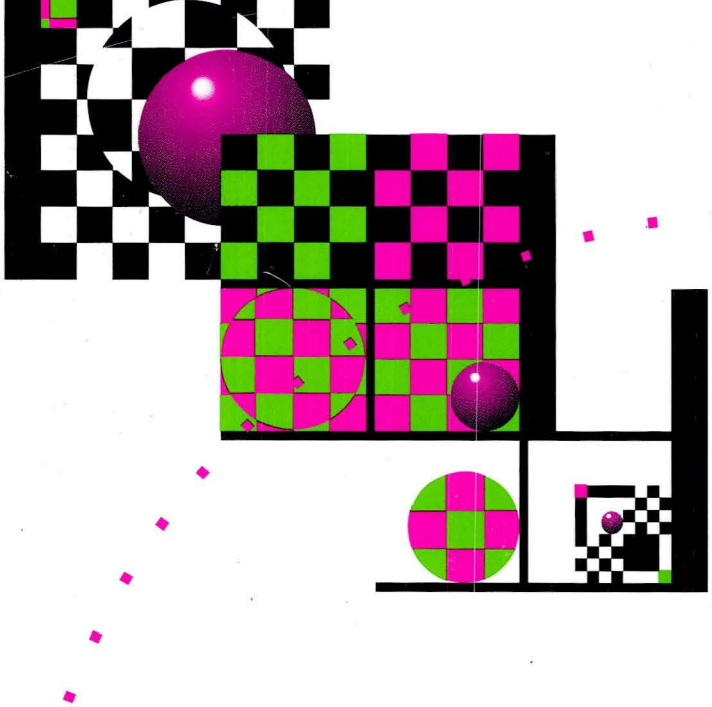


CONVEX

# CXdb Quick Reference

Third Edition

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

# CXdb Quick Reference

Order No. DSW-474

Third Edition
November 1994

**CONVEX Press**
Richardson, Texas
United States of America

# CXdb
# Quick Reference

Document No. 710-015630-002
Order No. DSW-474

Released with V3.1 of the CXdb software.



Printed in the United States of America

# Contents

# Using CXdb

1

## Compiling your program for CXdb

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 and places them in a subdirectory called .CTI.

The .CTI subdirectory is created in the same directory where the compiler places the object (.o) files for your program.

If you move any of your source files, executable files, or CTI data files after compiling or if your directory structure changes, then you must give CXdb the new locations of those files, as shown in the following table:

| If you move | Specify new location with |
|---|---|
| Source files | `add path` or `set path` command |
| Executable files | `add default path` or `set default path` command |
| CTI data files | `add path` or `set path` command |

## Invoking CXdb

Use the `cxdb` command to invoke CXdb in either of two interfaces:

- **X Windows mode**—Provides a window environment for X terminals that includes full use of a mouse to perform debugging functions in CXdb.
- **Line mode (tty interface)**—Allows you to use CXdb interactively without the graphical user interface on either windowless terminals (such as VT-100s) or X terminals.

### In X Windows mode

Assuming your DISPLAY environment variable is set correctly, invoke CXdb in X Windows mode from the shell with the `cxdb` command. For example:

```
% cxdb &
```

### In line mode (tty interface)

From the shell, invoke CXdb in line mode (tty interface) using the `-nw` option of the `cxdb` command. For example:

```
% cxdb -nw
```

### Using `cxdb` command options

The following table lists and describes selected `cxdb` command options. Refer to "Starting CXdb—cxdb command," on page 26 for complete syntax and options.

| Option | Purpose |
|---|---|
| *<executable>* or `-e` *<executable>* | Performs the `debug exec` command and loads the executable file to debug. |
| `-c` *<corefile>* | Performs the `debug core` command and loads the core file to debug. |
| `-a` *<process_id>* | Attaches to the specified process. |
| `-F` | Enables fixed scheduling (C Series only). |
| `-nw` | Invokes CXdb in line mode (tty interface). |
| `-csd` | Automatically incorporates aliases for `csd` debugger commands. Using this option also invokes CXdb in line mode. |

## Loading a program to debug

When you begin a debugging session, or if a process object does not exist, use the `debug exec` command on the CXdb command line to load your program. (The executable file should have been compiled with the `-cxdb` option.) For example,

```
(CXdb) debug exec <executable_file>
```

The `debug exec` command:

- Gives CXdb the name of the executable file you wish to debug.
- Creates a process object. A process object contains the image of the process, core file, or executable; debugging information; and process settings.
- Incorporates debugging information associated with the executable file into the process object so that symbolic debugging is possible.
- In X Windows mode, brings up a Source Code window by default that displays source files associated with your executable.

### Invoking CXdb with the name of an executable

You can invoke CXdb from the shell with the name of an executable. For example:

```
% cxdb <executable>
```

The above command invokes CXdb in X Windows mode. CXdb loads the executable file to debug by executing the `debug exec` command on the specified file. CXdb assumes that the first file that is not preceded by an option flag is the executable file.

### Invoking CXdb with the name of a core file

To invoke CXdb from the shell with the name of a core file to debug, use the `-c` option of the `cxdb` command. For example:

```
% cxdb -c <corefile> <executable>
```

The above command starts CXdb in X Windows mode. CXdb loads the core file to debug by executing the `debug core` command on the specified core file.

Specifying the executable file is optional— this incorporates debugging information from the executable that generated the core file. If you do not specify an executable, you can still debug the core image using absolute addresses.

## Specifying a different executable, a core file, or a process to debug

The following table describes CXdb commands for specifying a different executable file, a core or checkpoint file, or a process to debug. Any executable files specified should have been compiled with the `-cxdb` option.

| If you want to... | Use these commands... | Explanation |
|---|---|---|
| Specify a different executable file | `executable` <*executable_file*> | A process object must already exist (a process object is created when you invoke CXdb with the name of an executable file or when you execute the `debug exec`, `debug proc`, or `debug core` commands).<br>This command replaces the old debugging information with new information associated with the specified executable file. |
| Debug an existing process | `debug proc` <*process-id*><br>`executable` <*executable_file*> | This involves two comands. The `debug proc` command stops the process, then creates a process image that is incorporated into the process object.<br>The `executable` command incorporates the debugging information from the executable file that generated the process, so that symbolic debugging is possible. |
| Debug a core file | `debug core` <*core_file*><br>`executable` <*executable_file*> | This involves two commands. The `debug core` command creates a process object, creates an image based on the specified core file, and incorporates that image into the process object.<br>The `executable` command associates the appropriate debugging information from the executable file that generated the core image, so that symbolic debugging is possible. |

## User interface and windows

There are two user interfaces for CXdb: X Windows mode and line mode (tty interface).

This section explains how to create windows in X Windows mode, then lists common debugging tasks and their associated CXdb windows, along with line mode command equivalents, where applicable.

### Creating CXdb windows (X Windows mode)

To create CXdb windows:

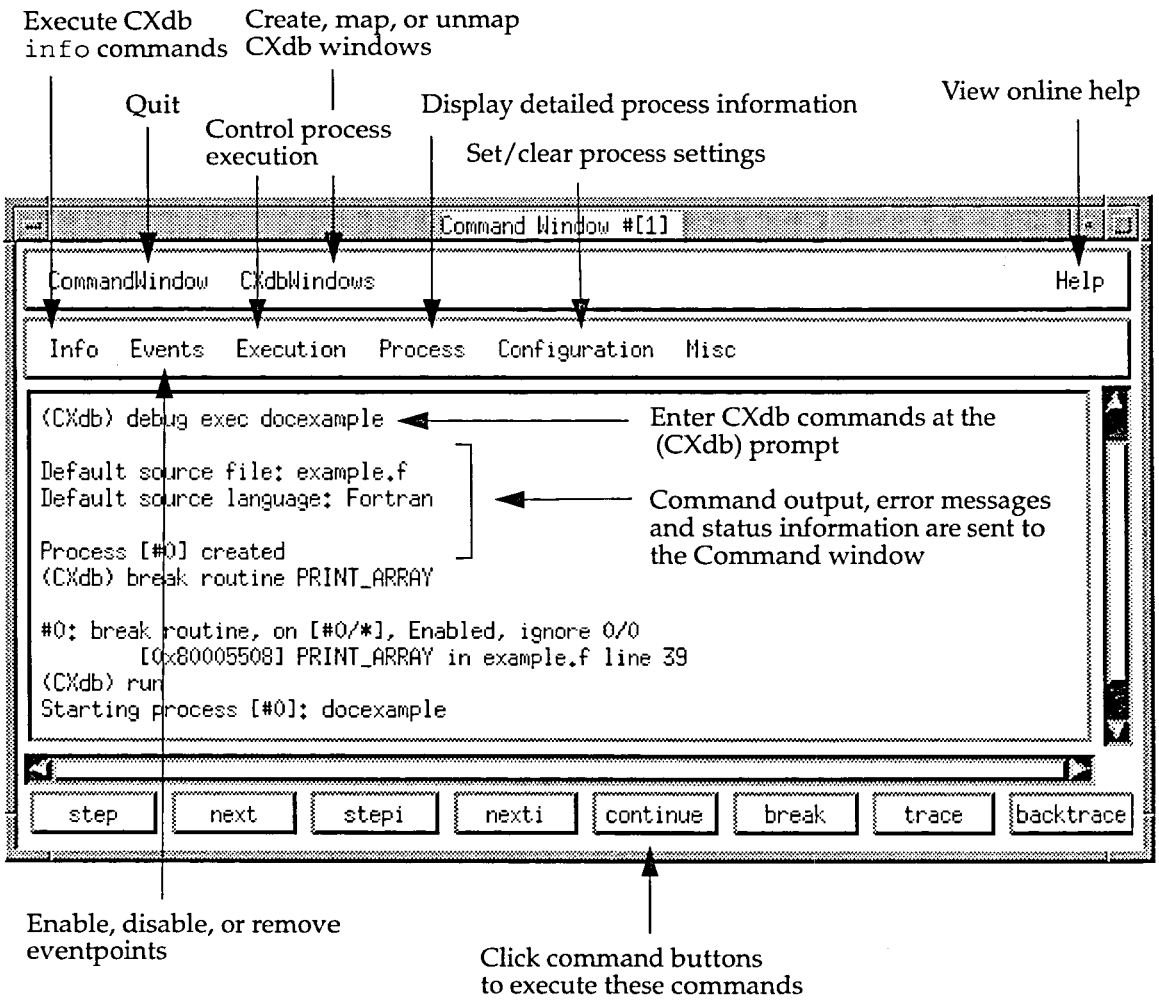
- **Command window**—Created automatically when you execute the `cxdb` command.
- **Source Code window**—By default, created automatically when you execute the `debug exec`, `debug proc`, `executable`, `display source`, or `display routine` commands. Can also be created by selecting Create window, then Source Code, from the CXdbWindows menu in the Command window.
- **Process interface window**—Created automatically when you execute the `run` or `rerun` command.
- **Assembly Code window**—Select Create window, then Assembly Code, from the CXdbWindows menu in the Command window, or use the `display disassembly` command.
- **Memory Display window**—Select Create window, then Memory Display, from the CXdbWindows menu, or execute the `display examine` command.
- **Stack Trace window, Thread Activity window or register windows**—Select Create window, then the appropriate item from the CXdbWindows menu in the Command window.
- **Help window**—Select the Window Overview, Using Help or Tutorial item from the Help menu in any CXdb window, or execute the `help` command.
- **Edit window**—Select the edit item from the Misc menu in the Command window, or execute the `edit` command.
- **Shell window**—Select the shell item from the Misc menu in the Command window, or execute the `shell` command.

