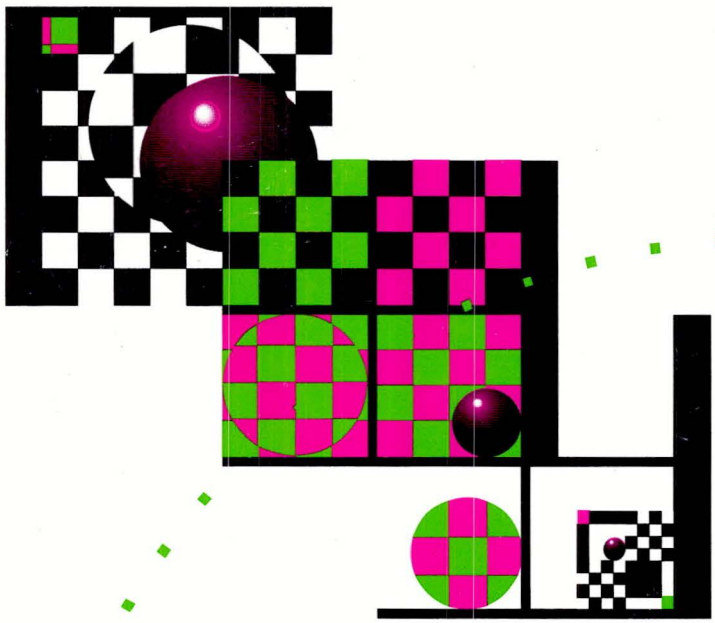


CONVEX

CXdb Reference

Volume 1
Commands and Parameters

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.

Commands

1

This chapter contains reference pages for CXdb commands. There is a separate reference page for each command. Each reference page can contain the following sections:

- **Description** — Text explaining the purpose and functionality of the command.
- **Syntax** — Format rules for the command and its parameters.
- **Examples** — One or more examples illustrating the use of the command.
- **Related Commands** — A list of CXdb commands related to the current command being described. The related commands are also described in this chapter.
- **Related Concepts** — A list of concepts related to the current command. Concepts are described in Chapter 3.
- **Related Parameters** — A list of parameters related to the current command. CXdb parameters are described more fully in Chapter 2.
- **Related Windows** — A list of CXdb windows that you can use with the current command if you are running CXdb in X Windows mode. The windows are described in Chapter 4.

The heading at the top of each command description contains the following lines of information:

Full command name → break line

Shortest abbreviation → bre l

Default alias or aliases (if any exist) → *bl*

You can invoke a command by using any of the above three forms, or you can create your own aliases and macros.

add cmderr

ad cmde

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

Syntax

`add cmderr` *<file-name>* **`[, ...]`**

Parameter	Meaning
<file-name>	The name of a file that is to receive and store CXdb messages. Each file name is relative to the console working directory unless it is qualified by a path name.
`[, ...]`	A list of additional file names. Multiple file names in the list must be separated by commas. Spaces between the list items are optional.

Description

The `add cmderr` command adds file names to the list of viewports for cmderr.

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

CXdb creates the specified 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 add viewports to cmderr.

```
(CXdb) add cmderr errmsgs
New cmderr: Window #1, errmsgs
```

The above command adds the file called errmsgs to the cmderr viewport list. The file name is relative to the console working directory in this case. Note that the list already contains Window #1 (the Command window), which is the default viewport for cmderr.

```
(CXdb) add cmderr /tmp/errlog, myerrlog
New cmderr: Window #1, errmsgs, /tmp/errlog, myerrlog
```

The above command adds two more files to the cmderr viewport list. The files are myerrlog in the console working directory and errlog in the directory /tmp. Window #1 and the file errmsgs are still included in the viewport list from the previous example.

Related Commands

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

Related Concepts

cmderr	cmdlog
cmdout	logging
redirection	viewports

Related Parameters

redirection-operator	viewport

add cmdlog

ad cmdl

Add file names to the list of files for logging CXdb input.

Syntax

add cmdlog *<file-name>* [, ...]

Parameter	Meaning
<file-name>	The name of a file that is to receive and store CXdb command-line input. Each file name is relative to the console working directory unless it is qualified by a path name.
[, ...]	A list of additional file names. Multiple file names in the list must be separated by commas. Spaces between the list items are optional.

Description

The `add cmdlog` command adds file names to the list of viewports for cmdlog.

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 equivalent 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. Therefore, 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.

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). Therefore, no input is sent to the cmdlog files unless you first use the `set logging` command to enable logging.

To display the current settings for logging and noclobber, as well as the current viewport list for cmdlog, use the `info cxdb` command.

Examples

The following examples illustrate how to add viewports to cmdlog.

```
(CXdb) add cmdlog logfile
New cmdlog: logfile
```

The above command adds the file called logfile to the cmdlog viewport list. The file name is relative to the console working directory in this case.

```
(CXdb) add cmdlog logfile2, /tmp/inputlog
New cmdlog: logfile, logfile2, /tmp/inputlog
```

The above command adds two more files to the cmdlog viewport list. The files are logfile2 in the console working directory and inputlog in the directory /tmp/inputlog. The file logfile is still included in the viewport list from the previous example.

Related Commands

`add cmderr`
`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

add cmdout

ad cmdo

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

Syntax

add cmdout *<file-name>* [, ...]

Parameter	Meaning
<file-name>	The name of a file that is to receive and store CXdb output. Each file name is relative to the console working directory unless it is qualified by a path name.
[, ...]	A list of additional file names. Multiple file names in the list must be separated by commas. Spaces between the list items are optional.

Description

The `add cmdout` command adds file names to the list of viewports for cmdout.

Cmdout is the list of viewports (destinations) that receive the normal output generated in response to CXdb commands. Cmdout is equivalent 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 includes the CXdb Command window (or stdout in line mode).

CXdb creates the specified 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 add viewports to cmdout.

```
(CXdb) add cmdout outputdata
New cmdout: Window #1, outputdata
```

The above command adds the file called `outputdata` to the cmdout viewport list. The file name is relative to the console working directory in this case. Note that the list already contains Window #1 (the Command window), which is the default viewport for cmdout.

```
(CXdb) add cmdout /tmp/outputlog, myoutputlog
New cmdout: Window #1, outputdata, /tmp/outputlog, myoutputlog
```

The above command adds two more files to the cmdout viewport list. The files are myoutputlog in the console working directory and outputlog in the directory /tmp. Window #1 and the file outputdata are still included in the viewport list from the previous example.

Related Commands

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

Related Concepts

cmderr
cmdlog
cmdout
logging
redirection
viewports

Related Parameters

redirection-operator
viewport

add default environment

ad d e
denv+

Add or modify environment variables in the default environment.

Syntax

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

Parameter	Meaning
<environment-variable>	An environment variable to add or change.
<string>	The value to be given to the environment variable.
[**,** ...]	An optional list of environment variable assignments. The assignments must be separated by commas.

Description

The `add default environment` command creates or changes the specified environment variables in the default environment.

If the environment variable already exists, its old value is replaced by the new value. If it does not exist, it is created. The default environment is passed to a new process if the process object does not have its own environment.

Examples

The following examples add environment variables to the default environment.

```
(CXdb) add default environment EDITOR = vi
```

The above command adds the environment variable `EDITOR` to the default environment. If the variable `EDITOR` did not exist in the default environment, the variable was created. If it did exist, its value was changed.

```
(CXdb) add default environment EDITOR = emacs
```

The above example changes the string of the environment variable `EDITOR` from `vi` to `emacs`. Because the variable exists from the previous example, CXdb replaces the old value of the variable with the new string.

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

The above command adds the environment variables `INITVAL` and `ENDVAL` to the default environment. Because the strings each contain a white space character (a blank), they must be delimited by either quotes or double quotes. The comma is required to separate the variable assignments.

Related Commands

`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
environment
process object

Related Parameters

environment-variable
string

add default path

ad d p
dp+

Add directories to the default search path.

Syntax

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

Parameter	Meaning
<directory-specifier>	A directory to be added to the default search path.
[, ...]	An optional list of directories. The directories must be separated by commas.

Description

The `add default path` command appends the specified directories to the default search path.

Each new process object receives the modified default search path as part of its search path. Relative path names specified in the `add default path` command use the console working directory as a base path name. The `add default path` command can be used in initialization files to create default search paths automatically.

The `add default path` command has the same effect as the `-D` parameter when invoking CXdb from the shell prompt.

The default search path can be displayed using the `info path` command.

Examples

The following examples add directories to the default search path.

```
(CXdb) add default path /mnt/jones/project/source
```

The above command adds the /mnt/jones/project/source directory to the default search path.

When a new process object is created it inherits the default search path, which now includes the /mnt/jones/project/source directory. This modification to the default search path occurs only for the current session of CXdb. To always include this as part of the default search path, place this line in an initialization file.

```
(CXdb) add default path libraries , ~/math/libraries
```

For the above example, assume the console working directory is /mnt/jones. The `add default` command adds two directories to the default search path. The first directory is relative to the console working directory. The second directory uses the tilde (~) to indicate that the directory is based from the home directory, which in this example is also the /mnt/jones directory.

The tilde character is expanded to /mnt/jones before the directory is added. The comma separates the directories in the list and is required.

Related Commands

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

Related Concepts

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

Related Parameters

directory-specifier

add environment

ad e

env+

Add or modify environment variables in the process environment.

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process object.
<environment-variable>	An environment variable to add or change.
<string>	The value of the environment variable.
[, ...]	An optional list of environment variable assignments. The assignments must be separated by commas.

Description

The `add environment` command adds the specified environment variables to the environment of a process object.

If the environment variable already exists, its old value is replaced by the new value. If it does not exist, it is created. If the process object does not yet have its own environment, the `add environment` command creates an environment for the process object consisting of the default environment and the environment variables specified in the command.

Each new process will receive the modified environment. A process that is currently executing is not affected.

Examples

The following examples add environment variables to the environment of the current process object. Assume that an environment has not yet been created for the current process object.

```
(CXdb) add environment EDITOR = vi
```

In the above example, the environment of the process object is created, and the variable `EDITOR` is added to it. The `add environment` command indicates to CXdb that you want to modify the environment for this process object. CXdb creates an environment for the process object that consists of a copy of the default environment and the variable `EDITOR`.

```
(CXdb) add environment EDITOR = emacs
```

The above example changes the value of the environment variable `EDITOR` to `emacs`. The environment for the process object was created in the first example. Because the environment variable already exists, CXdb replaces the old value with the new value.

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

The above command adds the environment variables `INITVAL` and `ENDVAL` to the environment. Because the strings contain a white space character (a blank) they must be delimited by either quotes or double quotes. The comma is required to separate the variable assignments.

Related Commands

`add default 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
environment
process object

Related Parameters

environment-variable
process-list
string

add path

ad p

p+

Add directories to the search path.

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process.
<directory-specifier>	The directory to be added to the search path of the process object.
[**,** ...]	An optional list of directories. The directories must be separated by commas.

Description

The `add path` command appends the specified directories to the search path of the process object. You may need to use this command if you have moved the source files or CTI data files for your program or if you have changed the names of any of those files.

CXdb uses the updated search path the next time it searches for a source file for the process object. Relative directory names use the console working directory as the base path name.

The `add path` command can be included in command files to create search paths automatically.

NOTE: If your source code was compiled using a version of the CONVEX Fortran compiler later than V7.0 or a version of the CONVEX C compiler later than V4.3, you may not need to specify a search path. These compilers embed the location of the source code and CTI data files for a program in the executable file itself. When using these newer compilers, you generally do not need the `add path` command unless you have changed the location of the source code.

Examples

The following examples add directories to the search path of the current process object.

```
(CXdb) add path /usr/smith
```

The above command adds the source file directory /usr/smith to the search path.

```
(CXdb) add path /usr/smith/programs,/usr/smith/data
```

The above example adds two more directories to the end of the search path. The comma separates the two directories specified in the list and is required.

Related Commands

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

Related Concepts

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

Related Parameters

directory-specifier
process-list

alias

al

Define an alias.

Syntax

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

Parameter	Meaning
<alias-name>	A character string that forms the name of the alias. If the name contains white space, enclose it within single or double quotes (' or "). The name cannot contain a forward or backward slash (/ or \). Alias names are case sensitive.
<string>	The character string that is substituted for the alias name whenever CXdb expands the alias.

Description

The `alias` command defines a synonym, or substitute, for a CXdb command.

An alias can represent the first part of a command line, a complete command, or multiple commands. Aliases cannot accept parameters. Aliases can be nested within other aliases, but not recursively.

When using an alias, you must enter it as the first item on the command line. CXdb expands the alias before parsing and executing the command.

Note that an alias definition remains in effect only during the current debugging session. Therefore, if you have a set of aliases that you want to use regularly, you should define them in a CXdb command file or initialization file.

To display current definitions of existing aliases, use the `info alias` command.

Examples

The following examples illustrate how to define aliases.

```
(CXdb) alias P print
```

The above example defines the name `P` as an alias for the CXdb command `print`.

```
(CXdb) alias ssl 'set step loop'
```

The above example defines `ssl` as an alias, or substitute, for the string `set step loop`.

```
(CXdb) alias 'step & print' 'step block; info locals'
```

The above example defines the string `step & print` as an alias for the two CXdb commands `step block` and `info locals`. Because the alias name contains white space, it must be enclosed in quotation marks. However, the quotation marks are not used when invoking the alias, as the following example shows.

```
(CXdb) step & print

Stepping process [#0/*] by 1 block
Process [#0/0] stopped by Bkpt 0, at [0x800050ec] EXAMPLE in example.f line 11
(CXdb) Process [#0/0]
Frame : 0; [0x800050ec] EXAMPLE in example.f line 11
Number of locals : 4
     1 :  ARRAY = INTEGER*4(1:4, 1:4) 0x80073ce8
     2 :  I = (INTEGER*4) 1
     3 :  NUMARGS = (INTEGER*4) 0
     4 :  J = (INTEGER*4) 2
```

The above example invokes the alias `step & print`. This alias steps the current process by one block and then displays information about the local variables.

```
(CXdb) alias start "debug exec a.out; break instruction SUB2; run"
```

The above example defines the name `start` as an alias for the following three CXdb commands:

```
debug exec a.out
break instruction SUB2
run
```

You might prefer to use a macro instead of an alias in this case because macros can accept parameters.

Related Commands

`info alias`
`info macro`
`macro`
`remove alias`
`remove macro`

Related Concepts

command files
initialization files

Related Parameters

string

attach

at

Debug the image of a running process.

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process object.
<remote-host>	The name of a remote host (C Series only). The name can be either an absolute Internet address or a name in the /etc/hosts file.
<process-id>	The process ID of the process you wish to attach to.

Description

The `attach` command lets you debug a process that is already running. The process image of the attached process becomes the image of the process object.

CXdb attaches to the process, brings it under control, and stops it. The image of the process replaces any existing process or core image in the process object. The process object must already have been created using one of the three `debug` commands.

You can debug an attached process as you would a new process, using absolute addresses. If the executable file that created the process is specified, you can debug the process image with global and static symbols. If the executable file was compiled with the `-cxdb` option, you can debug the process symbolically.

If you are already debugging a process image, you must kill that process before you can use the `attach` command. To kill an existing process, use the `kill process` command.

To debug a process running on a remote host (C Series only), precede the process ID with the remote host name and a colon.

Examples

The following command attaches CXdb to a running process.

```
(CXdb)  attach 12345
Attaching Process [#0] to pid 12345
Process [#0/0] stopped by attach at [0x800013de] FIB in program.f line 10
```

This command brings the running process `12345` under the control of CXdb. The process is stopped as soon as CXdb gains control of it. The image from this process is now the image being debugged.

Related Commands

`core`
`debug core`
`debug exec`
`debug proc`
`detach`
`executable`
`info cxdb`
`info process`
`kill process`
`rerun`
`run`

Related Concepts

process object
remote debugging

Related Parameters

process-list

backtrace

ba

bt

Display the frames of the call stack.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.
<frame-count>	The number of frames to display. The count must be an unsigned integer. The default is all frames of the specified threads and processes.

Description

The `backtrace` command displays summary information about the frames of the process stack. The information displays in the Command window.

Examples

The following examples illustrate how to display information about stack frames.

```
(CXdb) backtrace
Process [#0/0]
cf> 0 : 0x800015a4  in SUB1(MAXLEN = (INTEGER*4) 1000, A = (REAL*4(1:<TEMP0>)) ,
        B = (REAL*4(1:<TEMP0>, 1:<TEMP0>)) )  (para.f line 17)
    1 : 0x800014ce  in PARA()  (para.f line 11)
    2 : 0x80001d08  in _main(1, 0xffffcafc, 0xffffcb04)
    3 : 0x800010e0  in _ _ _ap$envret()

Process [#0/1]
cf> 0 : 0x8000159c  in SUB1(MAXLEN = (INTEGER*4) 1000, A = (REAL*4(1:<TEMP0>)) ,
        B = (REAL*4(1:<TEMP0>, 1:<TEMP0>)) )  (para.f line 17)
    1 : 0x800014ce  in PARA()  (para.f line 11)
    2 : 0x80001d08  in _main(1, 0xffffcafc, 0xffffcb04)
    3 : 0x800010e0  in _ _ _ap$envret()
```

The above command displays all stack frames for all threads of the current process (in this case, there are two threads). In the response, CXdb lists each frame by number starting with frame 0, which is always the top frame of the stack. For each frame, the displayed information includes the following, when applicable:

- The currently selected frame, indicated by the symbol `cf>`
- The execution address of the frame, in hexadecimal notation
- The name of the routine called by the frame
- A list of the arguments for the routine
- The name of the source file that contains the routine
- The line number for source code represented by the execution address

```
(CXdb) backtrace 2
Process [#0/0]
cf> 0 : 0x800015a4  in SUB1(MAXLEN = (INTEGER*4) 1000, A = (REAL*4(1:<TEMP0>)) ,
        B = (REAL*4(1:<TEMP0>, 1:<TEMP0>)) )  (para.f line 17)
    1 : 0x800014ce  in PARA()  (para.f line 11)
More frames follow...

Process [#0/1]
cf> 0 : 0x8000159c  in SUB1(MAXLEN = (INTEGER*4) 1000, A = (REAL*4(1:<TEMP0>)) ,
        B = (REAL*4(1:<TEMP0>, 1:<TEMP0>)) )  (para.f line 17)
    1 : 0x800014ce  in PARA()  (para.f line 11)
More frames follow...
```

The above command displays the top two frames of the stack for all threads of the current process (in this example, there are two threads). The response also indicates that there are more frames than the two displayed.

```
(CXdb) :T1 backtrace
Process [#0/1]
cf> 0 : 0x8000159c  in SUB1(MAXLEN = (INTEGER*4) 1000, A = (REAL*4(1:<TEMP0>)) ,
        B = (REAL*4(1:<TEMP0>, 1:<TEMP0>)) )  (para.f line 17)
    1 : 0x800014ce  in PARA()  (para.f line 11)
    2 : 0x80001d08  in _main(1, 0xffffcafc, 0xffffcb04)
    3 : 0x800010e0  in _ _ _ap$envret()
```

The above command displays all stack frames for thread 1 of the current process (the second thread in this case, because the first thread is numbered 0).

Related Commands	`frame`	`info frame`
	`info frame at`	`info stack`
Related Parameters	process-list	thread-list
Related Windows	Stack Frame Description dialog	Stack Trace window

break instruction

bre i
bi

Set a breakpoint at an instruction.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.
<language-expression>	A valid language expression whose evaluation is used as the instruction address.
<event-handler>	A sequence of CXdb commands enclosed within curly braces ({}). Each command must be terminated with a semicolon (;).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

The `break instruction` command sets a breakpoint at the start of the specified instruction address.

The address can be any valid language expression that evaluates to an address.

When the breakpoint is triggered, process execution stops, and the commands of the breakpoint's handler are executed. If the breakpoint does not have its own handler, the default handler for breakpoints, which displays a message, is executed. Unless the breakpoint handler includes the `resume` command, execution is not restarted.

Examples

The following examples set breakpoints at specific instruction addresses.

```
(CXdb) break instruction PRINT_ARRAY
#0: break instruction, on [#0/*], Enabled, ignore 0/0
[0x80005506] PRINT_ARRAY in example.f line 35
```

The above command sets a breakpoint at the first instruction of the routine `PRINT_ARRAY`. The evaluation of the language expression `PRINT_ARRAY` is used as the address for this breakpoint. When a routine name is used with a `break instruction` command, the breakpoint is placed before the preamble (which manages the stack) of the routine. In contrast, a routine name used with a `break routine` command places the breakpoint at the first executable source unit of the routine.

When you create a breakpoint, CXdb responds by displaying the following information:

- `#0:` — The eventpoint number used to identify this eventpoint in other CXdb commands. In this case the eventpoint number is 0.
- `break instruction` — The type of eventpoint.
- `on [#0/*], Enabled, ignore 0/0` — The breakpoint is set on process object 0, for all threads (`*`). It is enabled and does not have an ignore count.
- `[0x80005506]` — The hexadecimal address location of the breakpoint. In this case the address is `80005506`.
- `PRINT_ARRAY in example.f line 35` — The symbolic location of the breakpoint. In this case the breakpoint is in the routine `PRINT_ARRAY` at line 35 of the source file `example.f`.

You can also display the above information by using the `info event` command.

When the breakpoint is triggered, execution is stopped before the instruction at that address is executed.

The syntax for specifying an absolute address is different between Fortran and C. The next two examples demonstrate this difference.

Using Fortran syntax:

```
(CXdb) break instruction '80001a32'x

#1: break instruction, on [#0/*], Enabled, ignore 0/0
[0x80001a32] SUB5A in chapter5.f line 21
```

The above command sets a breakpoint at the absolute address `80001a32`. The breakpoint number is 1, located at address `80001a32` in routine `SUB5A`, and corresponds to line 21 of the file `chapter5.f`. The notation `'80001a32'x` is Fortran-specific and indicates the address is in hexadecimal notation.

Using C syntax:

```
(CXdb) break instruction 0x80004fba

#2: break instruction, on [#0/*], Enabled, ignore 0/0
[0x80004fba] chapter13C`subxb in chapter13C.c line 64
```

The above command sets a breakpoint at the absolute address `80004fba`. The `0x` is the C notation for a hexadecimal number. The symbolic location uses the scope path of `chapter13c`subxb` to indicate the source file and routine in which the breakpoint is located.

When you specify an absolute address, the breakpoint is set at the closest even boundary. Because of this, you must be sure that the address is actually the starting address for the instruction. If the breakpoint is placed at an address in the middle of an instruction, it will be interpreted as a portion of the instruction, which can cause unpredictable results.

```
(CXdb) break instruction SUB5A {echo 'routine SUB5A reached'; resume;}

#3: break instruction, on [#0/*], Enabled, ignore 0/0
        [0x80001a32] SUB5A in chapter5.f line 21
    {
         echo 'routine SUB5A reached';
         resume;
    }
```

The above command sets a breakpoint at address `80001a32`, the starting address of routine `SUB5A`. The breakpoint is given its own eventpoint handler. When the breakpoint is triggered, execution is stopped, the `echo` command is executed, and finally, execution is resumed.

```
(CXdb) break instruction '80005694'x \; $Break4

#4: break instruction, on [#0/*], Enabled, ignore 0/0
[0x80005694] CLEAR_ARRAY in example.f line 50
```

The above command creates a new breakpoint at the absolute address `80005694`. The `\;` is needed to separate the language expression from the debugger variable. The debugger variable `$Break4` is created and set equal to the number of this eventpoint. In subsequent commands you can use `$Break4` to refer to this breakpoint. Debugger variables allow you to refer to eventpoints without having to remember their eventpoint numbers.

Related Commands

`break line`
`break routine`
`break source`
`event reached instruction`
`trace instruction`

Related Concepts

breakpoints
debugger variables
eventpoints
eventpoint handlers
tracepoints

Related Parameters

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

break line

bre l
bl

Set a breakpoint at a source line.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.
<line-specifier>	The line number where the breakpoint is to be set. The line number must be an integer, and may be preceded by a source file name.
<event-handler>	A sequence of CXdb commands enclosed within curly braces (`{}`). Each command must be terminated with a semicolon (`;`).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

The `break line` command sets a breakpoint before the first statement of the specified line.

If the line number does not map to a source line (whether due to optimizations or the line being a comment line), CXdb asks if you want the breakpoint set at the next highest line number that does map to a source line.

When the breakpoint is triggered, process execution stops, and the commands of the breakpoint's handler are executed. If the breakpoint does not have its own handler, the default handler for breakpoints, which displays a message, is executed. Unless the breakpoint handler includes the `resume` command, execution is not restarted.

Examples

The following examples set breakpoints at specific source lines.

```
(CXdb) break line 18

#0: break line, on [#0/*], Enabled, ignore 0/0
    0x8000545e] EXAMPLE in example.f line 18
```

The above command sets a breakpoint at the starting address that corresponds to line 18 of the current source file.

When you create a breakpoint, CXdb responds by displaying the following information:

- `#0:` — The eventpoint number used to identify this eventpoint in other CXdb commands. In this case the eventpoint number is 0.
- `break line` — The type of eventpoint.
- `on [#0/*], Enabled, ignore 0/0` — The breakpoint is set on process object 0, for all threads (`*`). It is enabled and does not have an ignore count.
- `[0x8000545e]` — The hexadecimal address location of the breakpoint. In this case the address is `8000545e`.
- `EXAMPLE in example.f line 18` — The symbolic location of the breakpoint. In this case the breakpoint is in the routine `EXAMPLE` at line 18 of the source file example.f.

You can also display the above information by using the `info event` command.

When the breakpoint is triggered, execution is stopped before the first instruction of the first statement on that line is executed.

```
(CXdb) break line example.f:31

#1: break line, on [#0/*], Enabled, ignore 0/0
       [0x800054d8] EXAMPLE in example.f line 31
```

The above command sets a breakpoint at the starting address of line 31 of the source file example.f. This source file must be included in the current search path and must be part of the compilation of the current executable file, compiled with the `-cxdb` option.

```
(CXdb) break line 35 {echo 'Line 35 reached'; resume;}

#2: break line, on [#0/*], Enabled, ignore 0/0
        [0x80005506] PRINT_ARRAY in example.f line 35
    {
        echo 'Line 35 reached';
        resume;
    }
```

The above command sets a breakpoint at the starting address of line 35 of the current source file. An eventpoint handler is defined for the breakpoint. When the breakpoint is triggered, process execution stops, and the commands of the eventpoint handler are executed. The first command displays a message and the second command resumes process execution.

```
(CXdb) break line 50 $Break3

#3: break line, on [#0/*], Enabled, ignore 0/0
        [0x80005604] CLEAR_ARRAY in example.f line 50
```

The above command creates a new breakpoint at the starting address of line 50 in the current source file. The debugger variable `$Break3` is created and set equal to the number of this eventpoint. In subsequent commands you can use `$Break3` to refer to this breakpoint. Debugger variables allow you to refer to eventpoints without having to remember their eventpoint numbers.

```
(CXdb) break line 30

ERROR A59: No source statements found.

Line 31 is the next line with object code. Use it? y

#4: break line, on [#0/*], Enabled, ignore 0/0
         [0x800054d8] EXAMPLE in example.f line 31
```

The above example attempts to set a breakpoint at a source line that does not contain any source units. CXdb responds by asking if you want to set the breakpoint at the next source line containing a source unit. By responding with a y, the breakpoint is set at line 31.

Related Commands

`break instruction`
`break routine`
`break source`
`event reached line`
`trace line`

Related Concepts

breakpoints
debugger variables
eventpoints
eventpoint handlers
tracepoints

Related Parameters

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

break routine

bre r
br

Set a breakpoint at the beginning of a routine.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.
<language-expression>	A valid language expression whose evaluation is used as the instruction address.
<event-handler>	A sequence of CXdb commands enclosed within curly braces ({}). Each command must be terminated with a semicolon (;).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

The `break routine` command sets a breakpoint at the first executable source unit of the routine containing the specified instruction address. If there are multiple entry points into the routine, a breakpoint is set at each entry point.

The specified address can be any valid language expression that evaluates to an address. CXdb finds the routine that contains this address and places the breakpoint at its first executable source unit. The first executable source unit is usually the first statement of a routine, unless there are local variable initializations.

When the breakpoint is triggered, process execution stops, and the commands of the breakpoint's handler are executed. If the breakpoint does not have its own handler, the default handler for breakpoints, which displays a message, is executed. Unless the breakpoint handler includes the `resume` command, execution is not restarted.

Examples

The following examples set breakpoints at the first executable source units of routines.

```
(CXdb) break routine BLD_MATRIX

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

The above command sets a breakpoint at the first executable source unit of the routine `BLD_MATRIX`.

When you create a breakpoint, CXdb responds by displaying the following information:

- `#0:` — The eventpoint number used to identify this eventpoint in other CXdb commands. In this case the eventpoint number is 0.
- `break routine` — The type of eventpoint.
- `on [#0/*], Enabled, ignore 0/0` — The breakpoint is set on process object 0, for all threads (`*`). It is enabled and does not have an ignore count.
- `[0x800029bc]` — The hexadecimal address location of the breakpoint. In this case the address is `800029bc`.
- `BLD_MATRIX in chapter7F.f line 26` — The symbolic location of the breakpoint. In this case the breakpoint is in the routine `BLD_MATRIX` at line 26 of the source file chapter7F.f.

You can also display the above information by using the `info event` command.

When the breakpoint is triggered, execution is stopped before the first source unit in the routine is executed.

The following two examples set a breakpoint at the start of a routine by specifying an absolute address inside of that routine. CXdb finds the routine containing the absolute address and places the breakpoint at the first source unit. The syntax for specifying an absolute address is different between Fortran and C.

Using Fortran syntax:

```
(CXdb) break routine '80001a36'

#1: break routine, on [#0/*], Enabled, ignore 0/0
        [0x80001a36] SUB5A in chapter5.f line 22
```

The above command sets a breakpoint at the starting address of the routine that contains the absolute address `80001a36`. The breakpoint number is 1, located at address `80001a36` in routine `SUB5A` at line 22 of the file chapter5.f. The notation `'80001a36'x` is Fortran-specific and indicates that the address is in hexadecimal notation.

Using C syntax:

```
(CXdb) break routine 0x80004c32

#2: break routine, on [#0/*], Enabled, ignore 0/0
        [0x80004c32] chapter13C`subxa in chapter13.c line 45
```

The above command sets a breakpoint at the starting address that contains the absolute address `80004c32`. The breakpoint number is 1, located at address `80004c32` in routine `subxa` in the source file chapter13.c at line 45. The notation `0x` is C-specific and indicates that the address is in hexadecimal notation.

```
(CXdb) break routine LEVEL_02 {echo 'routine LEVEL_02 reached'; resume;}

#3: break routine, on [#0/*], Enabled, ignore 0/0
        [0x80004328] LEVEL_02 in chapter15.f line 88
    {
        echo 'routine LEVEL_02 reached';
        resume;
    }
```

The above command sets a breakpoint at the address of the first executable source unit of the routine `LEVEL_02`. An eventpoint handler is defined for the breakpoint. When the breakpoint is triggered, execution is stopped, the `echo` command is executed, and finally, execution resumes.

```
(CXdb) break routine '800029bc'x \; $Break4

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

The above command creates a new breakpoint at the first executable source unit of the routine containing the absolute address `800029bc`. The `\;` is needed to separate the language expression from the debugger variable. The debugger variable `$Break4` is created and set equal to the number of this eventpoint. In subsequent commands you could use `$Break4` to refer to this breakpoint. Debugger variables allow you to refer to eventpoints without having to remember their eventpoint numbers.

Related Commands

`break instruction`
`break line`
`break source`
`event reached routine`
`trace routine`

Related Concepts

breakpoints
debugger variables
eventpoints
eventpoint handlers
tracepoints

Related Parameters

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

break source

bre s
bs

Set a breakpoint at a source unit.

Syntax

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

Parameter	Meaning
<*process-list*>	A list of processes affected by this command. The default is the current process.
<*thread-list*>	A list of threads affected by this command. The default is all threads of the specified process.
<*source-unit*>	The source unit number where the breakpoint is to be set. The source unit number must be an integer, and may be preceded by a source file name.
<*event-handler*>	A sequence of CXdb commands enclosed within curly braces ({}). Each command must be terminated with a semicolon (;).
<*debugger-variable*>	The debugger variable assigned to this eventpoint.

Description

The `break source` command sets a breakpoint at the specified source unit number.

Source units are numbered by CXdb. The number of a particular source unit can be determined by using the `info line` command. You can gather information about a source unit by using the `info sourceunit` command.

When the breakpoint is triggered, process execution stops, and the commands of the breakpoint's handler are executed. If the breakpoint does not have its own handler, the default handler for breakpoints, which displays a message, is executed. Unless the breakpoint handler includes the `resume` command, execution is not resumed.

Examples

The following examples set breakpoints at specific source units.

```
(CXdb) break source 35

#0: break source, on [#0/*], Enabled, ignore 0/0
        [0x80005482] EXAMPLE in example.f line 21
```

The above command sets a breakpoint at the start of source unit 35 of the current source file.

When you create a breakpoint, CXdb responds by displaying the following information:

- `#0:` — The eventpoint number used to identify this eventpoint in other CXdb commands. In this case the eventpoint number is 0.
- `break source` — The type of eventpoint.
- `on [#0/*], Enabled, ignore 0/0` — The breakpoint is set on process object 0, for all threads (`*`). It is enabled and does not have an ignore count.
- `[0x80005482]` — The hexadecimal address location of the breakpoint. In this case the address is `80005482`.
- `EXAMPLE in example.f line 21` — The symbolic location of the breakpoint. In this case the breakpoint is in the routine `EXAMPLE` at line 21 of the source file example.f.

You can also display the above information by using the `info event` command.

When the breakpoint is triggered, execution is stopped before the first instruction of the source unit is executed.

```
(CXdb) break source example.f:94

#1: break source, on [#0/*], Enabled, ignore 0/0
         [0x800056a0] CLEAR_ARRAY in example.f line 54
```

The above command sets a breakpoint at the starting address of source unit 94 of the source file example.f. This source file must be part of the compilation of the current executable file and be included in the search path of the process object.

```
(CXdb) break source 35 {echo 'Source unit 35 reached'; resume;}

#2: break source, on [#0/*], Enabled, ignore 0/0
        [0x80005482] EXAMPLE in example.f line 21
    {
        echo'Source unit 35 reached';
        resume;
    }
```

The above command sets a breakpoint at the starting address of source unit 35 of the current source file. An eventpoint handler is defined for this breakpoint. When the breakpoint is triggered, process execution stops, and the commands of the eventpoint handler are executed. The first command displays a message, and the second command resumes process execution.

```
(CXdb) break source 35 $Break3

#3: break source, on [#0/*], Enabled, ignore 0/0
        [0x80005482] EXAMPLE in example.f line 21
```

The above command sets a breakpoint at the starting address of source unit 35 in the current source file. The debugger variable `$Break3` is created and set equal to the number of this eventpoint. In subsequent commands you can use `$Break3` to refer to this breakpoint. Debugger variables allow you to refer to eventpoints without having to remember their eventpoint numbers.

Related Commands

`break instruction`
`break line`
`break routine`
`event reached source`
`info line`
`info sourceunit`
`trace source`

Related Concepts

breakpoints
debugger variables
eventpoints
eventpoint handlers
tracepoints

Related Parameters

debugger-variable
event-handler
source-unit
process-list
thread-list

cd

cd

Change the console working directory.

Syntax

cd *<directory-specifier>*

Parameter	Meaning
<directory-specifier>	The directory to become the console working directory.

Description

The `cd` command changes the console working directory to the specified directory. The console working directory is used as the base path name for relative path names in CXdb commands.

Examples

The following commands change the console working directory.

```
(CXdb) cd /usr/smith/programs
```

The above command changes the console working directory to be the /usr/smith/programs directory. Relative path names will now use this directory as the base path name.

```
(CXdb) cd ..
```

The above command changes the console working directory to the directory above its current setting. In the previous example, the console working directory was set to the /usr/smith/programs directory. Now the console working directory is set to the /usr/smith directory.

Related Commands

`info cxdb`
`info process`
`pwd`
`set directory`

Related Concepts

console working directory
process object
process working directory

Related Parameters

directory-specifier

clear autocreate

cl a

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

Syntax

```
clear autocreate
```

Description

The `clear autocreate` command disables the dynamic creation of Source Code windows. When this setting is disabled, CXdb cannot automatically create new Source Code windows. By default, the autocreate setting is enabled. If autocreate is disabled, it can be enabled again using the `set autocreate` command or by toggling the autocreate option in the CommandWindow menu of the Command window.

If the `clear autocreate` command is used in an initialization file or if you invoke CXdb from the shell with the `-ns` option, CXdb does not create a Source Code window at start up when invoked with the name of an executable file.

You can also disable autocreation by:

- Toggling the autocreate option of the CommandWindow menu, located on the main menubar of the CXdb Command window
- 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
```

This command is not available in line mode.

Examples

The following example illustrates how to disable automatic creation of Source Code windows.

