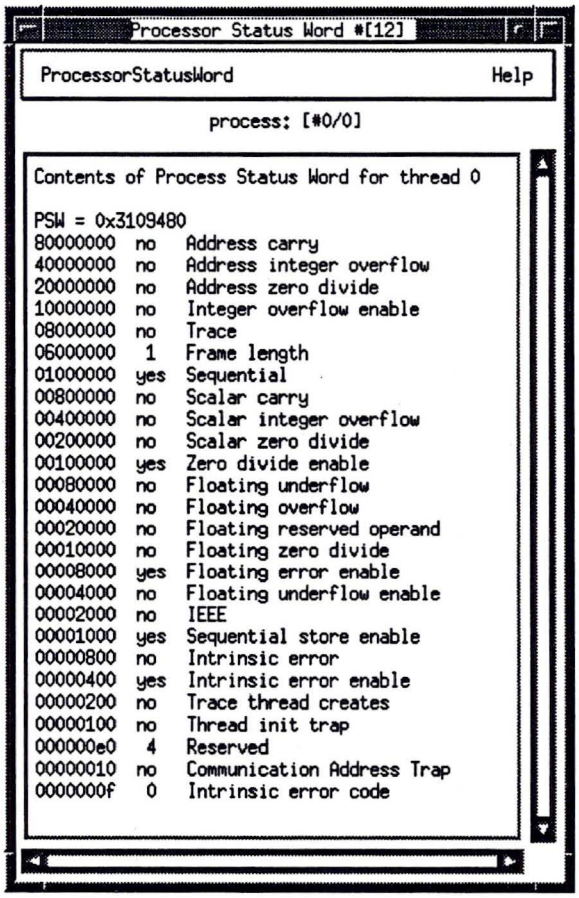
- Left column — A hexadecimal number that indicates the bit position(s) occupied by a field.
- Center column — The value or setting of the field.
- Right column — The name or purpose of the field.

The detailed breakdown lists the bits in order from most significant bit (bit 31) to least significant (bit 0). Some of the fields of the PSW occupy more than one bit.

The output shown in the Processor Status Word register varies between C Series and SPP Series systems. The following sections contain more detailed information about the Processor Status Word on these architectures.

Processor Status Word window

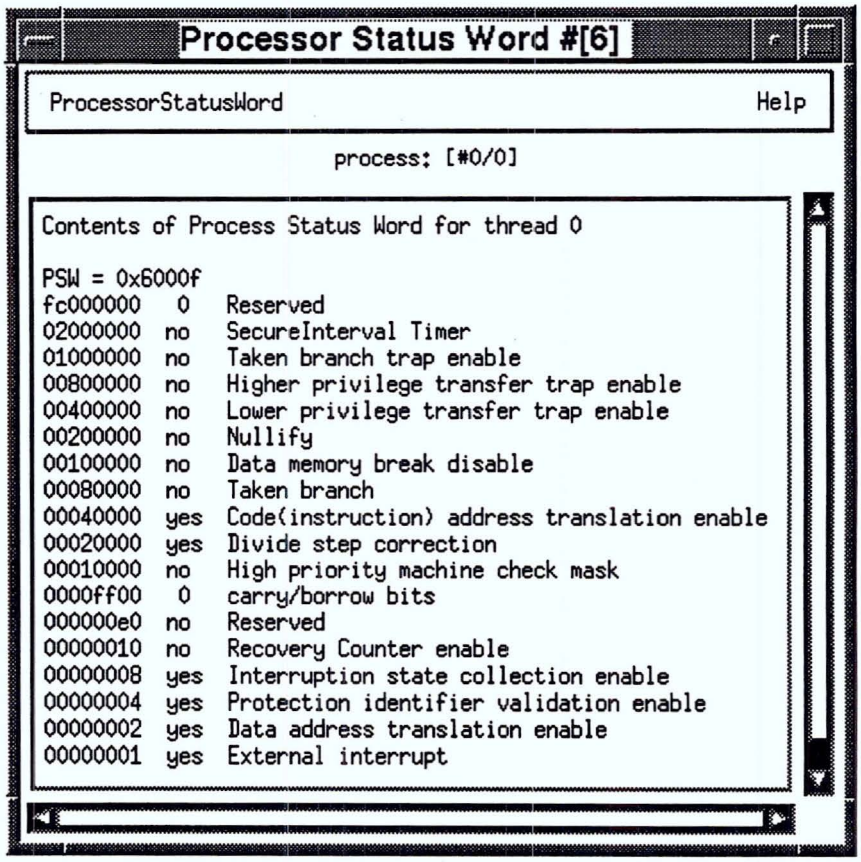
C Series systems



On C Series machines, the processor status word (PSW) is a user-accessible, 32-bit status register that indicates the processor state. This register contains flags that enable or disable exception processing and indicate the results of numerical operations

For more information about the PSW on C Series machines, refer to the *CONVEX Architecture Reference Manual (C Series).*

SPP Series systems



For information about the PSW on SPP Series machines, refer to the *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, available from Hewlett-Packard.

Menus

Name	Description
ProcessorStatusWord	Contains the following items: **Close** — Closes the Processor Status Word window. **Auto update** — Enables or disables automatic updating of the contents in the Processor Status Word window. **threads** — Brings up a Threads dialog where you can specify which thread is visible in the Processor Status Word window.

Help	Contains options for invoking the CXdb Help system. Refer to the "Help menu" online help topic or reference page for more information.

Context

The Processor Status Word window appears when you select Create window, then select the Processor Status Word item from the CXdbWindows menu in either the Command window or the Source Code window.

Related Windows

General Registers window
Scalar Registers window
Threads dialog

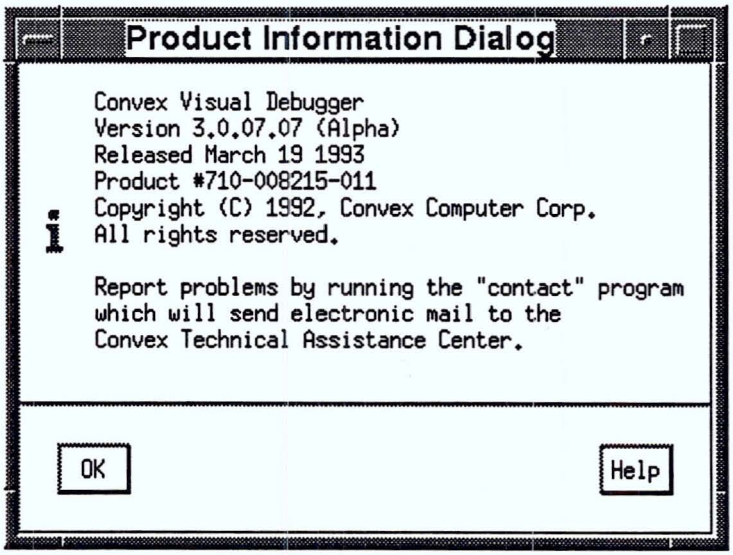
Related Menus

Help menu

Related Commands

```
clear seq
clear sqs
info psw
info registers
set seq
set sqs
```

Product Information dialog



Description

The Product Information dialog displays the following information about the version of CXdb you are running:

- Product name—Convex Visual Debugger
- Version
- Release date
- Product ID number
- Copyright information
- Reference to using the `contact` utility to submit problem reports (refer to the "Reporting problems" section for more information)

Getting technical assistance

If you have questions about CXdb, contact the Convex Technical Assistance Center (TAC). To contact the TAC, use one of the following phone numbers:

- Within the continental United States, call 1(800)952-0379.

- From Canada, call 1(800)345-2384.
- All other locations, contact the nearest Convex office.

Reporting problems

The Convex Technical Assistance Center (TAC) recommends using the contact utility to report a hardware, software, or documentation problem. The `contact` utility is an interactive program that helps the TAC track reports and route them to the Convex personnel most qualified to fix a problem.

After you invoke `contact`, it prompts you for information about the problem. When you finish your report, `contact` mails it to the TAC electronically. The TAC notifies you within 48 hours that your report has been received.

Using `contact` requires

- UNIX-toUNIX Communication Protocol (UUCP) connection to the TAC
- Full path name of the program or utility in question
- Version number of the program or utility in question

Refer to the contact(1) man page for complete details.

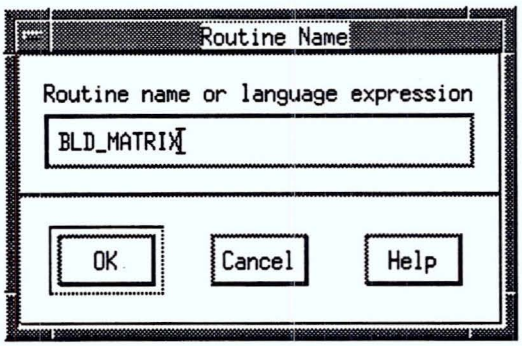
Buttons

Name	Action
OK	Closes the dialog.
Help	Displays the help text for the Product Information dialog in the Help window.

Context

The Product Information dialog appears when you select the Product Information item from the standard Help menu, accessible from all CXdb windows except the Help window.

Routine Name dialog



Description

The Routine Name dialog enables you to specify a different routine to display in the Source Code window.

Fields

Name	Description
Routine name	The value entered in this field can be a routine name or any language expression in the source language that evaluates to a valid address within the bounds of your process.

Buttons

Name	Action
OK	If a valid value is specified, CXdb closes the dialog and displays the source code of the named routine or the routine that contains the specified address in the Source Code window. If an incorrect value is specified, CXdb displays an error message in the Command window, and you must either enter a different address and then click OK or click Cancel to close the dialog.
Cancel	Closes the dialog without applying changes.
Help	Displays the "Routine Name dialog" help topic in the CXdb Help window.

Routine Name dialog

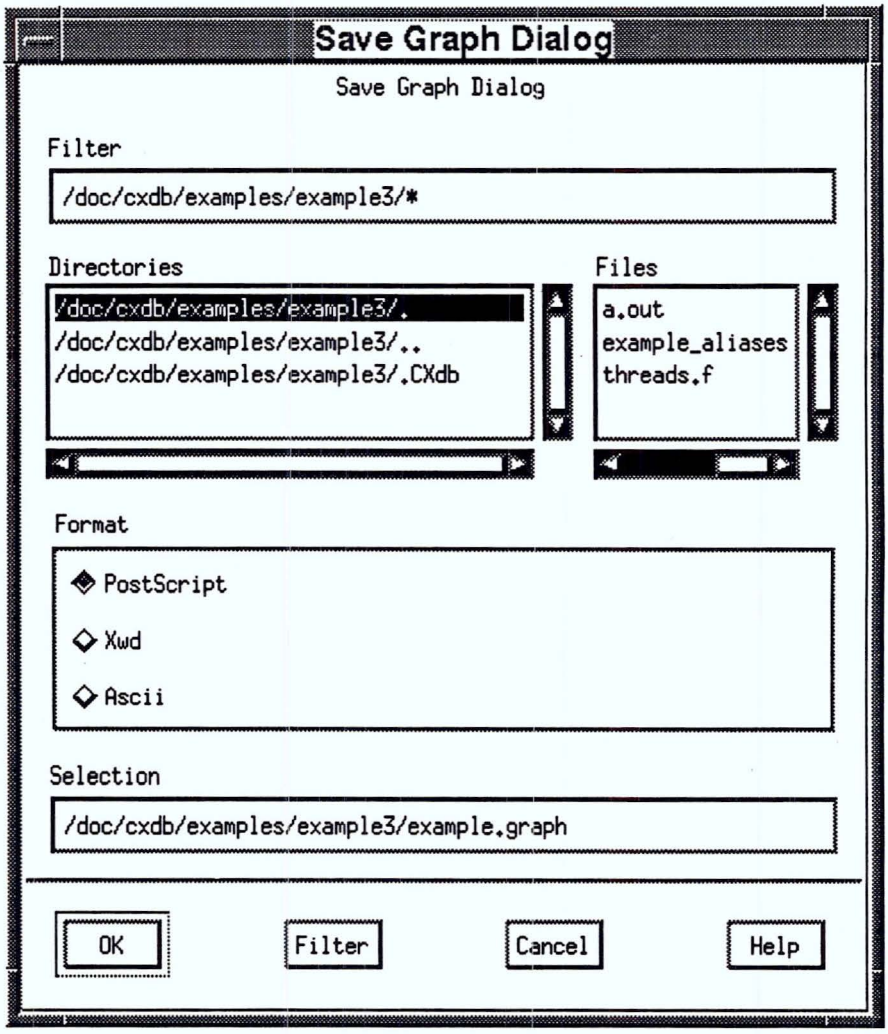
Context

The Routine Name dialog appears when you select the New routine option from the FileView menu in the Source Code window.

Related Windows

Source Code window

Save Graph dialog



Description

The Save Graph dialog enables you to save the 2-D thread activity graph to a file for later viewing or printing. You can specify either PostScrip, xwd, or ASCII format. The resulting PostScript, xwd, or ASCII file contains the entire graph, no matter what portion of the graph is displayed in the Thread Activity window.

Filtering files

When the file selection box appears, the Filter field contains the path to the current directory with *.graph appended to it. To use the filter feature:

1. Modify the path name in the Filter field by typing in the field or by clicking on one of the directories. You can use wildcards (*. or ?).
2. Press the Filter button. The Files list displays files, and the Directories list displays directories that match the specification in the Filter field (initially, *.graph).

Saving the Thread Activity graph to a file

Use the following procedure to save the Thread Activity graph to a file:

1. Specify the file name you want to save the output to, using one of the following methods:
   - Click the left mouse button on a file name in the Files list. The file name is highlighted and displayed in the Selection field.
   - Type the full path name to the file in the Selection field.
2. Select a file format by clicking on one of the Format radio buttons. You can specify either PostScript, Xwd, or ASCII.
3. Click OK or press RETURN.

Fields

Name	Value
Filter	Contains a path name, usually containing wildcards, that you set to determine which files and directories appear in the Files and Directories lists.
Directories	Lists all subdirectories in the directory specified in the Filter field.
Files	Lists all files in the selected directory in the Diresctories list that match the specified filter.
Selection	Contains the full path and file name that you want to save the Thread Activity graph in.

Buttons

Name	Action
OK	Closes the dialog and saves the Thread Activity graph to the file and format you have specified.
Filter	Updates the file and directory names displayed in the Files and Directories lists to match the specification in the Filter field.
Cancel	Closes the dialog without saving the Thread Activity graph to a file.
Help	Displays the "Save Graph dialog" help topic in the CXdb Help window.

Context

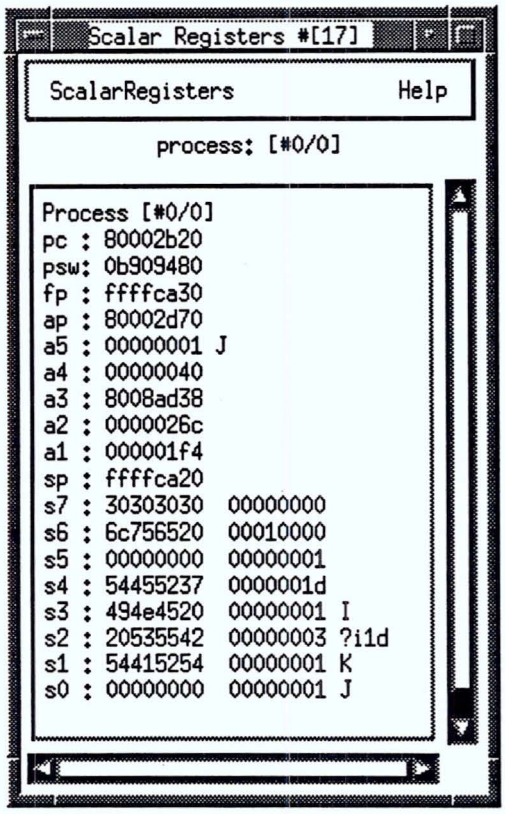
The Save Graph dialog appears when you select the Save item from the File menu in the Thread Activity window.

Related Windows

Thread Activity window

C Series only

Scalar Registers window



Description

The Scalar Registers window displays the contents of the address and scalar registers for the current thread of the specified process. By default, the display is in hexadecimal format.

If a variable has been loaded into a register, the name of the variable appears beside that register. This can be a program variable (for example, `I` or `NUMARGS`) or a compiler-generated synthesized variable (such as `?i1d`).

You can specify a different display format for the window, enable and disable automatic updating, and specify which threads are visible in the window.

Scalar Registers window

The registers displayed are:

- Program counter (`pc`)
- Processor status word (`psw`)
- Address registers (A0 through A*n*)—Three of the address registers have special names:
  - Register A7 is called the frame pointer (`fp`).
  - Register A6 is called the argument pointer (`ap`).
  - Register A0 is called the stack pointer (`sp`).
- Scalar registers (S0 to S*n*)
- Scalar stride registers (SS0 and SS1) — C4 Series machines only. C4600 Series CPUs contain two 32-bit scalar stride registers, SS0 and SS1. The `ld0` and `ld1` instructions use these registers to permit explicit cache prefetching under software control. This mechanism greatly improves the data cache hit rate for non-vectorizable routines operating on large data sets.

Menus

Name	Description
ScalarRegisters	Contains the following items: **Close**—Closes the window. **Auto update**—Enables and disables automatic screen updating. **Change format**—Brings up a Format dialog where you can specify a different display format for the values displayed in the Scalar Registers window. **threads**—Pops up a Threads dialog where you can specify which thread is visible in the Scalar Registers window.
Help	Contains options for invoking the CXdb Help system and displaying product information. Refer to the "Help menu" reference page or online help topic for more information.

Context

The Scalar Registers window appears when you select Create window, then select the Scalar Registers item from the CXdbWindows menu in either the Command window or the Source Code window.

Related Windows

Format dialog
Threads dialog
Help menu

Related Concepts

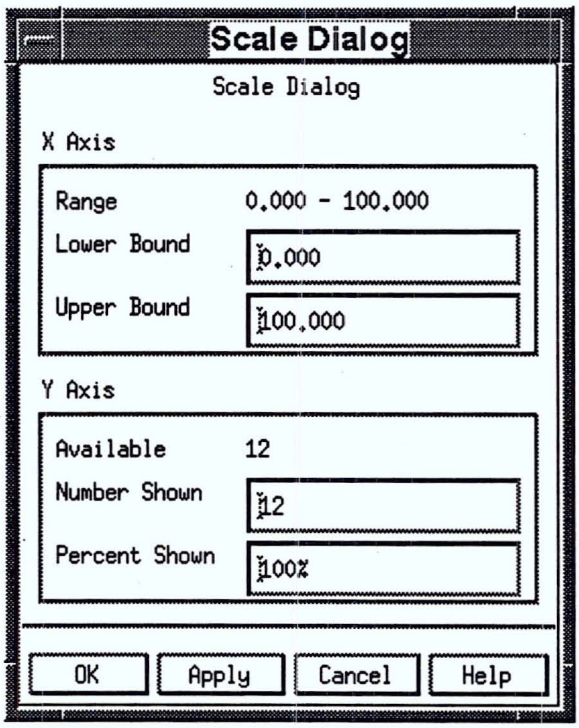
registers

Related Commands

`info registers`

Scalar Registers window

Scale dialog



Description

Use the Scale dialog to view or specify the range or number of data items displayed at one time in the Thread Activity window 2-D graph. This is especially useful in cases where there are a large number of data items (for example, files, routines, or threads) to graph and you want to focus on a subset of the data.

X-Axis scaling

The X Axis panel shows the range of X axis data values that are currently displayed in the 2-D graph—in this case, the number or percentage of threads in your program. You can limit the range of items displayed at one time in the window by specifying a lower and/or upper bound. When you scroll the X axis, the Lower Bound and Upper Bound fields update to reflect the range of values displayed in the graph.

Scale dialog

Y-Axis scaling

The Y Axis panel displays the total number of available items to graph on the Y-Axis—in this case, files or routines. You can limit the number of Y-Axis items displayed at one time in the window by specifying either a number or a percentage of items to display. If you change the number of items shown, the Percent Shown field updates to reflect the change; if you change the percentage of items shown, the Number Shown field updates to reflect the changes.

X Axis

Field	Description/Valid values
Range	Lists the range of data values that can be displayed in the graph.
Lower Bound	Sets the lower-bound of the X-axis. This value must be less than the value specified in the Upper Bound field, but can be smaller than the minimum data value.
Upper Bound	Sets the upper-bound of the X axis. This value must be greater than the value specified in the Lower Bound field, but can be larger than the maximum data value.

Y Axis

Name	Description/Valid values
Available	Shows the total number of available files or routines that can be displayed in the graph.
Number Shown	Shows the number of routines or files currently displayed. Valid values are integers from 1 to the number available, inclusive. Values larger than the number available are ignored.
Percent Shown	Shows the percentage of available routines or files currently displayed on the Y-axis. Valid values are percentages from 0 to 100. Values larger than 100 are ignored.

Buttons

Name	Action
OK	Applies the changes and closes the dialog.
Apply	Applies the changes without closing the dialog.
Cancel	Closes the dialog without making changes.

Help	Invokes the CXdb Help system and displays the help topic for the Scale dialog.