## CXdb windows and line mode equivalents

| If you want to... | Use this CXdb window... | Line mode (tty interface) command equivalent |
|---|---|---|
| Enter commands; view command output and messages | Command | Invoke CXdb from the shell with the `-nw` option and enter commands at the `(CXdb)` prompt. |
| View source code | Source Code | `list` |
| View assembly-language code | Assembly Code | `disassemble` |
| View contents of process memory | Memory Display | `examine` |
| View thread activity relative to source code | Thread Activity | None |
| View process stack frames | Stack Trace | `backtrace; info frame; info args; info locals` |
| View contents of processor status word register (PSW) | Processor Status Word | `info psw` |
| View contents of vector registers | Vector Registers | `info vregisters` |
| View contents of communication registers | Communication Registers | `info cregisters` |
| View contents of scalar and address registers | Scalar Registers | `info registers` |
| View contents of general registers | General Registers | `info registers` |
| View contents of space registers | Space Registers | `info space registers` |
| View contents of control registers | Control Registers | `info control registers` |
| View contents of floating point registers | Floating point Registers | `info floating point registers` |
| Provide interactive input to and output from the running process | Process interface window | Process input, output, and messages are sent to stdin, stdout, and stderr in line mode. |
| Execute shell commands | Shell | `shell` |

## Using the Command window

The Command window (shown below) is the primary way to communicate with CXdb. It appears automatically when you start CXdb with the `cxdb` command (assuming your DISPLAY environment variable is set correctly and you have not invoked CXdb with the `-nw` option).

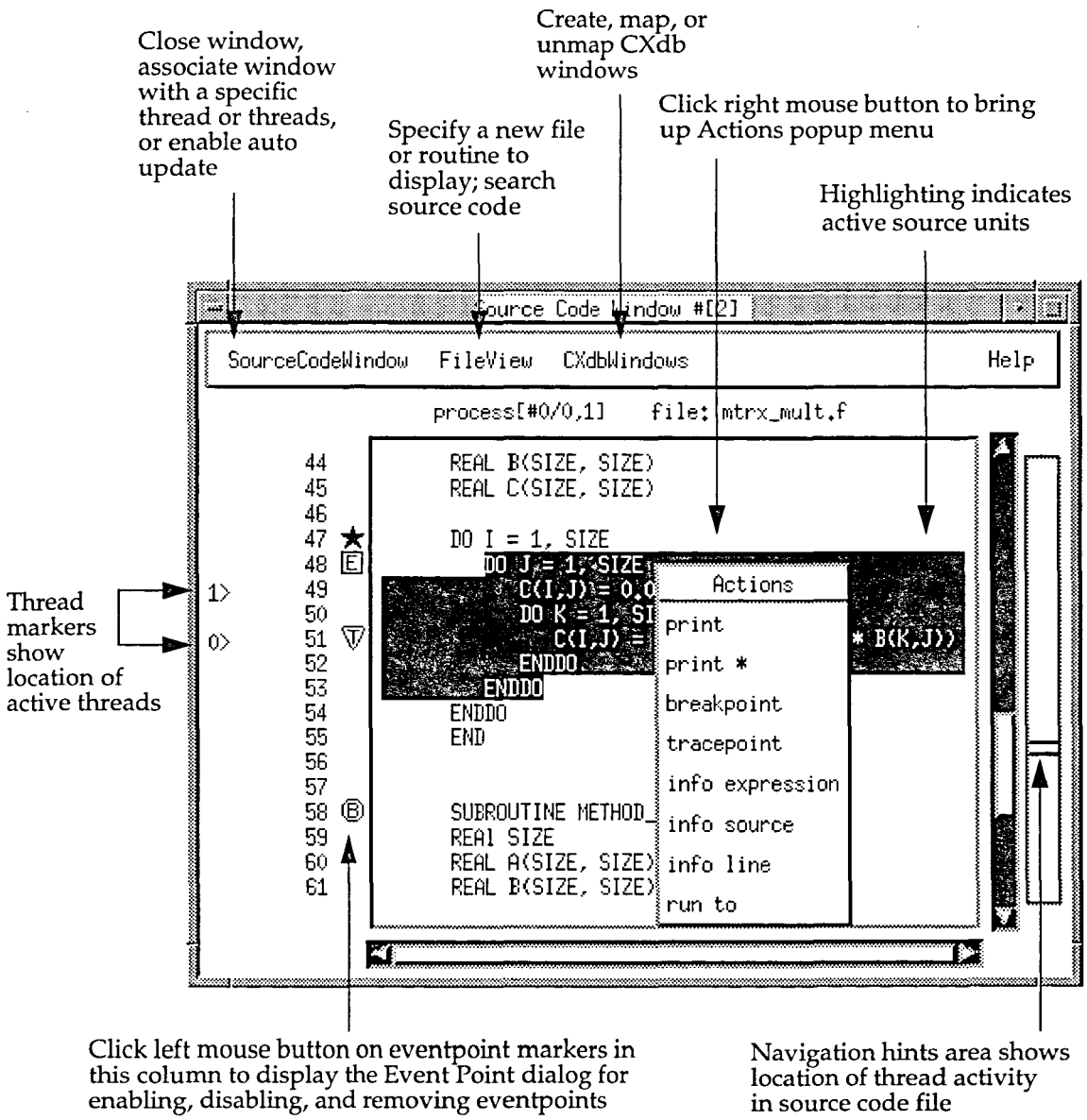



CXdb stores the last 100 commands you entered in a history buffer:

- Press **CTRL-p** to go to the previous line in the command history; use **CTRL-n** to go to the next line.
- Use the `info history` command to display the command history; use the `recall` command to retrieve and re-execute a command.

## Using the Source Code window

The Source Code window (shown below) displays the source code files associated with the executable file. By default, a Source Code window is automatically created when you specify an executable file to debug. You can have multiple Source Code windows open during a debugging session.



# Working with breakpoints, tracepoints, and watchpoints

This section lists commands for manipulating breakpoints, tracepoints, and watchpoints and also describes how to use the mouse in the Source Code window to manipulate them.

## Using commands

The following table contains a list of selected commands for working with breakpoints, tracepoints, and watchpoints. All of the commands listed are available in both X Windows mode and line mode (tty interface).

| If you want to... | Use these command(s)... |
|---|---|
| Display information about breakpoints, tracepoints, watchpoints, or all eventpoints | `info break`<br>`info trace`<br>`info watch`<br>`info event` |
| Set a breakpoint or tracepoint at a routine | `break routine`<br>`trace routine` |
| Set a breakpoint or tracepoint at a line | `break line`<br>`trace line` |
| Set a breakpoint or tracepoint at an instruction | `break instruction`<br>`trace instruction` |
| Set a watchpoint to monitor an address range | `watch` |
| Set a conditional breakpoint | `event relation` |
| Disable a breakpoint, tracepoint, or watchpoint | `disable event` |
| Enable a breakpoint, tracepoint, or watchpoint | `enable event` |
| Remove a breakpoint, tracepoint, or watchpoint | `remove event` |

Refer to following sections for complete syntax and additional commands:


- "Setting breakpoints—break commands" on page 28
- "Setting tracepoints—trace commands" on page 29
- "Displaying eventpoint information" on page 31
- "Removing eventpoints" on page 32
- "Eventpoint handler commands" on page 33

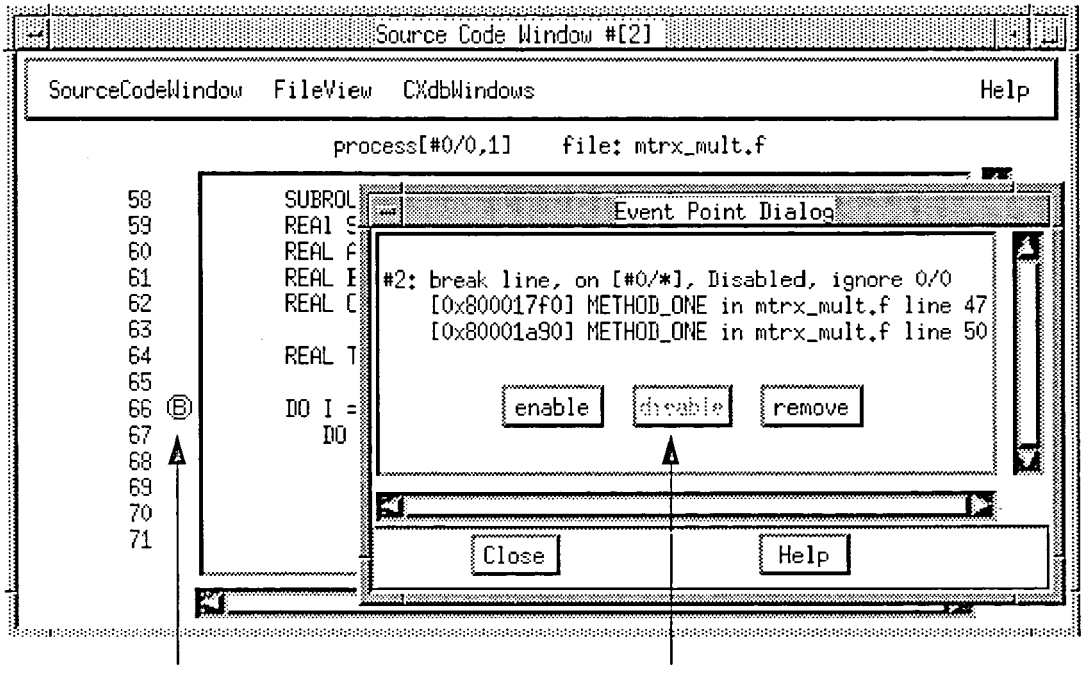
## Using the mouse in the Source Code window

In the Source Code window, you can use the mouse to set a breakpoint or tracepoint using the following method:

1. Position the mouse pointer over the line of source code at which you wish to set the breakpoint or tracepoint.
2. Click the right mouse button to display the popup action menu.
3. Select the breakpoint or tracepoint item.

To enable, disable, or remove a breakpoint or tracepoint in the Source Code window:

1. Left click on the eventpoint marker to bring up the Event Point dialog.
2. Locate the eventpoint you wish to work with in the Event Point dialog.
3. Select the action you wish to perform on the eventpoint (enable, disable, or remove).
4. Click the Close button.



## Controlling process execution

Before using any of the CXdb commands for controlling process execution, make sure that you have:

- Compiled your program with the `-cxdb` option.
- Loaded the executable file you want to debug.
- Set at least one breakpoint—CXdb does not automatically set a breakpoint at the first line of your program.