```
(CXdb) clear autocreate
```

The above command disables the autocreate option.

Related Commands

`cxdb`
`display source`
`list`
`set autocreate`

Related Concepts

initialization files

Related Windows

Command window

clear default environment

cl d e

Remove all environment variables from the default environment.

Syntax

```
clear default environment
```

Description

The `clear default environment` command clears the default environment of all environment variables.

The default environment is passed to a new process if the process object does not have its own environment.

Examples

The following example clears the default environment.

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

The above command clears the default environment of all environment variables. This command can be included in an initialization file if you want to ensure that processes start out with empty environments.

Related Commands

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

Related Concepts

default environment
environment
process object

C Series only

clear default fixed sched

cldfs

Disable fixed scheduling in the default settings.

Syntax

```
clear default fixed sched
```

Description

The `clear default fixed sched` command disables fixed scheduling in the CXdb default settings. The CXdb default setting for fixed scheduling is used by new process objects that have not explicitly had their fixed scheduling set with the `set fixed sched` or `clear fixed sched` commands. This command does not affect existing processes.

Fixed scheduling means that the process requires the simultaneous use of all the processors on a given machine. With fixed scheduling enabled, the process does not begin executing until all the processors become available. With fixed scheduling disabled, the process executes on any processors that are available during a given time slice.

The default is fixed scheduling disabled. To display the default setting for fixed scheduling, use the `info cxdb` command.

NOTE: Because of the additional system overhead involved with fixed scheduling, it is recommended for debugging multithreaded processes only.

Examples

The following example shows how to disable fixed scheduling.

```
(CXdb) clear default fixed sched
```

The above command disables fixed scheduling in the CXdb defaults.

Related Commands

`clear fixed sched`
`info cxdb`
`info process`
`set default fixed sched`
`set fixed sched`

Related Concepts

optimized code
threads

clear default handler

cl d h

Clear the default handler for all eventpoints.

Syntax

```
clear default handler
```

Description

The `clear default handler` command removes the default handler for all eventpoints. Eventpoints that do not have their own handler, or a handler for their type, now use the initial default handler for eventpoints. The initial default handler displays a message when the eventpoint triggers.

A default handler must already have been specified using the `set default handler` command. Use the `info event` command to view default handlers that have been specified with `set default handler`.

Examples

The following example clears the default handler.

```
(CXdb) clear default handler
```

The above command removes the default handler for all eventpoints. The default handler returns to its initial setting, which displays a message.

Related Commands

`clear handler`
`clear typehandler`
`info event`
`info eventtype`
`set default handler`
`set handler`
`set typehandler`

Related Concepts

breakpoints
eventpoints
eventpoint handlers
tracepoints
watchpoints

Related Parameters

event-handler
event-specifier

C Series only

clear default remotewd

cl de re

Clear the default remote working directory.

Syntax

```
clear default remotewd
```

Description

The `clear default remotewd` command clears the default remote working directory. The default remote working directory is used if the remote working directory of the process object has not been explicitly set using the `set remotewd` command.

The remote working directory acts as the console working directory for a remote host during a remote debugging session. The remote working directory is used as:

- The base for relative path names specified on a remote host
- The directory for executing new processes on a remote host if a process working directory has not been specified for the process object using the `set directory` command

If the remote working directory is not set (by either the `set default remotewd` or `set remotewd` command), CXdb uses the console working directory as the remote working directory. For more information on the remote working directory, refer to the concepts page on "remote debugging."

Examples

The following example shows how to clear the default remote working directory.

```
(CXdb) clear default remotewd
```

The above command clears the default remote working directory.

Related Commands

`cd`	`core`
`debug core`	`debug exec`
`executable`	`info process`
`pwd`	`run`
`rerun`	`set default remotewd`
`set remotewd`	

Related Concepts

console working directory	process object
process working directory	remote debugging

clear echo

cl ec

Disable echoing of input from initialization files and command files.

Syntax

```
clear echo
```

Description

The `clear echo` command disables echoing of input to CXdb. With echoing disabled, input coming from initialization files and command files is not echoed at the CXdb command line.

By default echoing is disabled. You can enable echoing using the `set echo` command. To display the current setting of echoing, use the `info cxdb` command.

Examples

The following example disables echoing.

```
(CXdb) clear echo
```

The above command disables echoing of input. If the `source` command is used to execute a CXdb command file, input from that command file is not echoed at the CXdb command line.

Related Commands

`info cxdb`
`set echo`
`source`

Related Concepts

cmdlog
command files
initialization files
logging

clear environment

cl en

Remove all environment variables from the process environment.

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process object.

Description

The `clear environment` command clears the environment of the process object of all environment variables.

If the process object does not yet have its own environment, the `clear environment` command creates an empty environment for the process object.

Each new process will receive the modified environment. A process that is running will not be affected.

Examples

The following example clears the current process object of all environment variables. Assume that the current process object does not yet have an environment.

```
(CXdb) clear environment
```

The above example creates an empty environment for the current process object. The `clear environment` command indicates to CXdb that you want to modify the environment for this process object. CXdb creates an environment that is empty.

This command is useful if you want to ensure that new processes do not inherit any environment variables when they are created.

Related Commands

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

Related Concepts

default environment
environment
process object

Related Parameters

process-list

C Series only

clear fixed sched

cl f s

cfs

Disable fixed scheduling for the current process.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.

Description

The `clear fixed sched` command disables fixed scheduling for the current process. With fixed scheduling disabled, the process executes on any processors that are available during a given time slice.

Fixed scheduling means that the process requires the simultaneous use of all the processors on a given machine. With fixed scheduling enabled, the process does not begin executing until all the processors become available.

The default is fixed scheduling disabled. To display the current setting for fixed scheduling, use the `info process` command.

NOTE: Because of the additional system overhead involved with fixed scheduling, it is recommended for debugging multithreaded processes only.

Examples

The following example shows how to turn off fixed scheduling.

```
(CXdb) clear fixed sched
```

The above command disables fixed scheduling for the current process.

Related Commands

`clear default fixed sched`
`info cxdb`
`info process`
`set default fixed sched`
`set fixed sched`

Related Parameters

process-list

Related Concepts

optimized code
threads

clear handler

cl h

Clear the handler for a specified eventpoint.

Syntax

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

Parameter	Meaning
<event-specifier>	An eventpoint to remove the handler of.
[, ...]	An optional list of additional eventpoints. Multiple eventpoints must be separated by a comma.

Description

The `clear handler` command removes the handler of the specified eventpoints. The eventpoints return to using the default handler for their type. If a default handler has not been defined for an eventpoint type, the eventpoint returns to using the default handler for all eventpoints.

The specified eventpoints must already have been created and given handlers. An eventpoint may be given a handler when it is created or after it is created using the `set handler` command.

Examples

The following examples clear handlers for existing eventpoints.

```
(CXdb) clear handler 1
```

The above command removes the handler for eventpoint 1. The next time eventpoint 1 triggers, the default handler executes.

```
(CXdb) clear handler 0,2
```

The above command clears the handler for eventpoints 0 and 2. The comma is required to separate multiple eventpoints.

Related Commands

`clear default handler`
`clear typehandler`
`info event`
`info eventtype`
`set default handler`
`set handler`
`set typehandler`

Related Concepts

breakpoints
eventpoints
eventpoint handlers
tracepoints
watchpoints

Related Parameters

event-specifier

clear logging
cl l

Disable logging of CXdb input to viewport files.

Syntax

```
clear logging
```

Description

The `clear logging` command disables logging to the viewport files for cmdlog.

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 equivalent to stdin in the shell.

When logging is enabled for cmdlog, everything you enter on the CXdb command line is also sent to the viewports of cmdlog. When logging is disabled, nothing is sent to the viewports of cmdlog.

The default is logging disabled. To enable logging, use the `set logging` command.

To display the current setting of the logging option, use the command `info cxdb`.

Examples

The following example illustrates how to disable logging.

```
(CXdb) clear logging
```

The above command disables logging of CXdb input to the viewport files.

Related Commands

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

Related Concepts

cmdlog
logging
viewports

clear noclobber

cl n

Disable the noclobber option for all viewport files.

Syntax

```
clear noclobber
```

Description

The `clear noclobber` command disables the noclobber option, which protects viewport files that are used for logging or redirection operations.

When noclobber is enabled, CXdb responds with an error if it tries to overwrite an existing viewport file or append to a viewport file that does not exist. When noclobber is disabled, CXdb may overwrite existing viewport files and create new files for appending.

The noclobber option applies to all files specified as viewports with the redirection operators or with the following commands:

```
add cmderr
add cmdlog
add cmdout
set cmderr
set cmdlog
set cmdout
```

The default is noclobber disabled (clear). To display the current setting of the noclobber option, use the command `info cxdb`.

Examples

The following example illustrates how to clear the noclobber option.

```
(CXdb) clear noclobber
```

The above command disables the noclobber option for all cmderr, cmdlog, and cmdout viewport files.

Related Commands

`add cmderr`
`add cmdlog`
`add cmdout`
`clear logging`
`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

C Series only

clear seq

cl se

Clear the sequential mode (SEQ) bit.

Syntax

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

Parameter	Meaning
<*process-list*>	A list of processes affected by this command. The default is the current process.
<*thread-list*>	A list of threads affected by this command. The default is all threads of the current process.

Description

The `clear seq` command clears the sequential mode (SEQ) bit of the processor status word (PSW) register.

The SEQ bit controls pipelining within the processor. If this bit is clear, the processor operates with maximum pipelining and overlap. If this bit is set, the processor executes all instructions sequentially: that is, the execution of the next instruction is initiated only after the previous instruction has been executed. The default is SEQ set.

For more information about the PSW and the SEQ bit, refer to *Convex C-Series Architecture* (DSW-300).

Examples

The following example illustrates how to clear the SEQ bit.

```
(CXdb) clear seq
```

The above command clears the SEQ bit for all threads of the current process.

Related Commands

`clear sqs`
`info psw`
`set seq`
`set sqs`

Related Parameters

process-list
thread-list

C Series only

clear sqs

cl sq

Clear the sequential store enable (SQS) bit.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the current process.

Description

The `clear sqs` command clears the sequential store enable (SQS) bit of the processor status word (PSW) register.

If the SQS bit is clear, stores to memory may occur in nonsequential order. If this bit is set, all stores to memory occur in instruction execution order. The default is SQS set.

For more information about the PSW and the SQS bit, refer to *Convex C-Series Architecture* (DSW-300).

Examples

The following example illustrates how to clear the SQS bit.

```
(CXdb) clear sqs
```

The above command clears the SQS bit for all threads of the current process.

Related Commands

`clear seq`
`info psw`
`set seq`
`set sqs`

Related Parameters

process-list
thread-list

clear step

cl st

Reset the stepping granularity to the default setting.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.

Description

The `clear step` command resets the stepping granularity (or step size) of the specified process to match the current setting of the CXdb default stepping granularity. If the CXdb default granularity is later changed with the `set default step` command, then the default granularity of the specified process also changes.

To display the default stepping granularity of a particular process, use the `info process` command. To display the CXdb default stepping granularity, use the `info cxdb` command.

Examples

The following example illustrates how to reset the default granularity for the stepping commands.

```
(CXdb) clear step
```

The above command resets the step size to the current value of the CXdb default stepping granularity. This command applies to all threads of the current process.

Related Commands

`finish`
`info process`
`next over`
`set step`
`step over`
`info cxdb`
`next`
`set default step`
`step`

Related Concepts

granularity
stepping
source units

Related Parameters

process-list

clear typehandler

cl t

Clear the handler for a specified type of eventpoint.

Syntax

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

Parameter	Meaning

<eventtype-specifier>	The type of eventpoint. Possible eventtypes are as follows:
	`break`
	`trace`
	`watch`
	`exec`
	`join`
	`modify`
	`reached`
	`relation`
	`signal`
	`spawn`
	`*` (all)

[, ...]	An optional list of additional eventtypes. Multiple eventtypes must be separated by a comma.

Description

The `clear typehandler` command removes the handler of the specified eventpoint types. Eventpoints of the specified type that do not have their own handler now use the default handler for eventpoints.

A handler must already have been specified for the given type. A handler may be given to an eventpoint type with the `set typehandler` command.

Examples

The following examples clear handlers for existing eventpoint types.

```
(CXdb) clear typehandler break
```

The above command removes the handler for breakpoints. If a breakpoint without a handler triggers, the default handler for eventpoint executes.

```
(CXdb) clear typehandler trace, watch
```

The above command clears the handler for tracepoints and watchpoints.

Related Commands

`clear default handler`
`clear handler`
`info event`
`info eventtype`
`set default handler`
`set handler`
`set typehandler`

Related Concepts

breakpoints
eventpoints
eventpoint handlers
tracepoints
watchpoints

Related Parameters

eventtype-specifier

continue

con

c

Continue execution of the process.

Syntax

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

Parameter	Meaning
<*process-list*>	A list of processes affected by this command. The default is the current process.
<*thread-list*>	A list of threads affected by this command. The default is all threads of the specified process.
<*count*>	The number of times to continue execution after execution has been stopped by an eventpoint.
&	Runs the command in the background.

Description

The `continue` command continues execution of a stopped process or stopped threads of a process.

The process must have already been created with the `run` or `rerun` command or have been brought under the control of CXdb with the `attach` command. Images from core files can not be continued.

Execution continues from the current program counter (PC). Process execution continues until the process terminates or is stopped. The process can be stopped by an eventpoint, the receipt of a signal, or by typing **CTRL-c** in the Command window.

Examples

The following examples illustrate how to continue process execution.

```
(CXdb) continue
Resuming execution of Process [#0/*]
```

The above command continues execution of all threads of the current process. Process execution continues until the process terminates or is stopped.

```
(CXdb) :T1 continue
Resuming execution of Process [#0/1]
```

The above command continues the execution of thread 1 only of the current process. Other threads of the current process remain stopped. The notation `[#0/1]` indicates that only execution of thread 1 of process 0 is resumed.

```
(CXdb) continue &
Command [#7] backgrounded

Resuming execution of Process [#0/*]
```

The above command continues execution of all threads of the current process. The command is run in the background. This causes the CXdb command prompt to return, allowing you to enter other CXdb commands that do not require the process to be stopped.

Related Commands

`attach`	`core`
`debug core`	`debug exec`
`debug proc`	`detach`
`executable`	`info cxdb`
`info process`	`kill process`
`rerun`	`resume`
`run`	`signal process`
`signal thread`	`stop`

Related Concepts

background execution
process object
process working directory

Related Parameters

process-list
thread-list

C Series only

copy
cop

Copy a memory region.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.
<source-address>	The starting address of the memory region to be copied. This can be any *<language-expression>* that evaluates to a valid address.
<ending-address>	The ending address of the memory region to be copied. This can be any *<language-expression>* that evaluates to a valid address.
<byte-count>	The number of bytes in the memory region to be copied. This can be any *<language-expression>* that evaluates to a positive integer. The default count is all bytes of the memory region specified by the source address.
\;	The language expression terminator.
<destination-address>	The starting address of the memory region that receives the copied data. This can be any *<language-expression>* that evaluates to a valid address.

Description

The `copy` command copies the contents of one memory region into another memory region.

Caution

If you do not specify the memory region properly with this command, it could result in overwriting unprotected areas of process memory that you do not want to change.

Examples

The following examples illustrate how to copy one region of memory to another.

```
(CXdb) copy ARRAY \; TABLE
```

The above command copies the contents of `ARRAY` into `TABLE`. Both arrays are in the current process. Since the command does not specify the number of bytes to copy, all bytes of `ARRAY` are copied. If `TABLE` is not the proper size or type to accommodate the copy, errors might result. The delimiter (`\;`) is required between the language expressions for the source and destination addresses.

```
(CXdb) copy ARRAY:40 \; TABLE
```

The above command copies the first `40` bytes of `ARRAY` into `TABLE`. If each element of `ARRAY` is one word (four bytes) long, then this is equivalent to copying the first 10 elements of `ARRAY` into `TABLE`.

```
(CXdb) copy '8008ace8'x..'8008ad34'x \; '80077058'x
```

The above command copies everything in the region from address `8008ace8` to `8008ad34` into the region starting at address `80077058`.

Related Commands

`disassemble`
`examine`
`fill`
`info expression`
`print`

Related Parameters

language-expression
process-list
thread-list

Related Concepts

modifying data

core

cor

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

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process object.
<remote-host>	The name of the remote host (C Series only). The name can be either an absolute Internet address or a name in the /etc/hosts file.
<file-name>	The name of the core file. Relative path names on the local host use the console working directory as a base.

Description

The `core` command retrieves the image from the specified core file. The `core` command can also retrieve images from checkpoint files.

The core image becomes the image being debugged, replacing any existing core or process image in the process object. The process object must already have been created using one of the three `debug` commands.

You can use the image of the core file to examine the state of a process that was abnormally terminated when an exception occurred. You can examine the contents of the core image using absolute addresses. If you specify the executable file that created the process, you can debug the process image with global and static symbols. If the executable file was compiled with the `-cxdb` option, then you can debug the process symbolically. Because the image is from a terminated process, you cannot use any process execution commands on the image (such as `step`).

If you are already debugging a process image, you must kill that process before you can use the `core` command. To kill an existing process, use the `kill process` command.

NOTE: On SPP Series machines, you must specify an executable file in order to perform any debugging on the core image. Use the `executable` command to specify the executable file.

A remote core file can be specified by preceding the core file with the name of the remote host (C Series only), a colon, and the path to the core file. Relative path names on a remote host use the remote working directory as a base. For more information, refer to the concepts page on "remote debugging."

Examples

The following example retrieves an image from a core file.

```
(CXdb) core core.para.25802
Core file from: para
thread 0 received signal 9 (Killed)
thread 1 received signal 5 (Trace/BPT trap)

Process [#0/0] stopped at [0x80001672] SUB1 in para.f line 22
Process [#0/1] stopped execution at [0x8000162c] SUB1 in para.f
       line 21
```

The above command retrieves the core image from the core file. The core image becomes the image in the process object. You can now examine the state of the terminated process when the exception occurred.

Related Commands

attach	debug core
debug exec	debug proc
detach	executable
info cxdb	info process
kill process	run
rerun	

Related Concepts

process object

remote debugging

Related Parameters

process-list

file-name

csd

csd

Enable compatibility with csd debugger commands.

Syntax

```
csd
```

Description

The csd command incorporates a set of predefined aliases for csd debugger commands. With the predefined aliases incorporated, you can type in a csd debugger 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.

You can also invoke CXdb from the command line with the -csd option. This automatically puts CXdb in line mode and incorporates the csd aliases.

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	No 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

csd command	CXdb equivalent

`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` commands. (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`
`thread`	`info process`
`treads false`	`remove eventtype spawn, join`
`threads true`	`event spawn; event join`

csd command	CXdb equivalent
`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`

Examples

The following example illustrates how to incorporate the predefined aliases for `csd` debugger commands.

```
(CXdb) csd
```

After executing the above command, you can enter `csd` debugger commands directly in the CXdb Command window.

To display a list of current CXdb aliases, use the `info alias` command.

Related Commands

`cxdb`
`gdb`
`info alias`

Related Concepts

csd debugger
gdb debugger

cxdb

cxdb

Invoke CXdb from the shell.

Syntax

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

Parameter	Meaning
-a *<process-id>*	Attaches CXdb to the process with the specified process ID. The image of the process is associated with the created process object. If an executable file has not yet been specified on the command line (using the `-e` option), then the `-a` option performs the `debug proc` command. If an executable file has been specified, then the `-a` option performs the `attach` command. You cannot use the `-a` option and the `-c` option (which specifies a core file) together.
-c *<file-name>*	Specifies a core file to debug. The image from the core file is associated with the created process object. If an executable file has not yet been specified on the command line (using the `-e` option), then the `-c` option performs the `debug core` command. If an executable file has been specified, then the `-c` option performs the `core` command. You cannot use the `-c` option and the `-a` option (which attaches CXdb to a process) together. The `-c` prefix is not required. If you omit it, then the second file name that is not preceded by an option flag is assumed to be the core file.
+core	Enables generation of CXdb or application core images from processes under CXdb control. By default CXdb does not generate core images when it terminates abnormally.

-csd
Invokes CXdb in line mode (same as the `-nw` option) and automatically incorporates aliases for `csd` debugger commands.

-D *<directory-specifier>*
Specifies a directory to be added to the default search path. This option performs the `add default path` command. You can use the `-D` option multiple times on the command line, once for each specified directory.

-e *<file-name>*
Specifies an executable file to debug. The executable file and any associated CTI data files, are associated with the created process object. If an image has not yet been specified on the command line (with the `-a` or `-c` options), then the `-e` option performs the `debug exec` command. If an image has been specified, then the `-e` option performs the `executable` command. The `-e` prefix is not required. If you omit it, then the first file name that is not preceded by an option flag is assumed to be the executable file.

-F (C Series only)
Enables fixed scheduling. This option performs the `set default fixed sched` command. Use this option when debugging multithreaded programs.

-f *<file-name>*
Executes the specified command file immediately after any default initialization files. This option performs the `source` command. You can use the `-f` option multiple times on the command line, once for each specified command file.

-ns
Disables the dynamic creation of Source Code windows. If used in an initialization file, this option prevents CXdb from creating a Source Code window at start-up when invoked with an executable file.

-nw
Invokes CXdb in line mode. Line mode allows you to enter CXdb commands interactively through command-line editing, without invoking any of the CXdb windows. In line mode, echoing of command files and initialization files is disabled by default.

-nx
Prevents CXdb from executing initialization files.

-x *<command-list>*	Specifies a list of CXdb commands that are executed upon start-up. Use a semicolon (;) to separate multiple commands in the list. If the list contains any spaces or semicolons, the entire list must be delimited with quotes. You can use the `-x` option multiple times on the command line.
<X-Toolkit-options>	Specifies an X Toolkit option. For more information about toolkit options, refer to your X Windows documentation.

Description

The `cxdb` command invokes CXdb from the shell prompt.

Parameters can be specified on the shell command line that enable you to:

- Attach CXdb to a process
- Run CXdb in line mode
- Execute `csd` debugger commands within CXdb
- Add directories to the default search path
- Prevent initialization file processing
- Debug an executable file
- Debug a core file
- Set fixed scheduling
- Execute CXdb commands upon start-up
- Source a command file
- Specify X flags (such as window geometries)

When CXdb is invoked, initialization files are executed first (unless you have used the `-nx` option). Then the options on the command line are executed in the order that they appear.

Examples

To invoke CXdb without any options, simply type the following at the shell prompt:

```
% cxdb
```

The above shell command invokes CXdb without any options. CXdb executes all initialization files found, starting with the .cxdbinit file in the /usr/lib/cxdb directory, then executing any .cxdbinit files found in your home directory, and finally the directory from which you invoke CXdb. After executing all initialization files, the Command window appears.

```
% cxdb -a 26435
```

The above command invokes CXdb. The `-a` option performs the `debug proc` command because an executable file is not specified. CXdb creates a process object and attaches to the process with a process ID of `26435` (obtained by using the `ps` shell command). The process object does not yet have an executable file associated with it.

```
% cxdb -a 26435 -e a.out
```

The above command again invokes CXdb. Because it was specified first, the `-a` option performs the `debug proc` command and creates a process object. The `-e` option performs the `executable` command and specifies an executable file for the newly created process object.

```
% cxdb -csd
CXdb version 3.0, Copyright (C) 1992, Convex Computer Corp.
(CXdb) clear echo
(CXdb) csd
(CXdb)
```

The above command invokes CXdb in line mode and automatically incorporates a set of predefined aliases for `csd` debugger commands. CXdb first executes the `clear echo` command to disable echoing of initialization files and command files. (To enable echoing again, use the `set echo` command.) Next CXdb executes the `csd` command to incorporate the predefined `csd` aliases. You can then enter many of the `csd` commands directly and they will be translated into equivalent CXdb commands. Refer to the `csd` command description for a list of the `csd` commands supported by CXdb.

```
% cxdb -D /mnt/jones -D /mnt/projects/smith
```

The above command invokes CXdb with the `-D` option. The `-D` option performs the `add default path` command and adds two directories to the default search path. Each newly created process object inherits these two directories as part of its search path.

```
% cxdb -nw
CXdb version 3.0, Copyright (C) 1992, Convex Computer Corp.
(CXdb) clear echo
(CXdb)
```

The above command invokes CXdb in line mode. CXdb first executes the `clear echo` command to disable echoing of initialization files and command files. (To enable echoing again, use the `set echo` command.) Next CXdb executes any initialization files, then it issues the `(CXdb)` prompt and waits for you to enter additional commands. In line mode, all output from CXdb goes to your standard output (STDOUT) and standard error (STDERR), unless you redirect it.

```
% cxdb -nx
```

The above command invokes CXdb but inhibits it from processing any initialization files.

```
% cxdb -x "alias l 'step loop'; alias r 'step routine'"
```

The above command invokes CXdb. After the Command window appears, the two `alias` commands are executed. The string of commands is delimited by double quotes. Note that the alias definitions must also be quoted because they contain spaces. To prevent the quotes for the alias commands from being treated as delimiters for the `-x` option, single quotes are used in the alias definitions.

```
% cxdb docexample
```

The above command invokes CXdb. The file name is taken to be an executable file, and CXdb performs the `debug exec` command. This creates a process object containing the executable file `docexample`. Any CTI data files for the executable file are associated with the process object.

```
% cxdb -c core.para.25802
```

The above command invokes CXdb. The `-c` option performs the `debug core` command and creates a process object containing the image retrieved from the `core.para.25802` file. Because no executable file was specified, the process object does not yet have an executable file.

```
% cxdb para core.para.25802
```

The above command invokes CXdb. The first file name is assumed to be an executable file, and CXdb performs the `debug exec` command. The second file name is assumed to be a core file, and CXdb performs the `core` command. The created process object has an executable file `para` and the image from the `core.para.25802` file.

Related Commands

`add default path`
`attach`
`core`
`debug core`
`debug exec`
`debug proc`
`executable`
`set fixed sched`
`source`

Related Concepts

command files
csd debugger
default search path
initialization files
process object
Xdefaults

Related Parameters

directory-specifier
file-name

Related Windows

Command window

debug core

deb c

dbgc

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

Syntax

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

Parameter	Meaning
<*core-file*>	The name of a core file. Relative path names use the console working directory as a base.
<*remote-host*>	The name of a remote host (C Series only). The name can be either an absolute Internet address or a name in the /etc/hosts file.
<*executable-file*>	The name of an executable file to provide the compiler-generated data for debugging the core image.
<*debugger-variable*>	The debugger variable for this process object.

Description

The `debug core` command creates a process object and specifies a core or checkpoint file to debug. The core image retrieved from the core file becomes the image of the process object.

When the `debug core` command is issued, CXdb performs two actions:

- Creates a process object.
- Retrieves the core image from the core file. This core image can be debugged using absolute addresses and global symbols.

You can always debug a core image using absolute addresses. To debug it symbolically, you must specify the name of the executable file that created the core file. You can specify the executable file with either the `executable` command or the `executable` option of the `debug core` command.

NOTE: On SPP Series machines, you must specify an executable file in order to perform any debugging on the core image.

You can use the image of the core file to examine the state of a process that was abnormally terminated when an exception occurred. Because the image is from a terminated process, you cannot use any process execution commands (such as `step`) on the image.

debug core

A remote core file can be specified by preceding the file name with the name of the remote host (C Series only), a colon, and the path to the core file. Relative path names use the remote working directory as a base. For more information, refer to the concepts page on "remote debugging."

To change core files during a debugging session, use the `core` command.

You can create a debugger variable for the process object. Future references to the process object can then use the debugger variable instead of the process object number.

The `debug core` command has the same effect as using the `-c` option by itself when invoking CXdb from the shell prompt.

Examples

The following example creates a process object with the image of a core file.

```
(CXdb) debug core corefile
Core file from: para
thread 0 received signal 9 (Killed)
thread 1 received signal 5 (Trace/BPT trap)

Process [#0/0] stopped at [0x80001672] SUB1 in para.f line 22
Process [#0/1] stopped execution at [0x8000162c] SUB1 in para.f
       line 21

Process [#0] created
```

This command creates a process object in CXdb. The process object consists of the image found in the core file. The image does not have an executable file or any compiler-generated data files associated with it. However, you can still examine the state of the process using absolute addresses.

Related Commands

`attach`	`core`
`debug exec`	`debug proc`
`detach`	`executable`
`info cxdb`	`info process`
`run`	`rerun`

Related Concepts

debugger variables	process object
remote debugging	

Related Parameters

debugger-variable	file-name

debug exec

deb e
dbg

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

Syntax

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

Parameter	Meaning
<remote-host>	The name of a remote host (C Series only). The name can either be an absolute Internet address or a name in the /etc/hosts file.
<executable-file>	The name of an executable file.
<debugger-variable>	The debugger variable for the process object.

Description

The `debug exec` command creates a process object and specifies an executable file. The executable file provides an executable image to debug, and it is the basis for CTI information in the process object.

When the `debug exec` command is issued, CXdb performs three actions:

- Creates a process object.
- Uses the executable file as the basis for the CTI information in the process object. This provides the debugging information needed to symbolically debug an image. The CTI information also consists of the locations of the CTI data files and source files specified in the executable file.
- Creates an executable image from the executable file. This becomes the image being debugged.

You can use an executable image to look at assembly-language code or global and static symbols. The executable image is also used to create a new process (with the `run` command).

To specify a different executable file during a debugging session, use the `executable` command.

A remote executable can be specified by preceding the executable name with the remote machine name (C Series only), a colon, and the path to the executable file. Relative path names use the remote working directory as a base. For more information, refer to the "remote debugging" concepts page.

If the executable file is on the local host, the directory in which CXdb finds the executable file is added to the search path of the process object.

CXdb uses the search path to find the CTI data files and source files specified in the executable file. The CTI data files exist only if the executable file was compiled with the `-cxdb` option of the CONVEX Fortran or C compilers. The CTI data files are located in the .CTI directory, which must be within the search path.

You can create a debugger variable for the process object. Future references to the process object can then use the debugger-variable name instead of the process object number.

The `debug exec` command has the same effect as using the `-e` parameter by itself when invoking CXdb from the shell prompt.

Examples

The following examples each create a process object with an executable file.

```
(CXdb) debug exec docexample

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

Process [#0] created
```

This command creates a process object in CXdb. The process object consists of information found in the executable file named `docexample`, which is located in the search path. If data files for this executable file exist in the .CTI directory, then these files are mapped to the executable file.

```
(CXdb) debug exec pixel:/doc/cxdb/examples/docexample

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

Process [#0] created
```

The above command creates a process object and uses the executable file located on a remote C Series host named `pixel` as the basis for the CTI information in the process object. An executable image is also created from this executable file and is now the image being debugged.

From this executable image, the `run` command would now create a new process on the remote host named `pixel`.

Related Commands

`attach`
`core`
`debug core`
`debug proc`
`detach`
`executable`
`info cxdb`
`info process`
`kill process`
`run`
`rerun`

Related Concepts

compiling for CXdb
debugger variables
process object
remote debugging

Related Parameters

debugger-variable

debug proc

deb p
dbgp

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

Syntax

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

Parameter	Meaning
<remote-host>	The name of a remote host (C Series only). The name can be an absolute Internet address or a name in the /etc/hosts file.
<process-id>	The process ID of the process you wish to attach to.
<executable-file>	The name of an executable file to provide the CTI base for debugging the process image.
<debugger-variable>	The debugger variable for the process object.

Description

The `debug proc` command creates a process object and specifies a running process to debug. The process image of the attached process is incorporated into the process object.

When the `debug proc` command is issued, CXdb performs three actions:

- Creates a process object
- Attaches to the specified process, brings it under CXdb control, and stops it
- Makes the process image of the attached process the image being debugged

You can always debug a process image using absolute addresses. To debug it symbolically, you must specify the associated executable file with either the `executable` command or the `executable` option of the `debug proc` command.

A remote process can be specified by preceding the process ID with the remote host name (C Series only) and a colon.

To attach to a different process during a debugging session, use the `attach` command.

You can create a debugger variable for the process object. Future references to the process object can then use the debugger variable name instead of the process object number.

The `debug proc` command has the same effect as using the `-a` parameter when invoking CXdb from the shell prompt.

Examples

The following examples each create a process object with the image from a process.

```
(CXdb) debug proc 12345
Process [#0] created
Attaching Process [#0] to pid 12345
Process [#0/0] stopped by attach at 0x80001402
```

This command creates a process object. CXdb attaches to the process and stops it. The process image is incorporated into the process object. The process object does not yet have any CTI information. You can specify the basis for CTI information by using the `executable` command.

```
(CXdb) debug proc pixel:23456
Process [#0] created
Attaching Process [#0] to pid 23456
Process [#0/0] stopped by attach at 0x80001402
```

The above command creates a process object in CXdb, then attaches to the remote process whose process ID is `23456` on the C Series host `pixel`. The process image of this process is now the image being debugged.

Related Commands

`attach`
`core`
`debug core`
`debug exec`
`detach`
`executable`
`info cxdb`
`info process`
`run`
`rerun`

Related Concepts

debugger variables
process object
remote debugging

Related Parameters

debugger-variable

detach

det

Detach from a process.

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process object.

Description

The `detach` command detaches CXdb from a process.

When you detach CXdb from a process, the image of the process is removed from the process object. Everything else in the process object remains intact. A process must be stopped in order to detach CXdb from it.

NOTE: If you have altered the flow of execution, detaching from the process may leave the process in an unstable state.

Examples

The following example detaches CXdb from a process.

```
(CXdb) detach
```

The above command detaches CXdb from the process of the current process object. The process is left running outside the control of CXdb. Another process can be attached, or, if an executable file has been specified, created.

Related Commands

`attach`
`core`
`debug core`
`debug exec`
`debug proc`
`executable`
`info cxdb`
`info process`
`run`
`rerun`

Related Concepts process object

Related Parameters process-list

dirpath
dir

Specify an alias for the directory path to the CTI data files.

Syntax

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

Parameter	Meaning
<original-directory>	The directory where the program object files were originally compiled. The directory path name must begin with root (/). Do not include the .CTI subdirectory as part of the path name.
<alternate-directory>	The directory where the CTI data files now reside. The path name does not have to begin with root (/). Do not include the .CTI subdirectory as part of the path name.

Description

The `dirpath` command creates an alias, or alternate path name, for the original CTI (Compiler-Tools Interface) directory path. Use the `dirpath` command only if you have moved the CTI data files from their original location or if your directory structure has changed.

NOTE: The `dirpath` command performs a global string substitution on all CTI directory path names. To change individual CTI directory paths, use the `add path`, `remove path`, and `set path` commands.

When you compile your source code with the `-cxdb` option, the compiler generates several CTI data files that provide CXdb with symbolic debugging information about your program. The compiler places these CTI data files in the same directory as the object (.o) files, under a subdirectory named .CTI. The compiler also permanently hard-codes this directory path name into the executable file for your program. If you move some or all of the CTI data files from their original directory, then you can use the `dirpath` command to tell CXdb the new path to these files.

The .CTI subdirectory must be the last branch in any CTI directory path. Therefore, the new directory where you relocate the CTI data files must have .CTI as its last branch. Because CXdb assumes that this subdirectory exists, you should not specify .CTI as part of the *<original-directory>* or *<alternate-directory>* path names in the `dirpath` command.

If you move the CTI data files to several different directories, you can execute the `dirpath` command multiple times to create a list of aliases for the CTI directory paths. To display this list, use the `info dirpath` command.

When you debug your program, CXdb searches for the associated CTI data files in the following order:

- The original CTI directory (hard-coded into the executable file)
- Alternate CTI directories specified with the `dirpath` command
- Directories specified in the search path (either by means of the default search path or with the `add path` and `set path` commands)

A directory path alias specified with the `dirpath` command remains in effect during the entire debugging session, unless you explicitly delete it with the `remove dirpath` command. If you want to use the same directory path aliases in future debugging sessions, you can put the appropriate `dirpath` commands in a CXdb command file or initialization file.

Examples

The following example creates an alias for the directory path to the CTI data files.

```
(CXdb) dirpath /usr/jones /usr/smith
```

The above command specifies /usr/smith as an alias for the directory path /usr/jones. CXdb then uses this alias to search the directory /usr/smith/.CTI for the CTI data files that were originally generated in the directory /usr/jones/.CTI. Note that this alias also applies to other paths that begin with /usr/jones. For example, CXdb would also search the directory /usr/smith/devl/proj2/.CTI for the CTI data files that were originally generated in the directory /usr/jones/devl/proj2/.CTI.

Related Commands

`add path`
`add default path`
`info dirpath`
`remove dirpath`
`set path`

Related Concepts

command files
Compiler-Tools Interface
initialization files
search path

disable event

disab event

dis

Disable eventpoints.

Syntax

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

Parameter	Meaning
<event-specifier>	A list of eventpoints to be disabled. The asterisk (*) is used to specify all eventpoints.
`[,` ...**`]`	An optional list of eventpoints. Multiple eventpoints are separated by commas.

Description

The `disable event` command disables all specified eventpoints.

CXdb treats disabled eventpoints as if they did not exist. However, they are not removed from the process object. A disabled eventpoint can be enabled again with the `enable event` command. All types of eventpoints can be disabled and enabled.

A disabled eventpoint is never reached. Because a disabled eventpoint will never be reached, its ignore count will not be incremented and it will not be triggered.

A disabled eventpoint can be affected by CXdb commands that affect eventpoints, such as the `remove event`, `set handler` and `set ignore` commands.

Examples

The following examples disable the specified eventpoints.

```
(CXdb) disable event 2
Eventpoint 2 disabled
```

The above command disables eventpoint 2. Eventpoint 2 can no longer be triggered. The eventpoint remains disabled until it is enabled or it is removed from the process object.

```
(CXdb) disable event 1,3
Eventpoint 1 disabled
Eventpoint 3 disabled
```

The above command disables eventpoints 1 and 3. Neither eventpoint can be triggered.