Context

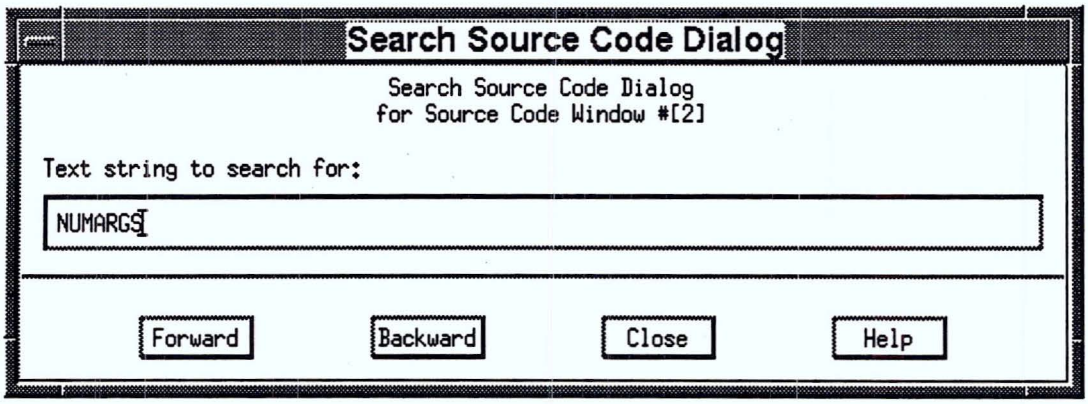
The Scale dialog appears when you select the Scale... item from the View menu in the Thread Activity window.

Related Windows

Sort dialog

Thread Activity window

Search Source Code dialog



Description

The Search Source Code dialog enables you to search backward or forward for a specified text string in the text that is currently displayed in the associated Source Code window. The number of the associated Source Code window is displayed in brackets (# [ ]) at the top of the dialog.

Searches are always peformed on the text that is currently shown in the Source Code window, even after you have selected a new file or routine to display.

When a match is found, it is highlighted in the Source Code window. If no match is found, a message is displayed in the Command window.

Fields

Name	Description
Text string to search for:	The value entered in this field can be any sequence of ASCII characters. Wildcard characters such as ? and * are interpreted literally (that is, not expanded). The text string can include white space (tabs or spaces).

Search Source Code dialog

Buttons

Name	Action
Forward	Executes a forward search for the specified text string through the file that is shown in the associated Source Code window, stopping at the first match. The matching text string is highlighted. Each time you click the Forward button, the search continues forward from the location where the last match occurred or from the beginning of the file, if there was no prior match. The search wraps from the end of the file to the beginning of the file, if necessary
Backward	Executes a backward search for the specified text string through the file that is shown in the associated Source Code window, stopping at the first match. Each time you click the Backward button, the search continues backward from the location of the last match, or from the end of the file, if there was no prior match. The search wraps from the beginning of the file to the end of the file, if necessary.
Close	Closes the dialog.
Help	Displays the "Search Source Code dialog" help topic in the CXdb Help window.

Context

The Search Source Code dialog appears when you select the Search Source Code item from the FileView menu in the Source Code Window.

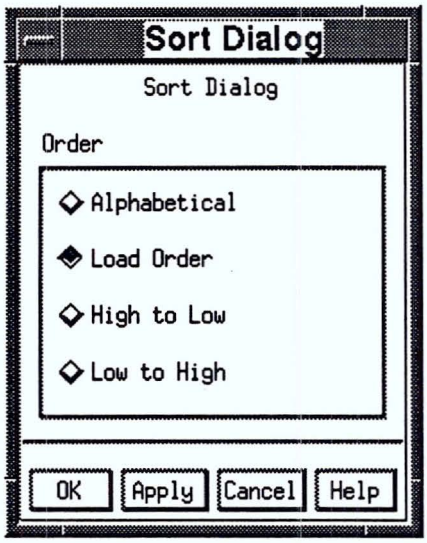
Related Windows

FileView menu Source Code window

Related Commands

`find source`

Sort dialog



Description

Use the Sort dialog to specify the ordering of the files or routines displayed on the Y-axis in the Thread Activity window 2-D graph. You can list them in alphabetical order, in the order specified by the loader (default), from highest to lowest thread activity, or from lowest to highest thread activity.

Clicking OK applies the changes and closes the dialog; clicking Apply allows you to apply the changes without closing the dialog.

Order

Choice	Action
Alphabetical	List files or routines on the Y-axis in alphabetical order.
Load Order	List files or routines on the Y-axis in the order specified by the loader.
High to Low	List files or routines from highest thread activity (number/percentage of active threads) to lowest.

Sort dialog

Low to High	List files or routines on the Y-axis from lowest thread activity (number/percentage of active threads) to highest.

Buttons

Name	Action
OK	Sorts the files or routines in the specified order and closes the dialog.
Apply	Sorts the files or routines in the specified order, but does not close the dialog.
Cancel	Closes the dialog without making any changes.
Help	Invokes the CXdb Help system and displays the Help topic for the Sort dialog.

Context

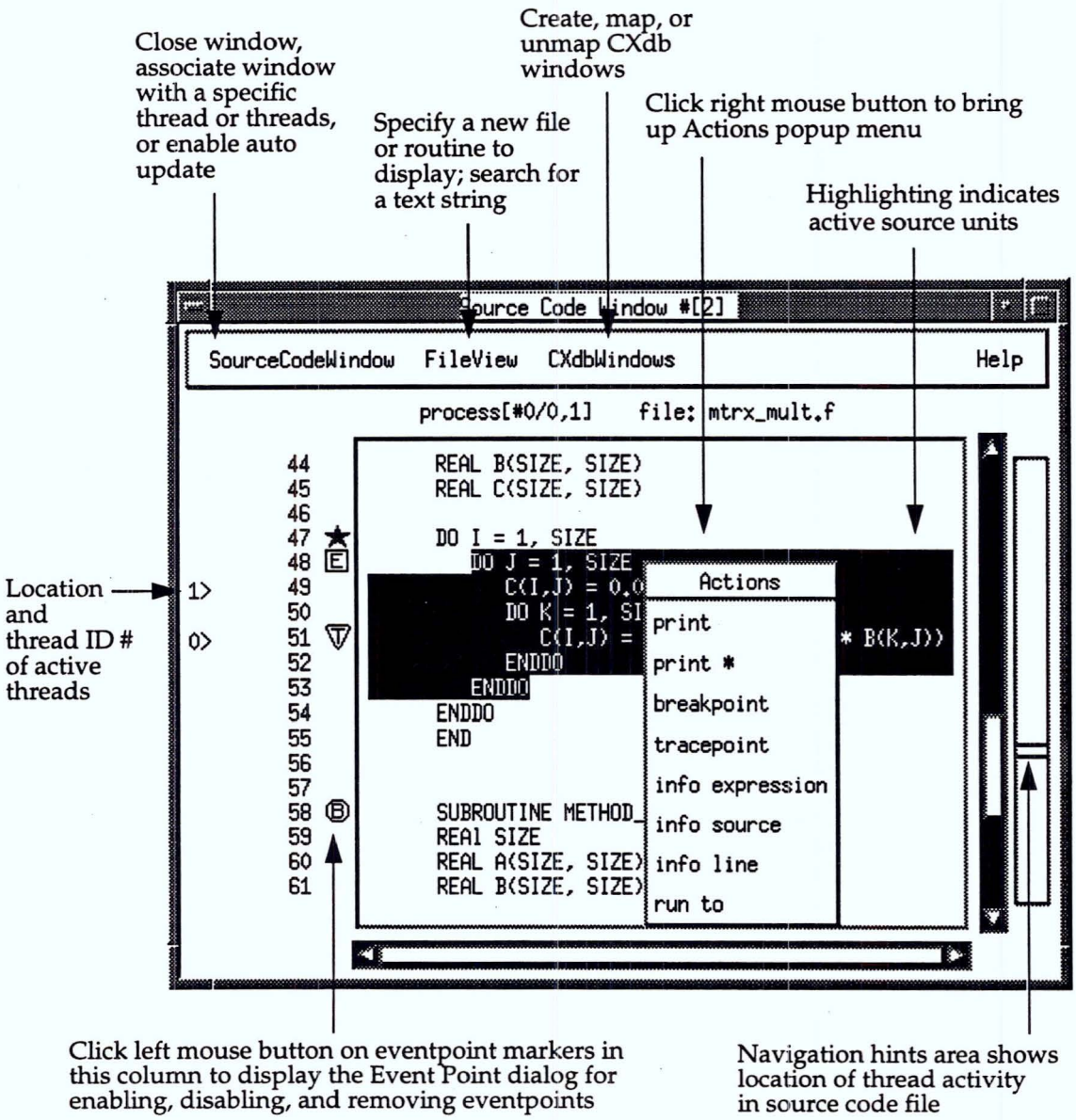
The Sort dialog appears when you select the Sort... item from the View menu in the Thread Activity window.

Related Windows

Scale dialog

Thread Activity window

Source Code window



Source Code window

Description

The Source Code window displays source code files associated with the executable file (specified with the `debug exec`, `debug core` or `executable` command). You can have multiple Source Code windows open during a debugging session. CXdb adds line numbers to identify each line in the source file.

The Source Code window also highlights active source units and displays markers to indicate the location of breakpoints, tracepoints, multiple eventpoints, and the current point of execution for each thread of your program. When the Source Code window first opens, all active threads are displayed, and automatic updating is enabled.

In the Source Code window, you can:

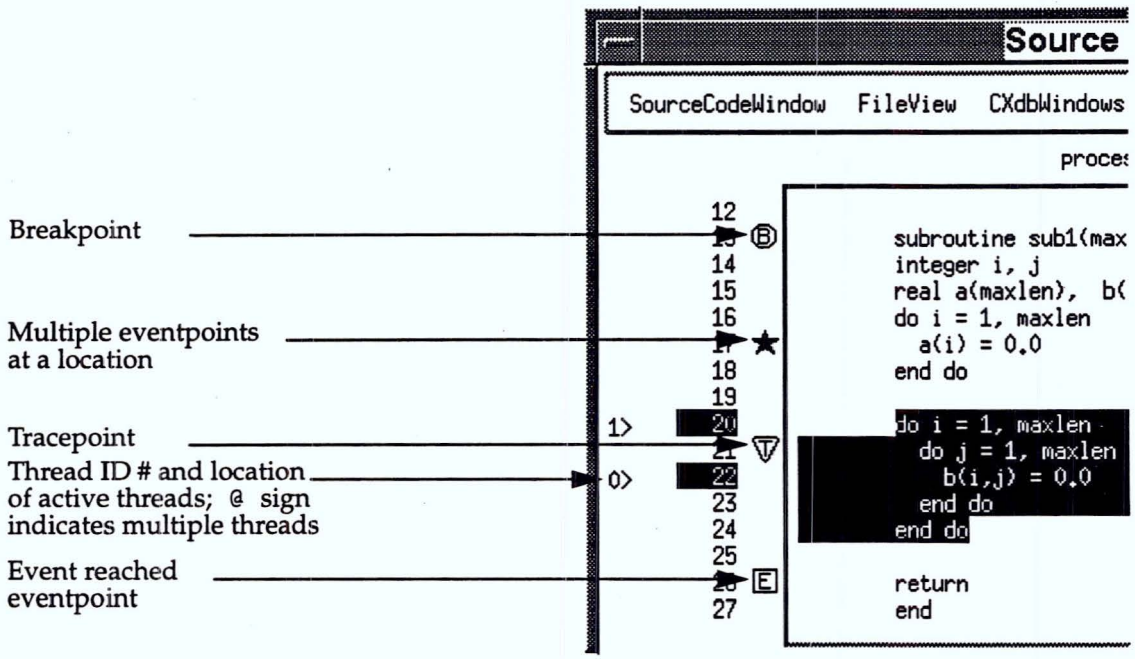
- Specify another file or routine in your program to display in the Source Code window
- Use the navigation hints area at the far right side of the Source Code window to locate source code that corresponds to active threads in your program. To view the source code at the location of the active thread, align the scroll bar with the horizontal bar in the Navigation hints area.
- Click with the right mouse button to bring up an Actions popup menu that allows you to:
  - Enable, disable, or remove eventpoints
  - Print the value of a variable or expression
  - Set a breakpoint or tracepoint
  - Run to a specific location
  - Display information about source units
  - Display information about an expression
  - Execute the `info line` command

  Refer to the "Using the actions popup menu" section of this reference topic for more information.
- Press **s** to execute the `step` command
- Press **n** to execute the `next` command
- Create new windows that display different types of information about the current process
- Search for a text string in the Source Code window by selecting the Search Source Code... item from the FileView menu

For more information about using mouse and keyboard shortcuts in the Source Code window, refer to the "mouse and keyboard shortcuts" reference topic.

Identifying symbols in the Source Code window

The following figure shows the location of markers and highlighting in the Source Code window.



You can click with the left mouse button on any eventpoint marker to bring up an Event Point Dialog where you can disable, enable, or remove an eventpoint.

Thread markers show the thread ID number and location of active threads in your program. If an `@ sign` appears in place of the >, then multiple threads are active at that location, and the number preceding the `@` indicates the number of active threads.

Highlighting in the Source Code window shows the location of active source units in your program. The level of detail in the highlighting is affected by the stepping granularity (step size) that you have specified. To specify a different step size, use the `set step` command. To determine the current stepping granularity, use the `info cxdb` command.

Specifying a different file or line number to display

To display a different file in the Source Code window or to display a different location in the current file:

1. Click on the FileView menu and select New file or line number. This brings up a File or Line number dialog.
2. Specify a file or line number in the File Name and/or Line Number field. Refer to the "File or Line Number dialog" reference topic for more information.
3. Click OK to display the new source file and close the dialog or click Cancel to dismiss the dialog without making any changes.

Specifying a different routine to display

1. Open the FileView menu and select New file or line number. This brings up a File or Line number dialog.
2. Fill in the Routine name or language expression field.

   The value entered in this field can be a routine name or any language expression in the source language that evaluates to a valid address within the bounds of your process.
3. Click OK to close the dialog.

   If a valid address is specified, the source code of the named routine or the routine that contains the specified address is displayed in the Source Code window.

   If the address specified is invalid, you must either specify a different address or click Cancel to close the dialog without making changes.

Enabling, disabling, and removing eventpoints

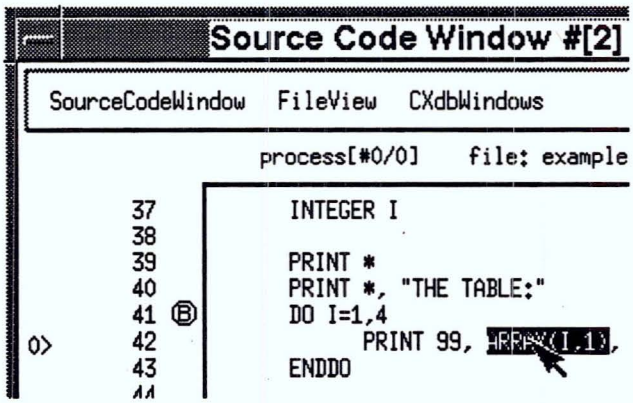
To enable, disable, or remove an eventpoint:

1. Position the mouse cursor over the eventpoint marker for the eventpoint (for example, a breakpoint or a tracepoint) you want to manipulate.
2. Click the left mouse button to bring up the Event Point Dialog box.
3. Click on the appropriate button to enable, disable, or remove the eventpoint, then click the Close button to close the dialog.

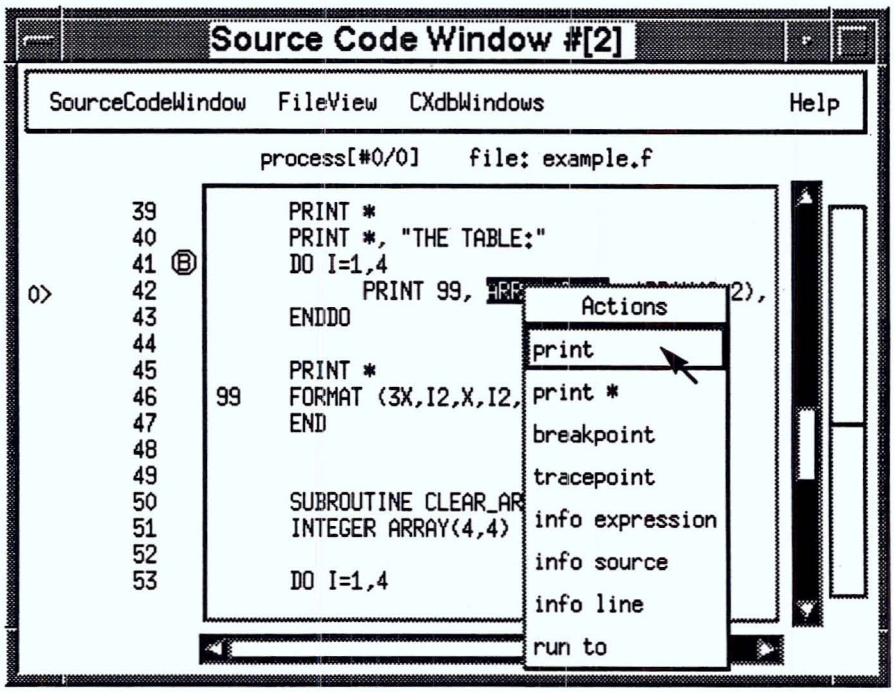
Using the actions popup menu

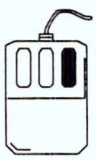
To use the actions popup menu:

1. Position the mouse over the source unit you want to perform the action on. .

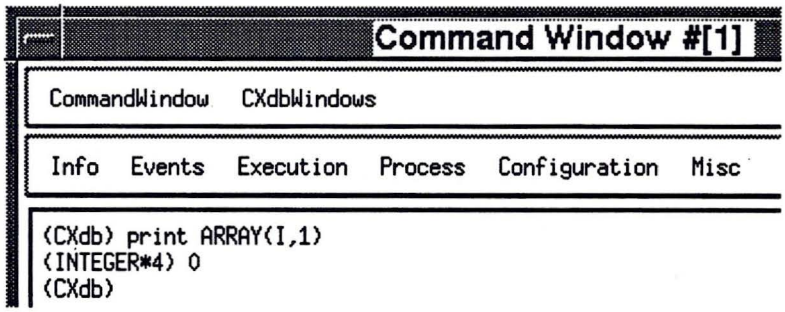


2. Click the right mouse button. This brings up the actions popup menu and highlights the source unit at the location of the mouse pointer. Drag the mouse to select the menu item that corresponds to the action you want to perform.





3. When you release the mouse button, CXdb performs the action (command) you selected using the highlighted source unit as its argument. The resulting command and output is displayed in the Command window.



The actions menu contains the following items:

- **print** — If the highlighted source unit is a valid language-expression, performs the `print` command on the highlighted source unit.
- **print * (print indirect)** — If the highlighted source unit is a C pointer variable, selecting this item prints the value referenced by the pointer (equivalent to `print *(`*ptr*`)`).
- **breakpoint** — Sets a breakpoint at the highlighted line number or source unit.
- **tracepoint** — Sets a tracepoint at the highlighted line number or source unit.
- **info expression** — If the highlighted source unit is a valid language-expression, performs the `info expression` command on the highlighted source unit.
- **info source** — Performs the `info source` command on the highlighted source unit.
- **info line** — Performs the `info line` command on the line containing the highlighted source unit.
- **run to** — Sets a breakpoint at the line number or source unit the mouse pointer is over, continues execution, stops execution at the breakpoint, and then removes the breakpoint.

Controlling automatic creation of Source Code windows

By default, a Source Code window is automatically created when you specify an executable file to debug using the `debug exec`, `debug core` or `executable` command. You can disable autocreation by:

- Toggling the autocreate option of the CommandWindow menu, located on the main menubar of the CXdb Command window
- Executing the `clear autocreate` command. (Use the `set autocreate` command to enable autocreation.)
- Specifying the `-ns` option when you invoke CXdb from the shell prompt with the `cxdb` command
- Putting the following line in your .Xdefaults file:

```
Cxdb.autocreate:        False
```

Menus

Item	Description
SourceCodeWindow	Contains items to close the Source Code window, toggle Auto update of the information in the Source Code window, and to select which threads are visible in the window. Refer to the "SourceCodeWindow menu" reference page or online help topic for more information.
FileView	Contains items to return the Source Code window display to the current point of execution, to specify another file or routine to view, and to search for a text string in the Source Code window. Refer to the "FileView menu" reference page or online help topic for more information.
CXdbWindows	Contains items for opening and controlling the visibility of other windows that display information about the current process. Refer to the "CXdbWindows menu" reference page or online help topic for more information.
Help	Contains items for invoking the CXdb Help system. Refer to the "Help menu" reference page or online help topic for more information.