The following table describes how to perform basic program execution tasks. Refer to "Stepping process execution" on page 12 for a description of stepping commands.

| If you want to... | Use this command or key... | Comments |
|---|---|---|
| Run your program | `run` | By default, automatically creates a Source Code window in X Windows mode. |
| Rerun your program using the previous argument list | `rerun` | |
| Continue execution of a stopped process | `continue` | A process object must have already been created with the `run` or `rerun` command. |
| Continue process execution from within an eventpoint handler | `resume` | Can only be used within an eventpoint handler. This is the only process execution command that can be used within an eventpoint handler. |
| Stop process execution | Type **CTRL-c** in the Command window | Aborts the command that started process execution. |
| Terminate a running process | `kill process` | |

In X Windows mode you can also control process execution by:

- Using the run and continue command buttons in the Command window
- Selecting process execution commands from the Execution menu in the Command window
- Selecting "run to" from the Actions popup menu in the Source Code window to run to a specific location

## Stepping process execution

The following table shows commands used for stepping process execution within CXdb.

The *<granularity>* (step size) you specify with stepping commands is optional, and can be `expression`, `statement`, `loop`, `routine`, or `block` If no granularity is specified, CXdb uses the default stepping granularity.

Refer to "Stepping commands" on page 35, for complete command syntax and additional commands.

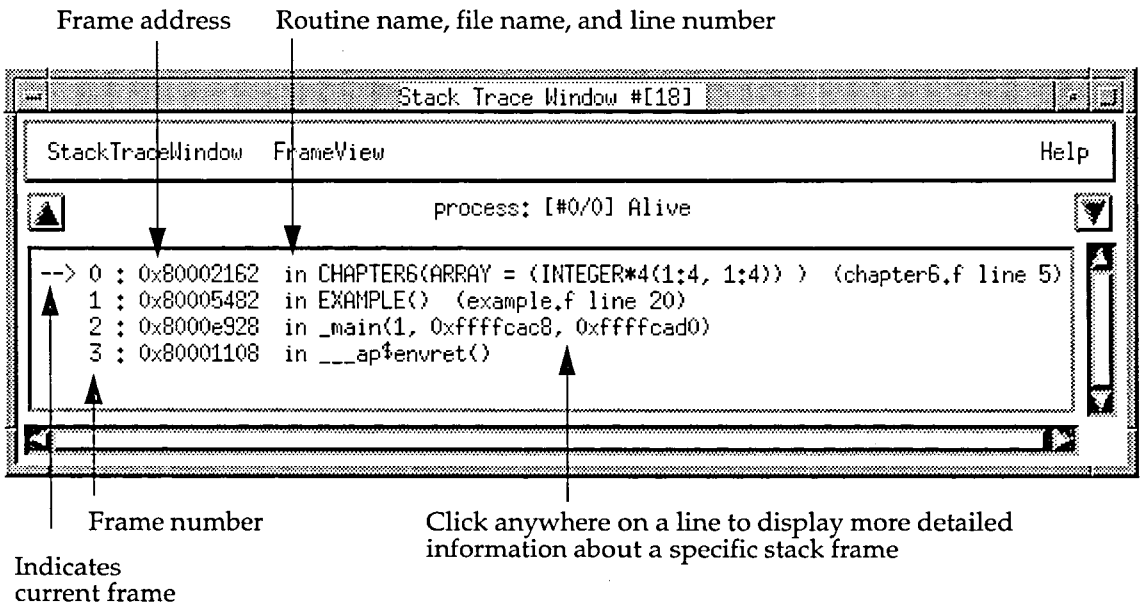
| To perform this stepping task... | Use this command... |
|---|---|
| Set the default stepping granularity for CXdb. | `set step [<granularity>]` |
| Step process execution by 1 source unit of the specified granularity, stepping into subroutine calls | `step [<granularity>]` |
| Step the process by 1 instruction, stepping into subroutine calls | `step instruction` |
| Finish executing (step out of) a source unit of the specified granularity | `finish [<granularity>]` |
| Step process execution to the next source unit of specified granularity, stepping over subroutine calls | `next [<granularity>]` |
| Step from the current source of the specified granularity to the next source unit of default granularity, not counting subroutine calls. | `next over <granularity>` |
| Step to the next instruction, stepping over subroutine calls. | `next instruction` |

In X Windows mode, you can also execute stepping commands by:

- Using the step, stepi, next, nexti command buttons in the Command window
- Selecting stepping commands from the Execution menu in the Command window
- Pressing the following keys while in the Source Code window (assuming you have not modified the default key bindings):
  - **s**—Step
  - **n**—Next
  - **c**—Continue

## Obtaining a stack backtrace

You can obtain a stack backtrace in X Windows mode by opening a Stack Trace window. You can do this by selecting the Stack Trace item from the CXdbWindows menu in either the Command window or the Source Code window. A sample Stack Trace window is shown below.




In X Windows mode, you can also click on the backtrace command button in the Command window.

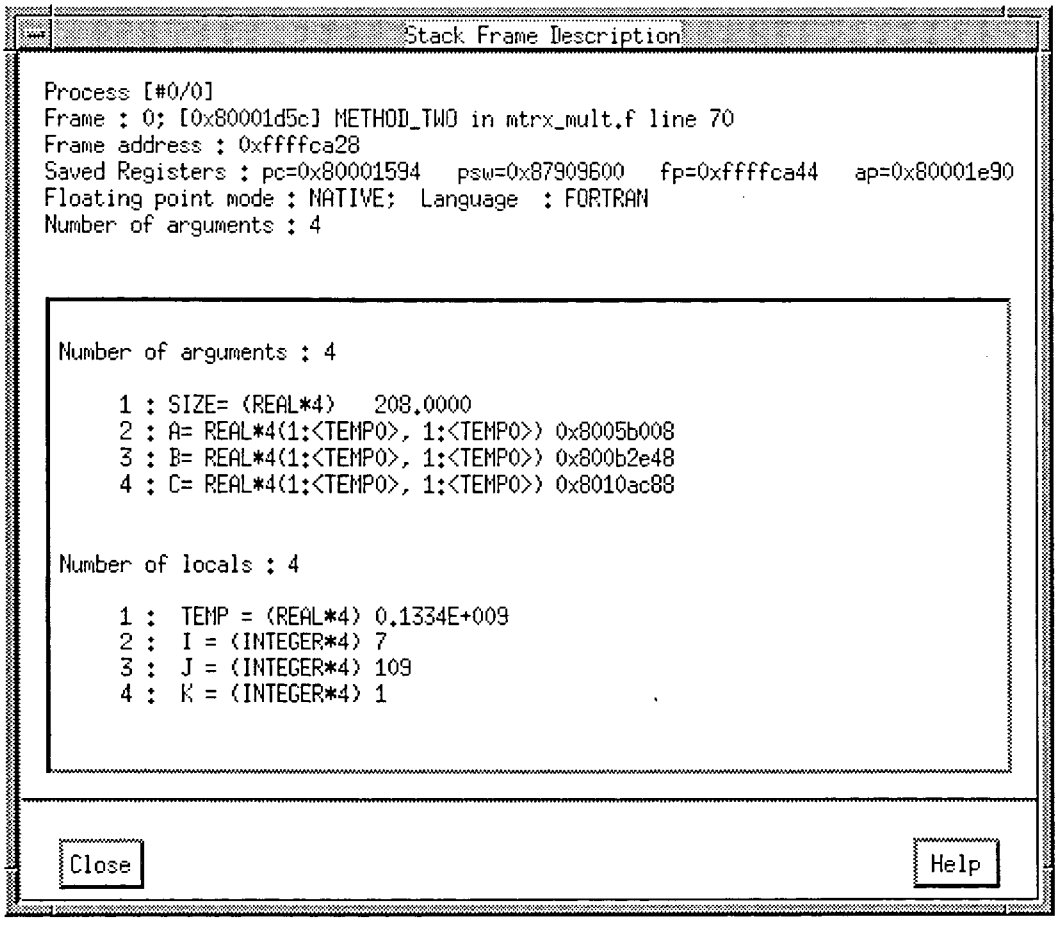
At the command line (in either line mode or X Windows mode), you can obtain a stack backtrace by executing the `backtrace` command at the (CXdb) prompt. For example:

```
(CXdb) backtrace
Process [#0/0]
cf> 0 : 0x80001658 in SUB1(MAXLEN = (INTEGER*4) 1000, A = (REAL*4(1:<TEMP0>)) ,
 B = (REAL*4(1:<TEMP0>, 1:<TEMP0>)) ) (para.f line 22)
 1 : 0x800014ce in PARA() (para.f line 11)
 2 : 0x80001d08 in _main(1, 0xffffcad8, 0xffffcae0)
 3 : 0x80001108 in ___ap$envret()
```

## Displaying information about stack frames

To obtain detailed information about a specific stack fame in X Windows mode:

1. Open a Stack Trace window by selecting Create window, then Stack Trace from the CXdbWindows menu in either the Command window or the Source Code window.
2. In the Stack Trace window, click with the left mouse button anywhere on the line describing the stack frame you want information for. This will bring up the Stack Frame Description dialog:




In line mode, you can use the following sequence of commands to obtain this information:

```
(CXdb) info frame; info args; info locals
```

## Displaying and modifying program data

This section describes commands for displaying or modifying the contents of the variables and memory segments associated with your program. CXdb provides commands that can access the variables either symbolically by identifer or directly by address.

Refer to the following sections for complete command syntax:


- "Using the print command" on page 39
- "Viewing and modifying memory" on page 40
- "Displaying register contents" on page 42
- "CXdb info commands" on page 44


| If you want to... | Use this command... | Comments |
|---|---|---|
| Display information about arguments. | `info args` | |
| Display the local variables of the current routine. | `info locals` | |
| Display all symbol names used in the program. | `info symbols` | |
| Display detailed information for a single variable, debugger variable, or language expression. | `info expression` | Shows the object type, storage location, size, data type, current value and *liveness* ranges (range of memory where the value of the variable is available for displaying) |
| Print a program variable or language expression. | `print` | Use the `info formatting` command to display current settings for print options. |
| Print an array slice. | `print` *<array_name>* *<starting-subscript>* `[..`*<ending-subscript>*`]` | If you do not specify an *ending_subscript*, the default array slice is the single element specified by the *starting_subscript*. |

| If you want to... | Use this command... | Comments |
|---|---|---|
| Specify format for displaying units of memory with `print` command. | `print/B`—Binary output<br>`print/x`—Hexadecimal output<br>`print/d`—Signed decimal output<br>`print/e`[*<width>.<precision>*]—Scientific notation<br>`print/f`[*<width>.<precision>*]—Floating point notation<br>`print/i`—Instruction | Note that there should not be a space between the word "`print`" and the display format specification. Not all print formats are available for all memory unit types. |
| Set print options. | `set printopts padding`<br>`set printopts nopadding`<br>`set printopts precision`<br>`set printopts maxarray` | Force alignment with leading zeros (padding).<br>Disable padding with leading zeros (default).<br>Set precision for real (floating-point) numbers.<br>Set maximum number of array elements to be printed at one time. |
| Display contents of memory. | `examine` | In X Windows mode, you can also open a Memory Display window. Refer to the "examine" reference topic for formatting options. |
| Display register contents for registers available on your system. | `info registers`<br>`info vregisters`<br>`info cregisters`<br>`info floating point registers`<br>`info psw`<br>`info space registers`<br>`info control registers` | In X Windows mode, you can open register windows for the registers available on your system. |
| Set floating-point mode for evaluating language expressions. | `set evalopts.fpmode {ieee \| native \| dual}` | Not available on SPP Series systems (on SPP Series systems, only IEEE floating-point mode is available.) |

## Viewing thread information

A *thread* is a sequence of machine instructions that executes on a single processor (CPU). If your machine has multiple CPUs, you can create multiple threads in your program by compiling your source code at optimization level `-O3`, including compiler directives (such as `FORCE_PARALLEL`) in your source code, or by using assembly-language instructions such as `spawn` or `pfork` to create threads at specific locations.