```
(CXdb) disable event *
Eventpoint 0 disabled
Eventpoint 4 disabled
Eventpoint 5 disabled

INFO 373: Event 1 is already disabled

INFO 373: Event 2 is already disabled

INFO 373: Event 3 is already disabled
```

The above command uses the asterisk to disable all existing eventpoints. CXdb displays all eventpoints that are disabled.

Related Commands

`disable eventtype`	`enable event`
`enable eventtype`	`info event`
`info eventtype`	`remove event`
`remove eventtype`	`set default handler`
`set handler`	`set ignore`
`set typehandler`	

Related Concepts

breakpoints	eventpoints
eventpoint handlers	tracepoints
watchpoints	

Related Parameters

event-specifier

disable eventtype

disab eventt

Disable all eventpoints of the specified type.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<eventtype-specifier>	A list of eventpoint types whose eventpoints are to be disabled. The asterisk (*) specifies all eventpoint types.
[**,** ...]	An optional list of additional eventpoint types. Multiple eventpoint types are separated by commas.

Description

The `disable eventtype` command disables all existing eventpoints of the specified type.

The following is a list of eventpoint types:

```
break
trace
watch
exec
join
modify
reached
relation
signal
spawn
```

CXdb treats disabled eventpoints as if they did not exist. However, they are not removed from the process object. The disabled eventpoints can be enabled again with the `enable event` or `enable eventtype` command. All eventpoint types can be disabled and enabled.

A disabled eventpoint can never be reached. The ignore count of a disabled eventpoint cannot be incremented by the process reaching it. Disabled eventpoints cannot be triggered.

Disabled eventpoints can be affected by any of the CXdb commands that affect eventpoints, such as the `remove event`, `set typehandler` and `set ignore` commands.

The `disable eventtype` command disables only existing eventpoints. New eventpoints of the specified type are not disabled.

Examples

The following examples disable the eventpoints of eventpoint types.

```
(CXdb) disable eventtype watch, break
Eventpoint 2 disabled
Eventpoint 1 disabled
```

The above command disables all watchpoints and breakpoints. CXdb displays all eventpoints that are disabled.

```
(CXdb) disable eventtype *

Eventpoint 3 disabled
Eventpoint 0 disabled

INFO 373: Event 2 is already disabled

INFO 373: Event 1 is already disabled
```

The above command disables all existing eventpoints, regardless of type. CXdb displays all eventpoints that are disabled.

Related Commands

`disable event`
`enable eventtype`
`info eventtype`
`remove eventtype`
`set handler`
`set typehandler`
`enable event`
`info event`
`remove event`
`set default handler`
`set ignore`

Related Concepts

breakpoints
eventpoint handlers
watchpoints
eventpoints
tracepoints

Related Parameters

eventtype-specifier
process-list

disassemble

disas

Display the assembly language code.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the current process.
<starting-address>	The first address to disassemble. This can be any *<language-expression>* that evaluates to an address in the syntax of the current source language. The default is the starting address of the current routine.
<ending-address>	The last address to disassemble. This can be any *<language-expression>* that evaluates to an address in the syntax of the current source language.
<instruction-count>	The number of instructions to disassemble. This can be any *<language-expression>* that evaluates to a positive integer in the syntax of the current source language. If no starting address is specified, the default count is all instructions of the current routine. If a starting address is specified, then the default count is 40 machine instructions.

Description

The `disassemble` command displays the assembly language code for the specified region of memory. The output of this command will vary, depending on whether you are debugging a process running on a C Series machine or on an SPP Series machine.

The region to disassemble can be specified either as an address range or as a starting address and the number of instructions to disassemble. If no region is specified, the default is the current routine, which is indicated by the current stack frame.

Examples

The following examples illustrate the syntax and output of the `disassemble` command. These examples show output from C Series machines only. While the assembly language instructions are different on C Series and SPP Series machines, the syntax of the `disassemble` command is the same, and the output contains the same type of information.

```
(CXdb) disassemble
Disassemble Process [#0/0] from 0x80002878 to 0x8000297c
   0x80002878  CHAPTER7:          sub.w   #0,a0
   0x8000287a  CHAPTER7+(0x2):    ldea    _bld_matrix_+(0x36c),a6
   0x80002880  CHAPTER7+(0x8):    calls   _for$s_wsle
          .
          .
          .
   0x80002978  CHAPTER7+(0x100):         ld.w    12(fp),a6
   0x8000297c  CHAPTER7+(0x104):         rtn
```

The above command displays the disassembled code for all threads of the current process. Because a memory region is not specified, the disassembly begins at the starting address of the routine indicated by the current stack frame. That routine in this case is called `CHAPTER7`. Because a count is not specified, the response displays all instructions for routine `CHAPTER7`. (The vertical ellipsis indicates that part of the response has been omitted.)

```
(CXdb) disassemble '800028dc'x .. '800028e4'x
Disassemble Process [#0/0] from 0x800028dc to 0x800028e4
   0x800028dc  CHAPTER7+(0x64):  mul.w   #4,a2
   0x800028de  CHAPTER7+(0x66):  ld.w    0(ap),a3          ;  ARRAY
   0x800028e2  CHAPTER7+(0x6a):  mov     a1,a4
   0x800028e4  CHAPTER7+(0x6c):  mul.w   #16,a4
```

The above command displays the disassembled code starting at address `800028dc` of the current process and continuing through address `800028e4`. The format used here for the addresses is the Fortran syntax for hexadecimal numbers. Variable names are shown at the far right; for example, the `ld.w` instruction references the variable `ARRAY`.

To represent the same address range in C syntax, the command would look like the following:

```
(CXdb) disassemble 0x800028dc .. 0x800028e4
Disassemble Process [#0/0] from 0x800028dc to 0x800028e4
   0x800028dc  CHAPTER7+(0x64):  mul.w   #4,a2
   0x800028de  CHAPTER7+(0x66):  ld.w    0(ap),a3          ;  ARRAY
   0x800028e2  CHAPTER7+(0x6a):  mov     a1,a4
   0x800028e4  CHAPTER7+(0x6c):  mul.w   #16,a4
```

The above command also displays the disassembled code starting at address `800028dc` and continuing through address `800028e4`. The output is the same as shown in the preceding example.

```
(CXdb) disassemble $pc..'8000299c'x
Disassemble Process [#0/0] from 0x80002994 to 0x8000299c
   0x80002994  ISQR+(0xe):   ld.w     mth$hwtype+(0x9c),s0
   0x8000299a  ISQR+(0x14):  rtn
   0x8000299c  ISQR+(0x16):  tac.b    0
```

The above command displays the disassembled code starting at the address stored in the current program counter (represented by the debugger variable `$pc`) and continuing through address `8000299c`.

```
(CXdb) disassemble ISQR
Disassemble Process [#0/0] from 0x80002986 for 40 machine instructions
   0x80002986  ISQR:  sub.w    #0,a0
   0x80002988  ISQR+(0x2):     ld.w     @0(ap),s0         ;  N
   0x8000298c  ISQR+(0x6):     mul.w    s0,s0
          .
          .
          .
   0x80002a26  BLD_MATRIX+(0x82):  add.w   #1,s0
   0x80002a2a  BLD_MATRIX+(0x86):  st.w    s0,J
   0x80002a30  BLD_MATRIX+(0x8c):  ld.w    J,s0
```

The above command displays the disassembled code starting at the address of the routine `ISQR` in the current process. Because a count is not specified, the response displays the default of 40 machine instructions starting at `ISQR`. (The vertical ellipsis indicates that part of the response has been omitted.)

```
(CXdb) disassemble ISQR:4
Disassemble Process [#0/0] from 0x80002986 for 4 machine instructions
   0x80002986  ISQR:  sub.w    #0,a0
   0x80002988  ISQR+(0x2):     ld.w    @0(ap),s0          ;  N
   0x8000298c  ISQR+(0x6):     mul.w   s0,s0
   0x8000298e  ISQR+(0x8):     st.w    s0,mth$hwtype+(0x9c)
```

The above command displays the disassembled code starting at the address of the routine `ISQR` in the current process and continuing for `4` instructions.

Related Commands

`display disassembly`
`examine`

Related Concepts

displaying data

Related Parameters

language-expression
process-list
thread-list

Related Windows

Assembly Code window
Memory Display window

display disassembly

disp d

Create an Assembly Code window in X Windows mode.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads used in evaluating the address to disassemble. The default is all threads in the current process.
<language-expression>	An expression that evaluates to a valid address within the bounds of the specified process. CXdb displays the assembly language code beginning at this address. The default is the starting address of the current routine.
\;	The language expression terminator.
<thread-number>	The number of the thread to associate with the new window. The default is no threads.
[**,** ...]	An optional list of threads to associate with the new Assembly Code window. Multiple thread numbers are separated by commas.

Description

The `display disassembly` command opens a new Assembly Code window to display the assembly language code beginning with the specified address. If an address is not specified, the current routine is disassembled.

The new Assembly Code window is associated with the threads specified at the end of the command. If no threads are specified, the window is not associated with any threads of the current process.

You can change the threads associated with an Assembly Code window by using the `set threads` command, or by using the threads option under the AssemblyCodeWindow menu.

This command has no effect when using CXdb in line mode (`-nw` option). Instead, use the `disassemble` command in line mode.

Examples

The following examples open new Assembly Code windows for the current process.

```
(CXdb) display disassembly
```

The above command opens a new Assembly Code window, displaying the assembly language code for the current routine. Because no threads were specified on the command line, the new window is not associated with any threads of the current process.

```
(CXdb) display disassembly SUB1
```

The above command opens a new Assembly Code window that displays the assembly language code for the `SUB1` routine. All instructions in the routine are disassembled. Again, the new Assembly Code window is not associated with any threads of the process.

```
(CXdb) display disassembly $pc \; 0, 1
```

The above command opens a new Assembly Code window that displays the assembly language code beginning at the address specified by `$pc`. The debugger variable `$pc` is a predefined debugger variable that contains the value of the program counter (PC). The language expression terminator (`\;`) separates the starting address from the thread list. The new window is associated with threads 0 and 1 of the current process.

Related Commands

`disassemble`
`display examine`
`display routine`
`display source`
`display stack`
`edit`
`examine`
`set threads`

Related Concepts

displaying data

Related Parameters

language-expression
process-list
thread-list

Related Windows

Assembly Code window

display examine

disp e

Create a Memory Display window in X Windows mode.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads used in evaluating the address to examine. The default is all threads in the current process.
<language-expression>	An expression that evaluates to a valid address within the bounds of the specified process. CXdb displays the section of memory beginning at this address.
\;	The language expression terminator.
<thread-number>	The number of the thread to associate with the new window. The default is the current thread.

Description

The `display examine` command opens a new Memory Display window to display the contents of memory beginning with the specified address.

The new Memory Display window is associated with the thread specified at the end of the command. If a thread is not specified, the window is associated with the current thread of the process.

You can change the thread associated with an Memory Display window by using the `set threads` command, or by using the threads option under the MemoryDisplayWindow menu.

This command has no effect when using CXdb in line mode (with the `-nw` option). Instead, use the `examine` command to display memory contents in line mode.

Examples

The following examples open new Memory Display windows for the current process.

Using Fortran syntax:

```
(CXdb) display examine loc(I)
```

The above command opens a new Memory Display window, displaying the contents of memory starting with the address of the variable `I`. The Fortran `loc()` function provides the starting address of the variable.

Using C syntax:

```
(CXdb) display examine &i
```

The above command opens a new Memory Display window to display the contents of memory, again beginning with the address of the variable `i`. The C address operator (`&`) provides the starting address of the variable.

```
(CXdb) display examine array \; 0
```

The above command opens a new Memory Display window that displays the contents of memory. The memory displayed begins with the starting address of the array named `ARRAY`. The `\;` separates the starting address from the thread list. The new window is associated with thread 0 of the current process.

Related Commands

`display disassembly`
`display routine`
`display source`
`display stack`
`examine`
`set threads`

Related Concepts

displaying data

Related Parameters

language-expression
process-list
thread-list

Related Windows

Memory Display window

display routine

disp r

Open a new Source Code window displaying source code for a routine.

Syntax

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

Parameter	Meaning
<*process-list*>	A list of processes affected by this command. The default is the current process.
<*thread-list*>	A list of threads to provide the context for evaluating the language expression.
<*language-expression*>	An expression that evaluates to a valid address within the bounds of the specified process. CXdb displays the routine that contains this address. The default is the current routine.
\;	The language expression terminator.
<*thread-number*>	The number of the thread to associate with the new Source Code window.
[, ...]	An optional list of threads to associate with the new Source Code window. Multiple thread numbers are separated by commas.

Description

The `display routine` command opens a new Source Code window and uses it to display the source code of the named routine or the routine that contains the specified address.

The new Source Code window is associated with the threads specified at the end of the command. If no threads are specified, the new window is not associated with any threads of the current process.

You can change the threads associated with a Source Code window by using the `set threads` command. In the X Windows interface, you can set the threads for a Source Code window using the Thread Selection dialog box. This dialog box can be opened using the threads option under the SourceCodeWindow menu.

This command has no effect when using CXdb in line mode (with the `-nw` option).

Examples

The following examples illustrate how to display a routine.

```
(CXdb) display routine INIT
```

The above command opens a new Source Code window, then it displays the source code for the routine `INIT` from the current process.

```
(CXdb) display routine '80001600'x
```

The above command opens a new Source Code window. Then it displays the source code for the routine that includes the address `80001600`.

Related Commands

`disassemble`	`display disassembly`
`display examine`	`display source`
`display stack`	`edit`
`examine`	`set threads`

Related Parameters

language-expression	process-list
thread-list	

Related Windows

Source Code window	SourceCodeWindow menu

display source

disp so

In X Windows mode, create a new Source Code window to display a source file.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<file-name>	The name of the source file to display. Relative file names use the console working directory as a base.
<thread-number>	The number of the thread to associate with the new Source Code window. The default is no threads.
[, ...]	An optional list of threads associated with the new Source Code window. Multiple thread numbers are separated by commas.

Description

The `display source` command opens a new Source Code window and uses it to display the source code of the named file. The source file must have been compiled as part of the current process.

The new Source Code window is associated with the threads specified at the end of the command. If no threads are specified, the new window is not associated with any threads of the current process.

You can change the threads associated with a Source Code window by using the `set threads` command. In the X Windows interface, you can set the threads for a Source Code window using the Thread Selection dialog box. This dialog box can be opened by selecting the threads item from the SourceCodeWindow menu.

This command has no effect when using CXdb in line mode (with the `-nw` option). Instead, use the `list` command to display source code in line mode.

Examples

The following examples open new Source Code windows for the current process.

```
(CXdb) display source main.f
```

The above command opens a new Source Code window that displays the source code for the main.f file.

```
(CXdb) display source sub.f 0,1
```

The above command opens a new Source Code window to display the source code for the sub.f file. The Source Code window is associated with threads 0 and 1 of the current process.

Related Commands

`disassemble`	`display disassembly`
`display examine`	`display routine`
`display stack`	`examine`
`list`	`set threads`

Related Parameters

file-name	process-list

Related Windows

Source Code window	SourceCodeWindow menu

display stack

disp st

Create a Stack Trace window in X Windows mode.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<frame-specifier>	A relative or absolute frame number. CXdb sets the current frame to this specified frame, then it displays the stack. If you do not specify a frame number, the current frame is not changed.
\;	The language expression terminator.
<thread-number>	The number of the thread to associate with the new window. The default is the current thread.

Description

The `display stack` command opens a new Stack Trace window to display the current stack for the specified thread. The specified frame becomes the current frame for the specified thread of the process.

A stack frame stores the registers for the context of the calling routine, temporary variables local to this context, and values necessary to manage the current stack frame, as well as a link to the previous frame. For more information about stacks and stack frames, refer to the *CONVEX Architecture Reference Manual (C Series).*

The new Stack Trace window is associated with the thread specified at the end of the command. If a thread is not specified, the window is associated with the current thread of the process.

You can change the thread associated with a Stack Trace window by using the `set threads` command, or by selecting the threads item under the StackTraceWindow menu.

This command has no effect when using CXdb in line mode (with the `-nw` option). Instead, use the `backtrace` command to display the stack in line mode.

Examples

The following examples open new Stack Trace windows for the current process.

```
(CXdb) display stack
```

The above command opens a new Stack Trace window, displaying the current stack. The current frame of the current process remains unchanged.

```
(CXdb) display stack 0
```

The above command opens a new Stack Trace window. The current frame of the current thread is set to frame 0.

```
(CXdb) display stack +1 \; 0
```

The above command opens a new Stack Trace window that displays the current stack for thread 0. The current frame for thread 0 is set to one greater (numerically) than the previous frame. Thus, if frame 2 was the current frame, the current frame is now frame 3. The `\;` separates the frame specifier from the thread list. The new Stack Trace window is associated with thread 0 of the current process.

Related Commands

`backtrace`	`display disassembly`
`display examine`	`display routine`
`display source`	`edit`
`examine`	`frame`
`info frame`	`info frame at`
`info stack`	`set threads`

Related Concepts

scope

Related Parameters

frame-specifier

process-list

Related Windows

Stack Frame Description dialog

Stack Trace window

echo

ec

Echo a character string.

Syntax

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

Parameter	Meaning
/n	A flag that prevents the string from being followed by a newline.
<*string*>	The string to be echoed to cmdout.
[...]	A list of additional strings to echo.

Description

The `echo` command echoes the specified strings to cmdout. The string need not be delimited by quotes. Only a string can be echoed. Any white space between specified strings is removed when the strings are echoed.

The `/n` flag prevents the string being followed by a newline. By using this flag you can display multiple messages on the same line.

The `echo` command echoes the string specified regardless of whether echoing has been enabled or disabled by the `set echo` or `clear echo` command.

Examples

The following examples echo a message.

```
(CXdb) echo "About to create eventpoints"
About to create eventpoints
```

The above command echoes the string to cmdout.

```
(CXdb) echo event point 1 created
eventpoint1created
```

The above command echoes the four separate strings specified. The white space between the words is removed before the strings are echoed. To retain the white space between words, you must enclose the entire sentence in quotes.

```
(CXdb) set handler 0 {echo/n "Caught signal: "; print $signal;}
```

The above command sets an eventpoint handler for eventpoint 0. The `echo` command inside the handler echoes the string without a newline. The output of the `print` command, which prints the value of the debugger variable `$signal`, appears on the same line as the string from the `echo` command.

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.

Related Commands

`clear echo`
`print`
`set echo`

Related Concepts

command files
eventpoints
eventpoint handlers
logging

Related Parameters

string

edit
ed

Edit a file.

Syntax

edit [*<file-name>*]

Parameter	Meaning
<file-name>	The name of the file to edit. The file name is relative to the console working directory, unless the path name is also specified.

Description

In X Windows mode, the `edit` command opens an editor window that enables you to edit the specified file. If the specified file does not exist, it is created. When you exit from the editor, the window closes.

In line mode, the `edit` command invokes your editor in a new shell.

The environment variable `$EDITOR` determines the editor invoked by this command. If `$EDITOR` is not set before you start CXdb, then the default editor is `vi`.

Examples

The following examples illustrate how to invoke an editor from within CXdb.

```
(CXdb) edit myfile.c
```

The above command invokes the default editor and opens the file myfile.c for editing. Because a path name was not specified in this case, myfile.c is in the console working directory.

```
(CXdb) edit /usr/local/Smith/prog4.f
```

The above command invokes editing for the file prog4.f, which is in the directory /usr/local/Smith.

Related Concepts console working directory

Related Parameters file-name

enable event

ena event

en

Enable eventpoints.

Syntax

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

Parameter	Meaning
<event-specifier>	An eventpoint to be enabled. The asterisk (*) is used to specify all eventpoints.
[**,** ...]	An optional list of eventpoints. Multiple eventpoints are separated by commas.

Description

The `enable event` command enables all specified eventpoints. An eventpoint is enabled when it is created. An eventpoint that is enabled is triggered when it is reached, unless it has an ignore count. If it has an ignore count, the ignore count is incremented.

The `enabled event` command is used to enable disabled eventpoints. Eventpoints can be disabled by the `disable event` command.

Examples

The following commands enable disabled eventpoints.

```
(CXdb) enable event 0
Eventpoint 0 enabled
```

The above command enables eventpoint 0. Eventpoint 0 can now be reached and therefore can be triggered.

```
(CXdb) enable event 1,2
Eventpoint 1 enabled
Eventpoint 2 enabled
```

The above command enables eventpoints 1 and 2. Both of these eventpoints can now be reached.

```
(CXdb) enable event *
Eventpoint 3 enabled

INFO 374: Event 2 is already enabled

INFO 374: Event 1 is already enabled

INFO 374: Event 0 is already enabled
```

The above command enables all existing eventpoints. CXdb displays the eventpoints that are enabled.

Related Commands

`disable event`
`disable eventtype`
`enable eventtype`
`info event`
`info eventtype`
`remove event`
`remove eventtype`
`set default handler`
`set handler`
`set ignore`
`set typehandler`

Related Concepts

breakpoints
eventpoints
eventpoint handlers
tracepoints
watchpoints

Related Parameters

event-specifier

enable eventtype

ena eventt

Enable all eventpoints of the specified type.

Syntax

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

Parameter	Meaning
<eventtype-specifier>	A type of eventpoint to enabled. The asterisk (*) is used to specify all eventpoint types.
[**,** ...]	An optional list of eventpoint types. Multiple eventpoint types are separated by commas.

Description

The `enable eventtype` command enables all existing eventpoints of the specified type. The following is a list of eventpoint types:

```
break
trace
watch
exec
join
modify
reached
relation
signal
spawn
```

When an eventpoint is created, it is enabled. An eventpoint that is enabled is triggered when it is reached, unless it has an ignore count. If it has an ignore count, the ignore count is incremented.

The `enabled eventtype` command is used to enable all disabled eventpoints of an eventpoint type. The eventpoints of an eventpoint type can be disabled by the `disable eventtype` command.

Examples

The following examples enable various types of eventpoints.

```
(CXdb) enable eventtype trace
Event 2 enabled
```

The above command enables all tracepoints. In this case, only eventpoint 2 is a tracepoint. Eventpoint 2 can now be triggered.

```
(CXdb) enable eventtype watch, break
Event 3 enabled
Event 1 enabled
```

The above command enables all watchpoints and breakpoints. CXdb displays the eventpoints that are enabled.

```
(CXdb) enable eventtype *
Eventpoint 0 enabled

INFO 374: Event 3 is already enabled

INFO 374: Event 2 is already enabled

INFO 374: Event 1 is already enabled
```

The above command enables all existing eventpoints, regardless of type. CXdb displays which eventpoints are enabled.

Related Commands

`disable event`
`disable eventtype`
`enable event`
`info event`
`info eventtype`
`remove event`
`remove eventtype`
`set default handler`
`set handler`
`set ignore`
`set typehandler`

Related Concepts

breakpoints
eventpoints
eventpoint handlers
tracepoints
watchpoints

Related Parameters

eventtype-specifier

evaluate

eva

Evaluate a language expression.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.
<language-expression>	Any expression that is valid in the current source language.

Description

The `evaluate` command evaluates the specified language expression. The language expression can include functions or subroutines from the specified process object.

The main purpose of the `evaluate` command is to assign values to debugger variables and to change the values of process variables.

The following related commands affect how the `evaluate` command performs its calculations:

- `set evalopts fpmode` (C Series only) — Selects the floating point mode (either native, IEEE, or dual) used to evaluate language expressions.
- `set evalopts iprecision` — Selects either 4-byte or 8-byte precision for integer constants.
- `set evalopts rprecision` — Selects either single precision (4-bytes) or double precision (8-bytes) for floating point constants.

Examples

The following examples illustrate various uses of the `evaluate` command.

```
(CXdb) evaluate K=3
```

The above command assigns the value `3` to the process variable `K`. Note that the value of `K` is now `3`, so this is the value the process will see when it resumes execution.

```
(CXdb) evaluate K=I+J
```

The above command evaluates the language expression `I+J` in the context of the current process. It then assigns the result of this evaluation to the process variable `K`. When the process resumes execution, it uses this new value for `K`. However, the process variables `I` and `J` are not changed.

```
(CXdb) evaluate $X=MATRIX(I,J,K) - 29.7/(I+J+K)
```

The above command evaluates the language expression `MATRIX(I,J,K) - 29.7/(I+J+K)` in the context of the current process. It then assigns the result of this evaluation to the debugger variable `X`.

```
(CXdb) evaluate $B=ISQR(K)
```

The above command evaluates the function `ISQR` in the context of the current process. The value returned by `ISQR` is stored in the debugger variable `B`. This command executes `ISQR` independent of the current process. When the function returns its value, the program counter (PC) and process stack are set back to the state they were in before the `evaluate` command was executed. Note that any eventpoints in `ISQR` may be triggered by this independent execution from the `evaluate` command.

Related Commands

`info cxdb`
`print`
`set evalopts fpmode`
`set evalopts ipprecision`
`set evalopts rprecision`

Related Concepts

C language expressions
debugger variables
Fortran language expressions
language expressions
modifying data
scope

Related Parameters

debugger-variable
language-expression
process-list
thread-list

event exec

eve e

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

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process.
<event-handler>	A sequence of CXdb commands enclosed in curly-braces (`{}`). Each command must be terminated with a semicolon (`;`).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

The `event exec` command sets an eventpoint to watch for your process to make the `exec(2)` system call.

When your process makes the system call, the eventpoint is triggered. When the eventpoint is triggered, process execution stops, and then the commands of the eventpoint's handler are executed. If the eventpoint does not have its own handler, then the default handler for eventpoints, which displays a message, is executed.

Examples

The following examples create `exec` eventpoints.

```
(CXdb) event exec

#0: exec on [#0], Enabled, ignore 0/0
```

The above command creates an eventpoint to watch for the current process to call the `exec(2)` function.

When you create an eventpoint, CXdb responds by executing the `info event` command on the new eventpoint. The output is explained below:

- `#0:` — The eventpoint number that identifies this particular eventpoint in other CXdb commands. In this case, the eventpoint number is 0.
- `exec` — The type of eventpoint.
- `on [#0], Enabled, ignore 0/0` — The eventpoint is set on process object 0. It is enabled and does not have an ignore count.

When the system call is made, the eventpoint is triggered.

```
(CXdb) event exec {echo "Process exec'ed"; resume;}
There is already an exec eventpoint active, continue? y

#1: exec on [#0], Enabled, ignore 0/0
    {
        echo "Process exec'ed";
        resume;
    }
```

The above command sets an eventpoint to watch for the current process to call the `exec` function. Because an `exec` eventpoint already exists from the first example, CXdb asks if you really want to create another `exec` eventpoint. If you answer with a **y**, CXdb creates the second eventpoint. If you do not, the command is aborted. When the eventpoint is triggered, the commands of its eventpoint handler are executed. The `echo` command is executed, and then process execution resumes.

Related Commands

`event join`	`event modify`
`event reached instruction`	`event reached line`
`event reached routine`	`event reached source`
`event relation`	`event signal`
`event spawn`	`set default handler`
`set handler`	`set typehandler`

Related Concepts

breakpoints	debugger variables
eventpoints	eventpoint handlers
tracepoints	watchpoints

Related Parameters

debugger-variable	event-handler
process-list	

event join

eve j

Set an eventpoint to trap a thread joining.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<event-handler>	A sequence of CXdb commands enclosed within curly-braces ({}). Each command must be terminated with a semicolon (;).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

The `event join` command creates an eventpoint to watch for a thread of the specified process to join.

When the threads join, the eventpoint is triggered. When the eventpoint is triggered, process execution stops, and then the commands of the eventpoint's handler are executed. If the eventpoint has not had an eventpoint handler defined for it, then the default handler for eventpoints, which displays a message, is executed.

Generally, only one join eventpoint is needed at a time. If you attempt to create another join eventpoint while the first is still enabled, CXdb asks if you want to continue to create the eventpoint. If you answer yes, the eventpoint is created. If you answer no, the `event join` command is terminated. If multiple join eventpoints are enabled and a thread spawns, both eventpoints are triggered.

For more information about threads, refer to the "threads" concepts page.

Examples

The following examples set eventpoints to watch for a thread of the current process to join.

```
(CXdb) event join

#0: join, on [#0], Enabled, ignore 0/0
```

The above command creates an eventpoint to watch for a thread of the current process to join.

When you create an eventpoint, CXdb responds by executing the `info event` command on the new eventpoint. The output is explained below:

- `#0:` — The eventpoint number that identifies this particular eventpoint in other CXdb commands. In this case, the eventpoint number is 0.
- `join` — The type of eventpoint.
- `on [#0], Enabled, ignore 0/0` — The eventpoint is set on process object 0. It is enabled and does not have an ignore count.

When the thread joins, the eventpoint is triggered. All threads of the process are stopped.

```
(CXdb) event join {echo "Thread joined"; resume;}
There is already a join eventpoint active, continue? y

#1: join, on [#0], Enabled, ignore 0/0
    {
        echo "Thread joined";
        resume;
    }
```

The above command creates an eventpoint to watch for a thread to join. Because a join eventpoint already exists from the first example, CXdb asks if you really want to create another join eventpoint. If you answer with a **y**, CXdb creates the second eventpoint. The eventpoint has been given its own eventpoint handler. When the eventpoint is triggered, the `echo` command is executed. Then all threads of the process are continued.

```
(CXdb) event join $Join
There is already a join eventpoint active, continue? y

#2: join, on [#0], Enabled, ignore 0/0
```

The above command again creates an eventpoint to watch for a thread to join. The eventpoint has been assigned to the debugger variable `$Join`. You can use the `$Join` debugger variable in other CXdb commands that affect this eventpoint. Debugger variables allow you to reference eventpoints without having to remember their eventpoint number.

Related Commands

`event exec`
`event spawn`
`info event`
`info eventtype`

Related Concepts

breakpoints
debugger variables
eventpoints
eventpoint handlers
optimized code
threads
tracepoints
watchpoints

Related Parameters

debugger-variable
event-handler
process-list

event modify

eve m

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

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process object.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process object.
<starting-address>	Any valid language expression whose evaluation is used as the starting address of the address range.
<ending-address>	Any valid language expresssion whose evaluation is used as the ending address of the address range.
<byte-count>	The total number of bytes to watch, including the start of the address range. The language expression describing this count must evaluate to a positive integer.
<event-handler>	A sequence of CXdb commands enclosed in curly-braces ({}). Each command must be terminated with a semicolon (;).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

The `event modify` command creates an eventpoint to watch the specified address range. A process must exist for a modify eventpoint to be created.

After the execution of each statement, CXdb tests to see if the value stored at the watched address has changed. If it has, the eventpoint is triggered.

When the eventpoint is triggered, process execution stops, and then the commands of the eventpoint's handler are executed. If the eventpoint does not have its own handler, then the default handler for eventpoints, which displays a message, is executed. Unless the eventpoint handler includes the `resume` command, execution is not restarted.

Modify eventpoints are triggered when the contents of the watched address changes. Therefore, they are not associated with a particular location in the executing code. Eventpoints of this type are known as asynchronous eventpoints. Multiple asynchronous eventpoints can be triggered at the same time. In such cases, only the eventpoint handler of the lowest-numbered asynchronous eventpoint is executed.

Specifying an address range to watch

The address range can be specified using one of the following three methods:

- Specify a starting address and ending address. Both addresses are language expressions whose evaluations are used to determine the address range.
- Specify a starting address and a number of bytes to watch. The number of bytes watched starts from the starting address. The number of bytes to watch is a language expression that must evaluate to a positive integer.
- Specify a starting address. The starting address is a language expression. If the address of a variable is given, the entire region of the variable is watched. If an absolute address is specified, only the one byte at that address is watched.

Examples

The following examples set watchpoints. The syntax for retrieving a variable's address is different between Fortran and C. The next two examples demonstrate this difference.

```
(CXdb) event modify loc(TABLE)
#1: modify 0x80077058..0x80077097, on [#0/0], Enabled, ignore 0/0
```

The above command sets an eventpoint to monitor the contents of the Fortran array `TABLE`. The Fortran function `loc()` provides the starting address of the array. When any value stored in the array `TABLE` changes, the eventpoint is triggered.

When you create an eventpoint, CXdb responds by displaying the following information:

- `#1:` — The eventpoint number used to identify this particular eventpoint in other CXdb commands. In this case the eventpoint number is 1.
- `modify` — The type of eventpoint.
- `0x80077058..0x80077097` — The address range that the eventpoint monitors.
- `on [#0/0], Enabled, ignore 0/0` — The eventpoint is set on process object 0, for thread 0. It is enabled and does not have an ignore count.

You can also display the above information by using the `info event` command.

```
(CXdb) event modify &sum

#3: modify 0x8008ace0..0x8008ace3, on [#0/0], Enabled, ignore 0/0
```

The above command watches the contents of the C variable `sum`. The C operator `&` provides the address of the variable. CXdb responds with the same type of information as shown in the Fortran example above. When the value stored in `sum` changes, the eventpoint is triggered.

The syntax for specifying an absolute address is different between Fortran and C. The next two examples demonstrate this difference.

```
(CXdb) event modify '80001234'x:4

#5: modify 0x80001234..0x80001237, on [#0/0], Enabled, ignore 0/0
```

The above command uses the `:` notation to specify an address range. The eventpoint monitors four bytes, starting with the address `80001234` and ending with the address `80001237`. The notation `'80001234'x` is Fortran-specific and indicates the address is in hexadecimal notation. When the value stored in this address range changes, the eventpoint is triggered.

```
(CXdb) event modify 0x80001234:4
#6: modify 0x80001234..0x80001237, on [#0/0], Enabled, ignore 0/0
```

The above command watches four bytes starting with address `80001234`. The notation `0x80001234` is C-specific and indicates the address is in hexadecimal notation. When the value stored in this range changes, the eventpoint is triggered.

```
(CXdb) event modify '80001234'x..'80001237'x

#7: modify 0x80001234..0x80001237, on [#0/0], Enabled, ignore 0/0
```

The above command uses the `..` notation to specify an address range, starting with the address `80001234` and ending with `80001237`. When the value stored in this address range changes, the eventpoint is triggered.

```
(CXdb) event modify '80002345'x:8 {echo "region B modified"; resume;}

#8: modify 0x80002345..0x8000234c, on [#0/0], Enabled, ignore 0/0
    {
        echo "region B modified";
        resume;
    }
```

The above command sets an eventpoint to watch the eight bytes starting from the specified address. A handler is defined for the eventpoint. When the eventpoint is triggered, the `echo` command in the handler is executed, then process execution resumes.

```
(CXdb) event modify loc(TABLE) \; $w1

#9: modify 0x80077058..0x80077097, on [#0/0], Enabled, ignore 0/0
```

The above command watches the array `TABLE`. The `\;` is needed to separate the language expression from the debugger variable. The debugger variable `$w1` has been assigned to the eventpoint. In future CXdb commands, you can use the debugger variable to refer to this eventpoint. Debugger variables allow you to refer to eventpoints without having to remember their eventpoint numbers.

Related Commands

`event relation`
`set default handler`
`set handler`
`set typehandler`
`watch`

Related Concepts

breakpoints
debugger variables
eventpoints
eventpoint handlers
tracepoints
watchpoints

Related Parameters

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

event reached instruction

eve rea i

Set an eventpoint at an instruction.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.
<language-expression>	A valid language expression whose evaluation is used as the instruction address.
<event-handler>	A sequence of CXdb commands enclosed within curly braces (`{}`). Each command must be terminated with a semicolon (`;`).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

The `event reached instruction` command sets an eventpoint at the specified instruction address.

The address may be any valid language expression that evaluates to an address.

When the eventpoint is triggered, process execution stops, and the commands of the eventpoint's handler are executed. If the eventpoint does not have its own handler, the default handler for eventpoints, which displays a message, is executed. Unless the eventpoint handler includes the `resume` command, execution is not restarted.

Examples

The following examples set eventpoints at specific instruction addresses.

```
(CXdb) event reached instruction PRINT_ARRAY

#0: reached instruction, on [#0/*], Enabled, ignore 0/0
        [0x80005506] PRINT_ARRAY in example.f line 35
```

The above command sets an eventpoint at the first instruction of the routine `PRINT_ARRAY`. The evaluation of the language expression `PRINT_ARRAY` is used as the address for this eventpoint. When a routine name is used with an `event reached instruction` command, the eventpoint is placed before the preamble (which manages the stack) of the routine. In contrast, a routine name used with an `event reached routine` command places the eventpoint at the first executable source unit of the routine.

When you create an eventpoint, CXdb responds by executing the `info event` command on the new eventpoint. The output is explained below:

- `#0:` — The eventpoint number that identifies this particular eventpoint in other CXdb commands. In this case, the eventpoint number is 0.
- `reached instruction` —The type of eventpoint.
- `on [#0/*], Enabled, ignore 0/0` — The eventpoint is set on process object 0, for all threads (`*`). It is enabled and does not have an ignore count.
- `[0x80005506]` — The hexadecimal address location of the eventpoint. In this case, the address is `80005506`.
- `PRINT_ARRAY in example.f line 35` — The symbolic location of the eventpoint. In this case, the eventpoint is in the routine `PRINT_ARRAY` at line 35 of the source file example.f.

When the eventpoint is triggered, execution is stopped before the instruction at that address is executed.

The syntax for specifying an absolute address is different between Fortran and C. The next two examples demonstrate this difference.

Using Fortran syntax:

```
(CXdb) event reached instruction '800015f0'x

#1: reached instruction, on [#0/*], Enabled, ignore 0/0
        [0x800015f0] CHAPTER4 in chapter4.f line 26
```

The above command sets an eventpoint at the absolute address `800015f0`. The eventpoint number is 1, located at address `800015f0` in routine `CHAPTER4` corresponding to line 26 of the file example.f. The notation `'800015f0'x` is Fortran-specific and indicates the address is in hexadecimal notation.

Using C syntax:

```
(CXdb) event reached instruction 0x80004c2e

#3: reached instruction, on [#0/*], Enabled, ignore 0/0
        [0x80004c2e] chapter13C`subxa in chapter13C.c line 41
```

The above command sets an eventpoint at the absolute address `80004c2e`. The `0x` is the C notation for a hexadecimal number. The symbolic location uses the scope path of `chapter13c`subxa` to indicate the source file and routine in which the eventpoint is located.

When you specify an absolute address, the eventpoint is set at the closest even boundary. Because of this, you must be sure that the address is actually the starting address for the instruction. If the eventpoint is placed at an address in the middle of an instruction, it will be interpreted as a portion of the instruction, which can cause unpredictable results.