Source Code window

Context

A Source Code window appears:

- Automatically, when you specify an executable to debug using the `debug exec`, `debug core`, or `executable` command, if the autocreate option is enabled
- When you select Create Window, then select the Source Code item from the CXdbWindows menu in either the Command window or the Source Code window
- When you execute the `display routine` command from the Command window
- When you execute the `display source` command

Related Windows

Event Point dialog
File or Line Number dialog
Routine Name dialog
Threads dialog

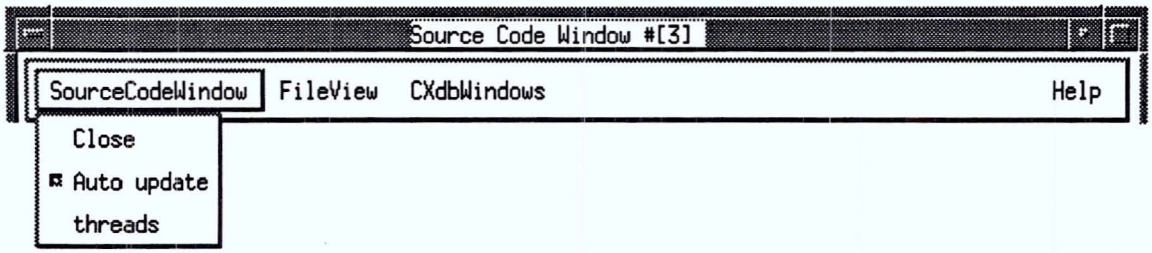
Related Menus

CXdbWindows menu
FileView menu
Help menu
SourceCodeWindow menu

Related Commands

`clear autocreate`
`debug exec`
`display routine`
`display source`
`executable`
`find source`
`set autocreate`

SourceCodeWindow menu



Description

The SourceCodeWindow menu contains items for closing the Source Code window, enabling or disabling automatic updating of information displayed in the Source Code window, and specifying which threads are visible.

Menu Items

Item	Action
Close	Closes the Source Code window.
Auto update	Toggles automatic updating of information displayed in the window. The default is enabled.
threads	Opens a Threads dialog for specifying which threads are visible in the Source Code window. By default, all threads of the current process are displayed.

Context

The SourceCodeWindow menu appears when you select SourceCodeWindow from the main menubar in the Source Code window.

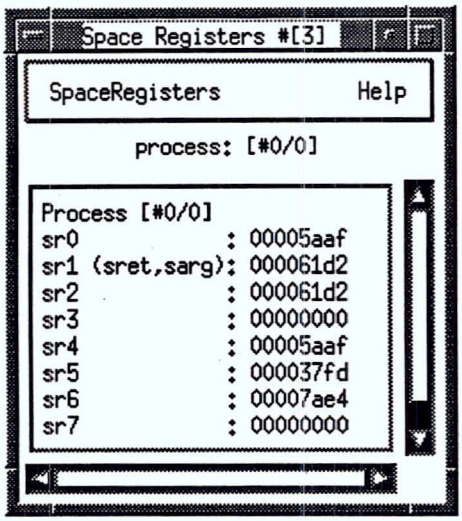
Related Windows

Source Code window

Threads dialog

SPP Series only

Space Registers window



Description

The Space Registers window displays the contents of the space registers for the specified process. There are 8 space registers, designated as sr0 through sr7. Register sr1 is also called the space argument (sarg) or space return (sret) register. The contents are displayed in hexadecimal format.

You can specify a different display format for the window, enable and disable automatic updating, and specify which threads are visible in the window.

For more information about these registers, refer to the *PA-RISC 1.1 Architecture and Instruction Set Reference Manual* and the *PA-RISC Procedure Calling Conventions Reference Manual*, both available from Hewlett-Packard.

Menus

Name	Description
SpaceRegistersWindow	Contains the following items:
	Close — Closes the window.
	Auto update — Enables and disables automatic screen updating.

	Change format — Brings up a Format dialog where you can specify a different display format for the values displayed in the Space Registers window. **threads** — Brings up a Threads dialog where you can control which thread is visible in the Space Registers window.
Help	Contains items for invoking the CXdb Help system. Refer to the "Help menu" reference page or online help topic for more information.

Context

The Space Registers window (SPP Series systems only) appears when you select Create window, then select the Space Registers item from the CXdbWindows menu in either the Command window or the Source Code window.

Related Windows

Control Registers window
Floating Point Registers window
Format dialog
General Registers window
Processor Status Word window
Space Registers window
Threads dialog

Related Menus

Help menu

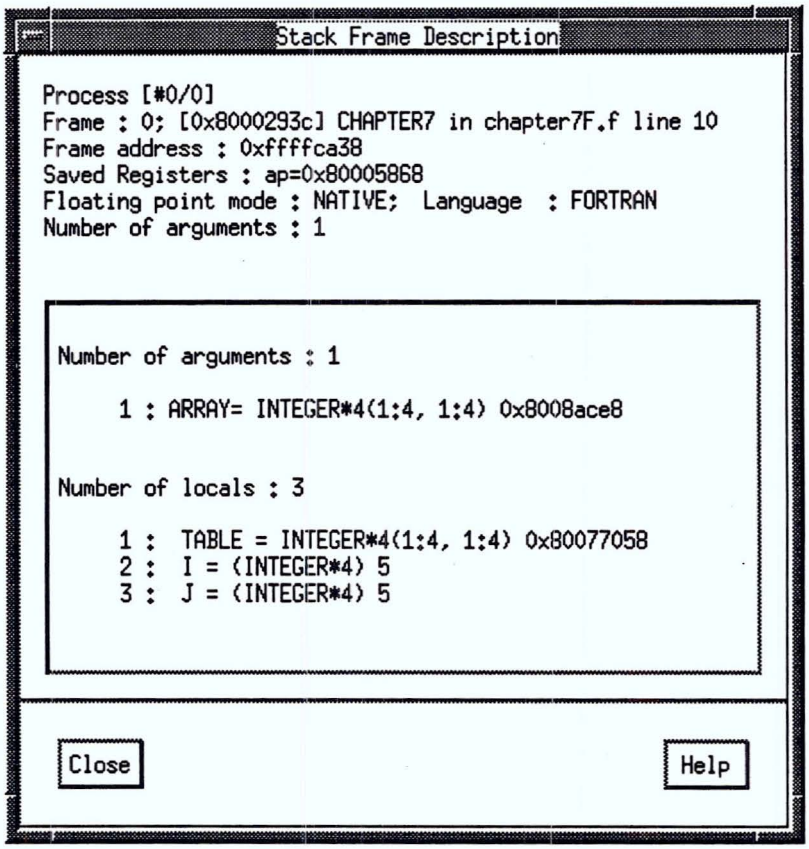
Related Commands

`info space registers`

Related Concepts

registers

Stack Frame Description dialog



Description

The Stack Frame Description dialog displays the following detailed information about a specified frame:

- **Process** — Process number and thread ID number for the specified frame.
- **Frame** — Frame number and value of the program counter (pc) for the frame and, when applicable, the line number, routine name, and file name for the point of execution.
- **Frame address** — Starting address of the area of memory where the frame is located.

- **Saved registers** — Contents of the registers saved in the frame.
- **Floating point mode** — Mode for performing floating point operations.
- **Language** — Source language for the routine represented by the frame.
- **Routine return type** — Data type for the value returned by the routine. Only displayed if the routine returns a value.
- **Number of arguments** — Number of arguments passed to the routine. For each argument, the argument name, data type, and current value (or, for arrays, the number of elements and starting address) are displayed.
- **Number of locals** — Number of local variables of the routine for the specified frame. For each local variable, the name, data type, size, and value is displayed, if applicable. For arrays, the number of elements and starting address are displayed.

SPP Series only

On SPP Series systems, the following additional information is displayed in the Stack Frame Description window:

```
Unwind Table Entry :
   region_start                 : 0x000257c0
   region_end                   : 0x000259e4
   Millicode                    : 0
   Millicode_save_sr0           : 0
   Region_description           : 0
   reserved1                    : 0
   Entry_SR                     : 0
   Entry_FR                     : 0
   Entry_GR                     : 0
   Args_stored                  : 0
   Variable_Frame               : 0
   Separate_Package_Body        : 0
   Frame_Extension_Millicode    : 0
   Stack_Overflow_Check         : 0
   Two_Instruction_SP_Increment : 0
   Ada_Region                   : 0
   reserved2                    : 0
   Save_SP                      : 0
   Save_RP                      : 1
   Save_MRP_in_frame            : 0
   reserved3                    : 0
   Cleanup_defined              : 0
   MPE_XL_interrupt_marker      : 0
   HP_UX_interrupt_marker       : 0
   Large_frame_r3               : 0
   reserved4                    : 0
   Total_frame_size             : 0x20
```

Stack Frame Description dialog

The Unwind Table Entry provides information about the stack size, register usage, and lengths of the entry and exit sequences for each unwind region, typically an entire procedure. Refer to the *PA-RISC Procedure Calling Conventions Reference Manual For the HP-UX and MPE XL Operating Systems, HP 9000 Series 600/700/800 and HP 3000 Series 900 Computer Systems*, available from Hewlett-Packard, for a detailed description of stack unwind descriptors.

Buttons

Name	Action
Close	Closes the dialog.
Help	Displays the "Stack Frame Description dialog" help topic in the CXdb Help window.

Context

To open a Stack Frame Description dialog, perform the following steps:

1. Open a Stack Trace window by selecting Create window, then selecting the Stack Trace item from the CXdbWindows menu in either the Command window or the Source Code window.
2. Click with the left mouse button anywhere on the line describing the stack frame you want information for.

Related Windows

Command window

Stack Trace window

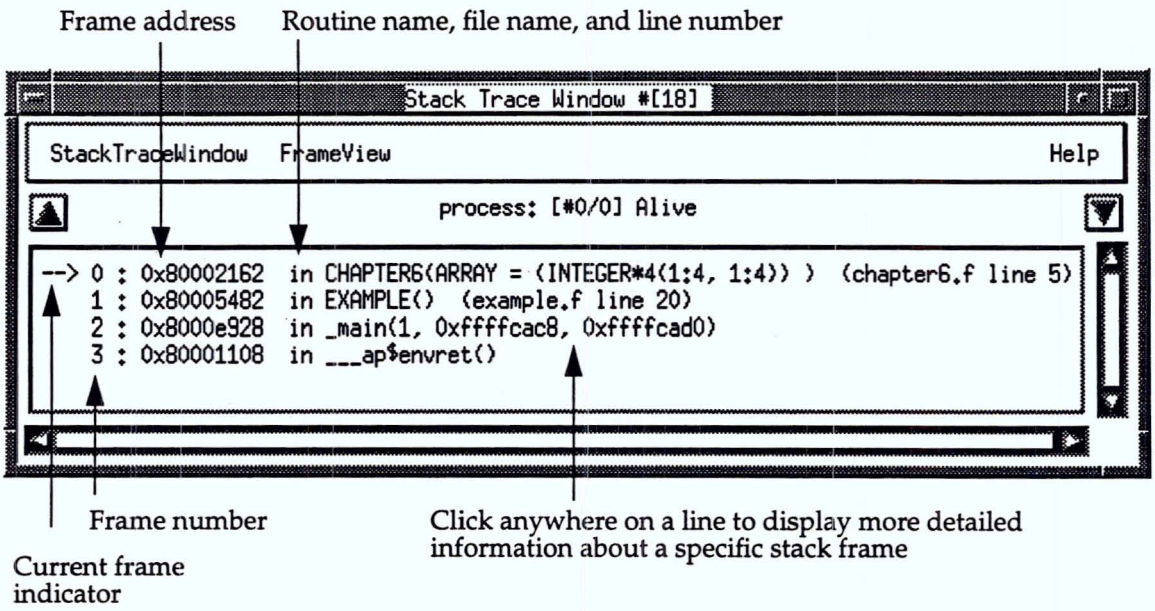
Related Commands

```
frame
info frame
```

```
info args
info locals
```

Stack Frame Description dialog

Stack Trace window



Description

The Stack Trace window displays the frames of the program stack. The --> symbol indicates the current frame. The information displayed includes the frame number, frame address, routine name, file name, and line number.

You can use the Stack Trace window to track the state of the program stack and to obtain information about the frames on the stack.

Initially, the Stack Trace window updates to reflect the current state of the stack each time process execution stops. Using menus and the mouse, you can:

- Change the current frame
- Specify which threads are visible in the window
- Get more detailed information about a particular frame
- Enable or disable automatic updating

Changing the current frame

Using the options under the FrameView menu you can select a particular frame from the process stack to be the current frame. The current frame defines the current scope for the process.

When you specify a different frame as the current frame, you are changing the context of the process for symbol mapping. Any unqualified identifiers used in subsequent CXdb commands are interpreted from the scope of the specified current frame. However, the context for process execution is not changed. Execution of the process still continues from the top of the stack, which is frame 0.

Getting more detailed information about stack frames

To display more detailed information about a specific frame, you can open aStack Frame Description dialog by clicking with the left mouse button anywhere on the line describing the stack frame in the Stack Trace window.

Using keyboard shortcuts in the Stack Trace window

When the mouse pointer is in the Stack Trace window, you can quickly change stack frames by pressing the following keys:

- **0** through **9** — Changes the current frame to the number pressed (executes a frame <*n*> command).
- **u** — Moves up the stack one frame at a time (executes a `frame +1` command).
- **d** — Moves down the stack one frame at a time (executes a `frame -1` command).
- **t** — Makes the top frame the current frame (executes a `frame 0` command).

Menus

Name	Description
StackTraceWindow	Contains the following options: **Close** — Closes the window. **Auto update** — Enables or disables automatic updating of window contents. **threads** — Pops up a Threads dialog where you can associate the Stack Trace window with a particular thread.

FrameView	Contains the following options: **Next frame** — Executes a `frame -1` command, selecting the next frame on the stack as the current frame. **Previous frame** — Executes a `frame +1` command, selecting the previous frame on the stack as the current frame. **Top of Stack** — Resets the current frame to frame 0 (executes a `frame 0` command).
Help	Contains items for invoking the CXdb Help system. Refer to the "Help menu" reference page or online help topic for more information.

Context

The Stack Trace window appears when you:

- Select Create window, then select the Stack Trace item on the CXdbWindows menu in either the Command Window or the Source Code window.
- Enter the `display stack` command at the `(CXdb)` prompt in the Command window.

Related Windows

Stack Frame Description dialog Threads dialog

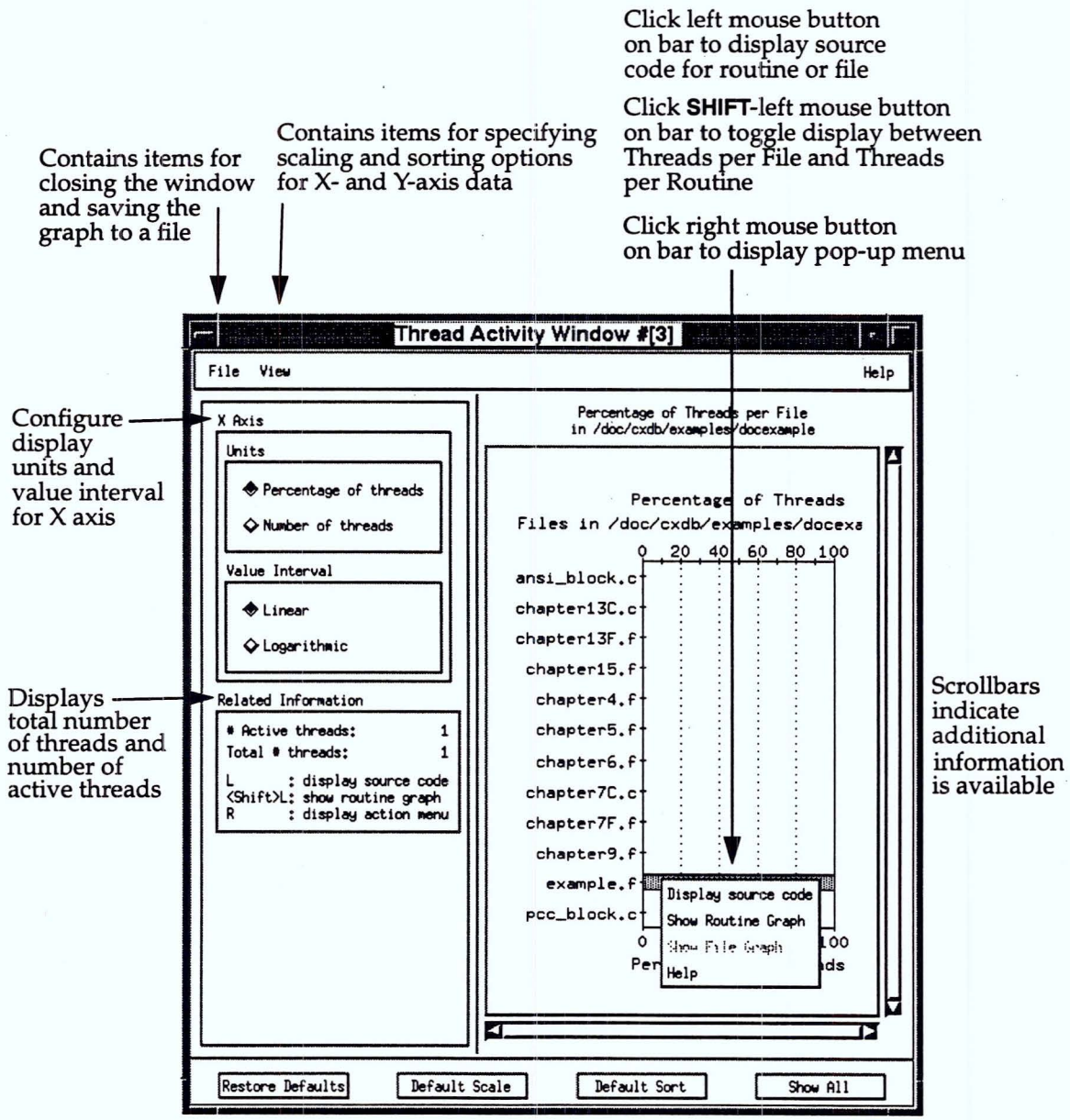
Related Menus

Help menu

Related Commands

`backtrace`
`display stack`
`info frame at`
`frame`
`info frame`

Thread Activity window



Thread Activity window

Description

The Thread Activity window displays a 2-D graph of thread activity as it relates to the source code of your program. Using this window, you can identify and display source files or routines associated with active threads in your program. This is especially useful when debugging programs with routine-level and asymmetric parallelism. The thread activity graph is updated when the process stops.

You can configure scaling and ordering options for the Y axis, as well as display units and value intervals for the X axis. Restore Defaults, Default Scaling, Default Sorting, and Show All buttons enable you adjust the graph view. You can save the 2-D thread activity graph to a file in PostScript, xwd, or ASCII format.

The bars in the 2-D graph are mouse-sensitive, allowing you to:

- Display source code in the Source Code window for a routine or file by clicking the left mouse button on the corresponding bar.
- Toggle the graph display between threads per routine in file or threads per file in program by holding down the **SHIFT** key and clicking the left mouse button on any bar in the graph.
- Bring up an actions popup menu by clicking the right mouse button on any bar in the graph. Items you can select from this menu are Display Source Code, Show Routine Graph, Show File Graph, and Help.

Changing the graph from Threads per File to Threads per routine

To toggle the thread activity graph between Threads per File and Threads per Routine, hold down the **SHIFT** key and click the left mouse button on any bar in the graph:

- If Threads per File are currently displayed, this displays the graph for the routines in the file represented by the bar you clicked on.
- If Threads per Routine are currently displayed, this displays the graph for the files in the program.

You can also toggle the graph display by clicking the right mouse button on any bar in the graph and selecting either Show Routine Graph or Show File Graph from the actions pop-up menu, depending on which is currently displayed.

Displaying related source code

To display the source code corresponding to a file or routine, click the left mouse button on the bar in the graph that corresponds to the file or routine you want to view.

Saving the thread activity graph to a file

To save the thread activity to a file for later viewing or printing, select Save from the File menu. This brings up a Save Graph dialog where you can specify the name of the file you want to save the graph to.

You can save the file in PostScript, xwd, or ASCII format. Refer to the "Save Graph dialog" reference topic for more information.

Changing the order of files or routines displayed on the Y-axis

To select a different ordering option for theY axis, select the Sort... item from the View menu to bring up the Sort dialog, then select the desired option.

You can order theY axis according to:

- Order specified by the loader (default)
- Alphabetical order
- Thread activity level (from high to low or from low to high)

Click OK to apply the changes and close the Sort dialog, or click Apply to apply the changes without closing the dialog.

Changing display units on the X-axis

You can graph thread activity by the number of threads in a file or routine or as a percentage of the total number of threads in your program per file or routine.

To change the display unit on the X-axis, click the appropriate button (either Percentage of threads or Number of threads) in the Units area of the X-axis configuration options area.

Changing range and scaling options (X and Y axes)

To specify range and scaling options for the number or percentage of items displayed at one time on the X and Y axes in the Thread Activity, perform the following steps.

1. Select the Scale... item from the View menu.This brings up the Scale dialog.
2. Specify an Upper and/or Lower Boundary for the range of items to display on the X axis.
3. Specify the number or percentage of available items to display at one time on the Y axis.
4. Click OK to apply the changes and close the dialog, or click Apply to appy the changes without closing the dialog.

Refer to the "Scale" dialog" online help topic or reference page for more information.

Changing the X-axis value interval

You can plot data values on the X-axis in a straight linear progression or as a logarithmic progression. You may want to use logarithmic value intervals when you wish to examine areas of a large program where only a tiny percentage of threads are active, as compared to large clusters of thread activity elsewhere in the program.

To change X-axis data value intervals, click the Linear or Logarithmic button in the Value Interval area of the X Axis configuration options panel in the Thread Activity window.