CXdb provides windows and commands that enable you to view information about threads in your program. In addition, you can associate various CXdb windows with specific threads, view navigation hints in the Source Code or Assembly Code windows, or use CXdb commands on individual threads. Refer to the "threads" reference topic page for more information.

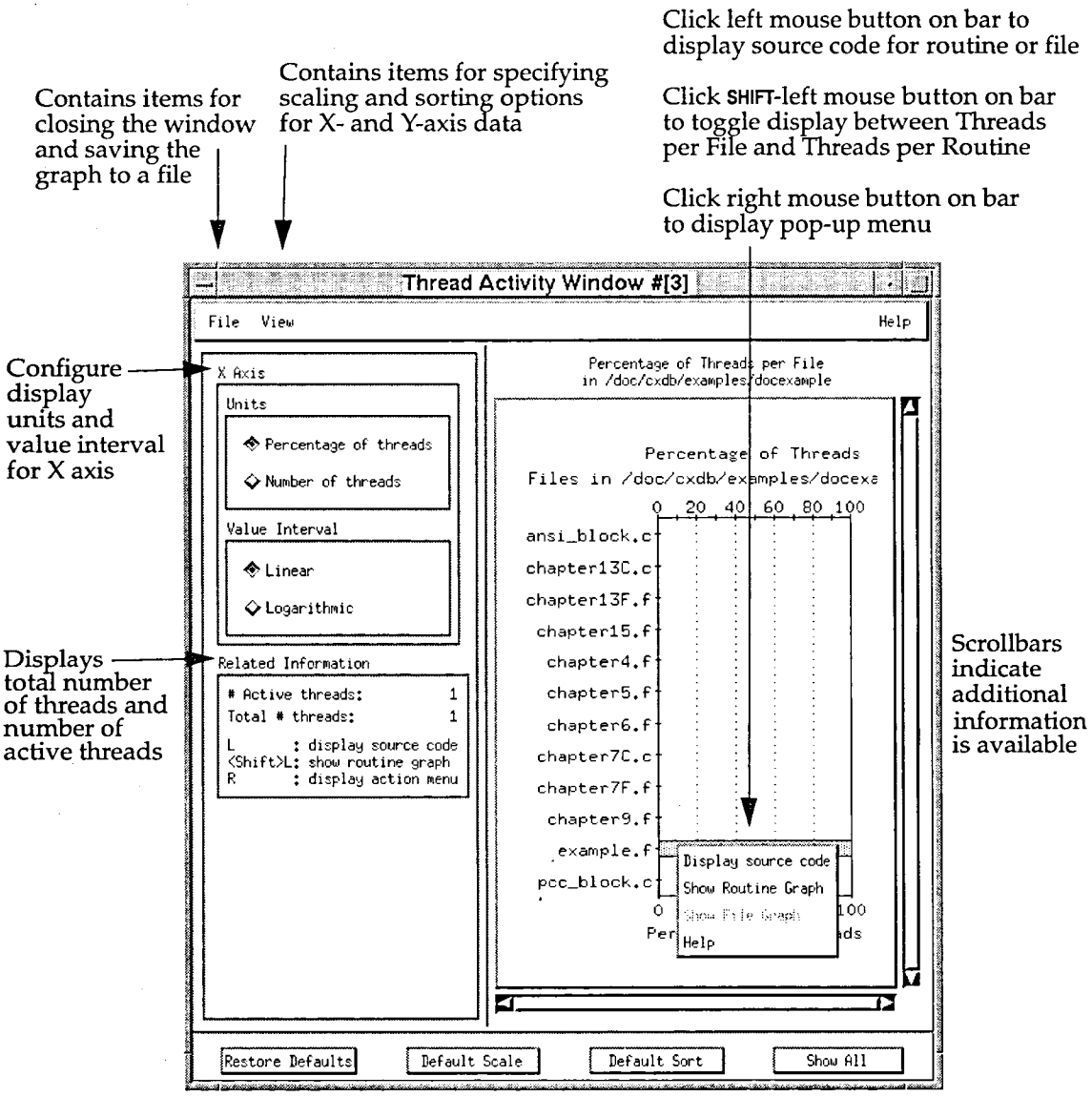
### Commands

The following table lists tasks associated with multithreaded debugging and describes corresponding CXdb commands.

| If you want to... | Use these commands... | Comments |
|---|---|---|
| Enable fixed scheduling. | `set fixed sched` | C Series systems only. Fixed scheduling reserves all the CPUs of the machine for each timeslice that your process executes.<br><br>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 nondeterminism. |
| Enable default fixed scheduling. | `set default fixed sched` | |
| Disable fixed scheduling. | `clear fixed sched` | |
| Disable default fixed scheduling. | `clear default fixed sched` | |
| Set an eventpoint to detect the creation of a new thread. | `event spawn` | When this eventpoint is triggered, it stops all threads of the process. |
| Set an eventpoint to detect when a thread joins. | `event join` | When this eventpoint is triggered, it stops all threads of the process. |
| Display thread information. | `info threads` | Lists the current state of each thread, ID numbers of active threads, and identifies the current thread. |

## Thread Activity window

The Thread Activity window (shown below) 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. To create this window, select Create window, then Thread Activity from the CXdbWindows menu in the Command window.



## Using CXdb in line mode

Line mode allows you to use CXdb interactively without the graphical user interface. Line mode is designed primarily for use on windowless terminals (for example, VT-100s), 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, 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.

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.

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

CXdb line mode provides command line editing functions similar to UNIX command line editing. You can display the key bindings for these editing functions by entering CTRL-? on the CXdb command line.

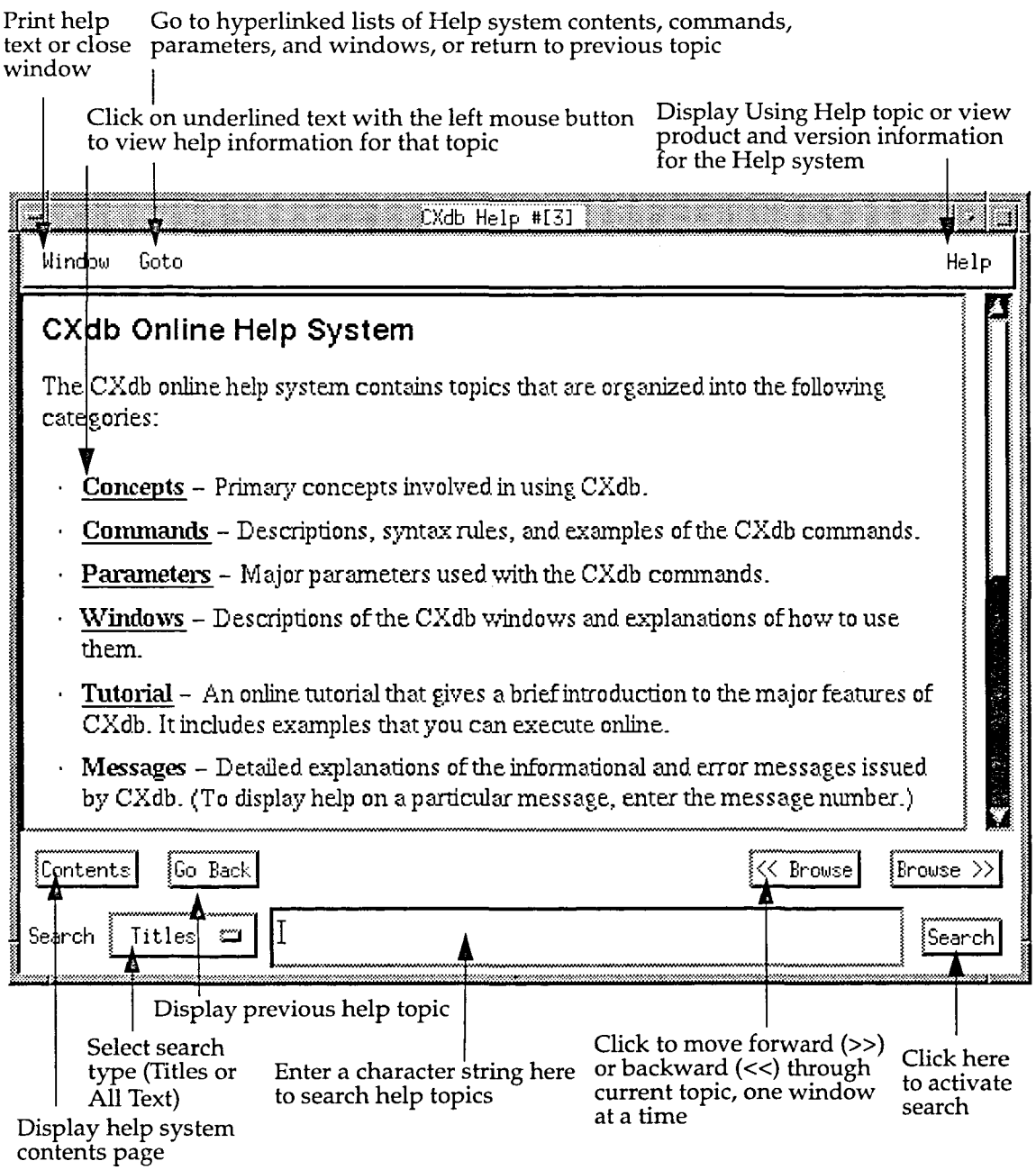
In line mode, whenever the process is stopped or the stack frame is changed, CXdb displays 5 lines of source code surrounding the line of source code at the point of execution. For example, lines of source code context are displayed:

- When the process is stopped by a breakpoint or an eventpoint of type spawn, join, signal, or reached
- When the process stops after executing commands affect process execution such as `step`, `stepi`, `next`, `finish`, or `goto`
- When you change the current stack frame by executing the `frame` command or the aliases `up` and `down`

Some CXdb windows and commands are not available in line mode. Refer to the section "CXdb windows and line mode equivalents" on page 6 for alternative or equivalent commands to use in line mode.

## Getting help online

To access the CXdb Help system and bring up the CXdb Help window, select the Window Overview, Using Help or Tutorial item from the Help menu in any CXdb window or execute the `help` command. The CXdb Help window is show below.



# Mouse and keyboard shortcuts

This section lists selected mouse and keyboard shortcuts for command-line editing and various CXdb windows, assuming that you have not modified the default button or key bindings.