```
(CXdb) event reached instruction subxa {echo 'routine subxa reached';}

#3: reached instruction, on [#0/*], Enabled, ignore 0/0
        [0x80004c2e] chapter13C`subxa in chapter13C.c line 41
    {
        echo 'routine subxa reached';
    }
```

The above command sets an eventpoint at address `80004c2e`, the starting address of routine `subxa`. The eventpoint is given its own eventpoint handler. When the eventpoint is triggered, execution is stopped, and then the `echo` command is executed.

```
(CXdb) event reached instruction '800015f0'x \; $Event4

#4: reached instruction, on [#0/*], Enabled, ignore 0/0
        [0x800015f0] CHAPTER4 in chapter4.f line 26
```

The above command creates a new eventpoint at the absolute address `800015f0`. The `\;` is needed to separate the language expression from the debugger variable. The debugger variable `$Event4` is created and set equal to the number of this eventpoint. In subsequent commands you can use `$Event4` to refer to this eventpoint. Debugger variables allow you to refer to eventpoints without having to remember their eventpoint numbers.

Related Commands

`break instruction`
`event reached line`
`event reached routine`
`event reached source`
`trace instruction`
`watch`

Related Concepts

breakpoints
debugger variables
eventpoints
eventpoint handlers
tracepoints
watchpoints

Related Parameters

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

event reached line

eve rea l

Set an eventpoint at a source line.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.
<line-specifier>	The line number where the eventpoint is to be set. The line number must be an integer, and may be preceded by a source file name.
<event-handler>	A sequence of CXdb commands enclosed within curly braces (`{}`). Each command must be terminated with a semicolon (`;`).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

The `event reached line` command sets an eventpoint before the first statement of the specified line.

If the line number does not map to a source line (whether due to optimizations or the line being a comment line), CXdb asks if you want the eventpoint set at the next highest line number that maps to a source line.

When the eventpoint is triggered, process execution stops, and the commands of the eventpoint's handler are executed. If the eventpoint does not have its own handler, the default handler for eventpoints, which displays a message, is executed. Unless the eventpoint handler includes the `resume` command, execution is not restarted.

Examples

The following examples set eventpoints at specific source lines.

```
(CXdb) event reached line 18

#0: reached line, on [#0/*], Enabled, ignore 0/0
        [0x8000545e] EXAMPLE in example.f line 18
```

The above command sets an eventpoint at the starting address that corresponds to line 18 of the current source file.

When you create an eventpoint, CXdb responds by executing the `info event` command on the new eventpoint. The output is explained below:

- `#0:` — The eventpoint number that identifies this particular eventpoint in other CXdb commands. In this case the eventpoint number is 0.
- `reached line` — The type of eventpoint.
- `on [#0/*], Enabled, ignore 0/0` — The eventpoint is set on process object 0, for all threads (*). It is enabled, and does not have an ignore count.
- `[0x8000545e]` — The hexadecimal address location of the eventpoint. In this case the address is `8000545e`.
- `EXAMPLE in example.f line 18` — The symbolic location of the eventpoint. In this case the eventpoint is in the routine `EXAMPLE` at line 18 of the source file example.f.

When the eventpoint is triggered, execution is stopped before the first instruction of the first statement on that line is executed.

```
(CXdb) event reached line example.f:31

#1: reached line, on [#0/*], Enabled, ignore 0/0
        [0x800054d8] EXAMPLE in example.f line 31
```

The above command sets an eventpoint at the starting address of line 31 of the source file example.f. This source file must have been part of the compilation of the current executable file and be included in the search path of the process object.

```
(CXdb) event reached line 18 {echo 'Line 18 reached'; resume;}

#2: reached line, on [#0/*], Enabled, ignore 0/0
        [0x8000545e] EXAMPLE in example.f line 18
    {
        echo 'Line 18 reached';
        resume;
    }
```

The above command sets an eventpoint at the starting address of line 18 of the current source file. An eventpoint handler is defined for the eventpoint. When the eventpoint is triggered, process execution stops, and the commands of the eventpoint handler are executed. The first command displays a message and the second command resumes process execution.

```
(CXdb) event reached line 18 $Event3

#3: reached line, on [#0/*], Enabled, ignore 0/0
        [0x8000545e] EXAMPLE in example.f line 18
```

The above command creates a new eventpoint at line 18. The debugger variable `$Event3` is created and set equal to the number of this eventpoint. In subsequent commands you can use `$Event3` to refer to this eventpoint. Debugger variables allow you to refer to eventpoints without having to remember their eventpoint numbers.

Related Commands

`break line`
`event reached instruction`
`event reached routine`
`event reached source`
`info event`
`info eventtype`
`trace line`
`watch`

Related Concepts

breakpoints
debugger variables
eventpoints
eventpoint handlers
tracepoints
watchpoints

Related Parameters

debugger-variable
event-handler

event reached routine

eve rea r

Set an eventpoint at the beginning of a routine.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.
<language-expression>	A valid language expression whose evaluation is used as the instruction address.
<event-handler>	A sequence of CXdb commands enclosed within curly braces (`{}`). Each command must be terminated with a semicolon (`;`).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

The `event reached routine` command sets an eventpoint at the first executable source unit of the routine containing the specified instruction address. If there are multiple entry points into the routine, an eventpoint is set at each entry point.

The specified address can be any valid language expression that evaluates to an address. CXdb finds the routine that contains this address and places the eventpoint at its first executable source unit. The first executable source unit is usually the first statement of a routine, unless there are local variable initializations.

When the eventpoint is triggered, process execution stops and the commands of the eventpoint's handler are executed. If the eventpoint does not have its own handler, the default handler for eventpoints, which displays a message, is executed. Unless the eventpoint handler includes the `resume` command, execution is not restarted.

Examples

The following examples set eventpoints at the first executable source units of routines.

```
(CXdb) event reached routine PRINT_ARRAY

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

The above command sets an eventpoint at the first executable source unit of the routine `PRINT_ARRAY`.

When you create an eventpoint, CXdb responds by executing the `info event` command on the new eventpoint. The output is explained below:

- `#0:` — The eventpoint number used to identify this particular eventpoint in other CXdb commands. In this case the eventpoint number is 0.
- `reached routine` — The type of eventpoint.
- `on [#0/*], Enabled, ignore 0/0` — The eventpoint is set on process object 0, for all threads (*). It is enabled and does not have an ignore count.
- `[0x80005508]` — The hexadecimal address location of the eventpoint. In this case the address is `80005508`.
- `PRINT_ARRAY in example.f line 39` — The symbolic location of the eventpoint. In this case the eventpoint is in the routine `PRINT_ARRAY` at line 39 of the source file example.f.

When the eventpoint is triggered, execution is stopped before the first source unit in the routine is executed.

The following two examples set an eventpoint at the start of a routine by specifying an absolute address inside of that routine. CXdb finds the routine containing the absolute address and places the eventpoint at the first source unit. The syntax for specifying an absolute address is different between Fortran and C.

Using Fortran syntax:

```
(CXdb) event reached routine '80005508'x

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

The above command sets an eventpoint at the starting address of the routine that contains the absolute address `80005508`. The eventpoint number is 1, located at address `80005508` in routine `PRINT_ARRAY` at line 39 of the file example.f. The notation `'80005508'x` is Fortran-specific and indicates that the address is in hexadecimal notation.

Using C syntax:

```
(CXdb) event reached routine 0x80004c32

#3: reached routine, on [#0/*], Enabled, ignore 0/0
        [0x80004c32] chapter13C`subxa in chapter13C.c line 45
```

The above command sets an eventpoint at the starting address of the routine that contains the absolute address `80004c32`. The `0x` is the C notation for a hexadecimal number. The symbolic location uses the scope path of `chapter13c'subxa` to indicate the source file and routine in which the eventpoint is located.

```
(CXdb) event reached routine subxa {echo 'routine subxa reached';}

#4: reached routine, on [#0/*], Enabled, ignore 0/0
        [0x80004c32] chapter13C`subxa in chapter13C.c line 45
    {
        echo 'routine subxa reached';
    }
```

The above command sets an eventpoint at the address of the first executable source unit of the routine `subxa`. An eventpoint handler is defined for the eventpoint. When the eventpoint is triggered, execution is stopped, and the `echo` command is executed.

```
(CXdb) event reached routine '80004c32'x \; $Event5

#5: reached routine, on [#0/*], Enabled, ignore 0/0
        [0x80004c32] chapter13C`subxa in chapter13C.c line 45
```

The above command creates a new eventpoint at the first executable source unit of the routine containing the absolute address `80004c32`. The `\;` is needed to separate the language expression from the debugger variable. The debugger variable `$Event5` is created and set equal to the number of this eventpoint. In subsequent commands you could use `$Event5` to refer to this eventpoint. Debugger variables allow you to refer to eventpoints without having to remember their eventpoint numbers.

Related Commands

`break routine`
`event reached instruction`
`event reached line`
`event reached source`
`event relation`
`event signal`
`info event`
`trace routine`
`watch`

Related Concepts

breakpoints
debugger variables
eventpoints
eventpoint handlers
tracepoints
watchpoints

Related Parameters

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

event reached source

eve rea s

Set an eventpoint at a source unit.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.
<source-unit>	The source unit number where the eventpoint is to be set. The source unit number must be an integer, and may be preceded by a source file name.
<event-handler>	A sequence of CXdb commands enclosed within curly braces (`{}`). Each command must be terminated with a semicolon (`;`).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

The `event reached source` command sets an eventpoint at the specified source unit number.

Source units are numbered by CXdb. The number of a particular source unit can be determined by using the `info line` command. You can also gather information about the source unit that a source unit number corresponds to by using the `info sourceunit` command.

When the eventpoint is triggered, process execution stops and the commands of the eventpoint's handler are executed. If the eventpoint does not have its own handler, the default handler for eventpoints, which displays a message, is executed. Unless the eventpoint handler includes the `resume` command, execution is not restarted.

Examples

The following examples set eventpoints at specific source units.

```
(CXdb) event reached source 25

#0: reached source, on [#0/*], Enabled, ignore 0/0
        [0x80005452] EXAMPLE in example.f line 17
```

The above command sets an eventpoint at the starting address of source unit `25` of the current source file.

When you create an eventpoint, CXdb responds by executing the `info event` command on the new eventpoint. The output is explained below:

- `#0:` — The eventpoint number. The eventpoint number is used to identify this particular eventpoint in other CXdb commands. In this case the eventpoint number is 0.
- `reached source` — The type of eventpoint.
- `on [#0/*], Enabled, ignore 0/0` — The eventpoint is set on process object 0, for all threads (*). It is enabled and does not have an ignore count.
- `[0x80005452]` — The hexadecimal address location of the eventpoint. In this case the address is `80005452`.
- `EXAMPLE in example.f line 17` — The symbolic location of the eventpoint. In this case the eventpoint is in the routine `EXAMPLE` at line 17 of the source file example.f.

When the eventpoint is triggered, execution is stopped before the first instruction of the source unit is executed.

```
(CXdb) event reached source example.f:100

#1: reached source, on [#0/*], Enabled, ignore 0/0
        [0x80001234] CLEAR_ARRAY in example.f line 55
```

The above command sets an eventpoint at the starting address of source unit 100 of the source file example.f. This source file must have been part of the compilation of the current executable file and be included in the search path of the process object.

```
(CXdb) event reached source 25 {echo 'Source unit 25 reached'; resume;}

#2: reached source, on [#0/*], Enabled, ignore 0/0
        [0x80005452] EXAMPLE in example.f line 17
    {
       echo 'Source unit 25 reached';
       resume;
    }
```

The above command sets an eventpoint at the starting address of source unit 25 of the current source file. An eventpoint handler is defined for the eventpoint. When the eventpoint is triggered, process execution stops, and the commands of the eventpoint handler are executed. The first command displays a message and the second command resumes process execution.

```
(CXdb) event reached source 25 $Event3

#3: reached source, on [#0/*], Enabled, ignore 0/0
        [0x80005452] EXAMPLE in example.f line 17
```

The above command sets an eventpoint at the starting address of source unit 25 in the current source file. A debugger variable has been assigned to this eventpoint. In subsequent commands you can use the debugger variable `$Event3` to refer to this eventpoint. Debugger variables allow you to refer to eventpoints without having to remember their eventpoint numbers.

Related Commands

`break source`
`event reached line`
`event reached routine`
`info line`
`info sourceunit`
`trace source`
`watch`

Related Concepts

breakpoints
debugger variables
eventpoints
eventpoint handlers
tracepoints
watchpoints

Related Parameters

debugger-variable
event-handler
source-unit
process-list
thread-list

event reached source

event relation

eve rel

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

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process object.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process object.
<language-expression>	A relational language expression to evaluate.
<event-handler>	A sequence of CXdb commands enclosed within curly braces (`{ }`). Each command must be terminated by a semicolon (`;`).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

The `event relation` command sets an eventpoint watching for the value of the specified language expression to become true.

The expression must evaluate to TRUE or FALSE, as defined by the current language. The expression cannot evaluate to TRUE when the eventpoint is created.

After the execution of each statement source unit, CXdb tests to see if any enabled relation eventpoint has become true. If it has, that eventpoint is triggered.

NOTE: Due to the extra checking involved with relation eventpoints, process execution is substantially slowed.

When the eventpoint is triggered, process execution stops, and the commands of the eventpoint's handler are executed. If the eventpoint does not have its own handler, then the default handler for eventpoints, which displays a message, is executed. Unless the eventpoint handler includes the `resume` command, execution is not restarted.

Relation eventpoints are not associated with a particular address. They are triggered when their condition evaluates to `TRUE`. Eventpoints of this type are known as asynchronous eventpoints. If multiple asynchronous eventpoints are reached at the same time only the first (lowest-numbered) eventpoint is triggered. Thus, only the handler of this eventpoint is executed.

Examples

The following examples set eventpoints for relations.

```
(CXdb) event relation i+j

#1: relation (i+j  ), on [#0/0], Enabled, ignore 0/0
Eventpoint 1 will be disabled when current stack frame returns
```

The above command sets an eventpoint to watch for the relation `i+j` to evaluate to `TRUE`.

When you create an eventpoint, CXdb responds by executing the `info event` command on the new eventpoint. The output is explained below:

- `#1:` — The eventpoint number that identifies this particular eventpoint in other CXdb commands. In this case, the eventpoint number is 0.
- `relation` —The type of eventpoint.
- `(i+j  )` — The relational expression the eventpoint is monitoring.
- `on [#0], Enabled, ignore 0/0` — The eventpoint is set on process object 0. It is enabled and does not have an ignore count.

When the relation becomes `TRUE`, the eventpoint is triggered, process execution stops, and the commands for the default handler for relation eventpoints is executed.

When the eventpoint is created, CXdb indicates that the eventpoint will be disabled when the current frame returns. Relation eventpoints are only enabled in the frame in which they are created. When process execution causes that frame to be removed from the stack, the eventpoint is disabled. Thus, the eventpoint is enabled only when the routine is executing or any routines it called are executing.

The language expression used to describe a relation is dependent upon the current language. The next two examples describe the same relation, first in Fortran and then in C.

Using Fortran syntax:

```
(CXdb) event relation ARRAY(1,1) .EQ. 4

#2: relation (ARRAY(1,1) .EQ. 4 ), on [#0/0], Enabled, ignore 0/0
Eventpoint 2 will be disabled when current stack frame returns
```

The above command sets an eventpoint to be triggered when the value of `ARRAY(1,1)` equals `4`. In this example the current language is Fortran, so the expression is in Fortran syntax. When `ARRAY(1,1)` equals `4`, the eventpoint is triggered, and the default handler for relation eventpoints is executed.

Using C syntax:

```
(CXdb) event relation i==4

#4: relation (i==4), on [#0/0], Enabled, ignore 0/0
Eventpoint 4 will be disabled when current stack frame returns
```

The above command sets an eventpoint to be triggered when the value of `i` equals `4`. In this example the current language is C, so the expression is in C syntax. When `i` equals `4`, the eventpoint is triggered, and the default handler for relation eventpoints is executed.

```
(CXdb) event relation i {print i;}
#6: relation (i ), on [#0/0], Enabled, ignore 0/0
    {
        print i;
    }
Eventpoint 6 will be disabled when current stack frame returns
```

The above command sets an eventpoint to be triggered when the value of `i` is nonzero. The eventpoint has been given its own handler. The eventpoint handler prints the value of `i`. A semicolon terminates each command in an eventpoint handler, even if there is only one command. It is important to remember that a relation eventpoint is associated with a particular frame and therefore to a particular scope.

```
(CXdb) event relation ARRAY(1,1) .EQ. TABLE(1,1) \; $Checkarrays

#7: relation (ARRAY(1,1) .EQ. TABLE(1,1) \;), on [#0/0], Enabled, ignore 0/0
Eventpoint 7 will be disabled when current stack frame returns
```

The above command sets an eventpoint to be triggered when the values of the two array elements at the specified locations are equal. A debugger variable has been assigned to this eventpoint. In subsequent commands, you can use `$Checkarrays` to refer to this eventpoint. Debugger variables allow you to refer to eventpoints without having to remember their eventpoint numbers.

Related Commands

`event exec`
`event modify`
`event reached instruction`
`event reached line`
`event reached routine`
`event reached source`
`event signal`
`info event`
`info eventtype`
`trace instruction`
`trace line`
`trace routine`
`trace source`
`watch`

Related Concepts

breakpoints
debugger variables
eventpoints
eventpoint handlers
tracepoints
watchpoints

Related Parameters

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

event signal

eve si

Set an eventpoint to catch a signal.

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process object.
<signal-specifier>	The signal to be caught.
<event-handler>	A sequence of CXdb commands enclosed within curly-braces (`{ }`). Each command must be terminated with a semicolon (`;`).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

The `event signal` command sets an eventpoint to catch the specified signal.

Any signals listed in the man pages for `sigvec(2)` can be caught. When an eventpoint is set to catch a signal, the eventpoint's handler takes precedence over the `stop`, `pass`, and `print` actions specified with the `set signal` command. The actions set with the `set signal` command are not removed, so if the eventpoint is ever disabled or removed, those actions will once again determine how CXdb handles a caught signal.

When the eventpoint is triggered, process execution stops, and the commands of the eventpoint's handler are executed. If the eventpoint does not have its own handler, the default handler for eventpoints, which displays a message, is executed. Unless the eventpoint handler includes the `resume` command, execution is not restarted.

When specifying a signal you can use its full name, its name without the `SIG` prefix, or its signal number. Signal names are not case sensitive. Signal names and signal numbers differ on C Series and SPP Series architectures. For more information, refer to the "signals" and "architecture dependencies" concepts pages.

Examples

The following examples create eventpoints to catch the signal `SIGINT`. Assume that during execution the signal `SIGINT` is sent to the process.

```
(CXdb) event signal SIGINT

#0: signal 2 on [#0], Enabled, ignore 0/0
```

The above command sets an eventpoint to catch the signal `SIGINT`. CXdb catches the signal before the process receives it.

When you create an eventpoint, CXdb responds by executing the `info event` command on the new eventpoint. The output is explained below:

- `#0:` — The eventpoint number that identifies this particular eventpoint in other CXdb commands. In this case, the eventpoint number is 0.
- `signal` —The type of eventpoint.
- `2` — the number of the signal to catch.
- `on [#0], Enabled, ignore 0/0` — The eventpoint is set on process object 0. It is enabled and does not have an ignore count.

When CXdb catches the signal, the eventpoint is triggered, and the process is stopped. Because the signal does not have its own eventpoint handler, the default handler for signals, which displays a message, is executed.

```
(CXdb) event signal SIGINT {eval $signal=0; resume;}

#1: signal 2 on [#0], Enabled, ignore 0/0
    {
        eval $signal=0;
        resume;
    }
```

The above command again sets an eventpoint to catch the signal `SIGINT`. CXdb catches the signal, triggers the eventpoint, and then the commands of the eventpoint's handler are executed. First, the debugger variable `$signal`, which holds the value of the current signal, is set to 0. Second, process execution resumes. When execution resumes, the signal stored in `$signal` is sent to the process. However, because `$signal` is zero, no signal is sent to the process. Thus, this eventpoint handler causes the signal `SIGINT` to be completely ignored.

```
(CXdb) event signal SIGINT $sigint_trap

#2: signal 2 on [#0], Enabled, ignore 0/0
```

The above command sets an eventpoint to catch the signal `SIGINT`. This time, the debugger variable `$sigint_trap` is assigned to the eventpoint. You can use the debugger variable in other CXdb commands that affect this eventpoint. Debugger variables allow you to reference eventpoints without having to remember their eventpoint numbers.

Related Commands

`event exec`	`event join`
`event modify`	`event reached instruction`
`event reached line`	`event reached routine`
`event reached source`	`event relation`
`event spawn`	`info event`
`info eventtype`	`info signal`
`set signal`	`breakpoints`

Related Concepts

architecture dependencies	breakpoints
debugger variables	eventpoints
eventpoint handlers	signals
tracepoints	watchpoints

Related Parameters

debugger-variable	event-handler
process-list	signal-specifier

event spawn

eve sp

Set an eventpoint to trap the spawning of a thread.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<event-handler>	A sequence of CXdb commands enclosed within curly-braces ({ }). Each command must be terminated with a semicolon (;).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

The `event spawn` command creates an eventpoint to watch for the process to spawn a thread.

When one or more threads spawn, the eventpoint is triggered. When the eventpoint is triggered, process execution stops, and then the commands of the eventpoint's handler are executed. If the eventpoint does not have its own handler, then the default handler for eventpoints, which displays a message, is executed.

Generally, only one spawn eventpoint is needed at a time. If you attempt to create another eventpoint of type spawn while the first is still enabled, CXdb asks if you want to continue to create the eventpoint. If you answer yes, the eventpoint is created. If you answer no, the `event spawn` command is terminated. If multiple spawn eventpoints are enabled and a thread spawns, both eventpoints are triggered.

For more information about threads, refer to the "threads" and "optimized code" reference pages.

Examples

The following examples set eventpoints to watch for a thread of the current process to spawn.

```
(CXdb) event spawn

#0: spawn, on [#0], Enabled, ignore 0/0
```

The above command creates an eventpoint to watch for a thread of the current process to spawn.

When you create an eventpoint, CXdb responds by executing the `info event` command on the new eventpoint. The output is explained below:

- `#0:` — The eventpoint number that identifies this particular eventpoint in other CXdb commands. In this case, the eventpoint number is 0.
- `spawn` — The type of eventpoint.
- `on [#0], Enabled, ignore 0/0` — The eventpoint is set on process object 0. It is enabled and does not have an ignore count.

After the thread spawns, the eventpoint is triggered. All threads of the process are then stopped.

```
(CXdb) event spawn {echo "A thread has spawned"; resume;}
There is already a spawn eventpoint active, continue? y

#1: spawn, on [#0], Enabled, ignore 0/0
    {
        echo "A thread has spawned";
        resume;
    }
```

The above command creates an eventpoint to watch for a thread to spawn. Because a spawn eventpoint already exists from the first example, CXdb asks if you really want to create another spawn eventpoint. If you answer with a **y**, CXdb creates the second eventpoint. The eventpoint has been given its own eventpoint handler. When the eventpoint is triggered, the `echo` command is executed, and then process execution resumes. In this case, all threads of the process resume execution.

Related Commands

`event exec`
`event join`
`event modify`
`event reached instruction`
`event reached line`
`event reached routine`
`event reached source`
`event relation`
`event signal`
`info event`
`info eventtype`
`info threads`
`watch`

Related Concepts

breakpoints
debugger variables
eventpoints
eventpoint handlers
optimized code
threads
tracepoints
watchpoints

Related Parameters

debugger-variable
event-handler
process-list

examine

exa

x

Display a region of memory.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.
<memory-unit>	The type of memory unit displayed. The memory unit specifications are: **b** — byte (8 bits) **h** — halfword (16 bits) **w** — word (32 bits) **l** — longword (64 bits) **q** — quadword (128 bits)
<format>	The format for displaying the memory units. The format specifications are: **c** — unsigned ASCII character **d** — decimal **e** — scientific notation **f** — floating point **o** — unsigned octal **u** — unsigned decimal **x** — unsigned hexadecimal **B** — binary **C** – Fortran complex **L** — logical

<fpmode>	The floating point mode. Available modes are: **D** — dual mode floating point (C Series only) **I** — IEEE mode floating point **N** — native mode floating point (C Series only)
<starting-address>	The first address to display. This can be any *<language-expression>* that evaluates to an address.
<ending-address>	The last address to display. This can be any *<language-expression>* that evaluates to an address.
<unit-count>	The number of memory units to display. The count can be any *<language-expression>* that evaluates to a positive integer. The default count is 20.

Description

The `examine` command displays the specified region of memory. The region to display may be specified either as an address range or as a starting address and the number of memory units to display.

The memory unit type, display format, and floating point mode can also be specified with the `examine` command. If you do not specify these settings on the command line, then CXdb looks for them in the following order and uses the first one it encounters:

- Process settings (displayed with the `info formatting` command)
- Default process settings (displayed with the `info cxdb` command)
- Words of memory (`w`) in hexadecimal format (`x`), with dual mode floating point (`D`)

Memory display formats

Only certain display formats are valid for a given type of memory unit. The valid combinations are:

- For bytes:
  c — unsigned ASCII character
  d — decimal
  o — unsigned octal
  u — unsigned decimal
  x — unsigned hexadecimal
  B — binary
  L — logical

- For halfwords:
  - `d` — decimal
  - `o` — unsigned octal
  - `u` — unsigned decimal
  - `x` — unsigned hexadecimal
  - `B` — binary
  - `L` — logical
- For words:
  - `d` — decimal
  - `e` — scientific notation
  - `f` — floating point
  - `o` — unsigned octal
  - `u` — unsigned decimal
  - `x` — unsigned hexadecimal
  - `B` — binary
  - `L` — logical
- For longwords:
  - `d` — decimal
  - `e` — scientific notation
  - `f` — floating point
  - `o` — unsigned octal
  - `u` — unsigned decimal
  - `x` — unsigned hexadecimal
  - `B` — binary
  - `C` – Fortran complex
  - `L` — logical
- For quadwords:
  - `e` — scientific notation
  - `f` — floating point
  - `o` — unsigned octal
  - `x` — unsigned hexadecimal
  - `B` — binary
  - `C` – Fortran complex
  - `L` — logical

Examples

The following examples illustrate how to display the contents of memory.

```
(CXdb) examine loc(ARRAY)
Examine Process [#0/0] from 0x8008ace8 to 0x8008ad34
 8008ace8:    00000000  00000002  00000003  00000004
 8008acf8:    00000002  00000004  00000006  00000008
 8008ad08:    00000003  00000006  00000009  0000000c
 8008ad18:    00000004  00000008  0000000c  00000010
 8008ad28:    00000005  00000000  00000005  00000000
```

The above command displays the region of memory beginning at the starting location of `ARRAY` in the current process. The Fortran function `loc()` provides the starting address of `ARRAY`. Since the memory units and display format are not specified, the command displays 20 memory units of default size (in this case, words) in the default format (in this case, hexadecimal) for the current process.

```
(CXdb) examine/bd loc(ARRAY):8
Examine Process [#0/0] from 0x8008ace8 to 0x8008acef
 8008ace8:       0     0     0     0     0     0     0     2
```

The above command displays `8` bytes (b) beginning at the starting location of `ARRAY` in the current process. The display format is decimal (d). Note that no white space is allowed between the command and the specification `/bd`.

Related Commands

`disassemble`	`display disassembly`
`info cxdb`	`info formatting`
`set default format`	`set default fpmode`
`set default memory`	`set format`
`set fpmode`	`set memory`

Related Parameters

displaying data	language-expression
process-list	thread-list

Related Windows

Memory Display window

executable

exe

Specify a new executable file to provide debugging data and the executable image.

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process object.
<remote-host>	The name of the remote host (C Series only). The name can be either an absolute Internet address or a name in the /etc/hosts file.
<file-name>	The name of the executable file. Relative path names use the console working directory as a base.

Description

The `executable` command specifies the executable file to use as the basis for the process object's CTI information. The executable file is also used to create an executable image, from which new processes are created.

Any existing debugging information or executable image is removed from the process object, and replaced by the new information and executable image. The process object must already have been created using one of the three `debug` commands.

If the executable file is on the local machine, the directory in which CXdb finds the executable file is added to the search path of the process object.

CXdb uses the search path to find the CTI (Compiler-Tools Interface) data files and source files specified in the executable file. The CTI data files provide all the debugging information for your program. These files exist only if the executable file was compiled with the `-cxdb` option of the CONVEX Fortran or C compilers. The .CTI directory, which contains the CTI data files, can be located in any directory on the search path.

You can specify a remote executable by preceding the executable name with the name of the remote host (C Series only), a colon, and the path to the executable file. Relative path names use the remote working directory as a base. For more information, refer to the "remote debugging" page.

Examples

The following example brings a new executable file into a process object.

```
(CXdb) executable para

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

The above command establishes the executable file named `para` located in the console working directory as the basis for the CTI information of the process object. The directory where the file was found is added to the search path of the process object.

An executable image is also created from this executable file. If neither a process image nor a core image exists in the process object, the executable image is now the image being debugged.

```
(CXdb) executable ben:/usr/smith/a.out

Default source file: ./prog.f
Default source language: Fortran
```

The above command uses the executable file located on the remote host named `ben` as the basis for the CTI information in the process object. An executable image is also created from this executable file.

The `run` command would now create a new process on the remote host named `ben` from this executable image.

Related Commands

`attach`	`core`
`debug core`	`debug exec`
`debug proc`	`detach`
`info cxdb`	`info process`
`run`	`rerun`

Related Concepts

process object	remote debugging

Related Parameters

file-name	process-list

fill
fil

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

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.
<memory-unit>	The type of memory unit to be filled. The possible types are: **b** — byte (8 bits) **h** — halfword (16 bits) **w** — word (32 bits) **l** — longword (64 bits) **q** — quadword (128 bits) If you do not specify a memory unit, CXdb uses the default memory unit for the specified memory region.
<starting-address>	The starting address of the memory region to be filled. This can be any *<language-expression>* that evaluates to a valid address.
<ending-address>	The ending address of the memory region to be filled. This can be any *<language-expression>* that evaluates to a valid address.
<unit-count>	The number of memory units to be filled. This can be any *<language-expression>* that evaluates to a positive integer. The default count is all units in the memory region specified by the starting address.

<language-expression>	Any valid expression in the source language. The resulting value of the expression is filled, or written, into the memory units of the specified memory region.

Description

The `fill` command fills the specified memory region with the value of the specified language expression. You can use the `info expression` command to determine the base size and total number of memory units for an expression.

You can view the results of the `fill` command for the specified memory region using the `examine` command or the `print` command..

Caution

If you do not specify the memory region properly with this command, it could result in overwriting unprotected areas of process memory that you do not want to change.

Examples

The following examples illustrate how to fill a region of memory with the result of a language expression.

```
(CXdb) fill ARRAY \; 3
```

The above command fills `ARRAY` with the value `3`. Since the command does not specify the number of memory units to fill, all elements of `ARRAY` are filled. The size of an element determines the default memory unit for how the fill takes place. For example, if each element of `ARRAY` is a word, then the value `3` is written to each word of the array. On the other hand, if each element of `ARRAY` is a byte, then the value `3` is written to each byte of the array.

Note that a delimiter (`\;`) is required to separate the memory region address from the fill value.

```
(CXdb) fill /w ARRAY:10 \; i+5
```

The above command fills the first `10` words (specified by `/w`) of `ARRAY` with the value of the language expression `i+5`. The fill takes place by words of memory, and each word is four bytes (32 bits). If each element of `ARRAY` is also a word, then the above command fills the first 10 elements of `ARRAY` with the value of `i+5`.

```
(CXdb) fill /b '800015da'x..'8000161a'x \; 1
```

The above command writes the value `1` into each byte (specified by `/b`) of memory in the address range `800015da` to `8000161a`.

Related Commands

`copy`
`disassemble`
`examine`
`info expression`
`print`

Related Concepts

C language expressions
Fortran language expressions
language expressions
modifying data

Related Parameters

language-expression
process-list
thread-list

find memory backward

find m b
fmb

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

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.
<memory-unit>	The type of memory unit to search. The memory unit specifications are: **b** — byte (8 bits) **h** — halfword (16 bits) **w** — word (32 bits) **l** — longword (64 bits) **q** — quadword (128 bits)
<byte-pattern>	The pattern to search for. It is expressed as a hexadecimal number. Because it takes two hexadecimal digits to represent one byte, the specified byte pattern must contain an even number of hexadecimal digits.
<lowest-address>	The starting (lowest) address of the memory region to search. It can be any *<language-expression>* that evaluates to a valid address.
<highest-address>	The ending (highest) address of the memory region to search. It can be any *<language-expression>* that evaluates to a valid address.
<byte-count>	The total number of bytes in the memory region to search. It can be any *<language-expression>* that evaluates to a positive decimal integer. The byte count added to the lowest address yields the highest address of the memory region.

find memory backward

Description

The `find memory backward` command searches backward in the specified memory region to find the specified byte pattern.

Because the search is backward, it starts at the highest address of the specified region and proceeds toward the lowest address. The command responds by listing the first address where the specified byte pattern is found.

Examples

The following examples illustrate how to search memory for a particular byte pattern.

```
(CXdb) find memory backward ff '80001164'x:100
Data found at 0x800011bb
```

The above command searches for the byte pattern `ff`. It searches backward through the 100-byte memory region at address `80001164`. In other words, the search starts at address `800011c8` (`80001164+64` hex) and proceeds toward address `80001164`. The first location where it finds the byte pattern is at address `800011bb`.

```
(CXdb) find memory backward/w 2b09 PRINT_ARRAY..CLEAR_ARRAY
Data not found within memory range [0x80005506..0x80005694]
```

The above command searches for the byte pattern `2b09`. It searches backward through the memory region that extends from the routine called `PRINT_ARRAY` to the routine called `CLEAR_ARRAY`. Because the command specifies a memory unit of words (`/w`), the search looks at only the first two bytes (four hexadecimal digits) of each word in the specified memory region. In this case, the byte pattern is not found anywhere in the specified region.

Related Commands

`disassemble`
`examine`
`find memory forward`
`print`

Related Concepts

process object

Related Parameters

language-expression
process-list
thread-list

find memory forward

find m f

fmf

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

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.
<memory-unit>	The type of memory unit to search. The memory unit specifications are: **b** — byte (8 bits) **h** — halfword (16 bits) **w** — word (32 bits) **l** — longword (64 bits) **q** — quadword (128 bits)
<byte-pattern>	The pattern to search for. It is expressed as a hexadecimal number. Because it takes two hexadecimal digits to represent one byte, the specified byte pattern must contain an even number of hexadecimal digits.
<starting-address>	The starting address of the memory region to search. It can be any *<language-expression>* that evaluates to a valid address.
<ending-address>	The ending address of the memory region to search. It can be any *<language-expression>* that evaluates to a valid address.
<byte-count>	The total number of bytes in the memory region to search. It can be any *<language-expression>* that evaluates to a positive decimal integer. The byte count added to the starting address yields the ending address of the memory region.

find memory forward

Description

The `find memory forward` command searches forward for the specified byte pattern in the specified memory region. The command responds by listing the first address where the specified byte pattern is found.

Examples

The following examples illustrate how to search memory for a particular byte pattern.

```
(CXdb) find memory forward ff '80001164'x:100
Data found at 0x80001170
```

The above command searches for the byte pattern `ff`. It searches through the 100-byte memory region that starts at address `80001164` in the current process. The first location where it finds the byte pattern is at address `80001170`.

```
(CXdb) find memory forward/w 2b09 PRINT_ARRAY..CLEAR_ARRAY
Data not found within memory range [0x80005506..0x80005694]
```

The above command searches for the byte pattern `2b09`. It searches through the memory region that starts at the routine called `PRINT_ARRAY` and ends at the routine called `CLEAR_ARRAY` in the current process. Because the command specifies a memory unit of words (`/w`), the search looks at only the first two bytes (four hexadecimal digits) of each word in the specified memory region. In this case, the byte pattern is not found anywhere in the specified region.

Related Commands

`disassemble`
`examine`
`find memory backward`
`print`

Related Concepts

process object

Related Parameters

language-expression
process-list
thread-list

find source

find s

fs

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

Syntax

`find source` <*string*> [<*file-name*>]

Parameter	Meaning
<*string*>	The character string to search for. The string can be any grep-style regular expression. To perform a literal match of a regular expression metacharacter (that is, `. * ? [ ^ $` or `\`), precede the metacharacter with a backslash (\). Refer to the regexp(1) man page for more information. If the string contains white space, enclose it in quotation marks.
<*file-name*>	The name of the source file to search. If no source file is specified, then the last file specified with the `find source` command is searched. If no file has been specified with the `find source` command, then the source file containing the main routine of the program is searched.

Description

The `find source` command shows all occurrences of the specified string in a source file.

The output of the `find source` command includes the line number and text of each line that contains the specified string. If no matches are found, nothing is displayed.

In line mode (when CXdb is invoked with the `-nw` option), the output of this command is sent through your pager.