Menus

Name	Items
File	Contains the following items: **Save** — Brings up the Save Graph dialog, which allows you to save the 2-D thread activity graph to a file in either PostScript, xwd, or ASCII format. **Close** — Closes the Thread Activity window.
View	Contains the following items: **Sort...** — Brings up the Sort dialog, where you can specify a different ordering for the files or routines displayed on the Y axis.. Refer to the "Sort dialog" reference page or online help topic for more information. **Scale...** — Brings up the Scale dialog, where you can specify a range, number, or percentage of items to display on the X and Y axes. Refer to the "Scale dialog" reference page or online help topic for more information.
Help	Contains items for invoking the CXdb Help system. Refer to the "Help menu" reference page or online help topic for more information.

Buttons

Name	Action
Restore Defaults	Restore default scaling and sorting options.
Default Scale	Restore defult scaling options only.

Default Sort	Restore default sorting options only.
Show All	Graphs all data items and values.

Context

The Thread Activity window appears when you select the Create window item from the CXdbWindows menu in either the Command window or the Source Code window, then select Thread Activity.

Related Concepts

optimized code

threads

Related Commands

`clear default fixed sched`
`event join`
`event spawn`
`info threads`
`set default fixed sched`
`set fixed sched`
`set threads`

Related Menus

Help menu

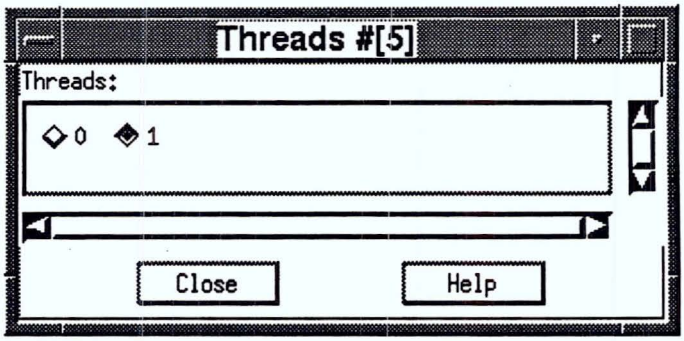
Related Windows

Threads dialog
Save Graph dialog
Scale dialog
Sort dialog

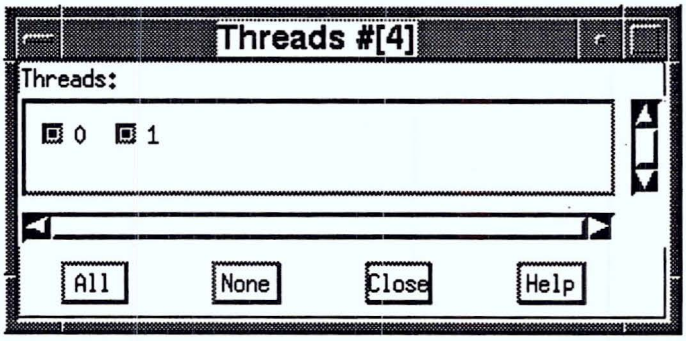
Thread Activity window

Threads dialog

Radio buttons indicate that you can only select and view information for one thread at a time in the associated window.



Selector buttons indicate that you can select and view information for multiple threads in the associated window.



Description

The Threads dialog enables you to select which threads are visible in an associated CXdb window. The Threads dialog title bar displays the number of the primary CXdb window associated with the dialog (the window from which the threads menu item was selected).

You can open a Threads dialog by selecting the threads item from the first (leftmost) menu in CXdb primary windows and register windows.

Toggling the selector buttons in the Threads dialog selects and deselects visible threads in the window. Numbers for threads that are not active in the current process are stippled out and cannot be selected. Use the horizontal and vertical scroll bars to scroll the dialog.

Threads dialog

Buttons

Name	Description
All	Makes all active threads visible in the associated CXdb window.
None	Deselects all threads.
Close	Closes the dialog.
Help	Displays the "Threads dialog" help topic in the CXdb Help window.

Context

To bring up a Threads dialog, select the threads item from the first (leftmost) menu of the following windows:

- Source Code
- Assembly Code
- Memory Display
- Stack Trace
- Processor Status Word
- Scalar Registers
- Vector Registers (C Series only)
- Scalar Registers (C Series only)
- General Registers (SPP Series only)
- Floating Point Registers (SPP Series only)
- Control Registers (SPP Series only)
- Space Registers (SPP Series only)

Related Windows

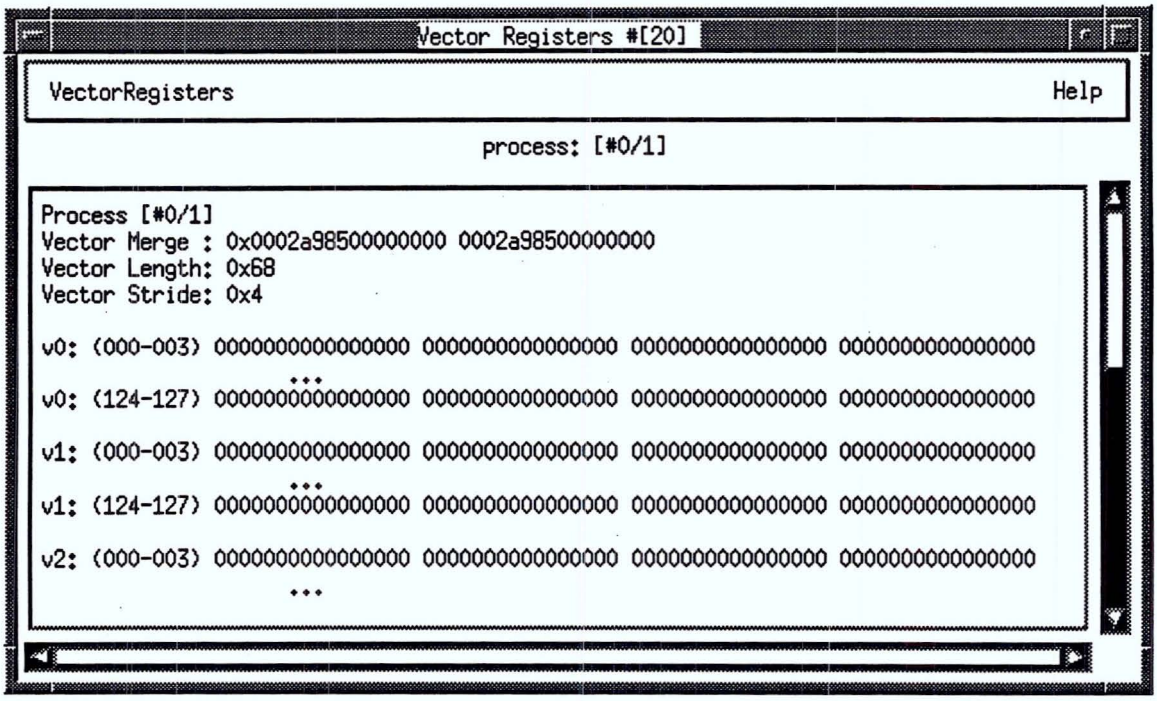
Assembly Code window
Control Registers window
Floating Point Registers window
General Registers window
Memory Display window
Processor Status Word window
Scalar Registers window
Source Code window
Space Registers window
Stack Trace window
Vector Registers window

Related Commands

`info threads`
`set threads`

C Series only

Vector Registers window



Description

The Vector Registers window displays the contents of the vector registers for the specified thread of the process. You can scroll the output in the window, enable or disable automatic updating, change the display format for the contents of the window, and specify which thread is visible in the window.

There are several types of registers in the vector register set:

- **Vector merge (VM)** — Holds the results of element-by-element array comparisons and controls miscellaneous array manipulations.
- **Vector length (VL)** — Contains the number of elements being used in the vector accumulators. The range of VL is from 0 to 128.

- **Vector stride (VS)** — A 32-bit register that contains the distance, in bytes, between adjacent array elements as they are loaded into memory.
- **Vector accumulators** — V0 to V7 on C2 and C3 Series machines; V0 to v15 on C4 Series machines. May contain from 0 to 128 64-bit register operands or elements.
- **Vector first (VF)** — C4 Series machines only. The vector first register (VF) specifies the first element of a vector register to be either read or written by a vector instruction.

Vectorization occurs under the following circumstances:

- The program is optimized to level `-O2` or higher
- The program contains assembly language code that explicitly uses the vector registers.
- The program calls a library routine that explicitly uses the vector registers.

All of the vector registers contain a value of zero unless one of the above is true. For more information about the vector registers on C Series machines, refer to the *CONVEX Architecture Reference Manual (C Series).*

Menus

Name	Description
VectorRegisters	Contains the following items: **Close**—Closes the window. **Auto update**—Enables and disables automatic screen updating. **Change format**—Brings up a Format dialog where you can specify a different display format for the values displayed in the Vector Registers window. **threads**—Brings up a Threads dialog where you can control which thread is visible in the Vector Registers window.
Help	Contains items for invoking the CXdb Help system. Refer to the "Help menu" reference page or online help topic for more information.

Context

The Vector Registers window (C Series only) appears when you select Create window, then select the Vector Registers item from the CXdbWindows menu in either the Command window or the Source Code window.

Related Windows

Command window
Source Code window
Format dialog
Threads dialog

Related Menus

Help menu

Related Commands

`info vregisters`

Related Concepts

registers

Vector Registers window

Messages

5

This chapter contains reference pages that explain the CXdb messages. The messages are listed in order by identification number (ID). Each explanation contains the following sections:

- **Message** — The exact text of the message. Variable parameters are enclosed in angle brackets (<>) and are shown in italic type. For example, *<char>* is a variable that represents an ASCII character.
- **Type** — The message type, which can be one of the following:
  - `INFO` — A condition that is normal or expected processing under the given circumstances.
  - WARNING — A condition that might not be normal or expected, but it is not severe enough to cause an error. CXdb continues to process your command after issuing the warning.
  - `ERROR` — A condition that prevents completion of the current CXdb command.
  - `FATAL` — A condition that prevents further execution of the process being debugged. Both the process and the CXdb session are terminated.
- **Explanation** — A more detailed explanation of the message, including possible actions to correct the situation.

ID		Description
A1	Message	Character <*character*> is not allowed in an alias name.
	Type	ERROR
	Explanation	The alias name contains the illegal character indicated in the error message. Please select a different name for your alias.
A2	Message	Alias <*alias name*> already exists. Please try a new alias name.
	Type	ERROR
	Explanation	An alias with this name already exists. Use a different name in the alias definition.
A3	Message	You must specify a name when creating an alias.
	Type	ERROR
	Explanation	You must specify a unique name for the alias in order to create it.
A4	Message	Alias <*alias name*> does not exist.
	Type	ERROR
	Explanation	There is no alias with this name. Try a different alias name, or use the "info alias" command to list the current aliases.
A5	Message	Command syntax error - <*expecting or missing*>.
	Type	ERROR
	Explanation	The command you entered contains a syntax error. For more information about this command, refer to the online help system, the CXdb Reference Manual, or the CXdb Quick Reference.
A6	Message	Fortran syntax error - <*expecting or missing*>.
	Type	ERROR
	Explanation	The Fortran language expression you entered contains a syntax error. Refer to your Fortran manual for details on the proper syntax.
A7	Message	The '<*string*>' display format cannot be used with '<*string*>' memory units.
	Type	ERROR
	Explanation	The display format you specified cannot be used with the indicated type of memory unit. Refer to the "set format" command for a list of the appropriate display formats.
A8	Message	Eventpoint <*event-id*> does not exist.
	Type	ERROR
	Explanation	The eventpoint number you entered is not currently defined. Try a different eventpoint number, or use the "info event" command to list the current eventpoints.

ID		Description
A9	Message	Thread number <*thread-id*> does not exist.
	Type	ERROR
	Explanation	The thread number you entered is not currently defined. Try a different thread number, or use the "info threads" command to list the current threads.
A10	Message	Cannot select thread <*thread-id*> for process <*process number*>.
	Type	ERROR
	Explanation	CXdb cannot select the requested thread as the current thread for the process because the thread number is not valid. Try the command again with a different thread number.
A11	Message	Process number <*process number*> does not exist.
	Type	ERROR
	Explanation	The process number you specified is not currently defined. Try a different process number, or use the "info cxdb" or "info process" command to list the current processes.
A12	Message	IOCTL error occurred on file descriptor <*file descriptor number*>.
	Type	ERROR
	Explanation	An error ocurred while performing an ioctl system call on the indicated file descriptor. This might lead to additional errors. Other error messages should indicate the reason the ioctl was being performed if the failure of the ioctl was considered harmful to that operation.
A13	Message	Macro <*macro-name*> already exists. Please try a new macro name.
	Type	ERROR
	Explanation	A macro with this name already exists. Use a different name in the macro definition.
A14	Message	You must specify a name when creating a macro.
	Type	ERROR
	Explanation	You must specify a unique name for the macro in order to create it.
A15	Message	Macro <*macro-name*> does not exist.
	Type	ERROR
	Explanation	There is no macro with this name. Try a different macro name, or use the "info macro" command to list the current macros.

ID		Description
A16	Message	Only the first *<count>* macro parameters are used; the rest are ignored.
	Type	WARNING
	Explanation	There is a limit to the number of parameters you can use in a macro. CXdb ignores any parameters beyond that limit. You can redefine the macro with fewer parameters.
A17	Message	Macro calls exceed maximum allowed nesting of *<count>* levels.
	Type	ERROR
	Explanation	You have exceeded the limit on the number of times macros can call each other. You might need to redefine some of your macros so that they do not make so many calls.
A18	Message	Dynamic memory exhausted.
	Type	FATAL
	Explanation	CXdb has used all the dynamic memory available to it. There is no recovery from this condition.
A19	Message	Process must be stopped before you can use this command.
	Type	ERROR
	Explanation	CXdb cannot execute the command you specified unless the process is stopped. Use CTRL-c or the "stop" command to stop the process immediately. If you have set an eventpoint for this process, you can also wait until the process is stopped by the eventpoint.
A20	Message	You must first create a process image with the "run", "attach", or "core" command.
	Type	ERROR
	Explanation	The command you entered requires the image of a running process. To create the image, use the "run", "attach", or "core" command.
A21	Message	Parameter *<formal parameter>* already exists for this macro.
	Type	ERROR
	Explanation	A parameter with this name already exists. Try a different parameter name.
A22	Message	You must assign a name to each parameter in a macro definition.
	Type	ERROR
	Explanation	You have omitted the name of a parameter in a macro definition. Redefine the macro and assign a name to the parameter.

ID		Description
A23	Message	Error in accessing system dependent information.
	Type	FATAL
	Explanation	CXdb encountered an error during the getsysinfo system call. There is no recovery from this error.
A24	Message	Unmatched parenthesis.
	Type	ERROR
	Explanation	The command contains an unmatched parenthesis. Try entering it again.
A25	Message	Cannot locate your home directory.
	Type	WARNING
	Explanation	None of the environment variables $DOTDIR, $LOGDIR, or $HOME are defined. Because of this, CXdb cannot locate your home directory to process any initialization files.
A26	Message	Pathname for <*filename*> is too long.
	Type	ERROR
	Explanation	The pathname constructed is longer than the maximum allowed by the system. Contact your system administrator.
A27	Message	Could not <*perform system call on*> file <*filename*>. System error text: <*errno description*>
	Type	ERROR
	Explanation	An operation on a file failed. The error message contains the text of the error code returned by the failing operation.
A28	Message	Executable file <*filename*> not found.
	Type	ERROR
	Explanation	The file you specified as an executable image could not be accessed. Either you do not have permissions to access the directories leading to the file, or the file does not exist.
A29	Message	File <*filename*> is not marked executable.
	Type	ERROR
	Explanation	The file you specified exists but does not have the execute permission bit set for you. It is either not executable, or you do not have the permissions to execute it.

ID		Description
A30	Message	Only one process/thread list allowed. All but first ignored.
	Type	WARNING
	Explanation	You have specified more than one process/thread list for a command, and only one list is allowed. All process/thread lists except the first have been ignored. The proper syntax is to specify the process first, followed by the thread list. For example, ":p0 :t1,2" specifies threads 1 and 2 of process 0.
A31	Message	You cannot specify threads for this command. Specifiers ignored.
	Type	WARNING
	Explanation	You have specified a thread list for a command that does not accept such a list. CXdb has ignored the thread list.
A32	Message	You cannot specify processes or threads for this command. Specifiers ignored.
	Type	WARNING
	Explanation	You have specified a process list or thread list for a command that does not accept such lists. CXdb has ignored the specifier list.
A33	Message	You must first create a process with the "debug exec", "debug proc", or "debug core" command.
	Type	ERROR
	Explanation	The command you entered requires a process. Use the "debug exec", "debug proc", or "debug core" command to create a process first. Then try your command again.
A34	Message	Process object <*process number*> no longer exists.
	Type	ERROR
	Explanation	A process that was on the global process list no longer exists. This is an internal error. Please report it to the CONVEX Technical Assistance Center.
A35	Message	Eventpoint <*event-id*> no longer exists.
	Type	ERROR
	Explanation	An eventpoint that was on the global event list no longer exists. This is an internal error. Please report it to the CONVEX Technical Assistance Center.
A36	Message	No such offset type.
	Type	ERROR
	Explanation	You have used an address offset type that does not exist. Try a different address specification.

ID		Description
A37	Message	Environment variable <*variable identifier*> not found.
	Type	WARNING
	Explanation	The environment variable you specified does not exist within the environment currently being used. This variable has been ignored.
A38	Message	Process number <*process number*> does not exist.
	Type	ERROR
	Explanation	The process number you specified does not exist. Try a different process number.
A39	Message	Cannot attach to more than one running process.
	Type	ERROR
	Explanation	You specified multiple processes with the "attach" command, but CXdb can attach to only one process at a time. Try the command again with only one process number.
A40	Message	Cannot attach to process <*process number*>. Check your permissions.
	Type	ERROR
	Explanation	CXdb could not attach to a child process. If you have just used the "attach" or "debug proc" command, check to see if you have the correct permissions to attach to the target process.
A41	Message	The system call "sigaction" failed.\nReason: <*errno description*>.
	Type	ERROR
	Explanation	CXdb tried to perform a "sigaction" system call to alter the way it handles signals. The reason for failure is listed in the error message. Please report this error to the CONVEX Technical Assistance Center.
A42	Message	This command cannot run in background. Ignoring "&" operator.
	Type	WARNING
	Explanation	You cannot run this command in background. CXdb has ignored the background operator, "&".
A43	Message	Process [#<*process number*>] is already running.
	Type	ERROR
	Explanation	You tried to run or attach to a process that is already running. If you want to restart the process, first use the "kill process" or "detach" command to terminate the current process. Then issue the "run" command again, or use the "rerun" command.

ID		Description
A44	Message	There is no executable file for process [#*<process number>*].
	Type	ERROR
	Explanation	You tried to run a process that has no executable file associated with it. Use the "executable" command to specify an executable file for your process.
A45	Message	Cannot block receipt of *<signal identifier>* signal.
	Type	FATAL
	Explanation	CXdb could not block the receipt of a signal. This is an internal error. Please report it to the CONVEX Technical Assistance Center.
A46	Message	Cannot unblock receipt of *<signal identifier>* signal.
	Type	FATAL
	Explanation	CXdb could not unblock the receipt of a signal. This is an internal error. Please report it to the CONVEX Technical Assistance Center.
A47	Message	Cannot create process [#*<process number>*].
	Type	ERROR
	Explanation	CXdb could not create the process image to debug. Ask your system administrator for assistance in verifying that CXdb is installed properly and that you have full access to it.
A48	Message	Cannot set fixed scheduling for process [#*<process number>*].
	Type	WARNING
	Explanation	CXdb could not set fixed scheduling for the target process. Ask your system administrator for assistance in verifying that CXdb is installed properly and that you have full access to it.
A49	Message	Cannot clear PIXINHERIT process [#*<process number>*].
	Type	ERROR
	Explanation	CXdb cannot clear the PIXINHERIT process mode. Ask your system administrator for assistance in verifying that CXdb is installed properly and that you have full access to it.
A50	Message	Cannot identify the parent process for process *<process number>*.
	Type	ERROR
	Explanation	CXdb could not identify the parent process of the target process it attached to. The target process was discarded. This is an internal error. Please report it to the CONVEX Technical Assistance Center.

ID		Description
A51	Message	Unknown SIGTRAP code <*SIGTRAP subcode*> in process <*process number*>.
	Type	WARNING
	Explanation	The process received a SIGTRAP signal, but the signal subcode is unkown. The SIGTRAP was ignored.
A52	Message	Cannot clear the Trace Trap PSW bit in process <*process number*>.
	Type	WARNING
	Explanation	CXdb was unable to clear the Trace Trap bit in one of the processes it is managing. This is only a warning, but it will probably lead to other errors later. If this error occurs during process startup, the process is terminated.
A53	Message	Unknown process <*process number*> issued an exec call.
	Type	ERROR
	Explanation	One of the processes that CXdb is controlling has an unknown state, and it performed an exec system call. The process has been detached. If this error occurs during process startup, the process is terminated.
A54	Message	Process [#<*process number*>] issued an exec call.
	Type	WARNING
	Explanation	The target process you are debugging has performed an exec system call. This will probably invalidate all the debugging information that CXdb has about the process. CXdb has removed all PC-based eventpoints from this process because their locations and data were no longer valid. You should use the "executable" command to inform CXdb about the executable file that the process is now executing.
A55	Message	Command [#<*command identifier*>] already executing on process [#<*process number*>].
	Type	ERROR
	Explanation	You tried to execute a command on the indicated process, but there was already another command executing in background on that process. Either wait until the previous command completes, or use the "stop" command to terminate it.
A56	Message	Thread [#<*process number*>/<*thread-id*>] is already running.
	Type	ERROR
	Explanation	You entered a command to continue execution of one or more threads of the process, but at least one of those threads is already running. Wait until all threads have stopped, or use CTRL-c or the "stop" command to stop them.