## Command-line editing

| Key | Function |
|---|---|
| CTRL-a | Beginning of line |
| CTRL-b | Back one character |
| META-b | Back one word |
| CTRL-d | Delete forward one character |
| META-d | Delete forward one word |
| CTRL-e | End of line |
| CTRL-f | Forward one character |
| META-f | Forward one word |
| CTRL-u | Beginning of line, delete line |
| CTRL-k | Delete to the end of the line |
| CTRL-v, PAGE UP | Scroll down one page (Command window only) |
| META-v, PAGE DOWN | Scroll up one page (Command window only) |
| TAB | Complete current command |
| CTRL-p | View previous entry in the command history |
| CTRL-n | View next entry in the command history |
| CTRL-l | Redraw screen |
| LEFT ARROW (<—) | Left one character (Command window only) |
| RIGHT ARROW (—>) | Right one character (Command window only) |
| CTRL-? | Lists all available key bindings (line mode only) |

## Source Code window

| Key/button | Function |
|---|---|
| r | Executes the `run` command. |
| s | Executes the `step` command using the default granularity. |
| c | Executes the `continue` command. |
| Left button in text area | Selects (highlights) text characters in the text scrolling area. |
| Left button on eventpoint marker | Brings up an Event Point dialog where you can disable, enable, or remove eventpoints. |
| 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:<br>print<br>print * (print indirect)<br>breakpoint<br>tracepoint<br>info expression<br>info source<br>info line<br>run to |
| CTRL-Left button | Highlights (selects) the source unit at the location of the mouse pointer. |

## Assembly Code window

| Key | Function |
|---|---|
| c | Executes the `continue` command. |
| n | Executes the `next instruction` command. |
| s | Executes 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. |

## Stack Trace window

| 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

| Button | 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 and Threads per Routine in file. |
| Right button | Click on bar in thread activity graph to pop up an actions menu. |

# Command syntax and descriptions

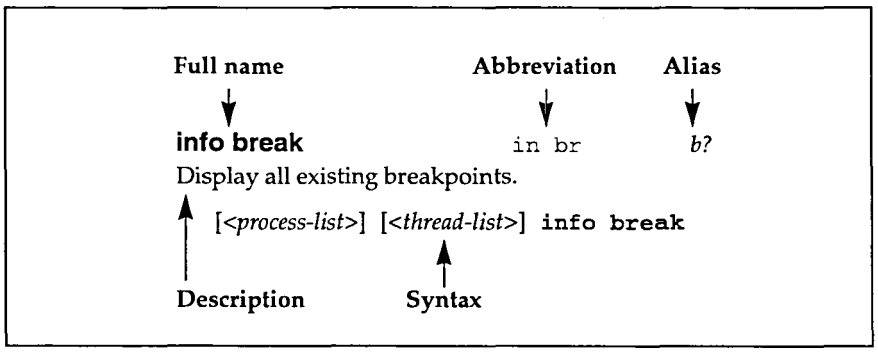
# 2



This section lists the following information for each CXdb command:

- Full command name
- Shortest abbreviation for the command
- Default alias for the command
- Brief description of what the command does
- Syntax of the command

The information is presented in the following format:



## Starting CXdb—cxdb command

Use the following command to invoke CXdb. To access symbolic debugging information, you must have compiled your program with the `-cxdb` option:

**cxdb** `cxdb`

Invoke CXdb from the shell.

```
cxdb [-a <process-id>] [-csd ]
    [-D <directory-specifier>[, ...]]
    [-f <command-file>] [-F ] [-ns ] [-nw ] [-nx ]
    [-x <command-list>] [[-e ] <file-name>]
    [[-c] <file-name>] [<X-Toolkit-options>]
```

| Parameter | Meaning |
|---|---|
| `-a` *<process-id>* | Attaches CXdb to the process with the specified process ID. |
| `-c` *<file-name>* | Specifies a core file to debug. |
| `-csd` | Invokes CXdb in line mode and incorporates aliases for `csd` debugger commands. |
| `-D` *<directory-specifier>* | Specifies a directory to add to the default search path. |
| `-e` *<file-name>* | Specifies an executable file to debug. |
| `-f` *<command-file>* | Executes the specified CXdb command file. |
| `-F` | Enables fixed scheduling. |
| `-ns` | Disables dynamic creation of Source Code windows. |
| `-nw` | Invokes CXdb in line mode. |
| `-nx` | Prevents CXdb from executing initialization files. |
| `-x` *<command-list>* | Specifies a list of CXdb commands to be executed at start-up. Separate multiple commands in the list with a semicolon (;). |
| *<X-Toolkit-options>* | Specifies an X Toolkit option. |

## quit command

**quit** `q`

Exit from CXdb.

```
quit
```

## help command

**help** `h` `?`

Invoke the CXdb help system.

```
help [<string>]
```

The string you enter can be a CXdb command, CXdb error message identifier, or other text string.

## Specifying a file or process to debug

**attach** `at`

Debug the image of a running process.

[*<process-list>*] `attach` [*<remote-host>*`:`] *<process-id>*

**core** `cor`

Debug the image of a core file or a checkpoint file.

[*<process-list>*] `core` [*<remote-host>*`:`] *<file-name>*

**debug core** `deb c` *dbgc*

Create a process object and debug the image of a core or checkpoint file.

`debug core` *<core-file>* [`executable` [*<remote-host>*`:`] *<executable-file>*] [*<debugger-variable>*]

**debug exec** `deb e` *dbg*

Create a process object and debug the image of an executable file.

`debug exec` [*<remote-host>*`:`] *<executable-file>* [*<debugger-variable>*]

**debug proc** `deb p` *dbgp*

Create a process object and debug the image from a running process.

`debug proc` [*<remote-host>*`:`] *<process-id>* [`executable` [*<remote-host>*`:`] *<executable-file>*] [*<debugger-variable>*]

**detach** `det`

Detach from a process.

[*<process-list>*] `detach`

**executable** `exe`

Specify an executable file to provide CTI data and the executable image.

[*<process-list>*] `executable` [*<remote-host>*`:`] *<file-name>*

## Setting breakpoints—`break` commands

**break instruction** `bre i` *bi*

Set a breakpoint at an instruction.

[<*process-list*>] [<*thread-list*>] `break instruction` <*language-expression*>
[ {<*event-handler*>} ] [<*debugger-variable*>]

**break line** `bre l` *bl*

Set a breakpoint at a source line.

[<*process-list*>] [<*thread-list*>] `break line` <*line-specifier*>
[ {<*event-handler*>} ] [<*debugger-variable*>]

**break routine** `bre r` *br*

Set a breakpoint at the beginning of a routine.

[<*process-list*>] [<*thread-list*>] `break routine` <*language-expression*>
[ {<*event-handler*>} ] [<*debugger-variable*>]

**break source** `bre s` *bs*

Set a breakpoint at a source unit.

[<*process-list*>] [<*thread-list*>] `break source` <*source-unit*>
[ {<*event-handler*>} ] [<*debugger-variable*>]

## Setting tracepoints—`trace` commands

**trace instruction** `t i` *ti*

Set a tracepoint at an instruction.

[<*process-list*>] [<*thread-list*>] `trace instruction` <*language-expression*> [ {<*event-handler*>} ][<*debugger-variable*>]

**trace line** `t l` *tl*

Set a tracepoint at a source line.

[<*process-list*>] [<*thread-list*>] `trace line` <*line-specifier*> [ {<*event-handler*>} ] [<*debugger-variable*>]

**trace routine** `t r` *tr*

Set a tracepoint at the beginning of a routine.

[<*process-list*>] [<*thread-list*>] `trace routine` <*language-expression*> [ {<*event-handler*>} ] [<*debugger-variable*>]

**trace source** `t s` *ts*

Set a tracepoint at a source unit.

[<*process-list*>] [<*thread-list*>] `trace source` <*source-unit*>[ {<*event-handler*>} ] [<*debugger-variable*>]

## Setting other types of eventpoints—`event` and `watch` commands

**event exec** `eve e`

Set an eventpoint to watch for an `exec(2)` system call by the process.

[*<process-list>*] `event exec` [ {*<event-handler>*} ] [*<debugger-variable>*]

**event join** `eve j`

Set an eventpoint to trap a thread joining.

[*<process-list>*] `event exec` [ {*<event-handler>*} ] [*<debugger-variable>*]

**event modify** `eve m`

Set an eventpoint to watch for a value change within an address range.

[*<process-list>*] [*<thread-list>*] `event modify` *<starting-address>*
[{ `..`*<ending address>* | `:`*<byte-count>* }] [ {*<event-handler>*} ]
[*<debugger-variable>*]

**event reached instruction** `eve rea i`

Set an eventpoint at an instruction.

[*<process-list>*] [*<thread-list>*] `event reached instruction`
*<language-expression>* [ {*<event-handler>*}][*<debugger-variable>*]

**event reached line** `eve rea l`

Set an eventpoint at a source line.

[*<process-list>*] [*<thread-list>*] `event reached line` *<line-specifier>*
[ {*<event-handler>*} ] [*<debugger-variable>*]

**event reached routine** `eve rea r`

Set an eventpoint at the beginning of a routine.

[*<process-list>*] [*<thread-list>*] `event reached routine`
*<language-expression>* [ {*<event-handler>*} ][*<debugger-variable>*]

**event reached source** `eve rea s`

Set an eventpoint at a source unit.

[*<process-list>*] [*<thread-list>*] `event reached source` *<source-unit>*[
{*<event-handler>*} ] [*<debugger-variable>*]

**event relation** `eve rel`

Set an eventpoint to watch for an expression to become true.

[*<process-list>*] [*<thread-list>*] `event relation` *<language-expression>* [
{*<event-handler>*} ] [*<debugger-variable>*]

**event signal** `eve si`

Set an eventpoint to catch a signal.

[*<process-list>*] `event signal` *<signal-specifier>* [ {*<event-handler>*} ] [*<debugger-variable>*]

**event spawn** `eve sp`

Set an eventpoint to trap the spawning of a thread.

[*<process-list>*] `event spawn` [ {*<event-handler>*} ][*<debugger-variable>*]

**watch** `w`

Set a watchpoint to monitor an address range.

[*<process-list>*] [*<thread-list>*] `watch` *<starting-address>*
[{ `..`*<ending address>* | `:`*<byte-count>*} ] [ {*<event-handler>*} ] [*<debugger-variable>*]

## Displaying eventpoint information

**info break** `in br` *b?*

Display all existing breakpoints.

[*<process-list>*] [*<thread-list>*] `info break`

**info event** `in event` *e?*

Display the specified eventpoints.

`info event` [*<event-specifier>*] [, ...]

**info eventtype** `in eventt` *et?*

Display all eventpoints of the specified type.

[*<process-list>*] `info eventtype` {*<event-specifier>* [, ...]

**info trace** `in tr` *t?*

Display all tracepoints.

[*<process-list>*] `info trace`

**info watch** `in w` *i w*

Display all watchpoints.

[*<process-list>*] `info watch`

## Enabling, disabling, and setting ignore counts for eventpoints

**enable event** `ena event` *en*

Enable eventpoints.