Examples

The following examples illustrate how to search for all occurences of a character string in a source file.

```
(CXdb) find source ARRAY example.f
2: INTEGER ARRAY(4,4)
11: ARRAY(I,J)=I*J
18: CALL CHAPTER4(ARRAY)
19: CALL CHAPTER5(ARRAY)
20: CALL CHAPTER6(ARRAY)
21: CALL CHAPTER7(ARRAY)
22: CALL chapter7c(ARRAY)
23: CALL CHAPTER8(ARRAY)
24: CALL CHAPTER13F(ARRAY)
25: CALL chapter13c(ARRAY)
35: SUBROUTINE PRINT_ARRAY(ARRAY)
36: INTEGER ARRAY(4,4)
42: PRINT 99, ARRAY(I,1), ARRAY(I,2), ARRAY(I,3), ARRAY(I,4)
50: SUBROUTINE CLEAR_ARRAY(ARRAY)
51: INTEGER ARRAY(4,4)
55: ARRAY(I,J)=0
```

The above command searches for all occurrences of of the string `ARRAY` in the source file example.f. The output displays the line number and text of each line in the source file that contains a match.

```
(CXdb) find source [eE]nd
```

The above command searches for all occurrences of the strings `end` and `End` in the source file example.f, since example.f was the last source file specfied with the `find source` command. No matches were found, so nothing is displayed.

```
(CXdb) find source bld_matrix chapter7C.c
15: bld_matrix(4,4,5,table);
27:void bld_matrix(n,m,l,slice)
```

The above command shows all occurrences of the string `bld_matrix` in the source file chapter7C.c.

```
(CXdb) find source "IF (NUMARGS" example.f
17: IF (NUMARGS .EQ. 0) THEN
```

The above command shows all occurrences of the string `"IF (NUMARGS"` in the source file example.f. The specified string is enclosed in quotes since the string contains white space.

```
(CXdb) find source "PRINT \*" example.f
7: PRINT *, "EXAMPLE PROGRAM STARTED"
31: PRINT *, "EXAMPLE PROGRAM FINISHED"
39: PRINT *
40: PRINT *, "THE TABLE:"
45: PRINT *
65: PRINT *, "The process interface window handles program input."
66: PRINT *, "As an example, enter your first name and press return."
```

The above command finds all occurrences of the string `PRINT *` in the source file example.f. The string is enclosed in quotes because it contains a white space. The backslash preceding the `*` forces a literal match of the `*` regular expression metacharacter.

Related Commands `display routine` `list`

Related Parameters string file-name

Related Windows Search Source Code dialog Source Code window

finish
fini

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

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.

<granularity>	The type of source unit, or step size. Available granularities are:
	`routine`
	`block`
	`loop`
	`statement`
	`expression`
	If you do not specify a granularity, CXdb uses the default granularity of the specified process.

&	Runs the command in the background.

Description

The `finish` command is a stepping command that completes execution of the innermost active source unit of the specified granularity. Execution stops at the next source unit of default granularity.

The innermost active source unit is the one of specified granularity whose address range (or extent) includes the current value of the program counter (PC). For example, the innermost active loop is the one that contains the current PC.

Examples

The examples shown below relate to the following Fortran source code, which has been compiled at optimization level -no:

```
1       SUBROUTINE BLD_MATRIX(N,M,L,SLICE)
2       INTEGER SLICE(N,M)
3       REAL MATRIX(5,5,5)
4
5       DO I = 1, 5
6         DO J = 1, 5
7           DO K = 1, 5
8             MATRIX(I,J,K) = 0
9           ENDDO
10        ENDDO
11      ENDDO
12      IF (N .GT. 5) THEN N=5
13      IF (M .GT. 5) THEN M=5
14      IF (L .GT. 5) THEN L=5
15      DO I = 1, N
16        DO J = 1, M
17          DO K = 1, L
18            MATRIX(I,J,K) = 3*K/7.5 - 29.7/(I+J+K) + SLICE(I,J)
19            PRINT 99, "I=",I,"J=",J,"K=",K,"MATRIX(I,J,K)=",MATRIX(I,J,K)
20          ENDDO
21          PRINT *
22        ENDDO
23        PRINT *
24      ENDDO
25  99  FORMAT (A, I2, 3X, A, I2, 3X, A, I2, 5X, A, F8.3)
26      RETURN
27      END
```

Assume that the default stepping granularity is `statement`. Also assume that the process is stopped, and the program counter (PC) points to the beginning of line 18.

```
(CXdb) finish
Finishing innermost statement in Process [#0/*]
Process [#0/0] stopped nexting at [0x80002b70] BLD_MATRIX in chapter7F.f line 19
```

Because `statement` is the default granularity, the above command finishes the innermost active statement (line 18). Also because of the default granularity, execution stops at the next statement (line 19). The PC now points to the beginning of line 19.

```
(CXdb) finish loop
Finishing innermost loop in Process [#0/*]
Process [#0/0] stopped nexting at [0x80002c6e] BLD_MATRIX in chapter7F.f line 21
```

The above command finishes the innermost active loop that contains the current PC (line 19). That loop begins on line 17 and ends on line 20. Because `statement` is the default granularity, execution stops at the next statement after line 20, which is on line 21.

```
(CXdb) finish loop
Finishing innermost loop in Process [#0/*]
Process [#0/0] stopped nexting at [0x80002cb0] BLD_MATRIX in chapter7F.f line 23
```

The above command finishes the innermost active loop that contains the current PC (line 21). That loop actually begins on line 16 and ends on line 22. Because `statement` is the default granularity, execution stops at the next statement, which is on line 23.

Assume that the default stepping granularity for this process has been changed to `loop`, and the process is stopped with the PC pointing at the beginning of line 12. You enter the following command:

```
(CXdb) finish loop
Finishing innermost loop in Process [#0/*]
Process [#0/0] stopped nexting at [0x80002a9a] BLD_MATRIX in chapter7F.f line 15
```

The above command finishes the innermost active loop at line 12, which is just a single statement. However, because the default granularity is now `loop`, execution does not stop at line 13. Instead, it continues until the process reaches the next loop after line 12. Thus, when the process stops, the PC points to the beginning of the loop on line 15.

Related Commands

`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
	stepping	

Related Parameters		
	granularity	process-list
	thread-list	

frame

fr

f

Change the current stack frame.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.

<frame-specifier>	A relative or absolute frame number.
	
	There are two default aliases for relative frame references:
	
	`down` — Alias for `"frame -1"`
	`up` — Alias for `"frame +1"`

Description

The `frame` command selects a particular frame from the process stack and makes it the current frame. The current frame defines the current scope for the process.

This command enables you to change the context of the process for symbol mapping. However, the context for process execution is not changed by this command.

NOTE: Selecting a stack frame with the `frame` command will affect the way CXdb interprets program symbols and identifiers used in subsequent CXdb commands. Refer to the "scope" reference page for more information.

Examples

The following examples illustrate how to change the current scope by using the `frame` command.

```
(CXdb) frame 1
  1 : 0x80002948  in CHAPTER7(ARRAY = (INTEGER*4(1:4, 1:4)) )  (chapter7F.f line 10)
```

The above command selects frame 1 as the current frame. The response indicates that this frame has an execution address of `80002948`, which is at line 10 in the file called `chapter7F.f`. The frame represents the routine `CHAPTER7`, which uses the values of a 4 x 4 matrix called `ARRAY`. Any unqualified identifiers used in subsequent CXdb commands are interpreted from the scope of this reference point in frame 1. However, execution of the process still continues from the top of the stack, which is frame 0.

```
(CXdb) frame -1
  0 : 0x800029bc  in BLD_MATRIX(N = (INTEGER*4) 4, M = (INTEGER*4) 4, L = (INTEGER*4)
      5, SLICE = (INTEGER*4(1:<TEMP0>, 1:<TEMP1>)) )  (chapter7F.f line 26)
```

The above command uses a relative frame number of `-1`. It sets the current frame back to frame 0 in this case.

```
(CXdb) up
  1 : 0x80002948  in CHAPTER7(ARRAY = (INTEGER*4(1:4, 1:4)) )  (chapter7F.f line 10)
```

The above command uses `up` as the alias for `frame +1`. It sets the current frame to frame 1 in this case.

Related Commands

`backtrace`
`info frame`
`info frame at`
`info locals`
`info process`
`info stack`

Related Concepts

scope

Related Parameters

frame-specifier
process-list
thread-list

Related Windows

Stack Frame Description dialog
Stack Trace window

gdb
gdb

Enable compatibility with gdb debugger commands.

Syntax

```
gdb
```

Description

The gdb command incorporates a set of predefined aliases for gdb debugger commands. With the predefined aliases incorporated, you can type in a gdb debugger 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 gdb command is not aliased and, where possible, suggests a CXdb command with the closest functionality 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`
`info comm-registers`	`info cregisters` (C Series only)
`info directories`	`info process`
`info display`	`info event`

gdb command	CXdb equivalent
`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 `edit` 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`
`set parallel off`	No equivalent.
`set parallel on`	`clear fixed sched`
`set pipeline off`	`set seq`
`set pipeline on`	`clear seq`
`set prettyprint`	No equivalent.
`set unionprint`	No equivalent.
`set prompt`	Done in .Xdefaults file.
`set screensize`	No equivalent.
`set verbose off`	No equivalent.
`set verbose on`	No equivalent.
`symbol-file`	Done automatically as needed.
`tbreak`	Use an eventpoint handler.

gdb command	CXdb equivalent
`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`

Examples

The following example illustrates how to incorporate the predefined aliases for gdb debugger commands.

```
(CXdb) gdb
```

After executing the above command, you can enter gdb debugger commands directly in the CXdb Command window.

To display a list of current CXdb aliases, use the `info alias` command.

Related Commands

`csd`	`cxdb`
`info alias`	`source`

Related Concepts

csd debugger

gdb debugger

C Series only

get
ge

Restore the contents of memory regions from a binary file.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes used in evaluating this command. The default is the current process.
<thread-list>	A list of threads used in evaluating this command. The default is all threads of the current process.
<file-name>	The name of the binary file from which to retrieve the memory contents. The file you specify must have been created with the `put` command. Relative path names use the console working directory as a base.
<starting-address>	The first address to load. This can be any *<language-expression>* that evaluates to a valid address in the syntax of the current source language. If the starting address is not followed by an ending address or byte count, CXdb determines the size of the memory region based on the type of structure at the starting address.
<ending-address>	The last address to load. This can be any *<language-expression>* that evaluates to a valid address in the syntax of the current source language. It must be preceded by two dots (**..**).
<byte-count>	The number of bytes to load. This can be any *<language-expression>* that evaluates to a positive integer in the syntax of the current source language. It must be preceded by a colon (**:**).
[**\;** ...]	An optional list of memory regions. Multiple memory regions are separated by the language expression terminator (**\;**).

Description

The `get` command reads the specified binary file and loads each memory region of the file into the corresponding memory regions specified on the command line. The `get` command can read only binary files created with the `put` command. If no memory regions are specified, CXdb attempts to load the variables back into the memory regions from which they originally came. Therefore, you should be at an equivalent scope in the process before issuing the `get` command.

If more memory regions than exist in the file are specified with the `get` command, an error occurs. If fewer memory regions are specified, only the necessary number of memory regions are loaded.

The `get` command can be used to restore Fortran common blocks and C structures. Fortran common block names can be given as address expressions by delimiting each block name with a slash (/).

The `get` command can be used to restore array slices saved with the `put` command; however, the contents must be restored into a contiguous memory region.

Caution

CXdb does not check for architectural data type dependencies when restoring data. If the `get` command is used to restore a floating point variable using IEEE format into a floating point variable using native format, the resulting value will be incorrect.

Examples

The examples below relate to the following Fortran source code, which has been compiled at optimization level `-no`.

```
SUBROUTINE SUB13B
CHARACTER*6 NAME
INTEGER*1 NUM(6), I
REAL AVG
COMMON /BLK/ NAME, NUM, AVG

SUM = 0
DO I=1,6,2
    SUM = SUM + (NUM(I)-48)*10 + (NUM(I+1) - 48)
ENDDO

AVG = SUM / 3

END
```

The examples below use the `get` command with the above Fortran source code.

```
(CXdb) get file1
```

The above command reads the contents of the file named file1 and restores the contents to the memory regions from which they came.

```
(CXdb) get file1 /BLK/:8
```

The above command reads file1 and restores the contents of the file to the first 8 bytes of common block `BLK`. If the size of the memory region in the file is larger than the size of the common block, CXdb reads only enough of the file to fill the common block.

```
(CXdb) get file3 loc(NUM(1))..loc(NUM(3))
```

The above example restores the contents of file3 to the specified memory region. The memory region extends from the starting address of array element `NUM(1)` to the *starting* address of element `NUM(3)`. Thus, only 2 array elements are restored, `NUM(1)` and `NUM(2)`. The Fortran `loc()` function provides the starting addresses of the array elements.

```
(CXdb) get file4 NUM:3
```

The above example restores the contents of file4 to the memory region beginning at the starting address of the array `NUM` and extending for 3 bytes (3 elements of the array, because each element is 1 byte long).

```
(CXdb) get file5 loc(NAME) \; loc(AVG)
```

The above command reads the values of the variables `NAME` and `AVG` from the file5 file. The language expression terminator (`\;`) separates the two language expressions. The Fortran `loc()` function provides the addresses of the variables.

Because only a starting address is specified for each variable, the size of the memory region is automatically set to the size of the variable.

Related Commands		
	`examine`	`put`

Related Concepts		
	C language expressions	console working directory
	Fortran language expressions	language expressions
	modifying data	saving data

Related Parameters		
	array-slice	file-name
	process-list	thread-list

goto address

g a

Branch to the specified address.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the current process.
<language-expression>	An expression that evaluates to a valid address in the default language of the specified process. The program counter (PC) is set to this address.

Description

The `goto address` command sets the program counter (PC) to the specified address. When you continue execution of the process, it branches unconditionally to the specified address. The process stack is *not* updated.

The address specified in this command must be the beginning of a machine instruction. If it is not, an error will result.

You can display the current value of the PC by using the `print $PC` command.

Caution

The `goto address` command drastically changes the order of execution for your program. Because of this, it can lead to unpredictable results, particularly with optimized code.

Examples

The following examples illustrate how to modify the PC with the `goto address` command.

```
(CXdb) goto address '8000289a'x
```

The above command sets the PC to the address `8000289a` (expressed in Fortran syntax). It affects all threads of the current process.

```
(CXdb) goto address 0x8000289a
```

The above command sets the PC to the address `8000289a` (expressed in C syntax). It affects all threads of the current process.

```
(CXdb) goto address BLD_MATRIX
```

The above command sets the PC to the starting address of the routine `BLD_MATRIX`. It affects all threads of the current process.

Related Commands

`disassemble`	`goto line`
`goto source`	`info frame`
`info line`	`info registers`
`info sourceunit`	`info stack`

Related Concepts

scope

Related Parameters

language-expression
process-list
thread-list

goto line

g l

Branch to the specified source line.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the current process.
<line-specifier>	The line specifier for the line of source code to branch to. The line specifier can include a source file name as well as the line number. (Line numbers are assigned at compilation time and are displayed in the Source Code window.) The program counter (PC) is set to the starting address of the specified line.

Description

The `goto line` command sets the program counter (PC) to the starting address of the specified line of source code. When you continue execution of the process, it branches unconditionally to the specified line. The process stack is *not* updated.

The source line specified in this command must contain a valid source unit of statement granularity. Blank lines, comment lines, and lines removed by optimization do not contain valid statement source units. Using the number of such a line in the `goto line` command results in an error.

You can display the current value of the PC by using the `print $PC` command.

Caution

The `goto line` command drastically changes the order of execution for your program. Because of this, it can lead to unpredictable results, particularly with optimized code.

Examples

The following examples illustrate how to modify the PC with the `goto line` command.

```
(CXdb) goto line 35
```

The above command sets the PC to the starting address of line 35 of the current source file. It affects all threads of the current process.

```
(CXdb) goto line chapter7C.c:35
```

The above command sets the PC to the starting address of line 35 in the source file chapter7C.c.

Related Commands

`disassemble`
`display routine`
`edit`
`goto address`
`goto source`
`info frame`
`info registers`
`info stack`

Related Concepts

scope

Related Parameters

line-specifier
process-list
thread-list

goto source

g s

Branch to the specified source unit.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the current process.
<source-unit>	The identifier of the source unit to branch to. The source unit identifier can include a source file name as well as the source unit number. The program counter (PC) is set to the starting address of the specified source unit.

Description

The `goto source` command sets the program counter (PC) to the starting address of the specified source unit. When you continue execution of the process, it branches unconditionally to the specified source unit. The process stack is *not* updated.

Each source unit has a unique identification number assigned to it by the compiler when you compile your program with the `-cxdb` option. You can use the `info line` command to display the source unit numbers for all source units that appear on a given line of source code.

The particular source unit specified in the `goto source` command must have executable object code associated with it. Some source units that have been removed by optimization do not have executable object code associated with them. Therefore, using such source units in the `goto source` command will result in errors.

You can display the current value of the PC by using the `print $PC` command.

Caution

The `goto source` command drastically changes the order of execution for your program. Because of this, it can lead to unpredictable results, particularly with optimized code.

Examples

The following examples illustrate how to modify the PC with the `goto source` command.

```
(CXdb) goto source 92
```

The above command sets the PC to the starting address of source unit `92` in the current source file. It affects all threads of the current process.

```
(CXdb) goto source chapter7C.c:92
```

The above command sets the PC to the starting address of source unit `92` in the source file `chapter7C.c`.

Related Commands

`disassemble`
`goto address`
`goto line`
`info frame`
`info line`
`info registers`
`info sourceunit`
`info stack`

Related Concepts

scope
source units

Related Parameters

process-list
source-unit
thread-list

help

h

?

Invoke the CXdb help system.

Syntax

help [<*string*>]

Parameter	Meaning
<*string*>	A character string used to search for help topics. All topic titles that contain the string are listed. The string can contain white space without being enclosed in quotes.

Description

The `help` command invokes the CXdb help system. The help system consists of numerous topics and associated text files that describe those topics.

The help topics cover the following categories:

- **Commands** — Full descriptions of every CXdb command, complete with examples.
- **Concepts** — Explanations of the terminology and major concepts associated with CXdb.
- **Parameters** — Detailed descriptions of the more complex parameters that can be used with various CXdb commands.
- **Messages** — Text and descriptions of messages that are generated as responses to CXdb commands. The messages are listed by their code numbers.
- **Windows** — Text, illustrations, and descriptions of CXdb windows, menus, and dialogs.

In X Windows mode, once you have invoked the help system with the `help` command, you can request help on a related topic simply by selecting that topic from the Help window. Refer to the "Help window" page for an explanation of how to use this window.

Examples

The following examples illustrate different methods of requesting help from CXdb.

```
(CXdb) help
```

The above command invokes the help system and displays text that describes the contents of the help system.

```
(CXdb) help break routine
```

The above command invokes the help system and brings up the text that describes the individual topic called `break routine`.

```
(CXdb) help break
```

The above command invokes the help system and displays a list of topics whose names contain the string `break`.

```
(CXdb) help set default
```

The above command invokes the help system and displays a list of topics whose names contain the string `set default`.

```
(CXdb) help A121
```

The above command invokes the help system and brings up the text that describes CXdb message `A121`.

Related Parameters

string

Related Menus

Help menu

Related Windows

Help window

if

if

Establish conditional execution of CXdb commands.

Syntax

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

Parameter	Meaning
<relational-expression>	A language expression whose evaluation determines the set of commands to execute (if any). The language expression must evaluate to TRUE or FALSE.
<command-set>	One or more CXdb commands. Each command must be terminated with a semicolon (;). If more than one command is used, the entire set must be enclosed in curly braces ({ }).

Description

The `if` command causes a particular set of CXdb commands to execute based on the value of a relational expression.

If the relational expression is TRUE, the first set of commands execute. If it is FALSE, and there is an `else` clause, the second set of commands execute. If it is FALSE, and there is not an `else` clause, the `if` command is finished.

The `if` command can be used to control the flow of execution inside of a command file or eventpoint handler. It can also be used on the CXdb command line.

Each command in a set must terminate in a semicolon (;). If more than one command is part of a set, all of the commands in the set must be enclosed in curly-braces ({}).

Examples

The following examples use the `if` command to control the flow of execution in an eventpoint handler set with the `set handler` command. The syntax used in the relational expressions is Fortran-specific.

```
(CXdb) set handler * {if ($signal .eq. 2) {evaluate $signal=0; resume;};}
```

The above command defines an eventpoint handler for all existing eventpoints. The handler consists of one command, the `if` command. Because the `if` command is part of an eventpoint handler, and all commands of a handler must end with a semicolon, the `if` command terminates with a semicolon.

The relational expression of the `if` command tests the value of the debugger variable `$signal`, which holds the number of the current signal. If the current signal has a number of 2 (the signal `SIGINT`), the first set of commands is executed. This set of commands changes the value of `$signal` to zero and then resumes process execution.

This handler causes the signal `SIGINT` to be ignored and process execution to resume.

Note that because the set consists of two commands (`evaluate` and `resume`), it is enclosed in curly braces ({ }).

```
(CXdb) set handler * {if ($signal .eq. 2) {evaluate $signal=0; resume;}
       else resume;;}
```

The above command again defines an eventpoint handler for all existing eventpoints. The `if` command now has an `else` clause.

If `$signal` is equal to 2, `$signal` is set to zero, and process execution resumes. If `$signal` does not equal 2, process execution resumes.

As with the previous example, this handler causes the signal `SIGINT` to be ignored and process execution to resume. However, with this handler, if the signal caught is not `SIGINT`, process execution resumes.

The following example sets the same eventpoint handler as the one above, but uses C syntax.

```
(CXdb) set handler * {if ($signal == 2) {evaluate $signal=0; resume;} else
       resume;;}
```

The above command defines an eventpoint handler as in the previous example. The relational expression uses C syntax rather than Fortran.

The following example uses the `if` command as it might appear in a command file. The entire command file is shown.

```
debug exec a.out
break line 10
run
if (A .lt. 1500) {\
   echo 'Number not large enough';\
   echo 'Increase X factor';}\
else {\
   echo 'Number large enough';\
   source cmdfile.2;}
echo 'command file finished'
```

The above command file demonstrates one possible use of the `if` command. The `if` command checks if `A` is less than 1500. If it is, the messages echo and the command file finishes.

If `A` is not less than 1500, the message echoes and the cmdfile.2 command file is sourced. Using `if` commands, you can control the flow of command files.

Related Commands

`evaluate`
`resume`
`set default handler`
`set handler`
`set typehandler`
`source`

Related Concepts

command files
debugger variables
eventpoints
eventpoint handlers
initialization files

Related Parameters

language-expression

info alias

in al

i al

Display aliases.

Syntax

info alias [*<regular-expression>*]

Parameter	Meaning
<regular-expression>	A search pattern specified as a regular expression. All aliases whose names match the specified search pattern are displayed.

Description

The `info alias` command displays the current definitions of existing aliases.

An alias definition remains in effect only during the current debugging session. Therefore, if you have a set of aliases that you want to use regularly, you should define them in a CXdb command file or initialization file.

Examples

The following examples illustrate how to display the current alias definitions.

```
(CXdb) info alias

!         "recall"
.         "source"
?         "help"
args      "info args"
b?        "info break"
bi        "break instruction"
bl        "break line"
          .
          .
          .
whatis    "info expression"
where     "info scope"
x         "examine"
```

The above command displays the current definitions of all existing aliases. In this case, the response lists all default aliases created by the default initialization file. The vertical ellipsis has been added to indicate that some of the output is omitted from the example.

```
(CXdb) info alias p
p         "print"
p+        "add path"
p-        "remove path"
p=        "set path"
p?        "info process"
```

The above command displays all the aliases whose names start with the letter *p*.

Related Commands

`alias` `macro`
`remove alias`

Related Concepts

command files initialization files

Related Parameters

regular-expression

info args

in ar

args

Display arguments of the current routine.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.

Description

The `info args` command displays information about the arguments of the current routine or function. The current routine is indicated by the current stack frame of the specified process.

For each argument, the following information is displayed:

- Argument name
- Data type
- Current value (or, for arrays, the number of elements and starting address)

Examples

The following example illustrates how to display information about arguments of the current function or routine.

```
(CXdb) info args
Process [#0/0]
Frame : 0; [0x800029bc] BLD_MATRIX in chapter7F.f line 26
Number of arguments : 4
     1 : N= (INTEGER*4) 4
     2 : M= (INTEGER*4) 4
     3 : L= (INTEGER*4) 5
     4 : SLICE= INTEGER*4(1:<TEMP0>, 1:<TEMP1>) 0x80077058
```

The above command displays the arguments for the current routine for all threads of the current process. The response shows that the current routine is `BLD_MATRIX`, which is in the source file called chapter7F.f. The current value of the program counter (PC) is `800029bc`, which is the address of the current source unit in line 26 of chapter7F.f. Frame 0 is the current frame, and it has four arguments. The names of the arguments are `N, M, L, and SLICE.`

`SLICE` is a two-dimensional, adjustable Fortran array. Each element of the array is a four-byte integer, and the starting address of the array is `80077058.` `<TEMP0>` and `<TEMP1>` are temporary variables created by the Fortran compiler to store the dimensions of the array (the compiler does this because `SLICE` is an adjustable array, and the values of `N` and `M` that define its dimensions can be changed within the `BLD_MATRIX` subroutine).

Each of the other arguments is a single four-byte integer. The current value of `N` is 4; `M` is 4; and `L` is 5.

Related Commands

`backtrace`
`info expression`
`info frame`
`info frame at`
`info symbols`
`info locals`
`info scope`
`info stack`
`print`

Related Parameters

process-list
thread-list

info break

in br

b?

Display all existing breakpoints.

Syntax

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

Parameter	Meaning
<*process-list*>	A list of processes affected by this command. The default is the current process.
<*thread-list*>	A list of threads affected by this command. The default is all threads of the current process.

Description

The `info break` command displays information about all existing breakpoints.

A small table displays the number, enabled setting, ignore count, process and thread numbers, instruction address, and symbolic location for each breakpoint. If the breakpoint has its own handler, the commands of the handler are displayed below the breakpoint.

Examples

The following examples display all existing breakpoints.

```
(CXdb) info break
Event    Enabled Ignore  proc/td Address          Where
#0         y     0/0     0/*     [0x80005508] PRINT_ARRAY in example.f line 39
```

The above command displays a table of the settings of all the breakpoints. The elements in the table are described below.

- `Event` — The eventpoint number.
- `Enabled` — A `y` indicates that the eventpoint is currently enabled. An `n` indicates that the eventpoint is currently disabled.
- `Ignore` — The ignore count for this eventpoint. The number before the slash is the number of times the eventpoint has been ignored, and the number after the slash is the ignore count.

- `proc/td` — The process number and the thread numbers at which the eventpoint is set. An asterisk in the threads position indicates that the eventpoint is set for all threads of the process.
- `Address` — The instruction address where the eventpoint is located.
- `Where` — The source code location of the eventpoint. The routine, source file, and line number are displayed.

```
(CXdb) info break
Event   Enabled Ignore  proc/td Address         Where
#0         y    0/0     0/*     [0x80005508] PRINT_ARRAY in example.f line 39
#1         y    0/0     0/*     [0x80005696] CLEAR_ARRAY in example.f line 53
    {
        print $pc;
        resume;
    }
```

The above command again displays all existing breakpoints. Breakpoint 1 has its own eventpoint handler, which is displayed below the breakpoint.

Related Commands

`break instruction`	`break line`
`break routine`	`break source`
`disable event`	`disable eventtype`
`enable event`	`enable eventtype`
`info event`	`info eventtype`
`info trace`	`info watch`
`remove event`	`remove eventtype`
`set default handler`	`set handler`
`set ignore`	`set typehandler`

Related Concepts

breakpoints	eventpoints
eventpoint handlers	tracepoints
watchpoints	

Related Parameters

process-list	thread-list

SPP Series only

info control registers

in co r
i co r

Display the control registers for SPP Series machines.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.

Description

The `info control registers` command displays the contents of the control registers for the specified process. The contents are displayed in hexadecimal format. 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.

Examples

The following example illustrates how to display the contents of the control registers.

```
(CXdb) info control registers
Process [#0/0]
cr0          : 0x00000000
cr1          : 0x00000000
cr2          : 0x00000000
cr3          : 0x00000000
cr4          : 0x00000000
cr5          : 0x00000000
cr6          : 0x00000000
cr7          : 0x00000000
cr8          : 0x00000000
cr9          : 0x00000000
cr10         : 0x00000000
cr11         : 0x00000000
cr12         : 0x00000000
cr13         : 0x00000000
cr14         : 0x00000000
cr15         : 0xfffffffe
```

```
cr16          : 0x00000000
cr17          : 0x00000000
cr18          : 0x00000000
cr19          : 0x00010004
cr20          : 0x000061b0
cr21          : 0x00000000
cr22 (psw)    : 0x0004000f
cr23          : 0x00000000
cr24          : 0x00000000
cr25          : 0x00000000
cr26          : 0xffffffff
cr27          : 0x00000000
cr28          : 0x00000000
cr29          : 0x00000000
cr30          : 0x00000000
cr31          : 0x00000000
```

The above command displays the control registers for the current process. In this case, there are 32 control registers, designated as cr0 through cr31. Register cr22 is the processor status word (psw).

Related Commands

```
info floating point registers
info registers
info space registers
print
```

Related Concepts

debugger variables

registers

Related Parameters

process-list

thread-list

Related Windows

Control Registers window

C2, C3, C4 only

info cregisters

in cr

i cr

Display the communication registers for C2, C3, or C4 Series machines.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.

Description

The `info cregisters` command displays the contents of the communication registers for the specified process. The contents are displayed in hexadecimal format.

The C100 Series machines do not use communication registers. Other CONVEX computer models may use different configurations for the communication registers. For more information about these registers, refer to the *CONVEX Architecture Reference Manual (C Series)*.

Examples

The following example illustrates how to display the contents of the communication registers.

```
(CXdb) info cregisters
Process [#0/0]
 C[00] 0x0000000000000000 (0)
 C[01] 0x0000000000000008 (1)
 C[02] 0x0000000000000000 (0)
        ...

 C[63] 0x0000000000000000 (0)
```

The above command displays the communication registers for the current process. In this case, there are 64 communication registers, designated as `C[00]` through `C[63]`. The ellipsis (`...`) indicates that all of the intervening communication registers have the same contents as `C[02]`. The value in parentheses at the end of each line is the lock bit for that register (in this example, all of the registers except `c[01]` have a lock bit value of `0`).

Related Commands	`info registers`	`info vregisters`
	`print`	
Related Concepts	debugger variables	registers
Related Parameters	process-list	
Related Windows	Communication Registers window	

info cxdb

in cx

i cx

Display the status of the current CXdb session.

Syntax

```
info cxdb
```

Description

The `info cxdb` command displays the current state of the CXdb session.

Examples

The following example illustrates how to display information about the current CXdb session.

```
(CXdb) info cxdb
Current CXdb state:

     ENVIRONMENT:
             pid: 3768
             cwd: /doc/cxdb/examples
   command modes: echo on, logging off, noclobber off
          cmdout: Window #1
          cmderr: Window #1
          cmdlog:
        evalopts: fpmode = native, iprecision = 4, rprecision = 4
           shell: tcsh

PROCESS DEFAULTS:
fixed scheduling: Off
       step size: statement
   process shell: csh
          fpmode: native
     memory size: (none)
  memory formats: byte=(none), halfword=(none), word=(none)
                  longword=(none), quadword=(none)
     search path:
         .
       PROCESSES:
    process [#0]: created pid 3770, state = running, executable = docexample
                  shell = csh

 ACTIVE COMMANDS:

command [#17] - continue &
```

The above response from CXdb includes the following information:

- ENVIRONMENT — The current environment for the CXdb session. This information includes the following:
  - `pid` — Process ID for CXdb itself.
  - `cwd` — Console working directory.
  - `command modes` — The settings for echo, logging, and noclobber.
  - `cmdout` — The default list of viewports for cmdout.
  - `cmderr` — The default list of viewports for cmderr.
  - `cmdlog` — The default list of viewports for cmdlog.
  - `evalopts` — The floating point mode, integer size, and real number precision used by CXdb to evaluate language expressions.
  - `shell` — The type of shell invoked by the `shell` command.
- PROCESS DEFAULTS — Default parameters for new process objects that have not explicitly had their values set.
  - `fixed scheduling` — The state for fixed scheduling.
  - `step size` — The default step size, or granularity.
  - `process shell` — The default shell used by the process object.
  - `fpmode` — The default mode for floating point operations done by the process.
  - `memory size` — The default type of memory unit used to display memory with the `examine` command.
  - `memory formats` — The default display formats for each type of memory unit.
  - `search path` — The default search path used by the process. Dot (.) is the current directory.
- PROCESSES — Information about existing process objects.
  - `process` — The process object number.
  - `pid` — The process ID of any active processes created from the process object.
  - `state` — The current state of the process.
  - `executable` — The name of the executable file for the process object.
  - `shell` — The current shell for the process.
- ACTIVE COMMANDS — A list of commands that are currently executing in background mode.

Related Commands

`add cmderr`	`add cmdlog`
`add cmdout`	`add default path`
`clear default fixed sched`	`clear echo`
`clear logging`	`clear noclobber`
`continue`	`cxdb`
`debug core`	`debug exec`
`debug proc`	`detach`
`executable`	`info process`
`kill process`	`quit`
`remove cmderr`	`remove cmdlog`
`remove cmdout`	`remove default path`
`rerun`	`resume`
`run`	`set cmderr`
`set cmdlog`	`set cmdout`
`set default fixed sched`	`set default format`
`set default fpmode`	`set default memory`
`set default path`	`set default pshell`
`set default step`	`set evalopts fpmode`
`set evalopts iprecision`	`set evalopts rprecision`
`set logging`	`set noclobber`
`set shell`	`stop`

Related Concepts

background execution	cmderr
cmdlog	cmdout
console working directory	default search path
logging	process object
viewports	Xdefaults

Related Parameters

granularity	viewport

info default environment

in d e
i d e

Display all default environment variables.

Syntax

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

Parameter	Meaning
<regular-expression>	A search pattern. All environment variables whose names match the search pattern are displayed. The default is all environment variables.

Description

The `info default environment` command displays all variables in the default environment. The default environment is initially a copy of the environment passed to CXdb.

Examples

The following example displays information about the default environment.

```
(CXdb) info default environment
Default environment:
HOME=/usr/smith
SHELL=csh
EDITOR=vi
NEWS=rn
PAGER=less
```

The above example displays all environment variables of the default environment along with their current values.

Related Commands

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

info default environment

Related Concepts	default environment	environment
Related Parameters	regular-expression	

info dirpath

i di

List all aliases for CTI directory paths, or list the names of object files that match a regular expression.

Syntax

info dirpath [*<regular-expression>*]

Parameter	Meaning
<regular-expression>	A search pattern specified as a regular expression. A process must exist before you can specify a regular expression with this command.

Description

The `info dirpath` command can be used in two different ways:

- Without a regular expression — To list the aliases for CTI directory paths that have been created with the `dirpath` command.
- With a regular expression — To list the names of object files for the current process that match the regular expression.

A process must exist before you can specify a regular expression with the `info dirpath` command.

Examples

The following examples illustrate both ways of using the `info dirpath` command.

```
(CXdb) info dirpath
     From                        To
 1. /doc/cxdb/examples -> /usr/smith
 2. /doc/cxdb/examples -> /usr/smith/data
 3. /doc/cxdb/examples -> /usr/smith/programs
```

The above command lists all aliases for CTI directory paths that have been created with the `dirpath` command during the current debugging session. In this example, the original path to the CTI data files is /doc/cxdb/examples. The three aliases for this path are /usr/smith, /usr/smith/data, and /usr/smith/programs.

```
(CXdb) (CXdb) info dirpath c
    Object File             Directory
 1. chapter13C.o            /doc/cxdb/examples/
 2. chapter13F.o            /doc/cxdb/examples/
 3. chapter15.o             /doc/cxdb/examples/
 4. chapter4.o              /doc/cxdb/examples/
 5. chapter5.o              /doc/cxdb/examples/
 6. chapter6.o              /doc/cxdb/examples/
 7. chapter7C.o             /doc/cxdb/examples/
 8. chapter7F.o             /doc/cxdb/examples/
 9. chapter9.o              /doc/cxdb/examples/
```

The above command lists the names of all the object files for the current process that begin with the letter `c`. It also lists the directory where each object file was originally created during compilation.

Related Commands

`add path`
`add default path`
`dirpath`
`remove dirpath`
`set path`

Related Concepts

Compiler-Tools Interface
process object
search path

C Series only

info dynamicobject

in dy
i dy

Display memory segments of dynamically loaded objects.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.

Description

The `info dynamicobject` command displays a table showing the memory segments of all object files that have been dynamically loaded into memory. Object files that have been dynamically loaded can be specified to CXdb during a debugging session by using the `load object` command.

NOTE: CONVEX does not provide or support a dynamic loader. CXdb only offers the capability to load dynamic object files to support the symbolic debugging of programs which provide their own dynamic loader.

Examples

The following example illustrates how to display information on all object files loaded into memory.

```
(CXdb) info dynamicobject

Dynamically Loaded Objects for Process [#0]:

  |--------------Segment Base Addresses------------|
Text Data tdata Bss tBss Name

0xc0000110 0xc0000558 0xc0000558 0x00000000 0x80013000 pcc_block.o
```

The above command displays a table showing the addresses for the `pcc_block.o` object file that has been dynamically loaded into memory. The base addresses for the text, data, thread data, bss, and thread bss segments are shown.

info dynamicobject

Related Commands `load object`

Related Parameters file-name

info environment

in en
env?

Display all process environment variables.