ID		Description
A57	Message	Not a valid floating point mode.
	Type	ERROR
	Explanation	The floating point mode you specified is either not recognized or not allowed with this command. Refer to the "set fpmode" or "set evalopts fpmode" command pages for a list of valid floating point modes.
A58	Message	No debugging information available.
	Type	ERROR
	Explanation	There is currently no debugging information available to the command you specified. If you have not already done so, compile your program with the -cxdb option. Then use the "executable" command to tell CXdb the name of the executable file that contains the debugging information for your program.
A59	Message	No source statements found.
	Type	ERROR
	Explanation	The location you specified does not contain any source statements. Try a different location.
A60	Message	No debugging information available for stepping thread <*thread-id*>.
	Type	ERROR
	Explanation	There is no debugging information available for the current location of the indicated thread. Stepping by source units is not possible at this location. Use the step instruction command or the break instruction command followed by a continue command,
A61	Message	Unable to clear bits <*PSW BIT* <*identifier*>S> in PSW.
	Type	FATAL
	Explanation	The indicated bits could not be cleared in the current PSW or on the stack. This will probably lead to more cascading errors. There is no way to recover from this condition.
A62	Message	Cannot reset memory occupied by breakpoint at <*address*>.
	Type	FATAL
	Explanation	CXdb could not restore the original value of a memory location currently occupied by a breakpoint. This will probably lead to incorrect operation of your program. You probably will not be able to continue execution of any thread past this address. This is an internal error. Please report it to the CONVEX Technical Assistance Center.

ID		Description
A63	Message	Cannot restore breakpoint at <*address*>.
	Type	FATAL
	Explanation	The breakpoint at the indicated address could not be restored. This will probably lead to incorrect operation of your program. It is probable that the current stepping operation will fail and further execution might also fail. This is an internal error. Please report it to the CONVEX Technical Assistance Center.
A64	Message	Source unit <*source unit index*> does not exist within file <*filename*>.
	Type	ERROR
	Explanation	The source unit number you specified does not exist within the indicated source file. Use the "info source" command to list all the source units at a particular line of source code.
A65	Message	Eventpoint <*event-id*> also set a breakpoint at address <*address*>.
	Type	INFO
	Explanation	You have created a breakpoint at a location where a previous breakpoint already exists. The last breakpoint created at that location takes precedence over previous ones. You can disable the previous eventpoints with the "disable event" command or remove them with the "remove event" command.
A66	Message	Debugger variable <*variable identifier*> is not defined.
	Type	ERROR
	Explanation	The CXdb debugger variable you specified does not exist. Refer to the "debugger variables" concepts page for more information about defining debugger variables.
A67	Message	Cannot delete debugger variable <*variable identifier*>. It is predefined by CXdb.
	Type	ERROR
	Explanation	You cannot delete the specified debugger variable because it is predefined by CXdb. You can delete only the debugger variables you have created. Refer to the "debugger variables" concepts page for more information.
A68	Message	Cannot modify debugger variable <*variable identifier*>. It is predefined by CXdb as read-only.
	Type	ERROR
	Explanation	You cannot change the value of this debugger variable because it is predefined by CXdb as a read-only variable. Refer to the "debugger variables" concepts page for more information.

ID		Description
A69	Message	Invalid command in eventpoint handler. Handler aborted.
	Type	ERROR
	Explanation	You used a command that is not allowed in eventpoint handlers, so the handler was aborted. Process execution commands such as "continue" are not allowed in eventpoint handlers. Use the "resume" command in the eventpoint handler if you want to continue process execution.
A70	Message	Command not valid interactively. This command is for eventpoint handlers only.
	Type	ERROR
	Explanation	The command you entered is for use in eventpoint handlers only, and it cannot be executed interactively. Refer to the command reference pages for more information about this command.
A71	Message	File <*filename*> is not a core file.
	Type	ERROR
	Explanation	The command you entered requires a core file. Try a different file name.
A72	Message	Address <*address*> is not within a defined region of the process.
	Type	ERROR
	Explanation	The address you specified was not within any of the section maps contained in the current process image. Either you entered an invalid address or the process image is not from the executable file you are debugging.
A73	Message	Remove old process image with "kill process" or "detach" command first.
	Type	ERROR
	Explanation	You tried to associate a new image with a process object that already contains an image. To remove the old image, use the "kill process" or "detach" command.
A74	Message	Vector register <*register identifier*> does not have enough space.
	Type	ERROR
	Explanation	The vector register you specified does not have enough space left for the operation you requested. Use the "info vregisters" command to check the current contents and size of the vector registers.
A75	Message	Your SHELL setting, '<*string*>', is not supported. Defaulting to /bin/sh.
	Type	WARNING
	Explanation	Your SHELL environment variable specifies a shell type that is not supported on this system. CXdb will use the default shell /bin/sh instead.

ID		Description
A76	Message	Regular expression <*string*> is not valid.
	Type	ERROR
	Explanation	The regular expression you entered is not valid. Refer to the ed(1) man page to determine the proper syntax of for regular expressions.
A77	Message	System call failed: <*perform system call on*> - <*errno description*>
	Type	ERROR
	Explanation	The indicated system call failed for the reason shown.
A78	Message	Cannot read scalar registers for [#<*process number*>/<*thread-id*>].
	Type	ERROR
	Explanation	CXdb encountered an error while trying to read the scalar registers. This will probably result in other errors. For example, CXdb might not be able to determine the state of your process correctly. This is an internal error. Please report it to the CONVEX Technical Assistance Center.
A79	Message	Cannot read vector registers for [#<*process number*>/<*thread-id*>].
	Type	ERROR
	Explanation	CXdb encountered an error while trying to read the vactor registers. This will probably result in other errors. For example, CXdb might not be able to determine the state of your process correctly. This is an internal error. Please report it to the CONVEX Technical Assistance Center.
A80	Message	You cannot resume execution of a core file.
	Type	ERROR
	Explanation	You cannot resume or continue execution of a core file. However, you can use the "run" command to start a new process from the executable file associated with this core file.
A81	Message	Cannot get trap addresses for thread <*thread-id*>.
	Type	ERROR
	Explanation	During process creation, CXdb was unable to read the user mode trap addresses from the process. This probably indicates a more serious problem, and the process object will be unusable. This is an internal error. Please report it to the CONVEX Technical Assistance Center.
A82	Message	Cannot read address <*address*> to get <*string*>.
	Type	ERROR
	Explanation	CXdb cannot read the specified location to get the requested data. This might prohibit further debugging. This is an internal error. Please report it to the CONVEX Technical Assistance Center.

ID		Description
A83	Message	Cannot write to address <*address*> to put <*string*>.
	Type	ERROR
	Explanation	:CXdb cannot write to the specified location to save (put) the requested data. This might prohibit further debugging. This is an internal error. Please report it to the CONVEX Technical Assistance Center.
A84	Message	Source file name is missing.
	Type	ERROR
	Explanation	You entered a command that requires a source file name. Enter the command again, specifying a source file name. You can also invoke a Source Code window with the "debug exec" command, and CXdb will use the source file associated with that window as the default.
A85	Message	Cannot construct a stack frame for [#<*process number*>/<*thread-id*>].
	Type	ERROR
	Explanation	CXdb could not construct a stack frame for the indicated process and thread. CXdb could not read the frame information memory or the registers. This is an internal error. Please report it to the CONVEX Technical Assistance Center.
A86	Message	Cannot evaluate the language expression.
	Type	ERROR
	Explanation	CXdb could not evaluate your language expression for the indicated reasons.
A87	Message	Cannot locate main routine. No default source file set.
	Type	WARNING
	Explanation	CXdb could not locate the main routine of your program. Because of this, CXdb did not select a default source file, but it did set the default source language to C. This condition arises if the main routine of your program is written in a language that CXdb does not support or if the main routine was compiled without the -cxdb option.
A88	Message	Too many threads selected (<*count*>). Only 1 allowed.
	Type	ERROR
	Explanation	You tried to select more than one thread for an command that allows only a single thread. Use the ":t" option to specify a single thread, and try the command again.
A89	Message	Formal parameter <*formal parameter*> was not used in the macro.
	Type	INFO
	Explanation	You did not use one of the parameters formally defined for this macro. This is only an informational message, and the macro definition is still valid.

ID		Description
A90	Message	Cannot create a Source Code window.
	Type	INFO
	Explanation	CXdb cannot create a Source Code window for one of the following reasons: 1. There is no executable file associated with your source file. Use the "executable" command to specify an executable file. 2. CXdb could not find the source file for the current location in your program. Use the "add path" command to specify the directory that contains your source file. 3. The current point of execution is in a portion of code (such as a library routine) for which CXdb has no debugging information.
A91	Message	Line <*count*> is out of bounds. Valid range is 1 to <*count*>.
	Type	ERROR
	Explanation	You have specified a line number that is not within the valid range for a source file. Try the command again with a different line number.
A92	Message	Source unit <*source unit index*> is out of bounds. Valid range is 0 to <*count*>.
	Type	ERROR
	Explanation	You have specified a source unit number that is not within the valid range for a source file. Try the command again with a different source unit number.
A93	Message	Line <*count*> has been optimized away.
	Type	ERROR
	Explanation	You have specified a line number that has no object code associated with it because of optimizations performed by the compiler. You cannot set an eventpoint at this line. Try the command again with a different line number.
A94	Message	Source unit <*source unit index*> has been optimized away.
	Type	ERROR
	Explanation	You have specified a source unit number that has no object code associated with it because of optimizations performed by the compiler. You cannot set an eventpoint at this source unit. Try the command again with a different source unit number.
A95	Message	A process already exists. Use the '<*string*>' command instead.
	Type	ERROR
	Explanation	A process already exists, but you have used a command that tries to create a new process. The current version of CXdb can operate on only one process at a time. To specify a new file to debug, use either the "executable", "core", or "attach" command.

ID		Description
A96	Message	Address <*address*> is not a valid PC setting for process [#<*process number*>/<*thread-id*>].
	Type	ERROR
	Explanation	You have specified an invalid address with a "goto" command. It is either outside the address regions of the program, or read permissions are not set for the page containing the address. Try the command again with a different address.
A97	Message	Source unit or line <*filename*>:<*count*> has multiple entry points.
	Type	ERROR
	Explanation	You have used a "goto" command to specify a source unit or line number that has multiple entry points. Because this is an ambiguous situation, the PC of the thread cannot be modified. Use the "info line" or "info sourceunit" command to list the entry point addresses, then try the "goto address" command with a specific address from that list.
A98	Message	Multiple threads selected; evaluating expression in thread <*thread-id*>.
	Type	WARNING
	Explanation	You have selected mutiple threads, but expressions must be evaluated in the context of a single thread. CXdb has selected the indicated thread for this evaluation. If this is not the thread you want to use, try the command again and use the ":t" option to specify the desire thread.
A99	Message	Expression is not a valid type for use as an address.
	Type	ERROR
	Explanation	The expression you entered could not be converted to an address because of its data type. CXdb can convert only integers to addresses. Try the command again with a different address expression. (In C, you can also try casting the expression to an integer type.)
A100	Message	Conversion of expression result to <*type identifier*> failed.
	Type	ERROR
	Explanation	CXdb could not convert your expression to the data type indicated. Try the command again with a different expression. (In C, you can also try casting the expression to a data type that can be converted.)
A101	Message	Signal number <*signal value*> does not exist.
	Type	ERROR
	Explanation	The signal number you entered does not exist. Use the "info signal" command to list the valid signal numbers.

ID		Description
A102	Message	Cannot push <*count*> bytes onto process stack. Function call failed.
	Type	ERROR
	Explanation	CXdb could not push the indicated number of bytes onto the process stack. This operation is done in preparation for calling a function within the target process (for example, when evaluating an expression that includes a function call). Because the data could not be placed on the stack, the function call was not performed. Further errors may result.
A103	Message	Cannot create file <*filename*> with noclobber set.
	Type	ERROR
	Explanation	The file you specified does not exist, and CXdb could not create it because the noclobber option was set. Either use the "clear noclobber" command to disable the noclobber option, or use the ">>!" redirection operator to override noclobber.
A104	Message	Cannot overwrite file <*filename*> with noclobber set.
	Type	ERROR
	Explanation	The file you specified already exist, and CXdb could not overwrite the file because the noclobber option was set. Either use the "clear noclobber" command to disable the noclobber option, or use the ">!" redirection operator to override noclobber.
A105	Message	Window number <*window identifier*> is not a valid viewport.
	Type	ERROR
	Explanation	The window you specified is not a valid viewport. Only the Command window (window number 1) is a valid viewport in this release of CXdb.
A106	Message	Cannot evaluate relational expression in eventpoint <*event-id*>.
	Type	ERROR
	Explanation	CXdb could not evaluate the relational expression you used in the indicated eventpoint. Try the eventpoint again with a different relational expression.
A107	Message	Relational expression already TRUE. Eventpoint not created.
	Type	ERROR
	Explanation	CXdb could not create the relational eventpoint because the relational expression you entered already evaluates to TRUE. Try the command again with a different relational expression.
A108	Message	Cannot read previous stack frame. Relational eventpoint not created.
	Type	ERROR
	Explanation	CXdb could not read the necessary stack frame. This will probably lead to additional errors. If you were trying to create a relational eventpoint when this error occurred, that eventpoint was not created.

ID		Description
A109	Message	Relation eventpoint <*event-id*> disabled; out of scope.
	Type	WARNING
	Explanation	CXdb disabled the indicated relational eventpoint because the stack frame associated with that eventpoint has been popped from the stack.
A110	Message	File <*filename*> is <*errno description*>.
	Type	ERROR
	Explanation	A file operation failed for the indicated reason.
A111	Message	Cannot recognize scope path <*scope path*>.
	Type	ERROR
	Explanation	CXdb cannot recognize the scope path you specified. Try a different pathname.
A112	Message	Evaluating expression in each thread context.
	Type	WARNING
	Explanation	CXdb has evaluated your language expression for each thread of the process. This can result in errors if the threads are operating in different scopes. To limit the expression evaluation to particular threads, use the ":t" option with the CXdb commands.
A113	Message	Address offset <*count*> is not valid.
	Type	ERROR
	Explanation	The address offset you entered is not valid. Try the command again with a different address specification.
A114	Message	Address range <*address*>..<*address*> is not valid.
	Type	ERROR
	Explanation	The address range you specified is invalid, either because it extends beyond the address range of the process or because the starting address is larger than the ending address. Try the command again with a different address range.
A115	Message	Address range refers to stack. Eventpoint will be disabled when frame is popped.
	Type	WARNING
	Explanation	The address range you specified lies within the bounds of the process stack. When the stack frame containing this region is popped from the stack, the eventpoint will be disabled automatically.

ID		Description
A116	Message	Debugger variable '*<string>*' is not of type *<type identifier>*. Redefine it or use another variable.
	Type	ERROR
	Explanation	The debugger variable you specified is not the proper data type. Either redefine the variable to make it the appropriate type, or use a different debugger variable.
A117	Message	Eventpoint *<event-id>* disabled when region being monitored was popped from the stack.
	Type	WARNING
	Explanation	CXdb disabled the indicated eventpoint when the data region being monitored was popped from the process stack. This eventpoint is no long valid and cannot be reused. Create a new eventpoint if you want to monitor the same region.
A118	Message	Expression is not a valid address.
	Type	ERROR
	Explanation	The language expression you entered is not a valid address specifier. Use the ":" or ".." notation to construct explicit address ranges. You can also use the "loc()" operator in Fortran or the "&" operator in C to obtain the starting address of an identifier.
A119	Message	Stopped translating alias *<alias name>* because it is recursive.
	Type	WARNING
	Explanation	CXdb stopped expanding the indicated alias because the alias definition is cyclical. That is, the alias references itself either directly or indirectly (through another alias). CXdb still processes the command line as far as possible. To avoid this problem, redefine the alias without cyclical calls.
A120	Message	C syntax error - *<expecting or missing>*
	Type	ERROR
	Explanation	The C language expression you entered contains a syntax error. Refer to your C language manual for details on the proper syntax.
A121	Message	Identifier '*<string>*' not visible from the current lexical scope.
	Type	ERROR
	Explanation	The identifier you specified is not visible from the current lexical scope. Refer to the "scope" concepts page for information on how to qualify an identifier with a scope path. You can also use the "frame" command to select a different stack frame and thereby change the current scope.
A122	Message	Floating point overflow.
	Type	ERROR
	Explanation	The result of a floating point operation was too large to store.

ID		Description
A123	Message	Left operand of '*<string>*' is a mixed mode operation.
	Type	WARNING
	Explanation	The left operand of the indicated operator has an internal floating point representation that is not compatible with the floating point mode established by the "set evalopts fpmode" command. Use the "set evalopts fpmode" command to specify the same floating point mode as the left operand.
A124	Message	Right operand of '*<string>*' is a mixed mode operation.
	Type	WARNING
	Explanation	The right operand of the indicated operator has an internal floating point representation that is not compatible with the floating point mode established by the "set evalopts fpmode" command. Use the "set evalopts fpmode" command to specify the same floating point mode as the right operand.
A125	Message	Divide by zero.
	Type	ERROR
	Explanation	The divisor has a value of zero. Try the operation again with a nonzero devisor.
A126	Message	Subscript truncated to integer.
	Type	WARNING
	Explanation	CXdb truncated the subscript expression to fit within the maximum precision allowed by the hardware architecture.
A127	Message	Subscript not an integer.
	Type	ERROR
	Explanation	The subscript expression did not result in an integer. Try converting the subscript by using the "INT()" operator in Fortran or the "int" operator in C.
A128	Message	Too many subscripts specified.
	Type	ERROR
	Explanation	You have specified more subscripts than are defined for the referenced array. Use the "info expression" command to determine the dimensions of the array.
A129	Message	Too few subscripts specified.
	Type	ERROR
	Explanation	You have specified fewer subscripts than are defined for the referenced array. Use the "info expression" command to determine the dimensions of the array.

ID		Description
A130	Message	Left-hand side of "." expression is not a structure or union.
	Type	ERROR
	Explanation	The left-hand side of the expression containing the selection operator "." is not a structure or union. Try the command again with a different expression, or use the "info expression" command to determine the data type of the expression.
A131	Message	Subscript required.
	Type	ERROR
	Explanation	The array reference requires a subscript. Use the "info expression" command to determine the dinemsions of the array.
A132	Message	Expression is not a pointer.
	Type	ERROR
	Explanation	The expression you entered is not a valid pointer. Try the command again with a different expression, or use the "info expression" command to determine the data type of the expression.
A133	Message	Illegal address reference.
	Type	ERROR
	Explanation	CXdb could not evaluate the address of the object you specified. Try a different address expression.
A134	Message	Cannot dereference a pointer to a void object.
	Type	ERROR
	Explanation	The address expression you entered attempts to point to an object that is void or has no value. Try the command again with a different address expression.
A135	Message	Illegal pointer/integer combination
	Type	ERROR
	Explanation	An attempt has been made to take the address of either (1) an operand that is not a function designator, or (2) an lvalue that designates an object which is a bit field, or (3) an object that has been declared with register storage class.
A136	Message	Variable <*variable identifier*> is not available at this time.
	Type	ERROR
	Explanation	There is no storage value available for the indicated variable at the current point of process execution. Use the "info expression" command to determine the ranges of process memory where the variable is available (liveness ranges).

ID		Description
A137	Message	Cannot find symbol name in loader symbol list.
	Type	ERROR
	Explanation	CXdb could not find the loader symbol you specified. Use the "info symbols" command to list the symbols currently available in your process.
A138	Message	Variable is not active yet.
	Type	ERROR
	Explanation	A variable you specified in the expression is currently visible, but it has not yet become active. Use the "run" or "rerun" command to start execution of the process, then try referencing the variable again.
A139	Message	Incomplete data type.
	Type	ERROR
	Explanation	An attempt has been made to evaluate an expression in which the definition of the object designated by the expression is not visible. An example of this situation frequently occurs when a pointer to a derived type is declared in the source file, yet the source never references the object except to assign (or access) a value to the pointer. In this case, try prefixing the tag identifier of the derived type with a scope path in which the type definition is visible in a cast expression.
A140	Message	Cannot evaluate sizeof(void).
	Type	ERROR
	Explanation	You cannot use the "sizeof" operator on an object that is a void data type. Try the operator again on a different object.
A141	Message	Cannot evaluate sizeof(function).
	Type	ERROR
	Explanation	You cannot use the "sizeof" operator on a function. Try the operator again on a different object.
A142	Message	Cannot evaluate sizeof(bitfield).
	Type	ERROR
	Explanation	You cannot use the "sizeof" operator on an object that is bit field. Try the operator again on a different object.
A143	Message	The object of the "sizeof()" operator is not visible in the current scope.
	Type	ERROR
	Explanation	The object you specified for the "sizeof" operator is not visible in the current scope. Try specifying the scope path for the object.