`enable event` <*event-specifier*> [, ...]

**enable eventtype** `ena eventt`

Enable all eventpoints of the specified type.

[<*process-list*>] `enable eventtype` <*eventtype-specifier*> [, ...]

**disable event** `disab event` *dis*

Disable eventpoints.

`disable event` <*eventpoint_specifier*> [, ...]

**disable eventtype** `disab eventt`

Disable all eventpoints of the specified type.

[<*process-list*>] `disable eventtype` <*eventtype_specifier*> [, ...]

**set ignore** `se i`

Set an ignore count for an eventpoint.

`set ignore` <*ignore-count*> <*event-specifier*> [, ...]

## Removing eventpoints

**remove event** `rem event` *e-*

Delete the specified eventpoints.

`remove event` <*event-specifier*> [, ...]

**remove eventtype** `rem eventt` *et-*

Delete all eventpoints of the specified type.

[<*process-list*>] `remove eventtype` <*eventtype-specifier*> [, ...]

## Eventpoint handler commands

**clear default handler** `cl d h`

Clear the default handler for all eventpoints.

```
clear default handler
```

**clear handler** `cl h`

Clear the handler for a specified eventpoint.

`clear handler` *<event-specifier>* [, ...]

**clear typehandler** `cl t`

Clear the handler for a specified type of eventpoint.

`clear typehandler` *<eventtype-specifier>* [, ...]

**echo** `ec`

Echo a character string.

`echo[/n]` *<string>* [ ... ]

**if** `if`

Establish conditional execution of CXdb commands.

`if` (*<relational-expression>*) *<command-set>* [ `else` *<command-set>*]

**resume** `res`

Continue execution of the process from within an eventpoint handler.

```
resume
```

**set default handler** `se de h`

Set the default handler for eventpoints.

`set default handler` {*<event-handler>*}

**set handler** `se h`

Set the handler for a specified eventpoint.

`set handler` *<event-specifier>* [, ...] {*<event-handler>*}

**set typehandler** `se t`

Define the default handler for all eventpoints of the specified type.

`set typehandler` *<eventtype-specifier>* [, ...] {*<event-handler>*}

## Process execution commands

**continue** `con` *c*

Continue execution of the process.

[*<process-list>*] [*<thread-list>*] `continue` [*<count>*] [ **&** ]

**detach** `det`

Detach from a process.

[*<process-list>*] `detach`

**goto address** `g a`

Branch to the specified address.

[*<process-list>*] [*<thread-list>*] `goto address` *<language-expression>*

**goto line** `g l`

Branch to the specified source line.

[*<process-list>*] [*<thread-list>*] `goto line` *<line-specifier>*

**goto source** `g s`

Branch to the specified source unit.

[*<process-list>*] [*<thread-list>*] `goto source` *<source-unit>*

**kill process** `k p` *k*

Terminate a running process and remove the image being debugged from the process object.

[*<process-list>*] `kill process`

**rerun** `rer` *rr*

Create and execute a new process, using the previous argument list.

[*<process-list>*] `rerun` [ **&** ]

**resume** `res`

Continue execution of the process from within an eventpoint handler.

`resume`

**return** `ret`

Return to the calling routine.

[*<process-list>*] [*<thread-list>*] `return` *<language-expression>*

**run** `ru` *r*

Create and execute a new process.

[*<process-list>*] `run` [*<argument-list>*] [ `&` ]

**stop** `sto`

Stop execution of the process.

[*<process-list>*] `stop`



## Stepping commands

The *<granularity>* (step size) you specify with CXdb stepping commands can be one of the following:

- `expression`
- `statement`
- `block`
- `loop`
- `routine`

**clear step** `cl st`

Reset the stepping granularity to the default setting.

[*<process-list>*] `clear step`

**finish** `fini`

Finish executing (step out of) the specified source unit.

[*<process-list>*] [*<thread-list>*] `finish` [*<granularity>*] [ **&** ]

**next** `n`

Step to the next source unit, ignoring subroutine calls.

[*<process-list>*] [*<thread-list>*] `next` [*<granularity>*] [*<count>*] [ `&` ]

**next instruction** `n i` *ni, nexti*

Step to the next instruction, stepping over subroutine calls.

[*<process-list>*] [*<thread-list>*] `next instruction` [*<count>*] [ `&` ]

**next over** `n o` *no*

Step from the current source unit of specified granularity to the next source unit of default granularity, stepping over subroutine calls.

[*<process-list>*] [*<thread-list>*] `next over` [*<granularity>*] [ `&` ] [*<count>*]

**set default step** `se de s`

Set the default stepping granularity.

`set default step` <*granularity*>

**set handler** `se h`

Set the handler for a specified eventpoint.

`set handler` <*event-specifier*> [, ...] {<*event-handler*>}

**set ignore** `se i`

Set an ignore count for an eventpoint.

`set ignore` <*ignore-count*> <*event-specifier*> [, ...]

**set step** `se st`

Set the stepping granularity.

[<*process-list*>] [<*thread-list*>] `set step` <*granularity*>

**step** `ste` *s*

Step to the next source unit.

[<*process-list*>] [<*thread-list*>] `step` [<*granularity*>] [<*count*>] [ & ]

**step instruction** `ste i` *si, stepi*

Step to the next instruction.

[<*process-list*>] [<*thread-list*>] `step instruction` [<*count*>] [ & ]

**step over** `ste o` *so*

Step from the current source unit of specified granularity to the next source unit of default granularity.

[<*process-list*>] [<*thread-list*>] `step over` [<*granularity*>] [<*count*>] [ & ]

## Viewing source code

**display routine** `disp r`

In X Windows mode, open a new Source Code window displaying source code for a routine. Not available in line mode.

[<*process-list*>] [<*thread-list*>] `display routine` <*language-expression*> [`\;` <*thread-number*> [`,` ...] ]

**display source** `disp so`

In X Windows mode, create a new Source Code window to display a source file. Not available in line mode.

[<*process-list*>] `display source` <*file-name*> [<*thread-number*> [`,` ...] ]

**list** `l`

List lines of source code.

`list` [ [<*file-name*>`:`] <*starting-line*>] [<*number-of-lines*>] ]
[`routine` <*language-expression*>]

## Viewing assembly-language code

**disassemble** `disas`

Display the assembly-language code.

[<*process-list*>] [<*thread-list*>] `disassemble`
[<*starting-address*> [ { `..`<*ending-address*> | `:`<*instruction-count*> } ] ]

**display disassembly** `disp d`

In X Windows mode, create an Assembly Code window displaying assembly-language code. Not available in line mode.

[<*process-list*>] [<*thread-list*>] `display disassembly` [<*address-expression*>]
[ `\;` <*thread-number*> [`,` ...] ]

## Displaying and modifying data

**evaluate** `eva`

Evaluate a language expression.

[<*process-list*>] [<*thread-list*>] `evaluate` <*language-expression*>

**info args** `in ar` *args*

Display arguments of the current routine.

[<*process-list*>] [<*thread-list*>] `info args`

**info expression** `in ex` *describe, whatis*

Display the characteristics of the specified language expression.

[<*process-list*>] [<*thread-list*>] `info expression` <*language-expression*>

**info line** `in li` *i li*

Display the source units for a specified line.

[<*process-list*>] `info line` <*line-specifier*>

**info locals** `in lo` *locals*

Display the local variables of the current routine.

[<*process-list*>] [<*thread-list*>] `info locals`

**info sourceunit** `in so` *i so*

Display the specified source unit.

[<*process-list*>] `info sourceunit` <*source-unit*>

**info symbols** `in sy` *globals, symbols*

Display program symbols.

[<*process-list*>] `info symbols` [<*regular-expression*>]

## Using the `print` command

**print** `pr` *p*

Evaluate a language expression and print the result.

[<*process-list*>] [<*thread-list*>] `print[/[n]`<*format*><*fpmode*>] <*language-expression*>

<*format*> options are as follows:

| | |
|---|---|
| `B` | Binary |
| `C` | Fortran complex |
| `L` | Fortran logical |
| `c` | ASCII character |
| `d` | Signed decimal |
| `e[`<*width*>`.`<*precision*>`]` | Scientific |
| `f[`<*width*>`.`<*precision*>`]` | Floating-point notation |
| `i` | Instruction |
| `n` | Suppresses newline character at end of printed data |
| `o` | Octal |
| `s` | String |
| `u` | Unsigned decimal |
| `x` | Hexadeximal |

**info formatting** `in fo` *ifo*

Display settings for print options/memory display formats.

[<*process-list*>] `info formatting`

**set printopts maxarray** `se pr m`

Set the maximum number of array elements to print.

`set printopts maxarray` <*number-of-elements*>

**set printopts nopadding** `se pr n`

Disable padding with leading zeros when printing.

`set printopts nopadding`

**set printopts padding** `se pr pa`

Enable padding with leading zeros when printing.

`set printopts padding`

**set printopts precision** `se pr pr`

Set the precision used to print floating point numbers.

`set printopts precision` <*width*>`.`<*precision*>

## Viewing and modifying memory

**copy** `cop`

Copy a memory region.

[*<process-list>*] [*<thread-list>*] `copy` *<source-address>*
[`{` `..`*<ending-address>* | `:`*<byte-count>* `}`] `\;`*<destination-address>*

**display examine** `disp e`

Create a Memory Display window in X Windows mode.

[*<process-list>*] [*<thread-list>*] `display examine` *<address-expression>*
[ `\;` *<thread-number>* ]

**examine** `exa` *x*

Display a region of memory.

[*<process-list>*] [*<thread-list>*] `examine`
[ `/` {*<memory-unit>* *<format>* *<fpmode>*}] *<starting-address>*
[{ `..`*<ending-address>* | `:`*<unit-count>*}]

**evaluate** `eva`

Evaluate a language expression.

[*<process-list>*] [*<thread-list>*] `evaluate` *<language-expression>*

**fill** `fil`

Fill a region of memory with the value of an expression.

[*<process-list>*] [*<thread-list>*] `fill` [ `/` *<memory-unit>*] *<starting-address>*
[{ `..`*<ending-address>* | `:`*<unit-count>* }] `\;`*<language-expression>*

**get** `ge`

Restore the contents of memory regions from a binary file.

[*<process-list>*] [*<thread-list>*] `get` *<file-name>* [*<starting-address>*
[{`..`*<ending-address>* | `:`*<byte-count>*}] ] [`\;` ...]

**info formatting** `in fo` *i fo*

Display the settings for memory display formats.

[*<process-list>*] `info formatting`

**put** `pu`

Save the contents of memory to a file for later retrieval.

[*<process-list>*] [*<thread-list>*] `put` *<file-name>* *<starting-address>*
[{`..`*<ending-address>* | `:`*<byte-count>*}] [`\;` ...]

**set format** `se fo`

Set the formats for displaying memory.

[*<process-list>*] [*<thread-list>*] `set format` *<memory-unit>* *<format>*

**set memory** `se m`

Set the unit size for displaying memory.

[*<process-list>*] [*<thread-list>*] `set memory` *<memory-unit>*

## Displaying stack frame information

**backtrace** `ba` *bt*

Display the frames of the call stack.