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process object.
<regular-expression>	A search pattern. All environment variables whose names match the search pattern are displayed. The default is all environment variables.

Description

The `info environment` command displays the environment of a process object. If the process object does not have its own environment, the default environment is displayed, and CXdb displays the message "`(from default environment)`".

Examples

The following example displays the environment for the current process object.

```
(CXdb) info environment
Process [#0] environment: (from default environment)
PATH=/usr/local/bin:/usr/bin
PRINTER=pscript1
USER=smith
TERM=xterm
SHELL=/bin/csh
EDITOR=emacs
PAGER=less
```

The above command displays all of the environment variables in the environment that are passed to a new process. The message "`(from default environment)`" indicates that the current process object does not have its own environment, and the environment displayed is the default environment.

```
(CXdb) info environment P
Process [#0] environment: (from default environment)
PATH=/usr/local/bin:/usr/bin
PRINTER=pscript1
PAGER=less
```

The above command displays all of the environment variables in the environment that are passed to a new process and that begin with the letter *p*. The message "`(from default environment)`" indicates that the current process object does not have its own environment, and the environment displayed is the default environment.

Related Commands

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

Related Concepts

default environment
environment

Related Parameters

process-list
regular-expression

info errno

in er

i er

Display the system error message received by the process.

Syntax

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

Parameter	Meaning
<*process-list*>	A list of processes affected by this command. The default is the current process.
<*thread-list*>	A list of threads affected by this command. The default is all threads of the specified process.

Description

The `info errno` command displays the system error message associated with the current value of `errno`.

Examples

The following example illustrates how to display system error messages received by the process.

```
(CXdb) info errno
thread 0: errno = 2 - No such file or directory
```

The above command displays the current system error message received by the process. In this case, the error message indicates that the program tried to access a file that does not exist.

Related Commands

`info process`

Related Parameters

process-list thread-list

info event

in event
e?

Display the specified eventpoints.

Syntax

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

Parameter	Meaning
<event-specifier>	An eventpoint identifier. The asterisk (*) is used to specify all eventpoints. The default is `*`.
[**,** ...]	An optional list of additional eventpoints. Multiple eventpoints are separated by commas.

Description

The `info event` command displays information about each of the specified eventpoints.

For each eventpoint the number, type, enabled setting, ignore count, address and symbolic location (if applicable) are displayed. If a handler is specified for the eventpoint, then the commands of the handler are displayed.

Examples

The following examples display information about eventpoints.

```
(CXdb) info event 0

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

The above command displays information about eventpoint 0. The information displayed on the first line is described below:

- `#0` — The eventpoint number.
- `break line` — The specific type of eventpoint.
- `on [#0/*]` — The process number and the threads of the process at which the breakpoint is set. The asterisk in the threads position indicates that the eventpoint is set for all possible threads of this process.

- `Enabled` — Indicates that the eventpoint is currently enabled. Its handler is executed by CXdb as long as it is the last-placed eventpoint at any given location. If this field indicates `Disabled`, then the eventpoint is not activated when execution reaches its location.
- `ignore 0/0` — The ignore count for this eventpoint. You can set an ignore count for an eventpoint using the `set ignore` command. If an enabled eventpoint has an ignore count, CXdb ignores the eventpoint as many times as is specified by the count. The number before the slash is the number of times the eventpoint has been ignored and the number after the slash is the ignore count.

On the second line, the following information is displayed:

- `[0x800053b0]` — The hexadecimal address of the eventpoint.
- `EXAMPLE in example.f line 7` — The symbolic location of the eventpoint. The eventpoint is located in the `EXAMPLE` routine of the source file example.f at line 7.

```
(CXdb) info event 1,2

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

#2: reached routine, on [#0/*], Enabled, ignore 0/0
        [0x80005696] CLEAR_ARRAY in example.f line 53
    {
        print I;
        resume;
    }
```

The above command displays information about eventpoints 1 and 2. The CXdb commands in the handler for eventpoint 2 are displayed.

```
(CXdb) info event *

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

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

#2: reached routine, on [#0/*], Enabled, ignore 0/0
        [0x80005696] CLEAR_ARRAY in example.f line 53
    {
        print I;
        resume;
    }
```

The above command displays information about all existing eventpoints.

If you have created a debugger variable for the eventpoint, you can use the debugger variable in place of the eventpoint number.

```
(CXdb) info event $break0

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

The above command displays information about the eventpoint corresponding to the debugger variable `$break0`.

Related Commands

`disable event`	`disable eventtype`
`enable event`	`enable eventtype`
`info break`	`info eventtype`
`info trace`	`info watch`
`remove event`	`remove eventtype`
`set default handler`	`set handler`
`set ignore`	`set typehandler`

Related Concepts

breakpoints
eventpoint handlers
watchpoints
eventpoints
tracepoints

Related Parameters

event-specifier

info eventtype

in eventt
et?

Display all eventpoints of the specified type.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<eventtype-specifier>	An eventtype whose eventpoints you want to display.
[, ...]	An optional list of additional eventpoint types. Multiple eventpoint types are separated by commas.

Description

The `info eventtype` command displays information about the eventpoints of the specified eventpoint types.

The following is a list of eventpoint types:

```
break
trace
watch
exec
join
modify
reached
relation
signal
spawn
```

Eventpoints are separated by type. If the default handler has been changed for the eventpoint type, the new default handler is displayed.

For each eventpoint, the number, type, enabled setting, ignore count, address and symbolic location (if applicable) are displayed. If a handler has been specified for the eventpoint, then the commands of the handler are displayed.

Examples

The following examples display the eventpoints of different eventpoint types.

```
(CXdb) info eventtype break

Status of eventpoints of type Breakpoint:

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

The above command displays information about all breakpoints. In this case there is only one breakpoint, breakpoint 0. The information displayed on the first line is described below:

- `#0` — The eventpoint number.
- `break line` — The specific type of eventpoint.
- `on [#0/*]` — The process number and the threads of that process at which the breakpoint is set. The asterisk in the threads position indicates that the eventpoint is set for all possible threads of this process.
- `Enabled` — Indicates that the eventpoint is currently enabled so that its handler will be executed by CXdb as long as it is the last-placed eventpoint at any given location. If this field displays `Disabled`, then the eventpoint would not be activated when execution reached its location.
- `ignore 0/0` — The ignore count for this eventpoint. You can set an ignore count for an eventpoint using the `set ignore` command. If an enabled eventpoint has an ignore count, CXdb ignores the eventpoint as many times as is specified by the count. The number before the slash is the number of times the eventpoint has been ignored, and the number after the slash is the ignore count.

On the second line the following information is displayed:

- `[0x800053b0]` — The hexadecimal address of the eventpoint.
- `EXAMPLE in example.f line 7`— The symbolic location of the eventpoint. The eventpoint is located in the `EXAMPLE` routine of the source file example.f at line 7.

```
(CXdb) info eventtype trace, reached

Status of eventpoints of type Reached:

#2: reached routine, on [#0/*], Enabled, ignore 0/0
        [0x80005508] PRINT_ARRAY in example.f line 39
    {
        print I;
        resume;
    }

Status of eventpoints of type Tracepoint:
Default type handler defined:
    {
        echo "Reached tracepoint: ";
        print $self;
        resume;
    }

#1: trace line, on [#0/*], Enabled, ignore 0/0
        [0x800053d4] EXAMPLE in example.f line 9
```

The above command displays information about all tracepoints and event reached eventpoints. The default handler is displayed for tracepoints because the handler has been changed from its initial setting. The user-defined handler for the event reached eventpoint is also displayed.

```
(CXdb) info eventtype *

Status of eventpoints of type Signal:

Status of eventpoints of type Relation:

Status of eventpoints of type Modify:

Status of eventpoints of type Reached:

#2: reached routine, on [#0/*], Enabled, ignore 0/0
        [0x80005508] PRINT_ARRAY in example.f line 39
    {
        print I;
        resume;
    }

Status of eventpoints of type Join:

Status of eventpoints of type Spawn:
```

```
Status of eventpoints of type Exec:

Status of eventpoints of type Watchpoint:

Status of eventpoints of type Tracepoint:
Default type handler defined:
    {
        echo "Reached tracepoint: ";
        print $self;
        resume;
    }

#1: trace line, on [#0/*], Enabled, ignore 0/0
        [0x800053d4] EXAMPLE in example.f line 9

Status of eventpoints of type Breakpoint:

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

The above command displays information about all the eventpoints of all the eventtypes.

Related Commands

`disable event`
`disable eventtype`
`info event`
`remove event`
`remove eventtype`
`set default handler`
`set handler`
`set typehandler`
`set ignore`

Related Concepts

breakpoints
eventpoints
eventpoint handlers
tracepoints
watchpoints

Related Parameters

debugger-variable
event-specifier
eventtype-specifier

info expression

in ex

describe, whatis

Display the characteristics of the specified language expression.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.
<language-expression>	Any expression that is valid in the current language of the specified process.

Description

The `info expression` command displays the following information about the specified language expression, when applicable:

- Object type — Type of object represented by the language expression.
- Location — Starting address of the storage location of the object.
- Size — Total size of the object.
- Type — Data type of the language expression.
- Value — Current value of the language expression.
- Liveness ranges — Regions of memory where the value of the variable has meaning. Outside the liveness ranges, the value of the variable is not available.
- Synthesized variables — Variables generated by the compiler at optimization levels `-O1` and higher to improve program performance.
- Orientation — Array type.
- Bounds — Array size.
- Base address — Array starting address.
- Entry point — Starting address of a function.
- Return type — Data type of the value returned by a function.

- Prototype — The prototype for a function.
- Var type — Type of object represented by a debugger variable.
- Writable — Write access to a debugger variable.

Examples

The following examples illustrate how to obtain information about language expressions.

```
(CXdb) info expression NUMARGS

object type: Fortran identifier
   location: 0x8008ad2c
       size: 4 bytes
       type: INTEGER*4
      value: 0
       3 liveness ranges:
                Start          End          Location
         1.  0x8000544c:0x80005452 - register s0
         2.  0x80005458:0x8000545e - register s0
         3.  0x80066000:0x80077000 - 0x8008ad2c
```

The above command displays information about the variable `NUMARGS` from the current process. The liveness ranges indicate where `NUMARGS` is stored when the PC (program counter) falls within the bounds given by the start and end addresses. Outside these address ranges, the value of `NUMARGS` is not available and cannot be displayed.

```
(CXdb) info expression (I .EQ. 1.0)

object type: Fortran expression result
       size: 4 bytes
       type: LOGICAL*4
      value: .False.
```

The above command displays information about the logical expression `(I .EQ. 1.0)`. This expression is evaluated in the context of the current process. Because the value of this expression is not stored by the process, no storage location or liveness ranges are listed.

```
(CXdb) info expression SLICE
object type: Fortran array
orientation: column
     bounds: INTEGER*4(1:<TEMP0>, 1:<TEMP1>)
  base type: INTEGER*4
  base size: 4 bytes
 total size: 64 bytes
  base addr: 0x80002d60
```

The above command displays information about the array `SLICE` in the current process. The variables `<TEMP0>` and `<TEMP1>` are generated by the compiler to store the upper bounds of the array subscripts.

```
(CXdb) info expression I

object type: Fortran identifier
   location: <none>
       size: 4 bytes
       type: INTEGER*4
      value: <unknown>
       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`, which is a loop induction variable (`<INDV>`). Because the program that contains `I` has been compiled with the `-O1` optimization option, the compiler replaces the use of `I` with the synthesized variables `?i5` and `?i6`. The equations that the compiler generates to calculate `I` are also displayed. Because `I` is not used, it is not stored. Therefore, no storage location or liveness ranges are listed for `I`.

In cases where the `info expression` command lists both liveness ranges and synthesized variable equations for a single program variable, CXdb first tries to solve the equations to determine the value of the program variable. If it cannot solve the equations, then CXdb reads the value from the appropriate storage location.

```
(CXdb) info expression BLD_MATRIX

object type: Fortran function
entry point: 0x800029a4
return type: void
  arg count: 4
 prototype: void BLD_MATRIX( INTEGER*4, INTEGER*4, INTEGER*4, INTEGER*4(1:<TEMP0>,
      1:<TEMP1>) )
```

The above command displays information about the Fortran subroutine `BLD_MATRIX` in the current process. The prototype shows that `BLD_MATRIX` does not return a value (void), but it accepts four arguments as input.

```
(CXdb) info expression subxa

object type: C function
entry point: 0x80004c2e
return type: int
return size: 4 bytes
  arg count: 2
 prototype: int subxa( struct info_block*, struct __ap$file* )
```

The above command displays information about the C function `subxa` in the current process. The prototype shows that `subxa` returns an integer value (`int`), and it accepts two arguments as input.

```
(CXdb) info expression $Break4

object type: debugger variable
   writable: yes
   var type: reference to eventpoint [#4]
```

The above command displays information about the debugger variable `$Break4`. This variable stores the eventpoint number for eventpoint 5.

Related Commands

`evaluate`	`info cxdb`
`info process`	`print`
`set evalopts fpmode`	`set evalopts iprecision`
`set evalopts rprecision`	

Related Concepts

displaying data	language expressions
optimized code	synthesized variables

Related Parameters

language-expression	process-list
synthesized-variable	thread-list

SPP Series only

info floating point registers

in fl p r
ifl p r

Display the floating point registers for SPP Series machines.

Syntax

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

Parameter	Meaning
<*process-list*>	A list of processes affected by this command. The default is the current process.
<*thread-list*>	A list of threads affected by this command. The default is all threads of the specified process.

Description

The `info floating point registers` command displays the contents of the floating point registers for the specified process. The contents are displayed in hexadecimal format. The floating point mode on SPP Series machines is IEEE.

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.

Examples

The following example illustrates how to display the contents of the floating point registers.

```
(CXdb) info floating point registers
Process [#0/0]
fr0              : 0x00000000  0x00000000
fr1              : 0x00000000  0x00000000
fr2              : 0x00000000  0x00000000
fr3              : 0x00000000  0x00000000
fr4 (farg0,fret): 0x00000000  0x00000000
fr5 (farg1)      : 0x00000000  0x00000000
fr6 (farg2)      : 0x00000000  0x00000000
fr7 (farg3)      : 0x00000000  0x00000000
fr8              : 0x00000000  0x00000000
fr9              : 0x00000000  0x00000000
fr10             : 0x00000000  0x00000000
fr11             : 0x00000000  0x00000000
fr12             : 0x00000000  0x00000000
```

```
fr13            : 0x00000000  0x00000000
fr14            : 0x00000000  0x00000000
fr15            : 0x00000000  0x00000000
fr16            : 0x00000000  0x00000000
fr17            : 0x00000000  0x00000000
fr18            : 0x00000000  0x00000000
fr19            : 0x00000000  0x00000000
fr20            : 0x00000000  0x00000000
fr21            : 0x00000000  0x00000000
fr22            : 0x00000000  0x00000000
fr23            : 0x00000000  0x00000000
fr24            : 0x00000000  0x00000000
fr25            : 0x00000000  0x00000000
fr26            : 0x00000000  0x00000000
fr27            : 0x00000000  0x00000000
fr28            : 0x00000000  0x00000000
fr29            : 0x00000000  0x00000000
fr30            : 0x00000000  0x00000000
fr31            : 0x00000000  0x00000000
```

The above command displays the floating point registers for the current process. In this case, 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).

Related Commands

```
info control registers
info registers
info space registers
print
```

Related Concepts

debugger variables

registers

Related Parameters

process-list

thread-list

Related Windows

Floating Point Registers window

info formatting

in fo
i fo

Display the settings for memory display formats.

Syntax

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

Parameter	Meaning
<*process-list*>	A list of processes affected by this command. The default is the current process.

Description

The `info formatting` command shows the current default settings of the print options and memory display formats for specified processes. These defaults affect the appearance of output from the `print` and `examine` commands.

The print options include:

- `padding` — Adds or removes leading zeroes for integer values displayed with the `print` command. Use the `set printopts nopadding` and `set printopts padding` commands to change this setting.
- `max array elements` — Is the maximum number of array elements printed in a single execution of the `print` command. Use the `set printopts maxarray` command to change this setting.
- `floating point format` — Is the precision used for printing floating point numbers. Use the `set printopts precision` command to change this setting.

Each memory format description consists of a memory unit size and its corresponding display format. For example, bytes of data can be displayed as characters, decimal numbers, binary numbers, or other formats. The memory unit types and their available formats are:

- `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.

Examples

The following example illustrates how to list the current default settings of the memory display formats.

```
(CXdb) info formatting

Global format settings:
               padding: Off
    max array elements: 20
 floating point format: 10.4

Process [#0] format settings:

   Selected memory size:        byte
Selected Memory Formats:
        byte=character, halfword=(none), word=hexadecimal
        longword=(none), quadword=(none)
```

The above command displays the print options and memory format settings for all threads of the current process.

The print options show that the maximum number of array elements printed at one time is `20`. The precision for printing floating point numbers is `10.4`. Padding is disabled, so leading zeroes will not be printed for values displayed using the `print` command.

The memory format settings show that the default memory unit is a `byte`, and the default format for displaying bytes is `character`. Therefore, using the `examine` command on this process results in a display of ASCII characters, with each character representing the contents of one byte of memory. The `examine` command has options that allow you to override these defaults.

Related Commands

`examine`	`info cxdb`
`set default format`	`set default fpmode`
`set default memory`	`set format`
`set fpmode`	`set memory`
`set printopts maxarray`	`set printopts nopadding`
`set printopts padding`	`set printopts precision`

Related Parameters process-list thread-list

info frame

in fr

i fr

Display a stack frame.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.
<frame-specifier>	A relative or absolute frame number.

Description

The `info frame` command displays information about the specified frame of the process stack. If a frame number is not specified, the default is the current frame. You can use the `backtrace` command to obtain summary information about stack frames, including the number of the currently selected frame.

A stack frame stores the registers for the context of the calling routine, temporary variables local to this context, and values necessary to manage the current stack frame as well as a link to the previous frame.

The output of the `info frame` command will vary, depending on whether you are debugging a process running on a C Series machine or on an SPP Series machine. For more information about stacks and stack frames on C Series machines, refer to the *CONVEX Architecture Reference Manual (C Series)*. For the stack on SPP Series machines, refer to the *PA-RISC Procedure Calling Conventions Reference Manual*, available from Hewlett-Packard.

Examples

The following examples illustrate how to display stack frames. Output for both C Series and SPP Series machines is shown.

C Series

```
(CXdb) info frame
Process [#0/0]
Frame : 1; [0x800028ca] CHAPTER7 in chapter7F.f line 7
Frame address : 0xffffca38
Saved Registers : pc=0x800028ca   psw=0x87109400   fp=0xffffca48   ap=0x80002d5c
Floating point mode : NATIVE;  Language  : FORTRAN
Number of arguments : 1
```

The above command displays information about the current frame for all threads of the current process. Note that the current frame is the last one selected with the `frame` command. This frame could be different from the frame for the current point of execution.

The displayed information can include the following:

- Process — The process number and thread number to which the frame applies.
- Frame — The frame number and value of the program counter (`pc`) for the frame. Whenever applicable, the line number, routine name, and file name for the point of execution are also given.
- Frame address — The starting address of the area of memory where the frame is stored.
- Saved registers — The contents of the registers saved in the frame. In the above example, the saved registers are:
  - The program counter, `pc`
  - The processor status word, `psw`
  - The frame pointer, `fp` (C Series only)
  - The arguments pointer, `ap` (C Series only)
- Floating point mode — The mode for performing floating point operations.
- Language — The source language for the routine represented by the frame.
- Routine return type — If the routine returns a value, the data type for the value returned. In the above example, the routine does not return a value.

- Number of arguments — The number of arguments passed to this routine.

```
(CXdb) info frame 2
Process [#0/0]
Frame : 2; [0x8000548e] EXAMPLE in example.f line 22
Frame address : 0xffffca48
Saved Registers : pc=0x8000548e   psw=0x7909400   fp=0xffffca5c   ap=0x80005868
Floating point mode : NATIVE;  Language  : FORTRAN
Number of arguments : 0
```

The above command displays information for frame `2` of the current process.

Assume that frame 2 is the current frame, and you enter the following command:

```
(CXdb) info frame -2
Process [#0/0]
Frame : 0; [0x80002988] ISQR in chapter7F.f line 17
Floating point mode : NATIVE;  Language  : FORTRAN
Routine return type : INTEGER*4
Number of arguments : 1
```

The above command uses a relative frame number of `-2`. Because the current frame is frame 2, the command displays frame 0.

SPP Series

```
(CXdb) info frame
Process [#0/0]
Frame : 1; [0x258a4] CHAPTER7 in chapter7F.f line 7
Frame address : 0x7b033530
Saved Registers : rp=0x000276ab
Language  : FORTRAN
Number of arguments : 1
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
```

The above command displays information about the current frame for all threads of the current process. Note that the current frame is the last one selected with the `frame` command. This frame could be different from the frame for the current point of execution.

Most of the frame information displayed on SPP Series machines is similar to that shown for C Series machines. The differences are:

- **Saved registers** — The contents of the registers saved in the frame. In the above example, the only saved register is the return pointer, `rp` (SPP Series only).

- **Floating point mode** — For SPP Series machines, the only available floating point mode is IEEE. Therefore, it is not displayed.
- **Unwind table entry (SPP Series only)** — Information needed for stack unwinding and context restoration.

Related Commands

`backtrace`
`frame`
`info args`
`info frame at`
`info locals`
`info process`
`info scope`
`info stack`

Related Concepts

scope

Related Parameters

frame-specifier
process-list
thread-list

Related Windows

Stack Frame Description dialog
Stack Trace window

info frame at

in fr a
i fr a

Display the stack frame at the specified address.

Syntax

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

Parameter	Meaning
<*process-list*>	A list of processes affected by this command. The default is the current process.
<*thread-list*>	A list of threads affected by this command. The default is all threads of the specified process.
<*language-expression*>	An expression that evaluates to a frame address in the source language.

Description

The `info frame at` command displays information about the stack frame stored at the specified address. This information includes:

- Registers for the context of the calling routine
- Temporary variables local to the context of the calling routine
- Values needed for managing the current stack frame
- A link to the previous frame

The output of the `info frame at` command will vary, depending on whether you are debugging a process running on a C Series machine or on an SPP Series machine. For more information about stacks and stack frames on C Series machines, refer to the *CONVEX Architecture Reference Manual (C Series)*. For the stack on SPP Series machines, refer to the *PA-RISC Procedure Calling Conventions Reference Manual*, available from Hewlett-Packard.

Examples

The following examples illustrate how to display stack frames. Output for both C Series and SPP Series machines is shown.

C Series

```
(CXdb) info frame at 'ffffca38'x
Process [#0/0]
[0x800028ca] CHAPTER7 in chapter7F.f line 7
Frame address : 0xffffca38
Saved registers : pc=0x800028ca   psw=0x87109400   fp=0xffffca48   ap=0x80002d5c
Floating point mode : NATIVE;  Language  : FORTRAN
Number of arguments : 1
```

The above command displays information about the frame stored at address `ffffca38`.

The displayed information can include the following:

- Process — The process number and thread number to which the frame applies.
- Frame — The frame number and value of the program counter (`pc`) for the frame. Whenever applicable, the line number, routine name, and file name for the point of execution are also given.
- Frame address — The starting address of the area of memory where the frame is stored.
- Saved registers — The contents of the registers saved in the frame. In the above example, the saved registers are:
  - The program counter, `pc`
  - The processor status word, `psw`
  - The frame pointer, `fp` (C Series only)
  - The arguments pointer, `ap` (C Series only)
- Floating point mode — The mode for performing floating point operations.
- Language — The source language for the routine represented by the frame.
- Routine return type — If the routine returns a value, the data type for the value returned. In the above example, the routine does not return a value.
- Number of arguments — The number of arguments passed to this routine.

```
(CXdb) info frame at $fp
Process [#0/0]
[0x800028ca] CHAPTER7 in chapter7F.f line 7
Frame address : 0xffffca38
Saved registers : pc=0x800028ca   psw=0x87109400   fp=0xffffca48   ap=0x80002d5c
Floating point mode : NATIVE;  Language  : FORTRAN
Number of arguments : 1
```

The above command displays information for the frame stored at the address indicated by the current value of the frame pointer (`$fp`).

SPP Series

```
(CXdb) info frame at '7b033530'x
Process [#0/0]
Frame : 1; [0x258a4] CHAPTER7 in chapter7F.f line 7
Frame address : 0x7b033530
Saved Registers : rp=0x000276ab
Language  : FORTRAN
Number of arguments : 1
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
```

The above command displays information about the frame stored at address `7b033530`.

Most of the frame information displayed on SPP Series machines is similar to that shown for C Series machines. The differences are:

- Saved registers — The contents of the registers saved in the frame. In the above example, the only saved register is the return pointer, `rp` (SPP Series only).
- Floating point mode — For SPP Series machines, the only available floating point mode is IEEE. Therefore, it is not displayed.
- Unwind table entry (SPP Series only) — Information needed for stack unwinding and context restoration.

Related Commands

`backtrace`	`frame`
`info args`	`info frame`
`info locals`	`info process`
`info scope`	`info stack`

Related Concepts

scope

Related Parameters

language-expression
process-list
thread-list

Related Windows

Stack Frame Description dialog
Stack Trace window

info history

in h
i h

Display the CXdb command history.

Syntax

info history [<*command-count*>]

Parameter	Meaning
<*command-count*>	The number of commands to display. The count must be a positive integer. The history display starts at the most recent command and proceeds toward the oldest one, for the specified count. If no count is specified, the entire history is displayed.

Description

The `info history` command displays the commands archived in the command history. The command history stores the last 100 commands entered on the CXdb command line.

Examples

The following examples illustrate how to display the command history.

```
(CXdb) info history

debug exec docexample
break line 7
run
step
step
break line 15
continue
next
next
info process
info locals
print I
info history
```

The above command displays all commands currently stored in the command history. In this case, there are 13 commands in the history. Notice that the last command in the history is the `info history` command that was just entered.

```
(CXdb) info history 5

info locals
info process
print I
info history
info history 5
```

The above command displays the five most recent commands from the command history. The last command in the history is the one just entered.

Related Commands

`recall`

Related Concepts

logging

info line

in li

i li

Display the source units for a specified line.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<line-specifier>	The line of source code whose source units you want to display. The line specifier can include a source file name as well as the line number.

Description

The `info line` command displays information about all source units for the selected line. The information displayed includes the source unit number, the address location, the row and location in the Source Code window, and the source code corresponding to the source unit. Source units are not always displayed in numerical order.

Examples

The following examples display the source units of source lines.

```
66:       OPTIONS -O1
67:       SUBROUTINE LEVEL_O1(M,N,A,B,X)
68:       REAL A(M,N), B(M,N)
69:
70:       DO J=1,N
71:          DO I=1,M
72:             TEMP = 3.0 * B(I,J)
73:             A(I,J) = TEMP/(2.0*X)
74:             B(I,J) = 2.0 * TEMP
75:          ENDDO
76:       ENDDO
77:
78:       RETURN
79:       END
```

The above Fortran code is used in the following examples.

```
(CXdb) info line 66
 Id    Address Boundaries          Start        End      Kind
1. (144)  0x8000427a:0x8000431c    66 x  7    79 x  9 <ROUT>  OPTIONS -O1  <...>  END
```

The above example displays information about the source units on line 66. The information displayed is broken into the following units:

- `Id` — The source unit number. This number can be used in commands that allow you to specify a source unit number, such as the `break source` and `info sourceunit` commands.
- `Address Boundaries` — The starting and ending address for the source unit. Some source units have multiple entry points. Each different entry point has a different starting address and is displayed on a separate line. In this case, there is only one starting point.
- `Start` — The starting line number by column number of the source unit. This information can be used to distinguish between source units with the same symbolic identifier.
- `End` — The ending line number by column number of the source unit. Source units may extend beyond a single line.
- `Kind` — The granularity of the source unit. The granularity types are:

  `ROUT` — Routine
  `BLOCK` — Block
  `LOOP` — Loop
  `STAT` — Statement
  `EXPR` — Expression

- Source code — The source code pertaining to the source unit. In this example the ellipsis, `<...>`, shows that lines of code have omitted.

```
(CXdb)  info line 70
 Id    Address Boundaries          Start        End      Kind
 1. (148)  0000000000:0000000000    70 x 12    70 x 12  <EXPR>  1
 2. (147)  0000000000:0000000000    70 x 10    70 x 12  <STMT>  J=1
 3. (149)  0x80004290:0x80004294    70 x 14    70 x 14   <EXPR>  N
           0x8000427e:0x80004282
 4. (146)  0x8000427e:0x8000431a   70 x  7   76 x 11  <LOOP>  DO J=1,N  <...>  ENDDO
 5. (145)  0x8000427e:0x8000431c   70 x  7   79 x  9  <BLOCK>  DO J=1,N  <...>  END
```

The above example displays the source units on line 70. There are five source units.

```
(CXdb) info line 73
 Id     Address Boundaries          Start        End      Kind
 1. (169)  0000000000:0000000000     73 x 28     73 x 30  <EXPR>  2.0
 2. (170)  0x800042b6:800042ba       73 x 32     73 x 32  <EXPR>  X
 3. (168)  0x800042b6:800042ba       73 x 28     73 x 32  <EXPR>  2.0*X
 4. (166)  0x800042e0:800042e6       73 x 22     73 x 25  <EXPR>  TEMP
 5. (167)  0x800042b6:800042ba       73 x 27     73 x 33  <EXPR>  (2.0*X)
 6. (163)  0000000000:0000000000     73 x 15     73 x 15  <EXPR>  I
 7. (164)  0000000000:0000000000     73 x 17     73 x 17  <EXPR>  J
 8. (165)  0x800042ea:0x800042ee     73 x 22     73 x 33  <EXPR>  TEMP/(2.0*X)
           0x800042e0:0x800042e6
           0x800042b6:0x800042ba
 9. (162)  0x800042f6:0x800042fa     73 x 13    73 x 33  <STMT>  A(I,J) = TEMP/(2.0*X)
           0x800042ea:0x800042ee
           0x800042e0:0x800042e8
           0x800042b6:ox800042be
```

The above command displays the source units on line 73. There are nine source units. Source units 165 and 162 have multiple entry points, with different starting addresses for each entry point.

Using the `info line` command, you can determine exactly how a source code line has been broken into source units and how these units are numbered.

Related Commands

`break source`
`event reached source`
`finish`
`goto source`
`info sourceunit`
`next`
`next over`
`set default step`
`set step`
`step`
`step over`
`trace source`

Related Concepts

breakpoints
eventpoints
granularity
optimized code
source units
stepping
tracepoints

Related Parameters

line-specifier
source-unit

info locals

in lo

locals

Display the local variables of the current routine.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the current process.

Description

The `info locals` command displays information about the local variables of the routine based on the current frame and program counter (PC).

By default, the current frame is the frame at which execution has stopped. You can select a different frame by using the `frame` command.

CXdb displays the type and value for each local variable, if possible. Information about the current frame is also displayed. If a thread list is specified, the local variables are those of the specified threads.

Information about the arguments to a routine can be displayed using the `info args` command. Information about visible identifiers can be displayed with the `info symbols` command.

Examples

The following example illustrates how to display information about local variables.

```
(CXdb) info locals
Process [#0/0]
Frame : 0; [0x800029da] BLD_MATRIX in chapter7F.f line 29
Number of locals : 7
     1 :  MATRIX = REAL*4(1:5, 1:5, 1:5) 0x800770ac
     2 :  I = (INTEGER*4) 1
     3 :  J = (INTEGER*4) 1
     4 :  K = (INTEGER*4) 3
     5 :  THENN = (REAL*4) 0.0000E+000
     6 :  THENM = (REAL*4) 0.0000E+000
     7 :  THENL = (REAL*4) 0.0000E+000
```

The above command displays the local variables of the current routine for the current frame. The current frame in this example is frame 0. For each local variable, the name, size, and value are displayed, if applicable. The variable `MATRIX` is a three-dimensional array of real numbers. The hexadecimal address shown is the starting address of the array.

Related Commands

`backtrace`
`evaluate`
`frame`
`info args`
`info frame`
`info symbols`

Related Concepts

displaying data

Related Parameters

process-list
thread-list

info macro

in m
i m

Display macros.

Syntax

info macro [*<regular-expression>*]

Parameter	Meaning
<regular-expression>	A search pattern specified as a regular expression. All macros whose names match the search pattern are displayed. Macro names are case sensitive.

Description

The `info macro` command displays the definitions of the specified macros.

Examples

The following examples illustrate how to display macro definitions.

```
(CXdb) info macro

SS( N:1 )       "step statement N; info locals"
p( X )  "print X; @p"
sl( N:1 )       "step loop N; info locals"
slp( N:1, x, y )        "step loop N; @p(x,y)"
```

The above command displays all macros that are currently defined.

```
(CXdb) info macro s

sl( N:1 )       "step loop N; info locals"
slp( N:1, x, y )        "step loop N; @p(x,y)"
```

The above command displays all macros that start with the letter *s*. Note that the definition of macro `SS` is not displayed because macro names are case sensitive.

Related Commands

`alias`
`info alias`
`macro`
`remove alias`
`remove macro`

Related Parameters

regular-expression

info objectmap

in o
i o

Display the object map.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.

Description

The `info objectmap` command displays summary information about the object map for the specified process. The object map information is available only for files compiled with the `-cxdb` option.

The information displayed includes address ranges and object file names for segments of memory, and differs between SPP Series machines and C Series machines.

C Series

On C Series machines, the memory section types are as follows:

- Text — Object code.
- Data — Initialized data.
- Tdata — Initialized thread-specific data.
- Bss — Uninitialized data.
- Tbss — Uninitialized thread-specific data.

SPP Series

On SPP Series machines, the memory section types include the following (additional types may be defined):

- Text 0— Normal object code.
- Text1 — Read-only data.
- Data0, Data1 — Initialized data.
- Far Shared Data — Initialized data in far shared memory.
- Far Shared BSS — Uninitialized data in far shared memory.

- Near Shared Data — Initialized data in near shared memory.
- Near Shared BSS — Uninitialized data in near shared memory.
- Node Private BSS — Uninitialized data in node-private memory.
- Short Data — Initialized data. This is generally scalar data loaded here to optimize memory references.
- Short BSS — Uninitialized data. This is generally scalar data loaded here to optimize memory references.
- Tdata — Initialized data in thread-private memory.
- Tbss — Uninitialized data in thread-private memory.

Examples

The following examples show how to display object map information for C Series and SPP Series machines.

C Series

```
(CXdb) info objectmap

Object Map for Process [#0]:
Executable:     para

   Address Range                Section
low             high            Type    Object

0x80001438      0x800016bc      Text    para.o
0x80053000      0x80053020      Tbss    para.o
0x80064008      0x804358b4      Bss     para.o
```

The above example displays object map information for a C Series machine. The `info objectmap` command displays the object map information for the current process:

- The executable file is `para`.
- The object file is `para.o`.
- The text segment of the object file starts at address `0x80001438` and continues through `0x800016bc`.
- The tbss segment is from `80053000` to `80053020`.
- The bss segment is from `80064008` to `804358b4`.

SPP Series

```
(CXdb) info objectmap
Object Map for Process [#0]:
Executable:para

     Address Range              Section
 low              high          Type          Object

0x40b0           0x40cc        Text1          para.o
0x41df8          0x42094       Text0          para.o
0x40001000       0x40001000    Far Shared Data    para.o
0x40001000       0x40001000    Near Shared Data    para.o
0x40003910       0x40003910    Short Data     para.o
0x40003ab0       0x40003ab0    Data1          para.o
0x40003ac8       0x40003ac8    Short BSS      para.o
0x4000d0b0       0x4000d0b0    Node Shared BSS    para.o
0x4000d0b0       0x4000d0b0    Node Private BSS    para.o
0x4000d0b0       0x4000d0b0    Far Shared BSS    para.o
```

The above example displays object map information for an SPP Series machine.

Related Commands

`disassemble` `examine`
`info process`

Related Concepts

process object

Related Parameters

process-list

info path

in pa

p+

Display the directories in the search path.

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process.

Description

The `info path` command displays the directories in the search path of the process object. CXdb uses the search path to find the source code for the current executable.

The search path can be modified using the `add path` and `remove path` commands. The `set path` command can be used to replace the entire search path with a new set of directories.

Examples

The following example displays the search path of the current process object.

```
(CXdb) info path
Default search list:
        .
        /usr/smith/programs

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

The above command displays the search path. In this example, the search path consists of the current working directory (.) and the directory /usr/smith/programs.

Related Commands

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

Related Concepts

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

Related Parameters

process-list

info process

in pr
p?

Display the status of the process.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.

Description

The `info process` command displays information about the current process object. The information displayed includes the status of the process, the image, and the search path.

Examples

The following example displays information about the process object.

```
(CXdb) info process

status of process [#0]:

      executable: docexample
       arguments: (none)
fixed scheduling: off
          pshell: csh
    image status: created pid 20731, state = stopped
     working dir: /doc/cxdb/examples
    default step: statement
default language: Fortran
         threads: 1
  current thread: 0

 thread 0 status: stopped at [0x800029bc] BLD_MATRIX in
      chapter7F.f line 26

source file search path:
        .
```

The above command displays information about the current process object. The actual information displayed depends upon the state of the process at the time you execute the command.