ID		Description
A144	Message	Illegal type combination.
	Type	ERROR
	Explanation	The type name you used includes conflicting type specifiers. An example of this would be a type name specification that includes both "float" and "struct foo" type specifiers.
A145	Message	Storage class specifier ignored.
	Type	INFO
	Explanation	The storage class you specified does not apply to the language expression. CXdb ignored the storage class specification, but the expression evaluation was not affected.
A146	Message	Illegal data type specification.
	Type	ERROR
	Explanation	The data type specification your entered is not valid. Try a different specification.
A147	Message	Ignoring "volatile" type qualifier.
	Type	INFO
	Explanation	The "volatile" type qualifier has no effect on the evaluation of the expression, therefore it has been ignored.
A148	Message	Ignoring "const" type qualifier.
	Type	INFO
	Explanation	The "const" type qualifier has no effect on the evaluation of the expression, therefore it has been ignored.
A149	Message	Type already declared "volatile".
	Type	ERROR
	Explanation	The type name you specified has already been declared "volatile", so you cannot declare it again.
A150	Message	Type already declared "const".
	Type	ERROR
	Explanation	The type name you specified has already been declared "const", so you cannot declare it again.

ID		Description
A151	Message	Illegal type in cast expression.
	Type	ERROR
	Explanation	The type name specified in the cast expression is not a scalar or void type. Examples of scalar types are char, int, float, and double, where each may be further modified by short/long or signed/unsigned and qualified by const and/or volatile. A pointer to one of the above simple types or to a derived type may be specified. All derived types (except pointer) are not allowed.
A152	Message	Cannot cast to "function" type.
	Type	ERROR
	Explanation	You cannot cast an expression to "function" type.
A153	Message	Operand of cast is an incompatible type.
	Type	ERROR
	Explanation	The operand in the specified cast expression is not a scalar. Examples of scalar types are char, int, float, and double, where each may be further modified by short/long or signed/unsigned and qualified by const and/or volatile. A pointer to one of the above simple types or to a derived type may be specified. All derived types (except pointer) are not allowed.
A154	Message	Constant expression required.
	Type	ERROR
	Explanation	The expression you entered must consist of constants (literals) only. Try the command again with a constant expression.
A155	Message	Subscript value must be greater than zero.
	Type	ERROR
	Explanation	The expression you entered is not a valid subscript value because it evaluates to zero or a negative number. Try the command again with a subscript value greater than zero.
A156	Message	Array elements cannot be a "void" data type.
	Type	ERROR
	Explanation	You cannot derive an array element from a "void" data type. Try the command again with a different data type.
A157	Message	Array elements cannot be a "function" data type.
	Type	ERROR
	Explanation	An array type cannot be derived from a function type.

ID		Description
A158	Message	Functions cannot return array types.
	Type	ERROR
	Explanation	The specified type name defines a function that returns an array type as its result. This is an illegal type constructor.
A159	Message	Functions cannot return function types.
	Type	ERROR
	Explanation	The specified type name defines a function that returns a function type as its result. This is an illegal type constructor.
A160	Message	Array dimension cannot be null.
	Type	ERROR
	Explanation	The constant expression describing the bounds of an array must evaluate to an integral value greater than zero. Try the command again with a different expression for the array dimensions.
A161	Message	Specified tag does not identify a struct type.
	Type	ERROR
	Explanation	The type tag specified in the structure type name does not designate a structure type.
A162	Message	Specified tag does not identify a union type.
	Type	ERROR
	Explanation	The type tag specified in the union type name does not designate a union type.
A163	Message	Cannot find struct/union type definition.
	Type	ERROR
	Explanation	CXdb could not find the type definition corresponding to the specified type tag within the current scope. Try specifying a scope path for the derived type.
A164	Message	Called function expects an array parameter.
	Type	INFO
	Explanation	The called function was expecting an array, but you passed it a parameter that is not an array. This will not inhibit evaluation of the function call.
A165	Message	Called function expects a function parameter.
	Type	INFO
	Explanation	The called function was expecting another function, but you passed it a parameter that is not a function. This will not inhibit evaluation of the called function.

ID		Description
A166	Message	Size of actual argument does not match formal declaration.
	Type	INFO
	Explanation	The actual argument passed to a subroutine or function is a different type and/or precision than the formal argument declaration.
A167	Message	Too many actual arguments specified.
	Type	INFO
	Explanation	You have called a function or subroutine with more actual arguments than the formal definition has declared. Excess arguments were ignored.
A168	Message	Too few actual arguments specified.
	Type	INFO
	Explanation	You have called a function or subroutine with fewer actual arguments than the formal definition has declared. Unused arguments were ignored.
A169	Message	Operand truncated to four-byte integer.
	Type	INFO
	Explanation	The result of an expression has been truncated to an integer with a precision of four bytes. This is a result of constraints imposed by the hardware architecture. The resulting value of this expression was used to obtain an address in the virtual memory space of the process.
A170	Message	Formal and actual parameter are not compatible data types.
	Type	ERROR
	Explanation	The data type of the actual parameter is incompatible with the declaration of the formal parameter. Use the "info expression" command on the called function to determine the data types of its formal parameters.
A171	Message	Ignoring the "const" qualifier.
	Type	INFO
	Explanation	To permit a more flexible debugging environment, CXdb has ignored the "const" portion of the language expression. This allows you to modify the value.
A172	Message	Cannot find scope block '*<string>*'.
	Type	ERROR
	Explanation	The block you specified is not visible from the current lexical scope. Try the command again with a different scope block specification.
A173	Message	Syntax Error - expression must be a constant.
	Type	ERROR
	Explanation	The real and imaginary components of a COMPLEX constant must be constant expressions.

ID		Description
A174	Message	Syntax Error - precision of constant expression is too high.
	Type	ERROR
	Explanation	The precision of the real and imaginary components of a COMPLEX constant must be either single or double precision. Quad (REAL*16) precision is not allowed.
A175	Message	Syntax Error - REAL or INTEGER constant required.
	Type	ERROR
	Explanation	The real and imaginary components of a COMPLEX constant must be of type INTEGER or type REAL.
A176	Message	Scope blocks for *<variable identifier>* are merged. Use a scope path.
	Type	ERROR
	Explanation	The compiler has performed optimizations that merged two or more scope blocks into a single scope block. As a result, two identifiers with the same name are now visible in the same scope block. CXdb preserves the true lexical scope, so you can use a scope path to reach the desired identifier.
A177	Message	No address returned for loader symbol *<variable identifier>*.
	Type	ERROR
	Explanation	No address was returned for the indicated loader symbol. This is an internal error. Please report it to the CONVEX Technical Assistance Center.
A178	Message	Cannot use a scope path with the "loader$" prefix.
	Type	ERROR
	Explanation	You cannot use a scope path with a loader or assembler language specifier (such as l$, loader$, or asm$). Try the command again without specifying a scope path.
A179	Message	CXdb floating point mode [*<fpmode>*] conflicts with process mode [*<fpmode>*].
	Type	INFO
	Explanation	The floating point mode for CXdb conflicts with the floating point mode of the process. If this is not acceptable, use the "set evalopts fpmode" command to change the floating point mode for CXdb or the "set fpmode" command to change the floating point mode of the process.
A180	Message	Floating point hardware not available; evaluation aborted.
	Type	ERROR
	Explanation	The IEEE floating point hardware is not available on this system, so CXdb cannot use this mode to evaluate your expressions. Use the "set evalopts fpmode" command to select a different floating point mode for CXdb.

ID		Description
A181	Message	Cannot create '*<string>*' from '*<string>*'.
	Type	ERROR
	Explanation	This is an internal evaluation error. Please report it to the CONVEX Technical Assistance Center.
A182	Message	Mixed-mode conversion from '*<string>*' to '*<string>*'.
	Type	INFO
	Explanation	Some elements of the language expression were defined in one floating point mode but used in a different mode.
A183	Message	Operand of '*<string>*' has incompatible data type.
	Type	ERROR
	Explanation	The operand of the indicated unary operator has a data type that is not compatible with unary operations. Try the command again with a different operand or operator.
A184	Message	Constant expression '*<string>*' too large to be represented.
	Type	ERROR
	Explanation	The specified integer constant expression is too large to be represented by the integral data types available in the source language. Try a different expression.
A185	Message	Identifier '*<string>*' is not an array/function type.
	Type	ERROR
	Explanation	You tried to use the indicated identifier in a function call or an array expression, but it is not an array or function. Try a different identifier.
A186	Message	Subscript value *<count>* is outside the range *<count>*..*<count>*.
	Type	ERROR
	Explanation	The value of the subscript is not within the bounds of the dimension defined for this array. Try the expression again with a different subscript.
A187	Message	Structure or union does not have a member '*<string>*'.
	Type	ERROR
	Explanation	The indicated member name is not part of the structure or union you are trying to reference. Try the expression again with a different member name or a different structure/union.

ID		Description
A188	Message	Incomplete type - size of type is unknown.
	Type	ERROR
	Explanation	The referenced object has an incomplete type. No size information is available to permit evaluation to continue. This can occur as a result of a forward declaration in which the definition is not visible from the specified scope.
A189	Message	Argument to intrinsic function has incorrect data type.
	Type	ERROR
	Explanation	The argument passed to the intrinsic function has a data type that is not appropriate for use by the intrinsic function. Refer to the Fortran reference manual for a description of the intrinsic function.
A190	Message	Too many arguments passed to intrinsic function.
	Type	ERROR
	Explanation	You have passed too many arguments to the intrinsic function. Refer to the Fortran reference manual for a description of the intrinsic function.
A191	Message	Cannot pass more than one character argument to ICHAR().
	Type	ERROR
	Explanation	The ICHAR() intrinsic function cannot accept more than one character argument at a time. Refer to the Fortran reference manual for a description of the intrinsic function.
A192	Message	Too few arguments passed to intrinsic function.
	Type	ERROR
	Explanation	You have passed too few arguments to the intrinsic function. Refer to the Fortran reference manual for a description of the intrinsic function.
A193	Message	Cannot get the address of a CXdb debugger variable.
	Type	ERROR
	Explanation	You cannot obtain the address of a CXdb debugger variable because these variables do not exist in the virtual memory space of the debugged process.
A194	Message	Mathematical result of expression is not defined.
	Type	ERROR
	Explanation	The language expression produces a mathematical result that is not defined. Verify the values of the components in the expression to ensure the integrity of the result.

ID		Description
A195	Message	Operands of '*<string>*' are not compatible data types.
	Type	ERROR
	Explanation	The operands of the specified operator are not compatible data types. Try a different operand.
A196	Message	Feature not supported in this release of CXdb: *<command line>*
	Type	ERROR
	Explanation	This release of CXdb does not support the indicated functionality.
A197	Message	Missing data type information. Cannot check or coerce arguments.
	Type	WARNING
	Explanation	CXdb cannot obtain the data type information necessary to check the number and type of actual arguments to the called routine, nor can it coerce them to the type of the formal argument. If possible, recompile the source file with the -cxdb option to provide the missing data type information.
A198	Message	Descending scope path specification requires a descendent block.
	Type	ERROR
	Explanation	To look forward in scope, choose the desired scope block (descendent) from which the target identifier is visible.
A199	Message	Left operand of "(:)" operator is not a character.
	Type	ERROR
	Explanation	The substring operator "(:)" requires a character as its left operand. Refer to the Fortran reference manual for a description of this operator.
A200	Message	Left-hand side of assignment must be an Lvalue.
	Type	ERROR
	Explanation	The left-hand side of the assignment operator must be a modifiable object (Lvalue).
A201	Message	Scalar register *<register identifier>* does not exist.
	Type	ERROR
	Explanation	The indicated scalar register does not exist on this system. Try a different scalar register.
A202	Message	Vector register *<register identifier>* does not exist.
	Type	ERROR
	Explanation	The indicated vector register does not exist on this system. Try a different vector register.

ID		Description
A203	Message	Address register <*register identifier*> does not exist.
	Type	ERROR
	Explanation	The indicated address register does not exist on this system. Try a different address register.
A204	Message	Cannot access debugger variable <*variable identifier*>.
	Type	ERROR
	Explanation	CXdb cannot access the indicated debugger variable.
A205	Message	Cannot access vector registers.
	Type	ERROR
	Explanation	CXdb encountered an address that is outside the range of vector register addresses.
A206	Message	Repeat count <*count*> is not valid.
	Type	ERROR
	Explanation	The repeat count you specified is not valid. Repeat counts for all "step", "next", and "continue" commands must be greater than zero.
A207	Message	Address <*address*> is not within a region of code.
	Type	ERROR
	Explanation	The address you specified is not within the defined text region of the process image. Use the "info objectmap" command to determine the object file regions that make up the text segment.
A208	Message	Find pattern length of <*count*> is larger than maximum of <*count*>.
	Type	ERROR
	Explanation	The search pattern you specified with the "find" command was too long. The maximum length is indicated in the message. Try the command again with a shorter pattern.
A209	Message	Find pattern must be an even number of digits. Length of <*count*> is not even.
	Type	ERROR
	Explanation	The search pattern you specified with the "find memory" command contains an odd number of characters, but an even number of characters is required. Try the command again with a search pattern that contains an even number of characters.

ID		Description
A210	Message	Character <*character*> is not valid in a find pattern.
	Type	ERROR
	Explanation	The search pattern you specified with the "find memory" command contains an invalid hex character. Try the command again with a search pattern that contains an even number of hexadecimal digits.
A211	Message	Abbreviation '<*string*>' is ambiguous. Could refer to:<*keyword list*>
	Type	ERROR
	Explanation	The abbreviation you entered matches more than one command keyword. Try the entry again with a different abbreviation or keyword.
A212	Message	Cannot read process memory at address <*address*>. Cause: <*errno description*>.
	Type	ERROR
	Explanation	CXdb could not read the virtual memory space of the debugged process for the reason indicated. Refer to the errno.h(3) man page for further details describing the failure code.
A213	Message	Cannot write to process memory address <*address*>. Cause: <*errno description*>.
	Type	ERROR
	Explanation	CXdb could not write to the virtual memory space of the debugged process for the reason indicated. Refer to the errno.h(3) man page for further details describing the failure code.
A214	Message	Format <*format specifier*> is not available.
	Type	ERROR
	Explanation	You entered a combination of memory unit size and display format that is not available. Try a different format combination, or use the "info formatting" command to list the default format settings.
A215	Message	Conflicting floating point mode and format type.
	Type	ERROR
	Explanation	You entered a format specification that contains conflicting floating point and non-floating point formats. Try a different format specification.
A216	Message	Floating point precision specification is not valid.
	Type	ERROR
	Explanation	You entered a floating point width and precision specification that is not valid. Try a different specification.

ID		Description
A217	Message	Pointer expression references an incomplete object type.
	Type	ERROR
	Explanation	The pointer expression you entered references an object that is an incomplete data type or a void data type. Use the "info expression" command to determine the data type of the object referenced by the pointer expression.
A218	Message	Cannot read symbolic location. Cause: *<errno description>*.
	Type	ERROR
	Explanation	CXdb could not read a symbolic location in the program data for the reason indicated. Refer to the errno.h(3) man page for further details describing the failure code.
A219	Message	Cannot write to symbolic location. Cause: *<errno description>*.
	Type	ERROR
	Explanation	CXdb could not write to a symbolic location in the program data for the reason indicated. Refer to the errno.h(3) man page for further details describing the failure code.
A220	Message	*<operation>* not available.
	Type	ERROR
	Explanation	The indicated feature is not available. Either the core file you are debugging was not produced on a C2 Series machine, or the current process is not running on a C2 Series machine. Use a different feature, or run the process on a C2 Series machine.
A221	Message	Abbreviation *'<string>'* is not unique among:*<field identifier list>*
	Type	ERROR
	Explanation	The abbreviation you entered matches more than one structure or union name. Try the entry again with a unique abbreviation.
A222	Message	Cannot allocate memory without "sbrk" linked in the process.
	Type	ERROR
	Explanation	You tried to perform an operation that requires permanently allocating memory within the address space of the target process. This cannot be done unless the process image contains the "sbrk" library function. Try relinking your application to include this library function, or remove the operation that requires memory allocation.

ID		Description
A223	Message	Cannot allocate <*count*> bytes in the target process.
	Type	ERROR
	Explanation	You tried to perform an operation that requires permanently allocating memory within the address space of the target process. This operation failed for one of the following reasons: swap space is exhausted, the program has exceeded its data size limit as set by setrlimit(2), or the maximum possible size of a data segment (compiled into the system) was exceeded.
A224	Message	Process [#<*process number*>] is already stopped.
	Type	ERROR
	Explanation	You tried to stop execution of a process that is already stopped. Use the "info process" command to determine the complete status of the process.
A225	Message	Cannot evaluate string literals without a process image.
	Type	ERROR
	Explanation	CXdb cannot evaluate string literals unless a process image exists. Use the "run", "rerun", or "attach" command to create a process image.
A226	Message	Cannot allocate memory for string literal.
	Type	ERROR
	Explanation	CXdb cannot allocate storage for your string literal in the virtual memory space of the debugged process.
A227	Message	Cannot find window [#<*window identifier*>].
	Type	ERROR
	Explanation	CXdb could not find the window number you specified. Try the command again with a different window number.
A228	Message	Window [#<*window identifier*>] is not a Source Code window.
	Type	ERROR
	Explanation	The command you entered operates only on Source Code windows, but the window you specified is not a Source Code window. Try the command again with the number for a Source Code window, or use a different command.
A229	Message	Cannot find a command matching string '<*string*>'.
	Type	ERROR
	Explanation	The text you specified in the "recall" command did not match any of the commands currently in the command history. Use the "info history" command to list the command history.

ID		Description
A230	Message	Operation not available in line mode.
	Type	ERROR
	Explanation	You requested an operation that is not available in line mode. Use the X Windows mode, or try a different command.
A231	Message	There is no previous command in the command history.
	Type	ERROR
	Explanation	The command history is empty, so there is no previous command to recall.
A232	Message	Entry '*<string>*' is not a valid environment variable for CXdb.
	Type	ERROR
	Explanation	When starting CXdb, your shell passed the indicated environment variable to CXdb. This variable was not defined properly, so CXdb ignored it. Use the "add default environemnt" command to add the variable to the environment for CXdb, or correct the variable definition in your shell and start CXdb again.
A233	Message	File '*<string>*' is not on the viewport list.
	Type	ERROR
	Explanation	You cannot remove the indicated viewport file because it is not on the current viewport list. Use the "info cxdb" command to list the current viewports.
A234	Message	Window '*<integer>*' is not on the viewport list.
	Type	ERROR
	Explanation	You cannot remove the indicated viewport window because it is not on the current viewport list. Use the "info cxdb" command to list the current viewports.
A235	Message	Stream '*<string>*' is not on the viewport list.
	Type	ERROR
	Explanation	You cannot remove the indicated viewport stream because it is not on the current viewport list. Use the "info cxdb" command to list the current viewports.
A236	Message	Command aborted because of null viewport filename.
	Type	ERROR
	Explanation	CXdb aborted the command because the viewport filename was null. This is an internal error.

ID		Description
A237	Message	Command aborted because window '*<integer>*' is not a valid viewport.
	Type	ERROR
	Explanation	CXdb aborted the command because the indicated window was not a valid viewport. This is an internal error.
A238	Message	Command aborted because of null viewport stream.
	Type	ERROR
	Explanation	CXdb aborted the command because the viewport stream was null. This is an internal error.
A239	Message	FormatCode is not valid.
	Type	ERROR
	Explanation	An internal error occurred while formatting a piece of data. Please report this to the CONVEX Technical Assistance Center.
A240	Message	Data type descriptor is not valid.
	Type	ERROR
	Explanation	An internal error occurred while formatting a piece of data. Please report this to the CONVEX Technical Assistance Center.
A241	Message	Cannot use REAL*16 in IEEE floating point mode.
	Type	ERROR
	Explanation	You cannot use REAL*16 data types with the IEEE floating point mode. Use the native floating point mode, or try a different data type.
A242	Message	Operand is not compatible with "/C" format.
	Type	ERROR
	Explanation	CXdb could not convert the data to a complex floating point format. Try a different operand or a different format.
A243	Message	Operand is not compatible with "/i" format.
	Type	ERROR
	Explanation	CXdb could not convert the data to an instruction. The data must be at least two bytes long to be converted to an instruction. Try a different operand or a different format.
A244	Message	Cannot find the pattern '*<string>*'.
	Type	ERROR
	Explanation	CXdb cannot find the indicated pattern in the Source Code window you specified. Try the command again with a different pattern or a different Source Code window.

ID		Description
A245	Message	Descending scope specification is missing a block specifier.
	Type	ERROR
	Explanation	Descending scope specifications (those that look forward in the scope chain) require at least one intervening block specifier before specifying the target identifier. Refer to the "scope" concepts page for more information.
A246	Message	Program scope has not yet been established.
	Type	ERROR
	Explanation	Descending scope specifications (those that look forward in the scope chain) require the process to have an active scope, but this process does not have a scope because it has not been started yet. Use the "run" or "rerun" command to start the process, or specify a complete scope path starting with a block specifier that is global to the entire program.
A247	Message	Operand is not compatible with data format.
	Type	ERROR
	Explanation	Data that requires sixteen bytes of storage, such as REAL*16 and COMPLEX*16, cannot be formatted as an integral. Use a different data format or a different operand.
A248	Message	Left operand of "()" is not a function.
	Type	ERROR
	Explanation	The operand of "()" must be a function. Try specifying a scope path for the operand, or use a different operand or a different operator.
A249	Message	Cannot format data as floating point.
	Type	ERROR
	Explanation	CXdb could not format your data as a floating point number because the data type is not compatible with floating point format. Try a different format.
A250	Message	Array slice notation is not valid in this context.
	Type	ERROR
	Explanation	You cannot use array slice notation in this case. Try using standard array subscripting.
A251	Message	Ignoring <*thread-id*> because it is not active.
	Type	WARNING
	Explanation	CXdb has ignored the specified thread because that thread was not active.