[*<process-list>*] [*<thread-list>*] `backtrace` [*<frame-count>*]

**display stack** `disp st`

Create a Stack Trace window in X Windows mode. Not available in line mode.

[*<process-list>*] `display stack` [*<frame-specifier>*] [`:`*<thread-number>*]

**frame** `fr` *f*

Change the current stack frame.

[*<process-list>*] [*<thread-list>*] `frame` *<frame-specifier>*

**info frame** `in fr` *ifr*

Display a stack frame.

[*<process-list>*] [*<thread-list>*] `info frame` [*<frame-specifier>*]

**info frame at** `in fr a` *ifr a*

Display the stack frame at the specified address.

[*<process-list>*] [*<thread-list>*] `info frame at` *<language-expression>*

## Displaying register contents

**info control registers** `in co r` *i co r*

Display the control registers (Exemplar systems only).

[<*process-list*>] [<*thread-list*>] `info control registers`

**info cregisters** `in cr` *i cr*

Display the communication registers (C Series only).

[<*process-list*>] `info cregisters`

**info floating point registers** `in fl p r` *i fl p r*

Display the floating-point registers (Exemplar systems only).

[<*process-list*>] [<*thread-list*>] `info floating point registers`

**info psw** `in ps` *i ps*

Display the processor status word (PSW) register.

[<*process-list*>] [<*thread-list*>] `info psw`

**info registers** `in r` *i r*

Display the contents of scalar and address registers (on C Series systems) or general registers (Exemplar systems).

[<*process-list*>] [<*thread-list*>] `info registers`

**info space registers** `in sp r` *i sp r*

Display the space registers (Exemplar systems only).

[<*process-list*>] [<*thread-list*>] `info space registers`

**info vregisters** `in v` *i vr*

Display the vector registers (C Series only).

[<*process-list*>] [<*thread-list*>] `info vregisters`

## Searching memory

**find memory backward** `find m b` *fmb*

Find the first occurence of a byte pattern within a memory region.

[<*process-list*>] [<*thread-list*>] `find memory backward`
[ / <*memory-unit*> ] <*byte-pattern*> <*lowest-address*>
{`..`<*highest-address*> | `:` <*byte-count*>}

**find memory forward** `find m f` *fmf*

Find the first occurrence of a byte pattern within a memory region.

[<*process-list*>] [<*thread-list*>] `find memory forward`
[ / <*memory-unit*>] <*byte-pattern*> <*starting-address*>
{`..`<*ending-address*> | `:`<*byte-count*>}

## Searching source code

**find source** `find s` *fs*

Search for all occurrences of a text string in a source file.

`find source` <*string*> [<*filename*>]

## CXdb `info` commands

**info alias** `in al` *i al*

Display aliases.

`info alias` [<*regular-expression*>]

**info args** `in ar` *args*

Display arguments of the current routine.

[<*process-list*>] [<*thread-list*>] `info args`

**info break** `in br` *b?*

Display all existing breakpoints.

[<process-list>] [<*thread-list*>] `info break`

**info cregisters** `in cr` *i cr*

Display the communication registers (C Series only).

[<*process-list*>] `info cregisters`

**info cxdb** `in cx` *i cx*

Display the status of the current CXdb session.

`info cxdb`

**info default environment** `in d e` *i d e*

Display all default environment variables.

`info default environment` [<*regular-expression*>]

**info dirpath** `i di`

List alternate CTI directory paths or the names of object files that match a regular expression.

`info dirpath` [<*regular-expression*>]

**info dynamicobject** `in dy` *i dy*

Display memory segments of dynamically loaded objects (C Series only).

[<*process-list*>] `info dynamicobject`

**info environment** `in en` *env?*

Display all process environment variables.

[<*process-list*>] `info environment` [<*regular-expression*>]

**info errno** `in er` *i er*

Display the system error message received by the process.

[<*process-list*>] [<*thread-list*>] `info errno`

**info event** `in event` *e?*

Display the specified eventpoints.

`info event` [<*event-specifier*>] [, ...]

**info eventtype** `in eventt` *et?*

Display all eventpoints of the specified type.

[<*process-list*>] `info eventtype`
<*eventtype-specifier*> [, ...]

**info expression** `in ex` *describe, whatis*

Display the characteristics of the specified language expression.

[<*process-list*>] [<*thread-list*>] `info expression` <*language-expression*>

**info formatting** `in fo` *i fo*

Display the settings for memory display formats.

[<*process-list*>] `info formatting`

**info frame** `in fr` *i fr*

Display a stack frame.

[<*process-list*>] [<*thread-list*>] `info frame` [<*frame-specifier*>]

**info frame at** `in fr a` *i fr a*

Display the stack frame at the specified address.

[<*process-list*>] [<*thread-list*>] `info frame at` <*language-expression*>

**info history** `in h` *i h*

Display the CXdb command history.

`info history` [<*command-count*>]

**info line** `in li` *i li*

Display the source units for a specified line.

[<*process-list*>] `info line` <*line-specifier*>

**info locals** `in lo` *locals*

Display the local variables of the current routine.

[<*process-list*>] [<*thread-list*>] `info locals`

**info macro** `in m` *i m*

Display macros.

`info macro` [<*regular-expression*>]

**info objectmap** `in o` *i o*

Display the object map.

[<*process-list*>] `info objectmap`

**info path** `in pa` *p+*

Display the directories in the search path.

[<*process-list*>] `info path`

**info process** `in pr` *p?*

Display the status of the process.

[<*process-list*>] `info process`

**info psw** `in ps` *i ps*

Display the processor status word (PSW) register.

[<*process-list*>] [<*thread-list*>] `info psw`

**info registers** `in r` *i r*

Display the contents of scalar and address registers (on C Series systems) or general registers (Exemplar systems).

[<*process-list*>] [<*thread-list*>] `info registers`

**info scope** `in sc` *where*

Display the current scope path.

[<*process-list*>] [<*thread-list*>] `info scope`

**info signal** `in si` *i si*

Display signal names, numbers, and settings for signal actions.

[<*process-list*>] `info signal` [<*signal-specifier*>] [, ...]

**info sourceunit** `in so` *i so*

Display the specified source unit.

[<*process-list*>] `info sourceunit` <*source-unit*>

**info stack** `in st` *i st*

Display information about the process stack.

[<*process-list*>] [<*thread-list*>] `info stack`

**info symbols** `in sy` *globals, symbols*

Display program symbols.

[<*process-list*>] `info symbols` [<*regular-expression*>]

**info threads** `in th` *i th*

Display threads of the current process.

[<*process-list*>] `info threads`

**info trace** `in tr` *t?*

Display all tracepoints.

[<*process-list*>] `info trace`

**info type** `in ty` *i ty*

Display information about type definitions in C programs.

[<*process-list*>] [<*thread*>] `info type` [<*regular-expression*>]

**info vregisters** `in v` *i vr*

Display the vector register (C Series only).

[<*process-list*>] [<*thread-list*>] `info vregisters`

**info watch** `in w` *i w*

Display all watchpoints.

[<*process-list*>] `info watch`

## Logging input, output, and messages

You can log the following types of information:

- cmderr—CXdb error and informational messages.
- cmdlog—User entries on the CXdb command line.
- cmdout—Normal output generated by CXdb in response to commands.

To append to an existing log file, use redirection operators. A viewport can be a file, the CXdb Command window (X Windows mode only), or stderr, stdin, and stdout in line mode.

**add cmderr** `ad cmde`

Add file names to the list of files that log CXdb messages.

```
add cmderr <filename> [, ...]
```

**add cmdlog** `ad cmdl`

Add file names to the list of files that log CXdb input.

```
add cmdlog <filename> [, ...]
```

**add cmdout** `ad cmdo`

Add file names to the list of files that log CXdb command output.

```
add cmdout <filename> [, ...]
```

**clear logging** `cl l`

Disable logging for cmdlog.

```
clear logging
```

**clear noclobber** `cl n`

Allow overwriting of existing log files or creation of new files for appending for all viewports.

```
clear noclobber
```

**remove cmderr** `rem cmde`

Delete file names from the list of files that log CXdb messages.

```
remove cmderr <filename> [, ...]
```

**remove cmdlog** `rem cmdl`

Delete file names from the list of files that log CXdb input.

```
remove cmdlog <filename> [, ...]
```

**remove cmdout** `rem cmdo`

Delete file names from the list of files that log CXdb command output.

```
remove cmdout <filename> [, ...]
```

**set cmderr** `se cmde`

Clear and redefine the viewport list for cmderr.

```
set cmderr <viewport> [, ...]
```

**set cmdlog** `se cmdl`

Clear and redefine the viewport list for cmdlog.

```
set cmdlog <viewport> [, ...]
```

**set logging** `se l`

Enable logging for cmdlog.

```
set logging
```

**set noclobber** `se n`

When noclobber is set, CXdb responds with an error message if you try to overwrite an existing log file.

```
set noclobber
```

## Redirecting process I/O

To redirect process I/O, use a shell redirection operator with the CXdb `run` command. Use a backslash (\) before the redirection operator to distinguish it from a CXdb redirection operator. For example, the command

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

starts execution of your program, with input from the file prog_input. Output from the program is directed to the file prog_output.

## CXdb redirection operators

### For command output

| Operator | Action |
|---|---|
| > | Overwrite existing command output file, unless noclobber is set. |
| >> | 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. |

### For error and information message output

| Operator | Action |
|---|---|
| >& | Overwrite existing message output file, unless noclobber is set. |
| >>& | 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. |

## Setting and clearing CXdb defaults

**clear autocreate** `cl a`

In X Windows mode, disable the dynamic creation of Source Code windows.
Not available in line mode.

```
clear autocreate
```

**clear echo** `cl ec`

Disable echoing of input from initialization files and command files.

```
clear echo
```

**clear noclobber** `cl n`

Allow overwriting of existing log files or creation of new files for appending for all viewports.

```
clear noclobber
```

**clear seq** `cl se`

Clear the sequential mode (SEQ) bit (C Series only).

[<*process-list*>] [<*thread-list*>] `clear seq`

**clear sqs** `cl sq`

Clear the sequential store enable (SQS) bit (C Series only).

[<*process-list*>] [<*thread-list*>] `clear sqs`

**set autocreate** `se a`

In X Windows mode, enable dynamic creation of Source Code windows.

```
set autocreate
```

**set echo** `se ec`

Enable echoing of input from command files and initialization files.

```
set echo
```

**set evalopts fpmode** `se ev f`

Set the floating point mode for evaluating expressions (C Series only).

```
set evalopts fpmode { ieee | native | dual }
```

**set evalopts iprecision** `se ev i`

Set the size of integer constants for CXdb.

```
set evalopts iprecision {4 | 8}
```

**set evalopts rprecision** `se ev r`

Set the size of real numbers for CXdb.

```
set evalopts rprecision {4 | 8}
```

**set format** `se fo`

Set the formats for displaying memory.

[*<process-list>*] [*<thread-list>*] `set format` *<memory-unit>* *<format>*

**set memory** `se m`

Set the unit size for displaying memory.