The headings are described below:

- `executable` — The name of the executable file.
- `arguments` — The list of arguments to be passed to the process shell if the `rerun` command is used.
- `fixed scheduling` — The state of fixed scheduling. This is either set to `on` or `off`. (For SPP Series machines, fixed scheduling is always off because it is not available on that architecture.)
- `pshell` — The process shell. The type of shell is either `csh` or `sh`. This is the shell in which your process is run.
- `image status` — If an image exists, this line shows the current process ID (PID) and state of your process. The state is either running or stopped. If an image does not exist, the message "`no image`" is displayed.
- `remote host` — If debugging is being done remotely, the name of the remote host is displayed.
- `working dir` — The process working directory. This is the directory from which your process is run.
- `default step` — The granularity used if a granularity is not specified in a stepping command. The possible granularities are routine, block, loop, statement, and expression. The initial default is statement.
- `default language` — The default language used by this program. CXdb determines the default language from the language of the main routine of your program.
- `threads` — The threads in your process. If multiple threads are active, their numbers are displayed.
- `current thread` — The current thread of your process that CXdb may use for commands that need information about threads in general. This saves you the trouble of specifying a thread with many CXdb commands.

The following information is displayed for each thread that is active in the current process (in this case `thread 0`):

- `thread 0 status` — The current state of the thread. If the thread is stopped, the current point of execution is displayed.

The following information is displayed about the search path:

- `source file search path` — The list of directories in the search path. A period (.) in the list indicates the current directory. The search path is used to find source files and compiler-generated data files.

Related Commands

`add path`	`attach`
`clear fixed sched`	`core`
`debug core`	`debug exec`
`debug proc`	`detach`
`executable`	`info cxdb`
`info threads`	`remove path`
`rerun`	`run`
`set fixed sched`	`set pshell`
`set path`	`set remotewd`

Related Concepts

console working directory	default search path
process object	process working directory
remote debugging	search path
stepping	threads

Related Parameters

process-list

info psw

in ps

i ps

Display the processor status word (PSW) register.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.

Description

The `info psw` command displays a bit-by-bit description of the processor status word (PSW) register for the current stack frame.

The various models of CONVEX computers use different configurations for the processor status word register. For more information about the PSW on C Series machines, refer to *Convex C-Series Architecture* (DSW-300). 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.

Examples

The following examples illustrate how to display the contents of the PSW. Because the PSW is different on C Series and SPP Series machines, output for both architectures is shown.

C Series only

```
(CXdb) info psw
Contents of Process Status Word for thread 0

PSW = 0x83109480
80000000  yes  Address carry
40000000  no   Address integer overflow
20000000  no   Address zero divide
10000000  no   Integer overflow enable
08000000  no   Trace
06000000   1   Frame length
01000000  yes  Sequential
00800000  no   Scalar carry
```

```
00400000  no   Scalar integer overflow
00200000  no   Scalar zero divide
00100000  yes  Zero divide enable
00080000  no   Floating underflow
00040000  no   Floating overflow
00020000  no   Floating reserved operand
00010000  no   Floating zero divide
00008000  yes  Floating error enable
00004000  no   Floating underflow enable
00002000  no   IEEE
00001000  yes  Sequential store enable
00000800  no   Intrinsic error
00000400  yes  Intrinsic error enable
00000200  no   Trace thread creates
00000100  no   Thread init trap
000000e0   4   Reserved
00000010  no   Communication Address Trap
0000000f   0   Intrinsic error code
```

The above command displays the contents of the PSW for all threads of the current process. In this example, the PSW is 32 bits long, and it has a hexadecimal value of `83108480.`

The detailed breakdown of the PSW 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). The most significant bit is called the carry bit, and its position in the PSW is indicated by the hexadecimal value `80000000.` In the above example, this bit has a value of 1, as indicated by the word `yes` in the center column.

Some of the fields of the PSW occupy more than one bit. For example, frame length occupies bits 25 and 26. These bit positions in the PSW are indicated by the hexadecimal value `06000000.` In the above example, the frame length is `1,` so bit 26 has a value of 0 and bit 25 has a value of 1.

SPP Series only

```
(CXdb) info psw
Contents of Process Status Word for thread 0

PSW = 0x4000f
fc000000    0   Reserved
02000000   no   SecureInterval Timer
01000000   no   Taken branch trap enable
00800000   no   Higher privilege transfer trap enable
00400000   no   Lower privilege transfer trap enable
00200000   no   Nullify
00100000   no   Data memory break disable
00080000   no   Taken branch
00040000  yes   Code(instruction) address translation enable
00020000   no   Divide step correction
00010000   no   High priority machine check mask
0000ff00    0   carry/borrow bits
000000e0   no   Reserved
00000010   no   Recovery Counter enable
00000008  yes   Interruption state collection enable
00000004  yes   Protection identifier validation enable
00000002  yes   Data address translation enable
00000001  yes   External interrupt
```

The above command displays the contents of the PSW for all threads of the current process. In this example, the PSW has a hexadecimal value of `4000f`.

The detailed breakdown of the PSW 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). The least significant bit is called the external interrupt bit, and its position in the PSW is indicated by the hexadecimal value `00000001`. In the above example, this bit has a value of 1, as indicated by the word `yes` in the center column.

Some of the fields of the PSW occupy more than one bit. For example, carry/borrow bits occupy bits 8 through 15. These bit positions in the PSW are indicated by the hexadecimal value `0000ff00`. In the above example, the carry/borrow bits all have a value of `0`.

Related Commands

`clear seq`	`clear sqs`
`frame`	`info frame`
`info frame at`	`print`
`set seq`	`set sqs`

Related Concepts

debugger variables

registers

Related Parameters

process-list

thread-list

Related Windows

Processor Status Word window

info registers

in r
i r

Display the general registers for SPP Series machines, or the scalar and address registers for C Series machines.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.

Description

The `info registers` command displays the contents of registers for the specified process. The display is in hexadecimal format. The registers differ depending on which type of CONVEX machine is running your process, as indicated below.

C Series only

On C Series machines, the registers displayed are:

- Program counter (PC)
- Processor status word (PSW)
- Address registers (A0 to A*n*)
- Scalar registers (S0 to S*n*)
- Scalar stride registers SS0 and SS1 (C4 Series only)

For more information about these registers, refer to the *CONVEX Architecture Reference Manual (C Series)*.

SPP Series only

On SPP Series Series machines, the registers displayed are:

- Program counter (pc, same as iioq_head)
- Processor status word (psw)
- General registers (r0 to r31)

For more information about these registers, refer to the *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, available from Hewlett-Packard.

Examples

The following examples illustrate how to display the register contents on both SPP Series and C Series machines.

C Series only

```
(CXdb) info registers
Process [#0/0]
pc : 0x8000149a
psw: 0x83109480
fp : 0xffffc978
ap : 0x80005850
a5 : 0x00000010
a4 : 0x8010b119
a3 : 0x00000040
a2 : 0x00000050
a1 : 0x00000004
sp : 0xffffc978
s7 : 0x0000000000000000
s6 : 0x0000000000004000
s5 : 0x0000000000004000
s4 : 0x2053544152544544
s3 : 0x2050524f00000019
s2 : 0x2045584100000000
s1 : 0x5354415200000019
s0 : 0xffffffff00000000
```

The above command displays the scalar and address registers for all threads of the current process. The values of the processor status word (psw) and the program counter (pc) are also displayed. The registers are:

- a0 to a7 — Address registers
- ap (same as a6) — Argument pointer
- fp (same as a7) — Frame pointer
- pc — Program counter
- psw — Processor status word
- s0 to s7 — Scalar registers
- sp (same as a0) — Stack pointer

SPP Series only

```
(CXdb) info registers
pc : 0x0002c470
psw: 0x0004000f
Process [#0/0]
r0              : 0x00000000
r1              : 0x7b033448
r2 (rp)         : 0x0002f333
r3              : 0x00000001
r4              : 0x7b0332f0
r5              : 0x7b0331e4
r6              : 0x7b0331dc
r7              : 0x7b0332a8
r8              : 0x400102a0
r9              : 0x00000198
r10             : 0x00000000
r11             : 0x00000023
r12             : 0x00000023
r13             : 0x00000001
r14             : 0x40009aa0
r15             : 0x400092a0
r16             : 0x400102a0
r17             : 0x40010aa0
r18             : 0x40010aa0
r19             : 0x7b0333c0
r20             : 0x00000000
r21             : 0x00000004
r22             : 0x00000010
r23 (arg3)      : 0x02000000
r24 (arg2)      : 0x46000000
r25 (arg1)      : 0x00000010
r26 (arg0)      : 0x7b0333c0
r27 (dp)        : 0x4000ca60
r28 (ret0)      : 0x00000000
r29 (ret1,sl)   : 0x7b0333fc
r30 (sp)        : 0x7b0334b0
r31 (mrp)       : 0x0002f2b7
```

The above command displays the general registers for all threads of the current process. The values of the processor status word (psw) and the program counter (pc) are also displayed. The registers are:

- arg0 to arg3 (same as r26 to r23) — Argument registers
- dp (same as r27) — Data pointer
- mrp (same as r31) — Millicode return pointer
- pc — Program counter (same as instruction queue head pointer)

- psw — Processor status word
- r0 to r31 — General registers
- 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

Related Commands

```
info control registers      info cregisters
info floating point registers
info frame                  info frame at
info psw                    info space registers
info stack                  info vregisters
print
```

Related Concepts

debugger variables — registers

Related Parameters

process-list — thread-list

Related Windows

General Registers window — Scalar Registers window

info scope

in sc

where

Display the current scope path.

Syntax

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

Parameter	Meaning
<*process-list*>	A list of processes affected by this command. The default is the current process.
<*thread-list*>	A list of threads affected by this command. The default is all threads of the current process.

Description

The `info scope` command displays information about the current scope.

The current scope is defined by the currently selected frame. The currently selected frame initially is the same as the point of execution. However, a different frame can be selected using the `frame` command.

The current scope determines the identifiers that are visible in the selected frame.

Examples

The following examples display the current scope.

```
(CXdb) info scope
Process [#0/0], frame 0 scope: f$PRINT_ARRAY
```

The above command displays the setting of the current scope. The information provided is described below:

- `Process [#0/0]` — The current process object and thread for which the current scope is being displayed.
- `frame 0` — The current frame number. The current frame defines the current scope.

- `scope:` — The current scope. The different parts of the scope path are listed below:

  `f$` — The current language. The `f` stands for Fortran.
  `PRINT_ARRAY` — The current routine name.

The current scope determines the identifier selected when you specify an unqualified identifier. An unqualified identifier is an identifier without a scope path.

```
(CXdb) info scope
Process [#0/0], frame 0 scope: c$chapter13C`subxa
```

The above example shows a scope path for a C program.

Related Commands

`frame`
`info frame`
`info cxdb`
`info process`

Related Concepts

scope

Related Parameters

process-list
thread-list

info signal

in si

i si

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

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<signal-specifier>	A signal whose settings are to be displayed.
[`,` ...]	An optional list of additional signals. Multiple signals are separated by commas.

Description

The `info signal` command displays the settings for the specified signals, or all signals if none are specified.

The signal name and number are displayed, as well as a `Yes` or `No` value for the signal actions of `Stop`, `Pass`, and `Print`. These actions are defined below:

- `Stop` — Stop process execution when CXdb catches the signal.
- `Pass` — Pass the signal on to the process when execution resumes.
- `Print` — Print a message when CXdb catches the signal.

If an eventpoint handler has been created for the receipt of a signal, the commands of the handler are displayed below the settings for the signal.

The output of the `info signal` command differs between C Series and SPP Series architectures. Refer to the "signals" reference page for more information.

Examples

The following examples display the settings for signals. Because the signals are different on C Series and SPP Series machines, output is shown for both architectures.

C Series

```
(CXdb) info signal
The current signal actions are:

Signal                                  Signal
number   Stop    Pass    Print   name
------   ----    ----    -----   ----------------------
0        No      No      No      Signal 0
1        Yes     Yes     Yes     Hangup
2        Yes     Yes     Yes     Interrupt
3        Yes     Yes     Yes     Quit
4        Yes     Yes     Yes     Illegal instruction
5        Yes     No      No      Trace/BPT trap
6        Yes     Yes     Yes     IOT trap
7        Yes     Yes     Yes     EMT trap
8        Yes     Yes     Yes     Floating point exception
9        Yes     Yes     Yes     Killed
10       Yes     Yes     Yes     Bus error
11       Yes     Yes     Yes     Segmentation fault
12       Yes     Yes     Yes     Bad system call
13       Yes     Yes     Yes     Broken pipe
14       No      Yes     No      Alarm clock
15       Yes     Yes     Yes     Terminated
16       No      Yes     No      Urgent I/O condition
17       Yes     Yes     Yes     Stopped (signal)
18       Yes     Yes     Yes     Stopped
19       No      Yes     No      Continued
20       No      Yes     No      Child exited
21       Yes     Yes     Yes     Stopped (tty input)
22       Yes     Yes     Yes     Stopped (tty output)
23       No      Yes     No      I/O possible
24       Yes     Yes     Yes     Cputime limit exceeded
25       Yes     Yes     Yes     Filesize limit exceeded
26       No      Yes     No      Virtual timer expired
27       No      Yes     No      Profiling timer expired
28       No      Yes     No      Window size changes
29       Yes     Yes     Yes     Resource Lost
30       Yes     Yes     Yes     User defined signal 1
31       Yes     Yes     Yes     User defined signal 2
```

The above command displays the settings of all the signals on a C Series machine. The categories displayed are described below:

- `Signal number` — The signal number.
- `Stop` — Whether or not to stop process execution when CXdb catches the signal. If the value is `Yes`, process execution stops.
- `Pass` — Whether or not to send the signal to the process when process execution resumes. If the value is `No`, the signal will not be given to the process.
- `Print` — Whether or not to print the signal name to cmdout when the signal is received. If the value is `No`, no message is printed.
- `Signal name` — The signal name. For more information about signals, refer the "signals" reference page.

The settings displayed above are the default settings for the signals.

```
(CXdb) info signal 10, 11, 12

The current signal actions are:

Signal                              Signal
number  Stop    Pass    Print       name
------  ----    ----    -----       ----------------------
10      Yes     Yes     Yes         Bus error
11      Yes     Yes     Yes         Segmentation fault
12      <specific eventpoint>       Bad system call

Specific signal eventpoints:

#0: signal 12 on [#0], Enabled, ignore 0/0
    {
        echo "Bad system call received.";
        resume;
    }
```

The above command displays the settings for signals 10, 11, and 12. An eventpoint handler has been defined for signal 12, `sigsys`. The commands of the handler are displayed below the settings for all signals.

SPP Series

```
(CXdb) info signal
The current signal actions are:

Signal                                  Signal
number   Stop     Pass     Print       name
------   ----     ----     -----       ---------------------
0        No       No       No
1        Yes      Yes      Yes         floating point exception
2        Yes      Yes      Yes         Interrupt
3        Yes      Yes      Yes         quit
4        Yes      Yes      Yes         Illegal instruction (not reset when caught)
5        Yes      No       No          trace trap (not reset when caught)
6        Yes      Yes      Yes         Process abort signal
7        Yes      Yes      Yes         EMT instruction
8        Yes      Yes      Yes         Floating point exception
9        Yes      Yes      Yes         kill (cannot be caught of ignored)
10       Yes      Yes      Yes         bus error
11       Yes      Yes      Yes         Segmentation violation
12       Yes      Yes      Yes         bad argument to system call
13       Yes      Yes      Yes         write on a pipe with no one to read it
14       No       Yes      No          alarm clock
15       Yes      Yes      Yes         Software termination signal from kill
16       Yes      Yes      Yes         user defined signal 1
17       Yes      Yes      Yes         user defined signal 2
18       No       Yes      No          death of a child
19       Yes      Yes      Yes         power state indication
20       No       Yes      No          virtual timer alarm
21       No       Yes      No          profiling timer alarm
22       No       Yes      No          asynchronous I/O
23       No       Yes      No          window size change signal
24       Yes      Yes      Yes         Stop signal (cannot be caught or ignored)
25       Yes      Yes      Yes         Interactive stop signal
26       No       Yes      No          Continue if stopped
27       Yes      Yes      Yes         Read from control terminal attempted
28       Yes      Yes      Yes         Write to control terminal attempted
29       No       Yes      No          urgent condition on IO channel
30       Yes      Yes      Yes         remote lock lost  (NFS)
```

The above command displays the settings of all the signals on an SPP Series machine. The categories displayed are described below:

- `Signal number` — The signal number.
- `Stop` — Whether or not to stop process execution when CXdb catches the signal. If the value is `Yes`, process execution stops.
- `Pass` — Whether or not to send the signal to the process when process execution resumes. If the value is `No`, the signal will not be given to the process.
- `Print` — Whether or not to print the signal name to cmdout when the signal is received. If the value is `No`, no message is printed.
- `Signal name` — The signal name. For more information about signals, refer the "signals" reference page.

The settings displayed above are the default settings for the signals.

Related Commands

`event signal`	`set signal`
`signal process`	`signal thread`

Related Concepts

eventpoints	signals

Related Parameters

process-list	signal-specifier

info sourceunit

in so

i so

Display the specified source unit.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<source-unit>	The identification number of the desired source unit.

Description

The `info sourceunit` command displays information about the specified source unit. To obtain the identification numbers of all source units on a given line of source code, use the `info line` command. The information includes the following:

- ID — The unique identification number of the source unit.
- Address Boundaries — The address range occupied by the source unit, expressed in hexadecimal notation.
- Start — The line and column number where the source unit starts in the source file.
- End — The line and column number where the source unit ends in the source file.
- Kind — The type of source unit. The possible types are:

```
routine
loop
block
statement
expression
```

- Text — The text, or source code, for the source unit.

NOTE: Source unit numbers are random and can change between different compilations of the same source file.

Examples

The following example illustrates how to display information about source units.

```
(CXdb) info sourceunit 25
 Id     Address Boundaries          Start        End      Kind
( 25)  0x80005452:0x8000545b     17 x 11     17 x 24  <EXPR>  NUMARGS .EQ. 0
```

The above command displays information about source unit `25` from the current process. The address range for this source unit is `0x80005452` to `0x8000545b`. The source unit starts on line `17`, column `11` of the source file and ends on line `17`, column `24`. The source unit is an expression (`EXPR`), and the text of the statement is `NUMARGS .EQ. 0`.

Related Commands

`break source`
`event reached source`
`goto source`
`info line`
`trace source`

Related Concepts

source units

Related Parameters

granularity
process-list
source-unit

SPP Series only

info space registers

in sp r
i sp r

Display the space registers for SPP Series machines.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.

Description

The `info space registers` command displays the contents of the space registers for the specified process. The contents are displayed in hexadecimal format. 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.

Examples

The following example illustrates how to display the contents of the space registers.

```
(CXdb) info space registers
Process [#0/0]
sr0             : 0x00005985
sr1 (sret,sarg): 0x00005f90
sr2             : 0x00005f90
sr3             : 0x00000000
sr4             : 0x00005985
sr5             : 0x00007c63
sr6             : 0x00007ae4
sr7             : 0x00000000
```

The above command displays the space registers for the current process. In this case, there are 8 space registers, designated as sr0 through sr7. Register sr1 is also called the space argument (sarg) or space return (sret) register.

info space registers

Related Commands

```
info control registers
info floating point registers
info registers
print
```

Related Concepts

debugger variables
registers

Related Parameters

process-list
thread-list

Related Windows

Space Registers window

info stack

in st

i st

Display information about the process stack.

Syntax

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

Parameter	Meaning
<*process-list*>	A list of processes affected by this command. The default is the current process.
<*thread-list*>	A list of threads affected by this command. The default is all threads of the specified process.

Description

The `info stack` command displays summary information about the stack of the specified thread and process. The information includes:

- Process number and thread number associated with the stack
- Number of frames in the stack
- The current frame and address at which it is stored
- The memory extent, or range of addresses, occupied by the stack

Examples

The following examples illustrate how to obtain a summary of the process stack.

```
(CXdb) info stack
Process [#0/0] stack:
        frames:  4
        current: 0
        extent:  0xffffcab0 - 0xffffca28
```

The above command displays stack information for all threads of the current process. In this example, the current process is process 0, and thread 0 is its only thread. There are four frames on this stack, and frame 0 is the current frame. The memory extent for the stack is `ffffcab0` (highest address) to `ffffca28` (lowest address).

```
(CXdb) :t1 info stack
Process [#0/1] stack:
        frames:  3
        current: 2 at 0xffffca64
        extent:  0xffffcab8 - 0xe007fff8
```

The above command displays stack information for thread 1 of the current process. There are three frames on this stack, and frame 2 is the current frame. Frame 2 is stored at address `ffffca64`.

Related Commands

`backtrace`
`frame`
`info frame`
`info frame at`
`info process`

Related Parameters

process-list
thread-list

Related Windows

Stack Trace window
Stack Frame Description dialog

info symbols

in sy

globals, symbols

Display program symbols.

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process object.
<regular-expression>	A regular expression. All program symbols that match the regular expression are displayed. The regular expression can be preceded by a scope path.

Description

The `info symbols` command displays information about all the program symbols matching the specified regular expression in the current scope.

Program symbols can include routines and local and global variables.

All program symbols in the current scope that match the regular expression are displayed. If the regular expression is omitted, or a period and asterisk (`.*`) are used, then all symbols are displayed.

Examples

The following examples display information about program symbols in the current scope.

```
(CXdb) info symbols
PARA
 1. SUBROUTINE PARA()
 2. INTEGER*4 MAXLEN
 3. REAL*4 A(1:1000)
 4. REAL*4 B(1:1000, 1:1000)
 5. SUBROUTINE SUB1(...)
```

The above command displays the program symbols found in the current scope. The symbols are listed by the routines in which they appear. The numbers are added to keep track of the number of symbols found.

```
(CXdb) info symbols p.*
PARA
 1. SUBROUTINE PARA()
```

The above command displays all program symbols matching the regular expression `p.*`.

Related Commands

`frame`	`info args`
`info expression`	`info locals`

Related Parameters

process-list	regular-expression

info threads

in th
i th

Display threads of the current process.

Syntax

`[<process-list>] info threads`

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process.

Description

The `info threads` command displays the status of all threads.

The information displayed includes the number of threads, the current thread, and, if multiple threads exist, which threads are active. The status of each active thread is also displayed.

Examples

The following examples display information about the threads of the current process.

```
(CXdb) info threads

Status of process [#0] threads:

current thread: 0

      Thread 0: stopped at [0x80005508] PRINT_ARRAY in example.f line 39
                by breakpoint
```

The above command displays information on all threads of the current process. The information displayed is described below:

- `thread count` — The number of threads in the process. The thread count is only displayed if there are multiple threads.
- `current thread` — The thread that is considered to be current by CXdb.

- `Thread 0`— The status of each active thread (in this case thread 0). A thread is said to be active if it is alive. If a process exists, at least one thread is alive. The information displayed for each thread is described below:
  - `stopped at` — The address at which the thread was stopped. In this case, the thread was stopped at address `80005508`.
  - `PRINT_ARRAY in example.f line 39`— The source code location where the thread was stopped. In this case, the thread was stopped in the routine `PRINT_ARRAY` on line 39 of the source file example.f.
  - `by breakpoint` — The means by which the thread was stopped. In this case the thread was stopped by a breakpoint.

```
(CXdb) info threads

Status of process [#0] threads:

  thread count: 4
active threads: 0,1,2,3
current thread: 0

      Thread 0: stopped at [0x4a8f8] _start in mmunge.f line 14
                by breakpoint
      Thread active.

      Thread 1: stopped at [0xf000df80] _end+0xeffb13b0
                by general process stop
      Thread active.

      Thread 2: stopped at [0x315bc] __cps_idle_loop+0x54
                by general process stop
      Thread in idle loop.

      Thread 3: stopped at [0x315c4] __cps_idle_loop+0x5c
                by general process stop
      Thread in idle loop.
```

The output for the above example was generated by a process running on an SPP Series system. In this example, all four threads are active. Thread 0, the current thread, was stopped by a breakpoint. Threads 2, 3, and 4 were stopped by a general process stop.

Threads 3 and 4, while still considered active, are in an idle loop within the `_cps_idle_loop` routine, which is an internal routine in the Convex Compiler Parallel Support Library (CPSlib). This means that these threads have been allocated for use by your program, but they are still waiting to be used (and, therefore, uninteresting from a debugging standpoint). Refer to the Convex *Exemplar Programming Guide* (DSW-067) for more information on threads and CPSlib on SPP Series machines.

Related Commands

`event join`	`event spawn`
`info cxdb`	`info process`

Related Concepts

eventpoints	threads

Related Parameters

process-list

Related Windows

Thread Activity window	Threads dialog

info trace

in tr
t?

Display all tracepoints.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.

Description

The `info trace` command displays information about all existing tracepoints.

The output of the `info trace` command is a table that contains the number, enabled setting, ignore count, process and thread numbers, instruction address, and symbolic location for each tracepoint. If the tracepoint has its own handler, the commands of the handler are displayed below the tracepoint.

Examples

The following examples display all tracepoints.

```
(CXdb) info trace

Event   Enabled Ignore  proc/td Address         Where
#0          y    0/0     0/*     [0x800053b0] EXAMPLE in example.f line 7
```

The above command displays a table of the settings of all currently existing tracepoints. The different elements in the table are described below.

- `Event` — The eventpoint number.
- `Enabled` — A `y` indicates that the eventpoint is currently enabled. An `n` indicates that the eventpoint is currently disabled.

- `Ignore` — The ignore count for this eventpoint. The number before the slash is the number of times the eventpoint has been ignored, and the number after the slash is the ignore count.
- `proc/td` — The process number and the thread numbers at which the eventpoint is set. An asterisk (`*`) in the threads position indicates that the eventpoint is set for all threads of the process.
- `Address` — The instruction address where the eventpoint is located.
- `Where` — The source code location of the eventpoint. The routine, source file, and line number are displayed.

```
(CXdb) info trace

Event   Enabled Ignore  proc/td Address         Where
#0          y    0/0      0/*     [0x800053b0] EXAMPLE in example.f line 7
#1          y    0/0      0/*     [0x80004328] LEVEL_O2 in chapter15.f line 88
    {
        print $pc;
        resume;
    }
```

The above command displays all existing tracepoints, this time with the addition of tracepoint 1. Tracepoint 1 has its own eventpoint handler, which is displayed below the tracepoint.

Related Commands

`disable event`	`disable eventtype`
`enable event`	`enable eventtype`
`info break`	`info event`
`info eventtype`	`info watch`
`remove event`	`remove eventtype`
`set default handler`	`set handler`
`set ignore`	`set typehandler`
`trace instruction`	`trace line`
`trace routine`	`trace source`

Related Concepts

breakpoints	eventpoints
eventpoint handlers	tracepoints
watchpoints	

Related Parameters

process-list

info type
in ty
i ty

Display type definitions.

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process object.
<thread>	A single thread to which this command applies. The default is the lowest numbered active thread.
<regular-expression>	A regular expression. All named types that match the regular expression are displayed. The regular expression can be preceded by a scope path.

Description

The `info type` command displays information about named type definitions of your program.

The output of the command displays the current scope and the definition of each named type that matches the specified regular expression. The type definitions displayed are those found in the current scope.

All named types in the current scope that match the regular expression are displayed. If the regular expression is omitted, or a period and asterisk (`.*`) are used, then all named types are displayed.

Examples

The following examples display information about the named types declared in the following C program in the prog.c source file:

```
typedef char *STRING ;

typedef struct box {
   STRING contents;
   struct box *above;
   struct box *below;
} BOXNODE, *BOXPTR;
        .
        .
        .
```

For the following examples, assume that execution has stopped in the middle of the program.

```
(CXdb) info type B
Scope: prog
  Type: BOXPTR
        struct box*

  Type: BOXNODE
        struct box
```

The above command displays information about all named types in the current source file that begin with `B`. The output shows the current source file and information about the two corresponding typedef's.

```
(CXdb) info type .*
Scope: prog
  Type: BOXPTR
        struct box*

  Type: BOXNODE
        struct box

  Type: box
        struct {
          char* contents;
          struct box* above;
          struct box* below;
        }

  Type: STRING
        char*
```

The above command displays all named types found in the current source file.

Related Commands

`info args` `info locals`
`info symbols` `print`

Related Concepts

scope

Related Parameters

process-list regular-expression
string

C Series only

info vregisters

in v
i vr

Display the vector registers for C Series machines.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.

Description

The `info vregisters` command displays the contents of the vector registers for the specified process. The display is in hexadecimal format.

There are several types of registers in the vector register set:

- Vector merge (VM)
- Vector length (VL)
- Vector stride (VS)
- Vector accumulators (on C2 and C3 Series machines, V0 to V7; on C4 Series machines, V0 to V15)
- Vector first register, VF (C4 Series machines only)

Each vector accumulator consists of 128 elements, numbered 000 through 127. Each of these elements is 64 bits long.

Vectorization occurs under any of 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 C2 and C3 Series machines, refer to the *CONVEX Architecture Reference Manual (C Series)*.

Examples

The following example illustrates how to display the contents of the vector registers.

```
(CXdb) info vregisters
Process [#0/0]
Vector Merge : 0x0400030200000000 0400030200000000
Vector Length: 0x80
Vector Stride: 0x4

v0: (000-003) 0x2020202041dd89d9 0x6c65732042013b14 0x64622f654213b13b 0x202f646f42262763
                    ...
v0: (124-127) 0x0000000044160000 0x0000000044172763 0x0000000044184ec5 0x0000000044197628

v1: (000-003) 0x0000000042900000 0x0000000042a80000 0x0000000042c00000 0x0000000042d80000
                    ...
v1: (124-127) 0x0000000044c30000 0x0000000044c48000 0x0000000044c60000 0x0000000044c78000
        .
        .
        .
v6: (000-003) 0x0000000000000000 0x0000000000000000 0x0000000000000000 0x0000000000000000
                    ...
v6: (124-127) 0x0000000000000000 0x0000000000000000 0x0000000000000000 0x0000000000000000

v7: (000-003) 0x0000000000000000 0x0000000000000000 0x0000000000000000 0x0000000000000000
                    ...
v7: (124-127) 0x0000000000000000 0x0000000000000000 0x0000000000000000 0x0000000000000000
```

The above command displays the vector registers for all threads of the current process. The contents of register `v0`, element `000`, is represented by the first hexadecimal value `0x2020202041dd89d9`. CXdb uses the horizontal ellipsis (...) to indicate that all intervening register elements have the same value, which is zero in this example. The vertical ellipsis has been added to indicate that some of the output is omitted from the example.

Related Commands

`info cregisters` `info registers` `print`

Related Concepts

debugger variables, registers

Related Parameters

process-list, thread-list

Related Windows

Vector Registers window

info watch

in w
i w

Display all watchpoints.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.

Description

The `info watch` command displays information about all existing watchpoints.

The output of this command is a table that displays the number, enabled setting, ignore count, process and thread numbers, and address region being watched for each watchpoint. If the watchpoint has its own handler, the commands of the handler are displayed below the watchpoint.

Examples

The following example displays all watchpoints.

```
(CXdb) info watch
Event   Enabled Ignore  proc/td         Region
#1          y   0/0     0/0     0x800029a4      0x800029a7
```

The above command displays information about all existing watchpoints in the current process. The elements in the table are described below:

- `Event` — The eventpoint number.
- `Enabled` — A `y` indicates that the eventpoint is currently enabled. An `n` indicates that the eventpoint is currently disabled.
- `Ignore` — The ignore count for this eventpoint. The number before the slash is the number of times the eventpoint has been ignored, and the number after the slash is the ignore count.
- `proc/td` — The process number and the thread numbers at which the eventpoint is set. An asterisk (`*`) in the threads position indicates that the eventpoint is set for all threads of the process.
- `Region` — The address region being watched.

```
(CXdb) info watch
Event   Enabled Ignore  proc/td         Region
#3         y    0/0     0/0     0x800029a4      0x800029a7
    {
        print $pc;
        resume;
    }
```

The above command displays information about all existing watchpoints. The commands of the eventpoint handler for watchpoint 3 are displayed.

Related Commands

`disable event`
`disable eventtype`
`enable event`
`enable eventtype`
`info break`
`info event`
`info eventtype`
`info watch`
`remove event`
`remove eventtype`
`set default handler`
`set handler`
`set ignore`
`set typehandler`
`watch`

Related Concepts

breakpoints
eventpoints
eventpoint handlers
tracepoints
watchpoints

Related Parameters

process-list

kill process

k p
k

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

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.

Description

The `kill process` command terminates a running process. The `kill process` command also removes the core or process image being debugged from the process object.

After the image is removed, the executable image of the process object becomes the image being debugged. If the process object does not have an executable image, then nothing is being debugged. You can specify an executable image for the process object using the `executable` command.

When you use the `kill process` command, CXdb asks you to confirm that you want to kill the specified process. If you answer yes, the process is terminated, and the image of the process is removed from the process object. If you answer no, the process is not terminated.

Examples

The following example kills a process.

```
(CXdb) kill process
Kill process [#0]? y
Terminated execution of Process [#0]
```

The above command kills the current process. The image of the process is removed from the process object. A new process can be created using the `run` or `rerun` command, or attached using the `attach` command.

Related Commands

`attach`	`core`
`debug core`	`debug exec`
`debug proc`	`detach`
`executable`	`info cxdb`
`info process`	`rerun`
`run`	`stop`

Related Concepts

process object

Related Parameters

process-list

list
l

List lines of source code.

Syntax

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

Parameter	Meaning
<file-name>	The name of the source file. Relative path names use the console working directory as a base.
<starting-line>	The line in the source file to start the listing.
<number-of-lines>	The number of lines to list. The default is 10.
<language-expression>	This can be any valid language expression that evaluates to an address within a routine that contains debugging information.

Description

The `list` command lists the source code from a source file or routine. If the number of lines to list or a routine is not specified, 10 lines are listed.

Subsequent `list` commands without arguments continue listing the same source file, 10 lines at a time.

If the starting line is not specified, CXdb displays the source code from 5 lines before the beginning of the active source unit. If there is not an active source unit, the list begins with the first line of the specified source file. If the source file is not specified, the source file containing the main routine is chosen.

You must specify a starting line number when specifying a source file. The source file must have been compiled as part of the process currently being debugged.

To list source code for a specific routine, use the `routine` *<language-expression>* parameter. The language expression you specify can be any valid language expression that evaluates to an address. CXdb finds the routine that contains this address and displays the source code for that routine.

Use this command to display lines of source code when you are using CXdb in line mode. In line mode, the output of the `list` command is sent through your default pager.

Examples

The following examples list lines of source code. For the purpose of these examples, assume that the program being debugged has not yet begun execution. The source file for the program is example.f.

```
(CXdb) list
 1:       PROGRAM EXAMPLE
 2:       INTEGER ARRAY(4,4)
 3:       INTEGER I, NUMARGS
 4:       EXTERNAL chapter7c
 5:       EXTERNAL chapter13c
 6:
 7:       PRINT *, "EXAMPLE PROGRAM STARTED"
 8:
 9:       DO I=1,4
10:          DO J=1,4
```

The above example displays 10 lines of source code, beginning with the first line of the source file that contains the main routine, because this is the first use of the `list` command.

```
(CXdb) list
11:             ARRAY(I,J)=I*J
12:          ENDDO
13:       ENDDO
14:
15:       NUMARGS = IARGC()
16:
17:       IF (NUMARGS .EQ. 0) THEN
18:          CALL CHAPTER4(ARRAY)
19:          CALL CHAPTER5(ARRAY)
20:          CALL CHAPTER6(ARRAY)
```

The above command displays 10 lines of source code from the same source file, but starting with the line after the last line listed. Line 10 was the last line listed in the first example, so this listing begins with line 11.

```
(CXdb) list 1
 1:        PROGRAM EXAMPLE
 2:        INTEGER ARRAY(4,4)
 3:        INTEGER I, NUMARGS
 4:        EXTERNAL chapter7c
 5:        EXTERNAL chapter13c
 6:
 7:        PRINT *, "EXAMPLE PROGRAM STARTED"
 8:
 9:        DO I=1,4
10:           DO J=1,4
```

The above command lists 10 lines of source code from the source file containing the `main` routine, starting with line 1.

```
(CXdb) list 7 5
 7:        PRINT *, "EXAMPLE PROGRAM STARTED"
 8:
 9:        DO I=1,4
10:           DO J=1,4
11:              ARRAY(I,J)=I*J
```

The above command lists 5 lines from the source file containing the `main` routine, starting with line 7.

```
(CXdb) list example.f:25 5
25:           CALL chapter13c(ARRAY)
26:   CALL CHAPTER15
27:        ELSE
28:           CALL EXAMPLE_INPUT
29:        ENDIF
```

The above command lists 5 lines from the source file example.f, starting at line 25.

For the next example, assume that program execution has stopped at line 35 of the example.f source file.

```
(CXdb) list
30:
31:       PRINT *, "EXAMPLE PROGRAM FINISHED"
32:       END
33:
34:
35:       SUBROUTINE PRINT_ARRAY(ARRAY)
36:       INTEGER ARRAY(4,4)
37:       INTEGER I
38:
39:       PRINT *
```

In the above example, 10 lines were listed surrounding the current point of execution. Line 35 contained the beginning of the active source unit, so the listing began from five lines above, at line 30.

The next two examples show how you can list lines of source code for a specific routine.

```
(CXdb) list routine PRINT_ARRAY
35:       SUBROUTINE PRINT_ARRAY(ARRAY)
36:       INTEGER ARRAY(4,4)
37:       INTEGER I
38:
39:       PRINT *
40:       PRINT *, "THE TABLE:"
41:       DO I=1,4
42:            PRINT 99, ARRAY(I,1), ARRAY(I,2), ARRAY(I,3),
      ARRAY(I,4)
43:       ENDDO
44:
45:       PRINT *
46: 99    FORMAT (3X,I2,X,I2,X,I2,X,I2)
47:       END
```

The above command lists all lines of source code for the routine named PRINT_ARRAY.