ID		Description
A252	Message	Upper bound of array slice is not valid.
	Type	ERROR
	Explanation	The upper bound you specified for an array slice must be greater than or equal to the lower bound. Try specifying the array slice again with a larger upper bound or a smaller lower bound.
A253	Message	Cannot access memory without a process image.
	Type	ERROR
	Explanation	CXdb could not complete your command because no process image exists. Use the "run", "rerun", or "attach" command to create a process image.
A254	Message	Region of <*size*> bytes is too large to watch.
	Type	ERROR
	Explanation	The region you specified in a "watch" or "event modify" command is too large for CXdb to monitor. Try specifying a smaller region to monitor.
A255	Message	Left operand of "[]" is not an array.
	Type	ERROR
	Explanation	The left operand of an array slice expression must be an array. If the operand you specified is a pointer, try casing it to an array and defining the bounds of the array.
A256	Message	All threads selected in process [#<*process number*>] are exiting.
	Type	ERROR
	Explanation	All threads you selected in the indicated process are exiting due to a join, wfork, or idle instruction. Try specifying at least one active thread along with the threads you previously selected.
A257	Message	Integer precision is too small or too large.
	Type	ERROR
	Explanation	Integer constants and loader symbols used as function designators have a precision of four bytes. This precision is either too small or too large to hold the result of the function expression you specified. Use the "info expression" command to determine the precision required by your function expression.
A258	Message	Cannot perform pointer arithmetic on function pointers.
	Type	ERROR
	Explanation	You cannot use function pointers in arithmetic expressions. For more information, refer to the C language manual.

ID		Description
A259	Message	Setting of *<count>* is not valid for maxarray. Allowed range is 1 to 2147483648.
	Type	WARNING
	Explanation	The value you specified with the "set printopts maxarray" command is not valid, so CXdb has ignored this setting. Try setting maxarray again with a value greater than zero and less than 2147483648.
A260	Message	Wildcard specifications are not allowed in this command.
	Type	ERROR
	Explanation	The wildcard character (*) is not allowed with this command. For the correct command syntax, refer to the CXdb Quick Reference or the CXdb Reference manual.
A261	Message	Cannot evaluate array slice due to memory request failure.
	Type	ERROR
	Explanation	CXdb could not evaluate your array slice expression because it could not allocate sufficient space on its internal heap. Please report this error to the CONVEX Technical Assistance Center.
A262	Message	Using memory unit size of bytes for character format.
	Type	WARNING
	Explanation	Character format requires a memory unit size of bytes. If you have specified any other memory unit size, your specification is ignored.
A263	Message	There are no text lines in file *<filename>*.
	Type	ERROR
	Explanation	There are no lines of text in the indicated source file. Try a different source file.
A264	Message	There are no source units for file *<filename>*.
	Type	ERROR
	Explanation	There are no source units available for the indicated source file. Try updating the search path so that CXdb can find the correct CTI data files for your source file.
A265	Message	Expression too complex: *<expression>*
	Type	ERROR
	Explanation	Your language expression is too complex for CXdb to analyze. This condition can occur if the expression is arbitrarily large, the expression is nested too deeply, or the expression is the result of successively expanding a recursive macro. Try to simplify the expression.

ID		Description
A266	Message	Command line is too complex.
	Type	ERROR
	Explanation	The command line your entered is too complex for CXdb to analyze completely. This condition can occur if a CXdb command is nested too deeply or the command has become arbitrarily complex as a result of successively expanding a recursive macro or alias. Try to simplify the command, then enter it again.
A267	Message	Initializer expression type is not compatible with *'<string>'*.
	Type	ERROR
	Explanation	The data type of the initializer expression is not compatible with the optional memory type you specified in the command. If the expression syntax is C, try casting the initializer expression to match the precision of the memory type specified in the command. If the expression is in Fortran, try using a built-in intrinsic (such as INT(), REAL(), QEXT(), CMPLX(), DCMPLX(), etc.) to convert the initializer to the precision of the specified memory type. You can also use the "info expression" command to determine the data type and precision of the initializer expression.
A268	Message	Precision of COMPLEX initializer is not compatible with *'<string>'*.
	Type	ERROR
	Explanation	The precision of the COMPLEX data type obtained from the initializer expression is not compatible with the optional memory type you specified in the command. Try using a built-in intrinsic (such as INT(), REAL(), QEXT(), CMPLX(), DCMPLX(), etc.) to convert the initializer to the precision of the specified memory type. You can also use the "info expression" command to determine the data type and precision of the initializer expression.
A269	Message	Precision of floating point initializer is not compatible with *'<string>'*.
	Type	ERROR
	Explanation	The precision of the floating point data type obtained from the initializer expression is not compatible with the optional memory type you specified in the command. If the expression syntax is C, try casting the initializer expression to match the precision of the memory type specified in the command. If the expression is in Fortran, try using a built-in intrinsic (such as INT(), REAL(), QEXT(), CMPLX(), DCMPLX(), etc.) to convert the initializer to the precision of the specified memory type. You can also use the "info expression" command to determine the data type and precision of the initializer expression.

ID		Description
A270	Message	Memory region <*address*>..<*address*> (<*count*> bytes) too small for initializer (<*count*> bytes). Initialization not complete.
	Type	WARNING
	Explanation	The specified memory region to be initialized does not contain enough space to permit an even multiple of stores. The last element cannot be written without exceeding the region specified. Therefore, these remaining bytes were not initialized.
A271	Message	Precision <*count*> of initializer exceeds size <*size*> of memory region.
	Type	ERROR
	Explanation	The precision of the initializer expression is greater than the capacity of the specified memory region. If the expression syntax is C, try casting the initializer expression to match the precision corresponding to the capacity of the specified memory region. If the expression is in Frotran, try using a built-in intrinsic (such as INT(), REAL(), QEXT(), CMPLX(), DCMPLX(), etc.) to convert the initializer to the precision of the specified memory type. You can also use the "info expression" command to determine the data type and precision of the initializer expression.
A272	Message	Evalopts precision setting must be 4 or 8.
	Type	ERROR
	Explanation	The evalopts settings for iprecision and rprecision must be either 4 or 8. Try the command again with a precision of 4 or 8.
A273	Message	Ignore count of <*count*> is not valid.
	Type	ERROR
	Explanation	The ignore count must be greater than or equal to zero. If you use a debugger variable to specify the ignore count, CXdb converts the debugger variable value to an integer before using it as the ignore count. Rounding or truncation can affect the result of this conversion. Try the command again with a different ignore count.
A274	Message	Too many statements in command line: <*command line*>
	Type	ERROR
	Explanation	The CXdb command line contains too many statements. This condition can result from expansion of a macro or alias that is directly or indirectly recursive. Try to simplify the command, then enter it again.
A275	Message	The '<*string*>' command expects a <*type identifier*>, but received a <*type identifier*>.
	Type	ERROR
	Explanation	The command you have completed with command composition was expecting one data type but received another. Try the command again with the appropriate data type.

ID		Description
A276	Message	Process [#*<process number>*] has invalid state (*<variable identifier>*): *<count>*.
	Type	ERROR
	Explanation	This is an internal error that will probably lead to other errors. Please report it to the CONVEX Technical Assistance Center.
A277	Message	PIXRUN error on process [#*<process number>*]: *<errno description>*
	Type	ERROR
	Explanation	This is an internal error. Please report it to the CONVEX Technical Assistance Center.
A278	Message	Error trying to *<get/set>* process attributes on process [#*<process number>*]: *<errno description>*
	Type	ERROR
	Explanation	This is an internal error. Please report it to the CONVEX Technical Assistance Center.
A279	Message	Error trying to *<clear/set>* the inherit mode on process [#*<process number>*]: *<errno description>*
	Type	ERROR
	Explanation	This is an internal error. Please report it to the CONVEX Technical Assistance Center.
A280	Message	Error trying to detach process [#*<process number>*]: *<errno description>*
	Type	ERROR
	Explanation	This is an internal error. Please report it to the CONVEX Technical Assistance Center.
A281	Message	Error trying to get signal subcode on thread [#*<process number>*]: *<errno description>*
	Type	ERROR
	Explanation	This is an internal error. Please report it to the CONVEX Technical Assistance Center.
A282	Message	Error trying to *<step/continue>* thread [#*<thread-id>*]: *<errno description>*
	Type	ERROR
	Explanation	This is an internal error. Please report it to the CONVEX Technical Assistance Center.
A283	Message	Error trying to select thread [#*<thread-id>*]: *<errno description>*
	Type	ERROR
	Explanation	This is an internal error. Please report it to the CONVEX Technical Assistance Center.

ID		Description
A284	Message	Error trying to get CWD for process [#*<process number>*]: *<errno description>*
	Type	ERROR
	Explanation	This is an internal error. Please report it to the CONVEX Technical Assistance Center.
A285	Message	Error trying to get thread count for process [#*<process number>*]: *<errno description>*
	Type	ERROR
	Explanation	This is an internal error. Please report it to the CONVEX Technical Assistance Center.
A286	Message	Process [#*<process number>*] with PID *<process number>* is terminated. Kill the process, then restart it.
	Type	ERROR
	Explanation	CXdb could not start the target process because that process appeared to be terminated. First use the "kill process" command on the target process, then try starting it again.
A287	Message	Process [#*<process number>*] with PID *<process number>* in unknown wait state. Kill the process, then restart it.
	Type	ERROR
	Explanation	CXdb could not start the target process because that process was in an unknown state. First use the "kill process" command on the target process, then try starting it again.
A288	Message	Error trying to get uarea for process [#*<process number>*]: *<errno description>*
	Type	ERROR
	Explanation	This is an internal error. Please report it to the CONVEX Technical Assistance Center.
A289	Message	Error trying to pattach process with pid *<process number>*: *<errno description>*
	Type	ERROR
	Explanation	This is an internal error. Please report it to the CONVEX Technical Assistance Center.
A290	Message	Event *<event-id>* is already disabled.
	Type	ERROR
	Explanation	You have tried to disable an eventpoint that is already disabled.

ID		Description
A291	Message	Event *<event-id>* is already enabled.
	Type	INFO
	Explanation	You have tried to enable an eventpoint that is already enabled.
A292	Message	Lower bound (*<count>*) cannot be larger than upper bound (*<count>*).
	Type	ERROR
	Explanation	The lower bound of an array slice must be less than or equal to the upper bound. Try the command again with a different upper or lower bound for the array slice.
A293	Message	Search string given to "recall" command is null.
	Type	ERROR
	Explanation	You cannot pass a null search string to the "recall" command. Try the command again with a different search string.
A294	Message	Number *'<integer>'* is not valid for history elements.
	Type	ERROR
	Explanation	The number of history elements you specified is not valid. Try the command again with a number greater than zero.
A295	Message	There is no source file corresponding to the current CTI object file.
	Type	ERROR
	Explanation	There is no source file corresponding to the currently active Compiler-Tools Interface object file. This is an internal error. Please report it to the CONVEX Technical Assistance Center.
A296	Message	No source file object was passed to a source window screen update function.
	Type	ERROR
	Explanation	This is an internal error. Please report it to the CONVEX Technical Assistance Center.
A297	Message	Cannot use a null string with the "find window" command.
	Type	ERROR
	Explanation	You cannot specify a null search string for the "find window" command. Try the command again with a different search string.
A298	Message	No source unit exists in an internal source unit tree.
	Type	ERROR
	Explanation	This is an internal error. Please report it to the CONVEX Technical Assistance Center.

ID		Description
A299	Message	No source unit exists at line <*count*> column <*count*> of the file '<*string*>'.
	Type	ERROR
	Explanation	This is an internal error. Please report it to the CONVEX Technical Assistance Center.
A300	Message	The currently selected source unit has no parent source unit.
	Type	ERROR
	Explanation	The currently selected source unit has no parent source unit. The most likely cause is that the currently selected source unit is a routine. Try the command again with a different source unit.
A301	Message	The currently selected source unit has no child source unit.
	Type	ERROR
	Explanation	The currently selected source unit has no child source units. The most likely cause is that the currently selected source unit is an innermost source unit. Try the command again with a different source unit.
A302	Message	The currently selected source unit has no sibling source unit.
	Type	ERROR
	Explanation	The currently selected source unit has no further sibling source units. However, it may have other siblings. These other sibling source units can be found by finding the parent of the current source unit and then selecting the child and further sibling source units of the parent.
A303	Message	File '<*string*>' is not a known source file.
	Type	ERROR
	Explanation	CXdb does not recognize the specified file as a source file. Try the command again with a different file name. or use the "add path" command to tell CXdb the location of your source file.
A304	Message	Aborting handler for eventpoint #<*event-id*> because another handler is already active.
	Type	ERROR
	Explanation	CXdb aborted your handler for the indicated eventpoint because another handler was already active for this eventpoint. This situation can occur when the eventpoint handler calls one of your program functions. Try defining the handler again without a function call.

ID		Description
A305	Message	Process [#*<process number>*/*<thread-id>*] stopped by *<signal identifier>* and cannot be resumed.
	Type	ERROR
	Explanation	A hardware signal such as SIGBUS, SIGSEGV, or SIGILL has stopped execution of your process. You cannot resume or continue execution of the process after one of these signals. Use the "run" or "rerun" command to start a new process.
A306	Message	Integer overflow on *<string>*.
	Type	ERROR
	Explanation	The number you entered is too large. All numbers used in CXdb commands (for repeat counts, line numbers, ignore counts, etc.) must be less than 2147483647, the maximum value of a 4-byte signed integer. Try the command again with a smaller number.
A307	Message	Data type *<type identifier>* is not compatible with function return value.
	Type	ERROR
	Explanation	You have specified a return value that is not compatible with the data type of the function. Use the "info expression" command to determine the data type of the return value expected by the function.
A308	Message	Void return type. Using type of expression result.
	Type	WARNING
	Explanation	You have specified a return value for a function that has a return type of void. CXdb used the type of the expression specified when performing the return. There is no harm done if the caller does not expect a return value but one is returned.
A309	Message	No CTI information on current function. Assuming *<type identifier>* return type.
	Type	WARNING
	Explanation	There is no CTI information pertaining to the current function. CXdb used the indicated return type when converting the return value you specified. (This messages is no longer used.)
A310	Message	Initial process failed to exec. Cause: *<errno description>*
	Type	ERROR
	Explanation	Startup of the target process failed for the indicated reason. This generally indicates something is wrong with the installation of CXdb. Please inform your system administrator.

ID		Description
A311	Message	<*type identifier*> process failed to exec. Startup terminated.
	Type	ERROR
	Explanation	Startup of the target process failed for one of the following reasons: the file is marked executable but is not in a format that the kernel will exec; the executable is not meant for the current architecture (such as running a C2 executable on a C1); or a system limit may have been exceeded.
A312	Message	Inconsistent induction value for <*variable identifier*>. Using <*string*>, from the set of {<*string list*>}.
	Type	INFO
	Explanation	Several equations are used to calculate the loop induction value. Sometimes these equations can produce inconsistent results. All calculated values are displayed here, and the lowest value is arbitrarily chosen for use in expressions.
A313	Message	Inconsistent induction value for <*variable identifier*> from the set of {<*string list*>}.
	Type	ERROR
	Explanation	The equations used to calculate the loop induction value have produced inconsistent results. All calculated values are displayed here, but the operation cannot proceed because of this inconsistency.
A314	Message	Calculated induction value for following <*count*> variable(s): <*string list*>
	Type	INFO
	Explanation	The loop induction variable was removed and replaced by the listed synthesized variables.
A315	Message	Object file name <*filename*> is ambiguous.
	Type	ERROR
	Explanation	CXdb could not find a unique object file with the indicated name. Try a different file name.
A316	Message	Breakpoint address <*address*> is not in a defined text region.
	Type	WARNING
	Explanation	The breakpoint address you specified does not lie within a defined text region of your program. This condition can occur normally if you are dynamically loading object code that does not lie in a text region. Otherwise, it is an error, and you should specify a different breakpoint address.

ID		Description
A317	Message	Alias name '*<string>*' conflicts with alias '*<string>*'.
	Type	ERROR
	Explanation	The alias name you tried to define conflicts with an alias name that already exists. For example, you cannot define both aliases "foo" and "foo bar". Choose a new alias name.
A318	Message	Invalid data for automatic dynamic load. *<errno description>*
	Type	ERROR
	Explanation	CXdb automatically processes dynamically loaded object files whenever the special function "_cxdb_dynamic_load" is called. The arguments to this function must contain the data necessary to load the CTI information for the object file. The arguments you specified did not contain the proper data, for the reason indicated. Try the function call again with new arguments.
A319	Message	Cannot assign value to *<variable identifier>*.
	Type	ERROR
	Explanation	CXdb could not assign a value to the indicated variable. Either the variable does not have storage assigned to it, or it is defined as a constant. Try a different variable or a different operation.
A320	Message	Subscript expression is not an arithmetic data type.
	Type	ERROR
	Explanation	Your subscript expression does not result in an arithmetic data type such as integer, logical, real or complex. Try a different subscript expression.
A321	Message	Cannot write to a core file.
	Type	ERROR
	Explanation	You cannot perform any operation that writes to a core file or tries to modify one of its variables.
A322	Message	Incompatible pointers for formal and actual parameter types.
	Type	WARNING
	Explanation	The data type of the actual parameter is incompatible with the declaration of the formal parameter. However, execution still continues because both parameters are pointers.
A323	Message	The "put" command requires a size for *<string>*.
	Type	ERROR
	Explanation	The "put" command requires a size specification for the indicated address expression. In this case, CXdb cannot infer the size from the address expression itself. Try the command again, using either an address range or a byte count for the size specification.

ID		Description
A324	Message	Cannot access file. Check read/write permissions.
	Type	ERROR
	Explanation	CXdb can neither read nor write to the current file. Check the file permissions to see if you have access to this file.
A325	Message	File <*filename*> is not a standard "put/get" file.
	Type	ERROR
	Explanation	You cannot use the "get" command on the indicate file because it was not created with the "put" command. Try a different file, or save the file first with the "put" command.
A326	Message	Cannot "get" data from file <*filename*>.
	Type	ERROR
	Explanation	The "get" command cannot restore the data from the indicated file because your program is not in the same state it was in when the file was created. Make sure the program counter (PC) is in the same location as when you executed the "put" command to create the file. Then try the "get" command again.
A327	Message	Size of address expression from command line is <*count*>, but size from data file is <*size*>.
	Type	WARNING
	Explanation	The size of the address expression you specified on the command line differs from the size used to store the data in the file. CXdb uses the smaller of the two sizes.
A328	Message	Cannot find file '<*string*>'.
	Type	ERROR
	Explanation	CXdb could not find the indicated file. Verify that the file exists and that you specified its name correctly.
A329	Message	Expression <*expression*> has invalid size of <*size*>.
	Type	ERROR
	Explanation	You specified a size value that is not valid for the indicated expression. Redefine the size of the expression.
A330	Message	Cannot find the following CTI files: <*filename*>
	Type	ERROR
	Explanation	CXdb could not find the CTI data files it needs to perform symbolic debugging of your program. These files should reside in your local .CTI directory. If you have moved these files, use the "dirpath" command to tell CXdb the new path to the files. If these files do not exist, recompile your program with the -cxdb option so that the compiler can create them in your local .CTI directory.

ID		Description
A331	Message	Cannot have a file on this viewport list.
	Type	ERROR
	Explanation	This command does not allow a file name on the viewport list. Use a window number for the viewport, or try a different viewport list.
A332	Message	Cannot determine expression value.
	Type	ERROR
	Explanation	CXdb cannot determine the value of your language expression. Try a different expression.
A333	Message	Cannot read from remote server. *<errno description>*
	Type	ERROR
	Explanation	CXdb could not read a packet from the remote server, for the reason indicated. This generally means that the connection to the remote server has been terminated or the network support services are failing for some reason. Try exiting from CXdb and starting your session over.
A334	Message	Encountered End-Of-File condition on remote server.
	Type	ERROR
	Explanation	CXdb encountered an EOF condition while trying to read a packet from a remote server. This generally means that the server has exited for some reason or the underlying network connection was terminated. Try exiting from CXdb and starting your session over.
A335	Message	Received invalid packet from remote server; length = *<count>*.
	Type	ERROR
	Explanation	CXdb received an improperly formed packet from the remote server. This means that the communication between the server and the client has been corrupted or is out of sequence in some way. This condition usually leads to more errors.
A336	Message	File *<filename>* is not an executable file.
	Type	ERROR
	Explanation	The file you specified is not an executable file. Try the command again with a different file name.
A337	Message	Cannot connect to remote host *<host name>*.
	Type	ERROR
	Explanation	CXdb cannot connect to the indicated remote host. Check your .rhosts file to see if this is a valid host name.