[*<process-list>*] [*<thread-list>*] `set memory` *<memory-unit>*

**set noclobber** `se n`

When noclobber is set, CXdb responds with an error message if you try to overwrite an existing log file.

```
set noclobber
```

**set printopts maxarray** `se pr m`

Set the maximum number of array elements to print.

`set printopts maxarray` *<number-of-elements>*

**set printopts nopadding** `se pr n`

Disable padding with leading zeros when printing.

```
set printopts nopadding
```

**set printopts padding** `se pr pa`

Enable padding with leading zeros when printing.

```
set printopts padding
```

**set printopts precision** `se pr pr`

Set the precision used to print floating point numbers.

`set printopts precision` *<width>*`.`*<precision>*

**set remotewd** `se r`

Set the remote working directory.

[*<process-list>*] `set remotewd` *<directory-specifier>*

**set seq** `se se`

Set the sequential mode (SEQ) bit (C Series only).

[*<process-list>*] [*<thread-list>*] `set seq`

**set shell** `se sh`

Set the type of shell invoked from within CXdb.

```
set shell {sh | csh | tcsh | ksh}
```

**set sqs** `se sq`

Set the sequential store enable (SQS) bit (C Series only).

[<*process-list*>] [<*thread-list*>] `set sqs`

**set typehandler** `se t`

Define the default handler for all eventpoints of the specified type.

`set typehandler` <*eventtype-specifier*> [, ...] {<*event-handler*>}

## Setting and clearing global process defaults

These commands affect any new process that is created by CXdb *after* the default is changed.

**add default environment** `ad d e` *denv+*

Add or modify environment variables in the default environment.

```
add default environment
   <environment-variable> = <string> [, ...]
```

**add default path** `ad d p` *dp+*

Add directories to the default search path.

`add default path` <*directory-specifier*> [, ...]

**clear default environment** `cl d e`

Remove all environment variables from the default environment.

```
clear default environment
```

**clear default fixed sched** `cl d f s`

Disable fixed scheduling in the default settings (C Series only).

```
clear default fixed sched
```

**clear default handler** `cl d h`

Clear the default handler for all eventpoints.

```
clear default handler
```

**clear default remotewd** `cl de re`

Clear the default remote working directory (C Series only).

```
clear default remotewd
```

**clear step** `cl st`

Reset the stepping granularity to the default setting.

[*<process-list>*] `clear step`

**remove default environment** `rem d e` *denv-*

Delete environment variables from the default environment.

`remove default environment` *<environment-variable>* [, ...]

**remove default path** `rem d p` *dp-*

Delete directories from the default search path.

`remove default path` *<directory-specifier>* [, ...]

**set default environment** `se de e` *denv=*

Clear and redefine the environment variables for the default environment.

`set default environment` *<environment-variable>* = *<string>* [, ...]

**set default fixed sched** `se de fi s`

Enable fixed scheduling in the default settings (C Series only).

`set default fixed sched`

**set default format** `se de fo`

Set the default formats for displaying memory.

`set default format` *<memory-unit>* *<format>*

**set default fpmode** `se de fp`

Set the default floating point mode for new processes (C Series only).
On Exemplar systems, the floating-point mode is IEEE.

`set default fpmode { ieee | native | dual }`

**set default handler** `se de h`

Set the default handler for eventpoints.

`set default handler {`*<event-handler>*`}`

**set default memory** `se de m`

Set the default unit size for displaying memory.

`set default memory` *<memory-unit>*

**set default path** `se de pa` *dp=*

Set the default search path.

`set default path` *<directory-specifier>* [, ...]

**set default pshell** `se de ps`

Set the default process shell.

`set default pshell {sh | csh}`

**set default remotewd** `se de re`

Set the default remote working directory.

`set default remotewd` *<directory-specifier>*

**set default step** `se de s`

Set the default stepping granularity.

`set default step` *<granularity>*

## Setting and clearing defaults for the current process

These commands affect default settings for the *current* process only.

**add environment** `ad e` *env+*

Add or modify environment variables in the process environment.

[*<process-list>*] `add environment` *<environment-variable>* = *<string>* [, ...]

**clear environment** `cl en`

Remove all environment variables from the process environment.

[*<process-list>*] `clear environment`

**clear fixed sched** `cl f s` *cfs*

Disable fixed scheduling in the process settings.

[*<process-list>*] `clear fixed sched`

**clear handler** `cl h`

Clear the handler for a specified eventpoint.

`clear handler` *<event-specifier>* [, ...]

**clear typehandler** `cl t`

Clear the handler for a specified type of eventpoint.

`clear typehandler` *<eventtype-specifier>* [, ...]

**remove environment** `rem en` *env-*

Delete environment variables from the process environment.

[*<process-list>*] `remove environment` *<environment-variable>* [, ...]

**remove path** `rem p` *p-*

Delete directories from the process search path.

[<*process-list*>] `remove path` <*directory-specifier*> [, ...]

**set environment** `se en` *env=*

Clear and redefine the environment variables for the process environment.

[<*process-list*>] `set environment` <*environment-variable*>=<*string*> [, ... ]

**set fixed sched** `se fi s` *sfs*

Enable fixed scheduling in the process settings (C Series only).

[<*process-list*>] `set fixed sched`

**set fpmode** `se fp`

Set the floating point mode for the process (C Series only).

[<*process-list*>] `set fpmode { ieee | native }`

**set path** `se pa` *p=*

Set the search path for the process.

[<*process-list*>] `set path` <*directory-specifier*> [, ...]

## Working with aliases and macros

**alias** `al`

Define an alias.

`alias` *<alias-name>* *<string>*

**csd** `csd`

Enable compatibility with `csd` debugger commands.

`csd`

**gdb** `gdb`

Enable compatibility with `gdb` debugger commands.

`gdb`

**info alias** `in al` *i al*

Display aliases.

`info alias` [*<regular-expression>*]

**info macro** `in m` *i m*

Display macros.

`info macro` [*<regular-expression>*]

**macro** `m`

Define a macro.

`macro` *<name>* [(*<parameter>*[:*<default>*][, ...]) ] *<string>*

**remove alias** `rem a`

Delete an alias.

`remove alias` *<alias-name>*

## Path and directory commands

**add path** `ad p` *p+*

Add directories to the search path.

[*<process-list>*] `add path` *<directory-specifier>* [`,` ...]

**cd** `cd`

Change the console working directory.

`cd` *<directory-specifier>*

**dirpath** `dir`

Specify an alternate directory path for the CTI data files.

`dirpath` *<original-directory>* *<alternate-directory>*

**info dirpath** `i di`

List alternate CTI directory paths or the names of object files that match a regular expression.

`info dirpath` [*<regular-expression>*]

**info path** `in pa` *p+*

Display the directories in the search path.

[*<process-list>*] `info path`

**pwd** `pw`

Display the name of the console working directory.

`pwd`

**remove dirpath** `rem di`

Remove CTI directory paths created with the `dirpath` command.

`remove dirpath` [*<original-directory>*]

**set directory** `se di`

Set the process working directory.

[*<process-list>*] `set directory` *<directory-specifier>*

**set remotewd** `se r`

Set the remote working directory.

[*<process-list>*] `set remotewd` *<directory-specifier>*

## Working with signals

**event signal** `eve si`

Set an eventpoint to catch a signal.

[*<process-list>*] `event signal` *<signal-specifier>*
[ {*<event-handler>*} ] [*<debugger-variable>*]

**info signal** `in si` *i si*

Display signal names, numbers, and settings for signal actions.

[*<process-list>*] `info signal` [*<signal-specifier>*]
[, ...]

**set signal** *se si*

Set the actions for the specified signal.

[*<process-list>*] `set signal` *<signal-specifier>*
[{**stop** | **nostop**},] [{**pass** | **nopass**},] [{**print** | **noprint**}]

**signal process** `sig p`

Send a signal to the process and continue process execution.

[*<process-list>*] `signal process` *<signal-specifier>*
[ **&** ]

**signal thread** `sig t`

Send a signal to a specific thread of the process, and continue execution of that thread.

[*<process-list>*] *<thread>* `signal thread` *<signal-specifier>* [ `&` ]

## Working with threads

**clear default fixed sched** `cl d f s`

Disable fixed scheduling in the default settings (C Series only).

```
clear default fixed sched
```

**event join** `eve j`

Set an eventpoint to trap a thread joining.

```
[<process-list>] event join
    [ {<event-handler>} ] [<debugger-variable>]
```

**event spawn** `eve sp`

Set an eventpoint to trap the spawning of a thread.

```
[<process-list>] event spawn [ {<event-handler>} ] [<debugger-variable>]
```

**info threads** `in th` *i th*

Display threads of the current process.

```
[<process-list>] info threads
```

**set default fixed sched** `se de fi s`

Enable fixed scheduling in the default settings (C Series only).

```
set default fixed sched
```

**set fixed sched** `se fi s` *sfs*

Enable fixed scheduling in the process settings (C Series only).

```
[<process-list>] set fixed sched
```

**set threads** `se th`

In X Windows mode, associate windows with particular threads.

```
set threads <window> [, ...]
    [<thread-number> [, ...] ]
```

**signal thread** `sig t`

Send a signal to a specific thread of the process, and continue execution of that thread.

```
[<process-list>] <thread> signal thread <signal-specifier> [ & ]
```

## Debugger variables

A debugger variable can store

- The results of a language expression
- The contents of a register
- A signal number
- A CXdb object number, such as an eventpoint number or window number

CXdb also has predefined debugger variables for registers and other values. Refer to the "debugger variables" reference page or online help topic for information and examples on creating and referencing debugger variables.



### remove variable

`rem v`

Delete a debugger variable.

`remove variable` *<debugger-variable>*

## CXdb history commands

### info history

`in h` *i h*

Display the CXdb command history.

`info history` [*<command-count>*]

### recall

`rec` *!*

Reexecute a previous command.

`recall [ ? ]`*<string>*

## Miscellaneous commands

### echo

`ec`

Echo a character string.

`echo[/n]` <*string*> [...]

### edit

`ed`

In X Windows mode, open an editor window and edit a file.

`edit` [<*file-name*>]

### if

`if`

Establish conditional execution of CXdb commands.

`if` (<*relational-expression*>) <*command-set*> [**`else`** <*command-set*>]

### load object

`l o`

Load CTI data for an object file that has been dynamically loaded (C Series only).

[<*process-list*>] `load object` <*file-name*> <*text-address*>
<*data-address*> <*tdata-address*> <*bss-address*> <*tbss-address*>

### shell

`sh`

Invoke a shell.

`shell` [/<*shell-specifier*>] [<*shell-commands*>]

### source

`sou`

Execute a CXdb command file.

`source` <*file-name*>

# Index

## Symbols



## A



## B



## C

## D



## E



## F



## G



## H

## Q



## R



## S

## T



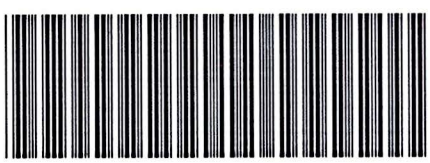
## V



## W

ORDER NUMBER
DSW-474

DOCUMENT NUMBER
710-015630-002



CONVEX PRESS