ID		Description
A338	Message	Cannot perform operation on command window <*window identifier*>.
	Type	ERROR
	Explanation	You cannot perform the specified operation on a command window. Try a different window number or a different operation.
A339	Message	Cannot create a Memory Display window.
	Type	ERROR
	Explanation	CXdb could not create a Memory Display window. Check to see if you have modified the X resources (Xdefaults) for this window.
A340	Message	Cannot create a Stack Trace window.
	Type	ERROR
	Explanation	CXdb could not create a Stack Trace window. Check to see if you have modified the X resources (Xdefaults) for this window.
A341	Message	No remote server associated with process <*process number*>.
	Type	ERROR
	Explanation	CXdb tried to access a process through a remote server, but there is currently no remote server for this process.
A342	Message	Cannot expand command line arguments. <*errno description*>
	Type	ERROR
	Explanation	CXdb could not expand the command line arguments you passed to the target process. Try the command again with different arguments.
A343	Message	The task priority <*count*> is out of range.
	Type	ERROR
	Explanation	The value you specified for an rtk task priority is out of range. Try the command again with a different priority value.
A344	Message	Cannot set task priority for task <*task-id*> of application <*count*>.
	Type	ERROR
	Explanation	CXdb could not set the task priority for the indicated task of the indicated rtk application.
A345	Message	Remote file system specification for process <*process number*> is not valid.
	Type	ERROR
	Explanation	The remote file system you specified for the indicated process is not valid. Try the command again with a different remote file system specification.

ID		Description
A346	Message	Cannot set remote file system *<filename>*.
	Type	ERROR
	Explanation	CXdb could not set the remote file system as specified. Try the command again with a different remote file system specification.
A347	Message	Cannot set memory size to *<size>*.
	Type	ERROR
	Explanation	CXdb could not set the maximum available memory to the size you specified for the remote application. Try the command again with a different maximum memory specification.
A348	Message	Cannot set maximum number of tasks to *<count>*.
	Type	ERROR
	Explanation	CXdb could not set the maximum number of tasks to the number you specified for the remote application. Try the command again with a different specification for maximum number of tasks.
A349	Message	Cannot set cpu affinity to *<count>*.
	Type	ERROR
	Explanation	CXdb could not set the cpu affinity to the value you specified for the remote application. Try the command again with a different specification for cpu affinity.
A350	Message	Cannot set default task priority to *<count>*.
	Type	ERROR
	Explanation	CXdb could not set the default task priority to the value you specified for the remote application. Try the command again with a different specification for default task priority.
A351	Message	Ring *<count>* is not a valid ring number.
	Type	ERROR
	Explanation	You have specified a ring number that is not valid. An application can execute in either user mode (ring 4) or kernel mode (ring 0). Try the command again using one of these two rings.
A352	Message	Cannot set ring mode to *<count>*.
	Type	ERROR
	Explanation	CXdb could not set the ring mode to the value you specified for the initial task of the remote application. Try the command again with a different specification for the ring mode.

ID		Description
A353	Message	Cannot set initial task private data size to <*size*>.
	Type	ERROR
	Explanation	CXdb could not set the initial task private data size to the value you specified for the remote application. Try the command again with a different specification for task private data size.
A354	Message	Cannot set initial task stack size to <*size*>.
	Type	ERROR
	Explanation	CXdb could not set the initial task stack size to the value you specified for the remote application. Try the command again with a different specification for task stack size.
A355	Message	Cannot set the name of application <*count*>.
	Type	ERROR
	Explanation	CXdb could not set the name of the application you specified. Try a different name.
A356	Message	Dynamic size <*size*> is not valid.
	Type	ERROR
	Explanation	The dynamic size stored in process memory was not valid. This could indicate some memory corruption.
A357	Message	You must first create a process with the "run", "rerun", or "attach" command.
	Type	ERROR
	Explanation	CXdb cannot access the process image because none exists. Use the "run", "rerun", or "attach" command to create a process image.
A358	Message	You cannot detach from an executable file.
	Type	ERROR
	Explanation	You cannot detach from a process image derived from an executable file. Use the "kill process" command to terminate the process. You can also use the "run" or "rerun" command to start a new process.
A359	Message	You cannot execute a remote command on a local process.
	Type	ERROR
	Explanation	The command you entered applies only to remote processes, but the current process is local. Try a different command.
A360	Message	Cannot create a remote process.
	Type	ERROR
	Explanation	CXdb could not create a remote process.

ID		Description
A361	Message	Cannot read the state of the remote process.
	Type	ERROR
	Explanation	CXdb could not read the state of the remote process. Either the remote process does not exist, or it cannot be accessed because it is currently running.
A362	Message	No primary selection exists.
	Type	ERROR
	Explanation	You performed a GUI action that requires a primary selection. Make a primary selection first, and then try the operation again.
A363	Message	Directory <*filename*> is not a rooted directory.
	Type	ERROR
	Explanation	This operation requires a rooted directory (one beginning with "/"). There is no default directory in this case. Try the operation again with a rooted directory.
A364	Message	The directory <*filename*> does not exist.
	Type	INFO
	Explanation	This directory does not exist. In this case, it is not an error.
A365	Message	Cannot generate file name <*filename*>.
	Type	ERROR
	Explanation	CXdb could not generate the indicated file name. CXdb uses the rules of the C shell (csh) for file name expansion. Try the operation again iwth a different file name.
A366	Message	Command <*command line*> is not supported on this architecture.
	Type	ERROR
	Explanation	The command you entered is not supported on this system. Refer to the "architecture dependencies" concepts page for a list of commands that are restricted to particular architectures.
A367	Message	Register <*register identifier*> does not exist.
	Type	ERROR
	Explanation	The register you specified does not exist on this system. Try a different register name, or refer to the "registers" concepts page for a list of accessible registers.

ID		Description
A368	Message	Cannot access register <*register identifier*> from current process object.
	Type	ERROR
	Explanation	You cannot access the indicated register from the context of the current process object. This situation can arise if architecture of the system where you are running CXdb is different than the architecture of the system where the target process was originally generated. To access this register, run CXdb on the same system as the target process you are trying to debug.
A369	Message	Cannot access register <*register identifier*>.
	Type	ERROR
	Explanation	CXdb could not access the specified register because it is reserved for system use only. Try a different register name.
A370	Message	Cannot access CIR register unless process is active.
	Type	ERROR
	Explanation	You cannot access the communication index register (CIR) from a core image or an executable image that has no running process associated with it. Use the "run", "rerun", or "attach" command to generate the image of a running process.
A371	Message	A system error occurred while attempting to acquire information for process <*process number or use of pathname*>.
	Type	ERROR
	Explanation	CXdb encountered an error while performing a system call on the indicated process. This can lead to additional errors. Subsequent error messages will indicate the reason for failure.
A372	Message	The address expression '<*integer*>' does not match any routine with CTI information.
	Type	ERROR
	Explanation	The address expression given for displaying a new routine in the source window did not evaluate to be within the range of any routine known to the compiler-debugger interface.
A373	Message	The line number '<*integer*>' is out of the bounds of the source file.
	Type	INFO
	Explanation	The line number given in the file name/line number specification was out of the range of line numbers for the given source file. The line number "1" was assumed.

ID		Description
A374	Message	Frame specified is out of range.
	Type	ERROR
	Explanation	You have specified a frame, either by absolute or relative number, which does not exist on the stack. You can use the "backtrace" command to see the frames currently on the stack.
A375	Message	Operation not available in batch mode.
	Type	ERROR
	Explanation	You requested an operation that is not available in batch mode. Use the line mode or X Windows mode, or try a different command.
A376	Message	Address <*address*> is not the base address of a known stack frame.
	Type	ERROR
	Explanation	You have specified a value that does not correspond to the base address of a known stack frame. (The fp register contains the base address of the current stack frame on C Series machines, and the sp register contains the base address on SPP Series machines.) You can use the "info frame" command to determine the base address of a particular stack frame.
A377	Message	Caller unknown. Cannot return from top stack frame.
	Type	ERROR
	Explanation	Either CXdb cannot unwind the stack from the current frame to find the caller, or you are attempting to return from process startup code.
A378	Message	Cannot recover saved frame context.
	Type	ERROR
	Explanation	CXdb could not recover the saved values of the registers necessary to restore the frame context. The necessary registers on SPP Series machines are sp, iioq_head, and iioq_tail; the necessary registers on C Series machines are fp, pc, ap, psw, and sp.
A379	Message	Cannot restore all callee-saved registers.
	Type	WARNING
	Explanation	CXdb could not restore the values of all the callee-saved registers. The return can still take place, but the program might produce unexpected results. This is a problem mainly if the calling routine (or a caller of it) is compiled with optimizations. It occurs if the called routine (or a routine it calls) has not been compiled with -cxdb. If possible, recompile the source file with the -cxdb option to provide the missing register spill information.

ID		Description
A380	Message	No CTI information on current function. C return semantics assumed.
	Type	`WARNING`
	Explanation	There is no CTI information pertaining to the current function. It is assumed that the code adheres to the parameter passing and return value calling convention applicable to a C program. CXdb used the type of the expression specified when performing the return.
A381	Message	Unknown return type. C return semantics assumed.
	Type	`WARNING`
	Explanation	CXdb does no know the expected return value type for the function. It is assumed that the code adheres to the parameter passing and return value calling convention applicable to a C program. CXdb used the type of the expression specified when performing the return.
A382	Message	Unknown return value area. Cannot return type larger than 8 bytes: <*type identifier*>.
	Type	`ERROR`
	Explanation	Types larger than 8 bytes (<*type identifier*>) are returned in a memory area specified by the caller. CXdb cannot deduce this area from the current program state.
A383	Message	Unknown return value area. Cannot return Fortran strings.
	Type	`ERROR`
	Explanation	Fortran strings are returned in a memory area specified by the caller. CXdb cannot deduce this area from the current program state.
A384	Message	Conversion of expression result to function return type <*type identifier*> failed.
	Type	`ERROR`
	Explanation	CXdb could not convert your expression to the function's expected return type. Try the "return" command again with a different expression. (In C, you can also try casting the expression to a data type that can be converted.)
A385	Message	Cannot read register <*register identifier*>.
	Type	`ERROR`
	Explanation	CXdb could not retreive the value of register <*register identifier*>. This is an internal error. Please report it to the CONVEX Technical Assistance Center.

ID		Description
A386	Message	Cannot write register <*register identifier*>.
	Type	ERROR
	Explanation	CXdb could not update the value of register <*register identifier*>. This is an internal error. Please report it to the CONVEX Technical Assistance Center.
A387	Message	File <*filename*> is not a command file.
	Type	ERROR
	Explanation	The file you specified as a command file is an excutable object image. Please enter the correct command file name or remove execution permission from the command file.
A388	Message	Zero length return type: <*type-ID*>. No value returned.
	Type	WARNING
	Explanation	The return type of the function has a size that is less than or equal to zero. This is a programming error that is not always detected by the compiler. Control has returned to the caller without setting any return value. The file you specified as a command file is an excutable object image. Please enter the correct command file name or remove execution permission from the command file.
A389	Message	Illegal pointer/integer combination.
	Type	WARNING
		An attempt has been made to take the address of either (1) an operand that is not a function designator, or (2) an lvalue that designates an object which is a bit field, or (3) an object that has been declared with register storage class.
A390	Message	Cannot allocate memory for expression result.
	Type	ERROR
		CXdb cannot allocate storage for the result of an expression evaluation in its internal heap. Try evaluating the expressions with a shorter string length, smaller array cross section, or using separate fields of a large structure.
A391	Message	Invalid argument mask. Arguments may be passed in incorrect registers.
	Type	WARNING
		The argument mask corresponding to the called routine is not valid. This can happen if the portion of the mask for the first or third argument has both bits set, which is only allowable for the second and fourth arguments. CXdb will not necessarily relocate the arguments to the appropriate registers in such a case.

ID		Description
A392	Message	:Out of order return from a CXdb call.
	Type	ERROR
		The program has attempted to return from a CXdb call while performing another CXdb call. The following sequence of events give rise to this case in the presence of multiple threads: the user invokes a CXdb call on a particular thread, execution is interrupted before it returns, another CXdb call is invoked on an different thread, exeuction is halted before it returns, then the former thread is resumed and attempts to return from the CXdb call before the latter thread returns.
C1	Message	Cannot open executable file <*filename*>.
	Type	ERROR
	Explanation	The indicated executable file could not be opened. Check to make sure the file exists and that you have permission to access it. If the file does exist, it might contain data that has been corrupted. In that case, try recompiling your program to create a new executable file.
C2	Message	Cannot open location range table for <*filename*>.
	Type	ERROR
	Explanation	Could not open the location range table (.lrt file) for your executable file. The .lrt file should reside in your local .CTI directory. If you have moved this file, use the "add path" command to specify the new location of the file. If the .lrt file does not exist (or if it exists but contains corrupted data), recompile your program with the appropriate option (-cxdb for CXdb and -cxpa, -cxpab, or -cxpar for CXpa) so that the compiler can generate the necessary CTI data files.
C3	Message	Cannot open source unit table for <*filename*>.
	Type	ERROR
	Explanation	Could not open the source unit table (.sut file) for your executable file. The .sut file should reside in your local .CTI directory. If you have moved this file, use the "add path" command to specify the new location of the file. If the .sut file does not exist (or if it exists but contains corrupted data), recompile your program with the appropriate option (-cxdb for CXdb and -cxpa, -cxpab, or -cxpar for CXpa) so that the compiler can generate the necessary CTI data files.

ID		Description
C4	Message	Cannot open type and scope information table for <*filename*>.
	Type	ERROR
	Explanation	Could not open the type and scope information table (.tsi file) for your executable file. The .tsi file should reside in your local .CTI directory. If you have moved this file, use the "add path" command to specify the new location of the file. If the .tsi file does not exist (or if it exists but contains corrupted data), recompile your program with the appropriate option (-cxdb for CXdb and -cxpa, -cxpab, or -cxpar for CXpa) so that the compiler can generate the necessary CTI data files.
C5	Message	Pathname <*filename*> is not a valid directory.
	Type	ERROR
	Explanation	The pathname you specified is not a directory. Try a different pathname.
C6	Message	Cannot open executable file for source file <*filename*>.
	Type	ERROR
	Explanation	Could not open the executable file associated with the indicated source file. Check to make sure the executable file exists and that you have permission to access it.
C7	Message	Cannot find source file for <*filename*> in search path.
	Type	ERROR
	Explanation	Could not find the source file associated with your executable file. If you have moved the source file since compiling it, use the "add path" command to specify the new location of this file.
C8	Message	Cannot find CTI data files for <*filename*> in search path.
	Type	ERROR
	Explanation	Cannot find the CTI (Compiler-Tools Interface) data files associated with your program. If you have moved these files since compiling your program, use the "add path" command to tell CXdb the new location of the CTI data files. If these files do not exist, you can generate them by recompiling your program with the appropriate option (-cxdb for CXdb and -cxpa, -cxpab, or -cxpar for CXpa).
C9	Message	Cannot open variable table for <*filename*>.
	Type	ERROR
	Explanation	Could not open the variable table (.vt file) for your executable file. The .vt file should reside in your local .CTI directory. If you have moved this file, use the "add path" command to specify the new location of the file. If the .vt file does not exist (or if it exists but contains corrupted data), recompile your program with the appropriate option (-cxdb for CXdb and -cxpa, -cxpab, or -cxpar for CXpa) so that the compiler can generate the necessary CTI data files.

C10 Message Ambiguous file name <*filename*>. Use a qualified pathname.

Type ERROR

Explanation The specified source file name is ambiguous. Use a path name to qualify the source file name.

C11 Message Source file <*filename*> is out of date. Last modified <*date*>; compiled <*date*>.

Type WARNING

Explanation The specified source file is out of date with respect to the version used for compilation. This is only a warning; the specified version of your source file will be used.

C12 Message Object file <*filename*> compiled with early version of compiler.

Type WARNING

Explanation The indicated object file was compiled with a version of the compiler that did not provide all the features currently available in CONVEX products. This might limit some of your results. For maximum capability, recompile the indicated file with the CONVEX Fortran V9.0 (or later) compiler or the CONVEX C V6.0 (or later) compiler.

C13 Message Cannot find file <*filename*> within current search path.

Type ERROR

Explanation Could not find the specified file in your current search path. Use the "info path" command to display the current search path, and use the "add path" command to add directories to the search path.

C14 Message Cannot open name space table for <*filename*>.

Type ERROR

Explanation Could not open the name space table (.ns file) for your executable file. The .ns file should reside in your local .CTI directory. If you have moved this file, use the "add path" command to specify the new location of the file. If the .ns file does not exist (or if it exists but contains corrupted data), recompile your program with the appropriate option (-cxdb for CXdb and -cxpa, -cxpab, or -cxpar for CXpa) so that the compiler can generate the necessary CTI data files.

C15 Message Data file <*filename*> is out of date. Last compiled <*date*>; created <*date*>.

Type ERROR

Explanation The indicated data file is out of date with respect to the executable file. Try relinking the executable file to include the correct version of the data file. You can also use the "add path" command to add directories to the search path if you have changed the location of the source file(s).

ID		Description
C16	Message	File <*filename*> compiled with prerelease compiler.
	Type	WARNING
	Explanation	The indicated source file was compiled with a prerelease version of the Fortran compiler. Recompile this file to obtain full symbolic debugging information.
C17	Message	Cannot read line <*count*> from file <*filename*>.
	Type	ERROR
	Explanation	Could not read from the indicated source file. If you have modified this file since its last compilation, try compiling the file again.
C18	Message	Cannot find CTI expression table information for <*filename*>.
	Type	ERROR
	Explanation	Could not open the expression table (.xpt file) for your executable file. The .xpt file should reside in your local .CTI directory. If you have moved this file, use the "add path" command to specify the new location of the file. If the .xpt file does not exist (or if it exists but contains corrupted data), recompile your program with the appropriate option (-cxdb for CXdb and -cxpa, -cxpab, or -cxpar for CXpa) so that the compiler can generate the necessary CTI data files.
C19	Message	File <*filename*> is not a valid object file. <*errno description*>
	Type	ERROR
	Explanation	The specified file is not a valid object file, for the reason indicated. Try a different object file name.
C20	Message	Cannot find CTI call relation table information for <*filename*>.
	Type	ERROR
	Explanation	Could not find the call relation table information for your executable file. This information should reside in the CTI data files in your local .CTI directory. If you have moved the CTI data files, use the "add path" command to specify the new location of these files. If the CTI data files do not exist (or if they exist but contain corrupted data), recompile your program with the appropriate option (-cxdb for CXdb and -cxpa, -cxpab, or -cxpar for CXpa) so that the compiler can generate the necessary CTI data files.

ID		Description
C21	Message	Cannot find CTI backend information for <*filename*>.
	Type	ERROR
	Explanation	Could not find the CTI backend information (.be file) for your executable file. The .be file should reside in your local .CTI directory. If you have moved this file, use the "add path" command to specify the new location of the file. If the .be file does not exist (or if it exists but contains corrupted data), recompile your program with the appropriate option (-cxdb for CXdb and -cxpa, -cxpab, or -cxpar for CXpa) so that the compiler can generate the necessary CTI data files.
C22	Message	Cannot find CTI monitor descriptor table information for <*filename*>.
	Type	ERROR
	Explanation	Could not find the monitor descriptor table information for your executable file. This information should reside in the CTI data files in your local .CTI directory. If you have moved the CTI data files, use the "add path" command to specify the new location of these files. If the CTI data files do not exist (or if they exist but contain corrupted data), recompile your program with the appropriate option (-cxdb for CXdb and -cxpa, -cxpab, or -cxpar for CXpa) so that the compiler can generate the necessary CTI data files.
D1	Message	Stub table not found in current executable.
	Type	ERROR
	Explanation	A critical data structure could not be found in your current program. This prevents the backtrace command from working. You may either continue debugging cautiously or try recompiling and relinking your application. If this persists, please submit a problem report.
D2	Message	The unwind table not found in current executable.
	Type	ERROR
	Explanation	A critical data structure could not be found in your current program. This prevents the backtrace command from working. You may either continue cautiously or try recompiling and relinking your application. If this persists, please submit a problem report.
D3	Message	Current executable is unreadable.
	Type	ERROR
	Explanation	The current executable cannot be read. No further work can continue on this executable. Try recompiling and relinking your application. If this persists, please submit a problem report.

ID		Description
D4	Message	File *<filename>* is not a core file.
	Type	ERROR
	Explanation	The given file is not recognized as a core file. A core file is a file created when your program aborts unexpectedly. No other file can be a core file. If this file was created by an unexpected abort, then it may have been corrupted. Recreate a new core file by running your program again and be sure that you have enough disk space. If this problem persists, please submit a problem report.
D5	Message	Cannot find symbolic support in current executable.
	Type	ERROR
	Explanation	The current executable does not have any symbolic support. Both debugging and performance analysis may precede; however, many important features will not work such as loop instrumentation, source code to program counter correlation, and printing user variables. If you want these features, please recompile your source code with the appropriate option (-cxdb for CXdb; -cxpa, -cxpar, or -cxpab for CXpa).

ID	Description

Master Index

This is a master index for both volumes of the *CXdb Reference*. Pages are numbered sequentially across both volumes and are distributed as follows:

- Pages 1 to 590— *CXdb Reference*, Volume 1 (Commands and Parameters)
- Pages 591 to 1038—*CXdb Reference*, Volume 2 (Concepts, Windows, and Messages)

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

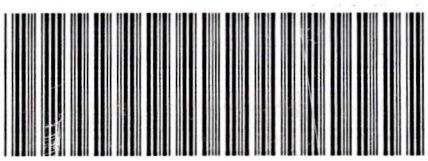
V

W

X

ORDER NUMBER
DSW-472

DOCUMENT NUMBER
710-015022-000



CONVEX PRESS