You can also specify a language expression with the `list routine` command, as shown in the following example.

```
(CXdb) list routine '80004390'x
83:        OPTIONS -O2
84:        SUBROUTINE LEVEL_O2(M,N,A,B,X)
85:        REAL A(M,N), B(M,N)
86:
87:        DO J=1,N
88:           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
```

The above command displays all lines of source code for the routine containing the address `'80004390'x` (the routine named `LEVEL_O2`).

Related Commands	`display routine`	`display source`
Related Concepts	console working directory	source units
Related Parameters	file-name	language-expression
Related Windows	Source Code window	

C Series only

load object

lo

Load CTI data for an object file that has been dynamically loaded.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<file-name>	The name of the object file loaded. The search path of the process object is used to find the object file.
<text-address>	The base address for the text segment.
<data-address>	The base address for the data segment.
<tdata-address>	The base address for the thread data segment.
<bss-address>	The base address for the bss segment.
<tbss-address>	The base address for the thread bss segment.

Description

The `load object` command loads the CTI data for an object file that has been dynamically loaded into memory. Once the CTI data has been loaded, normal debugging can be performed with the object file.

CXdb supports dynamically loaded object files that are relocatable with either fixed or relative addresses. The dynamic object file must have been created using `ld -r` or `ld -p`.

NOTE: CONVEX does not provide or support a dynamic loader. CXdb only offers the capability to load dynamic object files to support the symbolic debugging of programs which provide their own dynamic loader.

There is a second method of loading the CTI data of a dynamically loaded object file. You can include a C routine named `_cxdb_dynamic_load()` in your program. Each time the program loads a dynamic object, pass the base addresses of the loaded object to this routine. Each time the routine is called, CXdb correctly updates its debugging information to reflect the object file just loaded.

An example of a `_cxdb_dynamic_load` routine is shown below:

```
void _cxdb_dynamic_load(char *name, int len, int bool,
                        char *text, char *data,
                        char *bss, char *tdata,
                        char *tbss) {}
```

The parameters passed to the routine are the object file name, the length of the object file name, a `1` for loading the object file's CTI data, then the base addresses of the text, data, bss, tdata, and tbss segments.

When using the `_cxdb_dynamic_load()` routine, you cannot stop execution at the beginning of this routine using an eventpoint. CXdb performs a special task when this routine is called. However, you can stop execution inside this routine using an eventpoint.

NOTE: The dynamic object file loaded should not duplicate any routines that already exist in memory. If it does, CXdb will not be able to distinguish both locations of the routine.

After the CTI data has been loaded, you can display information about the object using the `info dynamicobject` command.

Examples

The following example illustrate how to load a dynamic object file into CXdb.

```
(CXdb) load object pcc_block.o 0xc0000110 0xc0000558 0x00000000 0x80013000 0x0000000
```

The above command loads into memory the CTI data of the object file named `pcc_block.o`. The addresses correspond to the base addresses for the text, data, tdata, bss, and tbss sections.

Related Commands

`info dynamicobject`

Related Parameters

file-name

macro

m

Define a macro.

Syntax

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

Parameter	Meaning
<name>	The name of the macro. The name may consist of alphanumeric characters only, and it is case sensitive.
<parameter>	A positional parameter that is passed to the macro. If a comma is to be passed as part of the parameter, a backslash (\) must precede the comma.
<default>	The default value for the specified parameter. A colon (:) must separate the parameter name from its default value.
[, ...]	Additional parameters and their corresponding default values. Multiple parameters in the list must be separated by commas. Spaces between the entries are optional.
<string>	The text that forms the body of the macro. If the text contains spaces, then it must be enclosed in quotation marks. Wherever CXdb encounters the macro name, it substitutes the body of the macro along with the specified parameters.

Description

The `macro` command defines a macro in the CXdb command language.

CXdb treats a macro as a text substitution. 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. Macros may be nested within other macros, and they can execute iteratively.

To invoke a macro, use an at-sign (@) in front of its name. Whenever the macro is invoked, CXdb expands it by substituting the text of the macro body in place of the macro name.

A macro definition remains in effect only during the current debugging session. Therefore, if you have a set of macros that you want to use regularly, you should define them in a CXdb command file or initialization file.

Examples

The following examples illustrate how to define and invoke macros.

```
(CXdb) macro sl(N:1) "step loop N; info locals"
```

The above command defines a macro called `sl`. This macro contains two CXdb commands: `step loop` and `info locals`. The parameter `N` represents the number of loops to step, and the default value for `N` is `1`.

To invoke the above macro, enter the following:

```
(CXdb) @sl
Stepping process [#0/*] by 1 loop
Process [#0/0] stopped stepping at [0x800053d4] EXAMPLE in example.f line 9
Process [#0/0]
Frame : 0; [0x800053d4] EXAMPLE in example.f line 9
Number of locals : 4
     1 :  ARRAY = INTEGER*4(1:4, 1:4) 0x8008ace8
     2 :  I = (INTEGER*4) 0
     3 :  NUMARGS = (INTEGER*4) 0
     4 :  J = (INTEGER*4) 0
```

The above example invokes the macro `sl`, which steps the current process and prints the values of its local variables. Since the number of steps was not specified when `sl` was invoked, the default value of `1` is used. Using `@sl(1)` to invoke the macro is equivalent to `@sl` in this case.

The above response from CXdb shows the result of stepping as well as the information about the local variables. The commands from the macro definition are not echoed in the Command window as the macro is expanded.

To pass a value to the above macro, enter the following:

```
(CXdb) @sl(2)
Stepping process [#0/*] by 2 loops
Process [#0/0] stopped stepping at [0x800053de] EXAMPLE in example.f line 10
Process [#0/0]
Frame : 0; [0x800053de] EXAMPLE in example.f line 10
Number of locals : 4
     1 :  ARRAY = INTEGER*4(1:4, 1:4) 0x8008ace8
     2 :  I = (INTEGER*4) 2
     3 :  NUMARGS = (INTEGER*4) 0
     4 :  J = (INTEGER*4) 5
```

The above example invokes the macro `sl` and passes it the value `2`. This steps the current process by 2 loops and prints the values of its local variables.

```
(CXdb) macro p(x) "print x; @p"
```

The above command defines a macro called `p`. This macro prints the value passed to it by parameter `x`, then it invokes itself to print the next parameter in the list. This macro can continue to invoke itself iteratively until it prints all of the parameters that are passed to it.

To invoke the above macro, enter the following:

```
(CXdb) @p("The values are:", I,NUMARGS,J)
(CHARACTER*15) 'The values are:'
(INTEGER*4) 2
(INTEGER*4) 0
(INTEGER*4) 5
```

The above example invokes the macro `p` to print four items. The first item is a literal string, and the other three items are variables from the current process.

```
(CXdb) macro slp(N:1,X,Y) "step loop N; @p(X,Y)
```

The above command defines the macro `slp`. This macro contains the CXdb command `step loop` and the macro `p`. The parameter `N` represents the number of loops to step, and the parameters `X` and `Y` represent the items to be printed by macro `p`.

To invoke the above macro, enter the following:

```
(CXdb) @slp(2, I,J)

Stepping process [#0/*] by 2 loops
Process [#0/0] stopped stepping at [0x800053de] EXAMPLE in example.f line 10
(INTEGER*4) 4
(INTEGER*4) 5
```

The above example invokes the macro `slp` to step the current process by 2 loops. It then prints the values of the process variables `I` and `J`, which are `4` and `5` respectively.

```
(CXdb) @slp(, PICK, J)

Stepping process [#0/*] by 1 loop
Process [#0/0] stopped stepping at [0x800014ca] CHAPTER4 in chapter4.f line 8
(INTEGER*4) 0
(INTEGER*4) 0
```

The above example invokes the macro `slp` to step the process and print the values of the process variables `PICK` and `J`. Note that the number of steps is not specified in the call to `slp`, so the default value of `1` is used.

Within the macro definition, you can use token pasting to concatenate a variable parameter with a fixed character string. The concatenation operator `##` performs the token pasting, as shown in the following example.

```
(CXdb) macro FL(LN:1) "example.f:##LN"
```

The above command creates the macro `FL`. This macro accepts the integer parameter `LN` and concatenates it to the character string `example.f:`. The default value for `LN` is `1`.

You can use this macro to set breakpoints at various line numbers of the file example.f, as shown in the following example.

```
(CXdb) break line @FL(22)

#1: break line, on [#0/*], Enabled, ignore 0/0
        [0x8000548e] EXAMPLE in example.f line 22
```

The above example sets a breakpoint at line 22 of the file example.f.

Related Commands

`alias`
`info alias`
`info macro`
`remove alias`
`remove macro`

Related Parameters

string

next

n

Step to the next source unit, not counting subroutine calls.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.

<granularity>	The type of source unit, or step size. Available granularities are:
	`routine`
	`block`
	`loop`
	`statement`
	`expression`
	If you do not specify a granularity, CXdb uses the default granularity of the specified process.

<count>	The number of times to repeat this command. The default is 1.
&	Runs the command in the background.

Description

The `next` command is a stepping command that continues execution of your process until it reaches the next source unit of the specified granularity. In searching for the specified source unit, the `next` command does not count any of the source units inside called subroutines.

You can also use the next command button in the Command window to execute the `next` command.

Examples

The 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) points to the beginning of line 4.

```
(CXdb) next
Nexting process [#0/*] by 1 statement
Process [#0/0] stopped nexting at [0x800017ec] CHAPTER5 in chapter5.f line 5
```

Because the default granularity is statement, the above command steps the current process by one statement. When execution stops, the PC points to the beginning of line 5.

```
(CXdb) next 2
Nexting process [#0/*] by 2 statements
Process [#0/0] stopped nexting at [0x80001836] CHAPTER5 in chapter5.f line 7
```

The above command steps the current process by two statements, again because the default granularity is statement. The PC now points to the beginning of line 7, which is a call to a subroutine.

```
(CXdb) next loop
Nexting process [#0/*] by 1 loop
Process [#0/0] stopped nexting at [0x800018c2] CHAPTER5 in chapter5.f line 11
```

The above command steps the process to the beginning of the next loop. However, before the command executed, the PC was at the beginning of line 7, which is a subroutine call. The `next` command does not count any source units in a called subroutine. Therefore, the process continues executing until it reaches the next loop after returning from subroutine `SUB5A`. When the process stops, the PC points to the loop at the beginning of line 11.

Now assume that the default stepping granularity for this process has been changed to loop. You enter the following command:

```
(CXdb) next
Nexting process [#0/*] by 1 loop
Process [#0/0] stopped nexting at [0x800018cc] CHAPTER5 in chapter5.f line 12
```

The above command steps the process to the next loop, which begins on line 12.

Related Commands

`finish`	`info cxdb`
`info line`	`info process`
`info sourceunit`	`next instruction`
`next over`	`set default step`
`set step`	`step`
`step instruction`	`step over`

Related Concepts

process object	source units
stepping	

Related Parameters

granularity	process-list
thread-list	

next instruction

n i

ni, nexti

Step to the next instruction, not counting subroutine calls.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.
<count>	The number of times to repeat this command. The default is 1.
&	Runs the command in the background.

Description

The `next instruction` command steps the process by the specified number of machine instructions. In executing the specified number of instructions, this command does not count any machine instructions in a called routine.

To display the machine instructions for the process, use the `disassemble` command or open the Assembly Code window.

You can also use the nexti button in the Command window to execute the `next instruction` command with the default count (1 machine instruction).

Examples

The following examples illustrate how to step a process by machine instructions.

```
(CXdb)  next instruction
Nexting process [#0/*] by 1 instruction
Process [#0/0] stopped nexting at [0x800053b6] EXAMPLE in example.f line 7
```

The above command steps the current process by one machine instruction.

```
(CXdb)  next instruction 5
Nexting process [#0/*] by 5 instructions
Process [#0/0] stopped nexting at [0x800053d4] EXAMPLE in example.f line 9
```

The above command steps the current process by five machine instructions. Instructions inside a called subroutine are not included in the step count.

Related Commands

`disassemble`	`finish`
`next`	`next over`
`step`	`step instruction`
`step over`	

Related Concepts

process object — stepping

Related Parameters

process-list — thread-list

next over

n o

no

Step from the current source unit of specified granularity to the next source unit of default granularity, not counting subroutine calls.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.

<granularity>	The type of source unit, or step size. Available granularities are:
	`routine`
	`block`
	`loop`
	`statement`
	`expression`
	If you do not specify a granularity, CXdb uses the default granularity of the specified process.

<count>	The number of times to repeat this command. The default is 1.
&	Runs the command in the background.

Description

The `next over` command is a stepping command that continues execution of your process until it reaches the next source unit of default granularity. In searching for the target source unit, the `next over` command does not count the current source unit of specified granularity. It also does not count any of the source units inside called subroutines.

The current source unit is one that 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, and all of these source units can be current at the same time. However, the only one of interest here is the current source unit that has the granularity specified in the `next over` command. If none of the current source units are of the specified granularity, then the `next over` command uses the current source unit of default granularity.

Examples

The 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) points to the beginning of line 4.

```
(CXdb) next over
Nexting process [#0/*] by 1 statement outside current statement
Process [#0/0] stopped nexting at [0x800017ec] CHAPTER5 in chapter5.f line 5
```

Because the default granularity is statement, the above command steps the process over the current statement and stops execution at the beginning of the next statement. Before this command was executed, line 4 was the current source unit of statement granularity. When execution stops, the PC points to the beginning of line 5, which is the next source unit of statement granularity.

```
(CXdb) next over 3
Nexting process [#0/*] by 3 statements outside current statement
Process [#0/0] stopped nexting at [0x80001846] CHAPTER5 in chapter5.f line 8
```

The above command steps the process over the current statement and stops execution at the beginning of the next statement after that. Again, this is because the default granularity is statement. A repetition factor is specified, so the command executes three times. Notice that line 7 is a call to subroutine `SUB5A`. The call was executed, but none of the statements in `SUB5A` were counted toward the repetition factor of 3 because the `next over` command does not count any source units inside called routines. Therefore, when the process stops, the PC points to the beginning of line 8.

```
(CXdb) next over loop
Nexting process [#0/*] by 1 statement outside current loop
Process [#0/0] stopped nexting at [0x800017f6] CHAPTER5 in chapter5.f line 6
```

The above command steps the process over the current loop and stops execution at the next statement after that. Before this command was executed, the PC pointed to line 8. There is no current *loop* source unit at line 8. The `DO` loop that begins on line 5 is active because it contains the current point of execution, but it is not the current loop because the PC is not pointing directly at its starting address. Therefore, the `next over` command ignores the specified granularity of loop and reverts to the default granularity of statement. The net result is that the process begins the second cycle of the `DO` loop and stops with the PC pointing at line 6.

```
(CXdb) next over block 4
Nexting process [#0/*] by 4 statements outside current block
Process [#0/0] stopped nexting at [0x800017f6] CHAPTER5 in chapter5.f line 6
```

The above command steps the process over the current block and stops execution at the next statement after that. The repetition factor is `4`, so the command executes four times. Before this command was executed, the PC pointed to line 6, which is the beginning of the block for the `DO` loop on line 5. Thus, line 6 is the beginning of the current block. Because the block is inside a loop, the `next over` command keeps encountering this same block during all four repetitions. The net result is that the above command executes four cycles of the `DO` loop on line 5. When the process stops, the PC points to line 6, and the value of program variable `I` is 6.

Assume that the default stepping granularity has been changed to loop. The PC is still pointing to line 6, and the value of `I` is 6:

```
(CXdb) next over block 5
Nexting process [#0/*] by 5 loops outside current block
Process [#0/0] stopped nexting at [0x800018c2] CHAPTER5 in chapter5.f line 11
```

The above command steps the process over the current block and stops execution at the next loop after that. The current block is the one starting on line 6, and it is contained within the `DO` loop that starts on line 5. The above command repeats five times, taking the `DO` loop through all of its remaining cycles. Because the default granularity is loop, the process does not stop until it reaches the next loop, which is on line 11.

Related Commands

`finish`
`info line`
`info sourceunit`
`next instruction`
`set step`
`step instruction`
`info cxdb`
`info process`
`next`
`set default step`
`step`
`step over`

Related Concepts

process object
stepping
source units

Related Parameters

granularity
thread-list
process-list

print
pr
p

Evaluate a language expression and print the result.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.
n	A line control that suppresses the newline character at the end of the printed data.
<format>	The format for displaying the memory units. If a format is not specified, the default format for the specified process is used. The format specifications are: • **B** — binary • **C** — Fortran complex • **L** — Fortran logical • **c** — ASCII character • **d** — signed decimal • **e**[*<width>*.*<precision>*] — scientific notation • **f**[*<width>*.*<precision>*] — floating point notation • **i** — instruction • **o** — octal • **s** — string • **u** — unsigned decimal • **x** — hexadecimal

<fpmode>	The mode for floating point calculations, which can be one of the following: • **D** — Dual floating point mode (C Series only) • **I** — IEEE floating point mode • **N** — Native floating point mode (C Series only)
<language-expression>	Any expression that is valid in the current source language.

Description

The `print` command evaluates the specified language expression and prints the result. The language expression can include functions or subroutines from the specified process object.

Because the `print` command evaluates the language expression before printing it, this enables you to assign values to debugger variables and to change the values of process variables.

By default, the output of the `print` command is in the proper format for the type of data being printed. For example, a one-byte character field prints as the appropriate ASCII character. However, the `print` command also allows you to specify different formats for the data. For example, you can print a one-byte character field as a decimal number. CXdb converts the data to the specified format before printing it.

NOTE: No spaces are allowed within the format specification or between the format specification, the floating point mode specification, and the `print` command verb.

Obviously, some print formats are not logically possible for certain data types. For example, it is not possible to print a one-byte character field as a complex number. If you specify a print format that is not appropriate for the data to be printed, CXdb responds with an error message.

A special case occurs when the data to be printed is a variable-length character string or a pointer to a variable-length character string. For these data types, the default print format displays the contents of the data field. However, if you specify a different print format for these data types, then the `print` command evaluates the starting address of the character string and prints that *address* in the format you specified.

Examples

The following examples illustrate various uses of the `print` command. In all of these examples, assume that the default format is decimal.

```
(CXdb) print K
(INTEGER*4) 6
```

The above command prints the current value of the variable `K` from the current process. The particular instance of variable `K` that is referenced here is the one in the current scope. The default format of decimal is used to print the value.

```
(CXdb) print K+2
(INTEGER*4) 8
```

The above command evaluates the expression `K+2` and prints the result. The current value of `K` is not modified.

```
(CXdb) print/B K+2
(INTEGER*4) 0000 0000 0000 0000 0000 0000 0000 0100
```

The above command evaluates the expression `K+2` and prints the result in binary format (`/B`). Note that no white space is allowed between the command and the specification `/B`.

```
(CXdb) print f$SUB13B`I
(INTEGER*1) 0
```

The above command prints the value of the variable `I`. Since `I` is not in the current scope, its scope path is specified.

```
(CXdb) print J=3
(INTEGER*4) 3
```

The above command assigns the value `3` to the process variable `J`; then it prints the result. The process uses this new value for `J` when it resumes execution.

```
(CXdb) print ARRAY
INTEGER*4(1:4, 1:4)
(1..4,1) :   000000001  000000002  000000003  000000004
(1..4,2) :   000000001  000000004  000000009  000000016
(1..4,3) :   000000001  000000008  000000027  000000064
(1..4,4) :   000000001  000000016  000000081  000000256
```

The above command prints the elements of array `ARRAY`. The array has 16 elements and is four rows by four columns. (The command `set printopts maxarray` determines how many array elements are printed at one time.)

```
(CXdb) print ARRAY(1..4,2)
INTEGER*4(1:4, 2:2)
(1..4,2) :   000000001  000000004  000000009  000000016
```

The above command prints the second row of the array `ARRAY`. The subscripts of `ARRAY` in this example follow the syntax for specifying Fortran array slices in CXdb.

```
(CXdb)  print/e6.3 MATRIX
REAL*4(1:5, 1:5, 1:5)
(1..5,1,1) :  -.750E+001 -.303E+001 0.460E+000 0.345E+001 0.000E+000
(1..5,2,1) :  -.203E+001 0.646E+001 0.165E+002 0.282E+002 0.000E+000
(1..5,3,1) :  0.446E+001 0.215E+002 0.502E+002 0.967E+002 0.000E+000
(1..5,4,1) :  0.125E+002 0.442E+002 0.126E+003 0.317E+003 0.000E+000
```

The above command prints the array `MATRIX`. The format for the output is scientific notation (e) with a field size of `6.3`. No white space is allowed between the command and the specification `/e6.3` on the command line. With this formatting, the output values are rounded to compensate for the digits that are not displayed.

```
(CXdb) print/f6.3N MATRIX
REAL*4(1:5, 1:5, 1:5)
(1..5,1,1) :  -7.500 -3.025  0.460  3.450 0.000E+000
(1..5,2,1) :  -2.025  6.460 16.450 28.157 0.000E+000
(1..5,3,1) :   4.460 21.450 0.502E+002 0.967E+002 0.000E+000
(1..5,4,1) :  12.450 0.442E+002 0.126E+003 0.317E+003
      0.000E+000
```

The above command also prints array `MATRIX`. The format for the output is native mode (`N`) floating point notation (`f`) with a field size of `6.3`. No white space is allowed between the command and the specification `/f6.3N`. With this formatting, the output values are rounded to compensate for the digits that are not displayed.

```
(CXdb) print $X=MATRIX(1,1,1)
(REAL*4)    -7.5000
```

The above command assigns the value of the process variable `MATRIX(1,1,1)` to the debugger variable `X`. It also prints that value.

```
(CXdb) print ISQR(N)
(INTEGER*4)  000000016
```

The above command evaluates the function `ISQR` in the current process and prints the value returned by that function. This command executes function `ISQR` independent of the current process. When the function returns its value, the program counter (PC) and process stack are set back to the state they were in before the `print` command was executed. Note that any eventpoints in `ISQR` may be triggered by this independent execution from the `print` command.

Related Commands

`evaluate`	`info formatting`
`set default format`	`set default memory`
`set format`	`set memory`
`set printopts maxarray`	`set printopts precision`

Related Concepts

C language expressions	debugger variables
displaying data	Fortran language expressions
language expressions	scope

Related Parameters

array-slice	debugger-variable
language-expression	process-list
redirection-operator	thread-list

C Series only

put
pu

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

Syntax

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

Parameter	Meaning
<process-list>	A list of processes used in evaluating this command. The default is the current process.
<thread-list>	A list of threads used in evaluating this command. The default is all threads of the current process.
<file-name>	The name of the file where the memory contents are saved. Relative path names use the console working directory as a base.
<starting-address>	The first address to save. This can be any *<language-expression>* that evaluates to a valid address in the syntax of the current source language. If the starting address is not followed by an ending address or byte count, CXdb determines the size of the memory region based on the type of structure at the starting address.
<ending-address>	The last address to save. This can be any *<language-expression>* that evaluates to a valid address in the syntax of the current source language. It must be preceded by two dots (**..**).
<byte-count>	The number of bytes to save. This can be any *<language-expression>* that evaluates to a positive integer in the syntax of the current source language. It must be preceded by a colon (**:**).
[**\;** ...]	An optional list of memory regions. Multiple memory regions must be separated by the language expression terminator (**\;**).

Description

The `put` command creates the specified file and stores the size and contents of the specified memory regions in the file. The contents can then be loaded back into the same memory regions at a later time using the `get` command.

The `put` command can be used to save and restore Fortran common blocks and C structures. Fortran common block names can be given as address expressions by delimiting each block name with a slash (/). You can also save an array slice, but an array slice can be restored only to a contiguous memory region.

If the specified file exists, it is overwritten unless the `NOCLOBBER` option is enabled.

Examples

The examples below relate to the following Fortran source code, which has been compiled at optimization level `-no`.

```
      SUBROUTINE SUB13B
      CHARACTER*6 NAME
      INTEGER*1 NUM(6), I
      REAL AVG
      COMMON /BLK/ NAME, NUM, AVG

      SUM = 0
      DO I=1,6,2
          SUM = SUM + (NUM(I)-48)*10 + (NUM(I+1) - 48)
      ENDDO

      AVG = SUM / 3

      END
```

The examples below use the `put` command with the above Fortran source code.

```
(CXdb) put file1 /BLK/:8
```

The above command saves the first 8 bytes of the common block `BLK` in the binary file file1. The 8 bytes include all 6 characters of `NAME` and the first 2 elements (1-byte integers) of the array `NUM`.

```
(CXdb) put file2 NUM
```

The above command creates a binary file named file2 and stores the contents of the array `NUM` in the file. Because only a starting address was specified, CXdb assumes the size of the region to be the size of the entire array.

```
(CXdb) put file3 loc(NUM(1))..loc(NUM(3))
```

The above example places the contents of the memory region into a new binary file named file3. The memory region extends from the starting address of the array element `NUM(1)` to the *starting* address of element `NUM(3)`. Thus, the memory region contains only 2 array elements, `NUM(1)` and `NUM(2)`. The Fortran `loc()` function is used to determine the starting addresses of the array elements.

```
(CXdb) put file4 NUM:3
```

The above example stores the contents of the specified memory region into file4. The memory region begins at the starting address of the array `NUM` and extends for 3 bytes (3 elements of the array, because each element is 1 byte long).

```
(CXdb) put file5 loc(NAME) \; loc(AVG)
```

The above command places the values of the variables `NAME` and `AVG` into the file5 file. The language expression terminator (`\;`) is necessary to separate the two address expressions. The Fortran `loc()` function is used to determine the addresses of the variables.

Because only a starting address is specified for each variable, the size of the memory region is automatically set to the size of the variable.

The `put` command can also be used to save the contents of C language structures. If the name of a C language structure is given as the starting address of a memory region to save, CXdb determines the size of the structure and saves the appropriate memory region.

Related Commands

`clear noclobber`
`examine`
`get`
`set noclobber`

Related Concepts

C language expressions
console working directory
Fortran language expressions
language expressions
saving data

Related Parameters

array-slice
file-name
process-list
thread-list

pwd
pw

Display the name of the console working directory.

Syntax

```
pwd
```

Description

The `pwd` command displays the current setting of the console working directory. The console working directory is the base path for relative path names used in CXdb commands.

Examples

The following example shows how to display the setting of the console working directory.

```
(CXdb) pwd
/doc/cxdb/examples
```

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

Related Commands

`cd`

`info cxdb`

Related Concepts

console working directory

process working directory

quit

q

Exit from CXdb.

Syntax

```
quit
```

Description

The `quit` command is the normal means of exiting from CXdb.

If a process is still running when you `quit`, CXdb asks you if you want to kill the process. The default is to kill the process, so if you press **RETURN** without responding to this question, CXdb kills the process.

If you `attach` to a process started outside of CXdb, then CXdb asks you whether or not you want to `detach` from the process before quitting. The default is to `detach`. If you do not `detach` the process, CXdb kills it.

When you `quit` the debugger, CXdb automatically closes all files and windows that it opened.

Caution

If you exit from CXdb in any other way (for example, by executing a `kill -9` command from the shell), files might not be closed and processes might not terminate properly.

Examples

The following example illustrates how to exit from CXdb.

```
(CXdb) quit
```

The above command ends the debugging session and returns you to the shell prompt.

Related Commands

`attach`
`cxdb`
`detach`
`kill process`

Related Concepts

process object

recall

rec

!

Re-execute a previous command.

Syntax

recall [**?**]<*string*>

Parameter	Meaning
?	An operator that searches for the specified string anywhere within each command of the command history.
<*string*>	The character string that is compared to each command in the command history.

Description

The `recall` command retrieves a previously entered command and automatically executes it again. CXdb does not ask for confirmation before executing the retrieved command.

The `recall` command searches backward through the command history to find the first command that matches the string you have specified. If the beginning characters of a command match the specified string, then CXdb retrieves that command and executes it immediately.

The command history contains the last 100 commands entered in the Command window.

The `recall` command is not available in line mode (when you invoke CXdb with the `-nw` option).

You can step forward through the command history, one command at a time, by typing **CTRL-n**. You can also step backward through the command history, one command at a time, by typing **CTRL-p**. This kind of stepping retrieves the command but does not execute it automatically. To execute the retrieved command, press **RETURN**. This method works in both line mode and X Windows mode.

Examples

The following examples illustrate how to search through the command history and re-execute a previous command.

```
(CXdb) recall print
(CXdb) print SUM
(REAL*4) 294.0000
```

The above example recalls the most recent `print` command and executes it again. In this case, the recalled command prints the value of the variable `SUM` from the current process. Note that CXdb does not ask for confirmation before executing the recalled command.

```
(CXdb) recall 'print $'
(CXdb) print $count=I
(INTEGER*1) 7
```

The above example recalls the most recent `print` command that printed the value of a variable whose name begins with `$`. In this case, the recalled command actually assigns the value of the variable I to the debugger variable `$count`, and then it prints its value.

```
(CXdb) recall ?$c
 print $count=I
(INTEGER*1) 7
```

The above command recalls the most recent command that contains the string `$c` anywhere on the command line. In this case, the recalled command references the debugger variable `$count`.

Related Commands

`info history`

Related Parameters

string

remove alias

rem a

Delete an alias.

Syntax

remove alias *<alias-name>*

Parameter	Meaning
<alias-name>	A character string that forms the name of the alias. This string is delimited by white space, so the name cannot contain any spaces. Alias names are case sensitive.

Description

The `remove alias` command deletes the definition of the specified alias.

Once you remove an alias, that alias is no longer available during the debugging session unless you redefine it.

NOTE: If any command files, macros, or other aliases try to reference an alias that was removed, errors will result.

An alias definition remains in effect only during the current debugging session. If you have a set of aliases that you want to use regularly, you can define them in a CXdb command file or initialization file.

You can use the `info alias` command to display the current definitions of existing aliases.

Examples

The following example illustrates how to delete an alias.

```
(CXdb) remove alias P
```

The above command deletes the definition of the alias named `P`.

Related Commands

`alias` `info alias`
`macro`

Related Concepts

command files initialization files

remove cmderr

rem cmde

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

Syntax

remove cmderr *<file-name>* [**,** ...]

Parameter	Meaning
<file-name>	The name of a file that is to receive and store CXdb messages. Each file name is relative to the console working directory unless it is qualified by a path name.
[**,** ...]	A list of additional file names. Multiple file names in the list must be separated by commas. Spaces between the list items are optional.

Description

The `remove cmderr` command removes file names from the list of viewports for cmderr.

Cmderr is the list of viewports (destinations) that receive all error and informational messages generated in response to CXdb commands. Cmderr is equivalent 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 includes the CXdb Command window (or stderr in line mode). You cannot delete the Command window (or stderr in line mode) from the viewport list.

To display the current viewports for cmderr, use the command `info cxdb`.

Examples

The following examples illustrate how to remove viewports from cmderr.

Assume that the viewport list for cmderr currently contains the following entries:

```
Window #1 (the Command window)
errmsgs
/usr/smith/errlog
myerr.log
```

The following command removes the file errmsgs from the viewport list.

```
(CXdb) remove cmderr errmsgs
New cmderr: Window #1, /usr/smith/errlog, myerr.log
```

The response to the above command reflects the updated viewport list.

The following command removes the other files from the viewport list.

```
(CXdb) remove cmderr /usr/smith/errlog, myerr.log
New cmderr: Window #1
```

The response to the above command indicates that Window #1 (the CXdb Command window) is the only remaining viewport for cmderr.

Related Commands

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

Related Concepts

cmderr
cmdlog
cmdout
logging
redirection
viewports

Related Parameters

viewport

remove cmdlog

rem cmdl

Delete file names from the list of files for logging CXdb input.

Syntax

`remove cmdlog` *<file-name>* **`[, ...]`**

Parameter	Meaning
<file-name>	The name of a file that is to receive and store CXdb command-line input. Each file name is relative to the console working directory unless it is qualified by a path name.
`[, ...]`	A list of additional file names. Multiple file names in the list must be separated by commas. Spaces between the list items are optional.

Description

The `remove cmdlog` command removes file names from the list of viewports for cmdlog.

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 equivalent 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.

Instead of permanently removing viewports with the `remove cmdlog` command, you can temporarily disable logging to the cmdlog viewports with the `clear logging` command.

To display the current setting for logging and the current viewports for cmdlog, use the `info cxdb` command.

Examples

The following examples illustrate how to remove viewports from cmdlog.

Assume that the viewport list for cmdlog currently contains the following entries:

```
log_file
cxdb_input
/usr/smith/data/input.log
```

The following command removes the file cxdb_input from the viewport list for cmdlog.

```
(CXdb) remove cmdlog cxdb_input
New cmdlog: log_file, /usr/smith/data/input.log
```

The response to the above command reflects the updated viewport list.

The following command removes the other files from the viewport list for cmdlog.

```
(CXdb) remove cmdlog log_file, /usr/smith/data/input.log
New cmdlog:
```

The above response indicates that the cmdlog viewport list is now empty.

Your entries in the CXdb Command window (or stdin in line mode) are always automatically echoed. Therefore, there is no need to add the Command window to the viewport list for cmdlog. In fact, doing so will cause each of your entries to appear twice in the Command window.

Related Commands

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

Related Concepts

cmderr
cmdlog
cmdout
logging
redirection
viewports

Related Parameters

viewport

remove cmdout

rem cmdo

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

Syntax

remove cmdout <*file-name*> [, ...]

Parameter	Meaning
<*file-name*>	The name of a file that is to receive and store CXdb output. Each file name is relative to the console working directory unless it is qualified by a path name.
[, ...]	A list of additional file names. Multiple file names in the list must be separated by commas. Spaces between the list items are optional.

Description

The `remove cmdout` command removes file names from the list of viewports for cmdout.

Cmdout is the list of viewports (destinations) that receive the normal output generated in response to CXdb commands. Cmdout is equivalent 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 includes the CXdb Command window (or stdout in line mode). You cannot delete the Command window (or stdout in line mode) from the viewport list.

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

Examples

The following examples illustrate how to remove viewports from cmdout.

Assume that the viewport list for cmdout currently contains the following entries:

```
Window #1
output_data
/usr/smith/data/output
myoutput.log
```

The following command removes the file output_data from the viewport list for cmdout.

```
(CXdb) remove cmdout output_data
New cmdout: Window #1, /usr/smith/data/output, myoutput.log
```

The response to the above command reflects the updated viewport list for cmdout.

The following command removes the other files from the viewport list for cmdout.

```
(CXdb) remove cmdout /usr/smith/data/output, myoutput.log
New cmdout: Window #1
```

The response to the above command indicates that Window #1 (the CXdb Command window) is the only remaining viewport for cmdout.

Related Commands

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

Related Concepts

cmderr	cmdlog
cmdout	logging
redirection	viewports

Related Parameters

viewport

remove default environment

rem d e
denv-

Delete environment variables from the default environment.

Syntax

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

Parameter	Meaning
<environment-variable>	An environment variable to remove.
[, ...]	A list of more environment variables to remove. Multiple environment variables are separated by commas.

Description

The `remove default environment` command removes the specified environment variables from the default environment.

If the variable does not exist, CXdb gives you a warning message. The default environment is passed to a new process if the process object does not have its own environment.

Examples

The following examples remove environment variables from the default environment.

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

In the above example, the environment variable `EDITOR` is removed from the default environment. This change to the default environment can only affect new processes whose process object does not have its own environment. Existing processes that were passed the default environment are not affected.

You can remove multiple environment variables by separating each one with a comma.

```
(CXdb) remove default environment PAGER, PRINTER
```

In the above command, the two environment variables `PAGER` and `PRINTER` are removed from the default environment.

remove default environment

Related Commands

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

Related Concepts

default environment
environment
process object

Related Parameters

environment-variable

remove default path

rem d p
dp-

Delete directories from the default search path.

Syntax

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

Parameter	Meaning
<directory-specifier>	A directory to be removed.
`[, ...]`	An optional list of additional directories to be removed. Multiple directory names are separated by commas.

Description

The `remove default path` command removes the specified directories from the default search path.

Relative directory names use the console working directory as the base path name. Each new process object that is created after this command receives the new default search path as part of its search path.

The `add default path` command can be used in initialization files to create default search paths automatically.

The default search path can be displayed using the `info path` command.

Examples

The following examples remove directories from the default search path.

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

The above command removes the /usr/smith/programs directory from the default search path. When CXdb creates a new search path for a new process object, it will not include the /usr/smith/programs directory.

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

The above command removes the directories /usr/smith/programs and /usr/smith/data from the default search path.

Related Commands

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

Related Concepts

console working directory
default search path
process object
process working directory
search path

Related Parameters

directory-specifier

remove dirpath

rem di

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

Syntax

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

Parameter	Meaning
<original-directory>	The directory where the program object files were originally compiled. If you do not specify this parameter, all aliases for CTI directory paths are removed. The directory path name must begin with root (/). Do not include the .CTI subdirectory as part of the path name.

Description

The `remove dirpath` command removes one or more of the alias names for CTI (Compiler-Tools Interface) directory paths. You can remove only the aliases created with the `dirpath` command. Use the `info dirpath` command to list all CTI directory aliases that have been created during the current debugging session.

NOTE: If you do not specify a directory alias to remove, CXdb removes all CTI directory aliases created with the `dirpath` command during the current debugging session.

Examples

The following examples remove CTI directory path aliases.

```
(CXdb) remove dirpath /doc/cxdb/examples
```

The above command removes all alias names for the directory path /doc/cxdb/examples. These aliases were previously specified with the `dirpath` command.

```
(CXdb) remove dirpath
```

The above command removes *all* directory path aliases created with the `dirpath` command during the current debugging session.

Related Commands

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

Related Concepts

Compiler-Tools Interface
search path

remove environment

rem en
env-

Delete environment variables from the process environment.

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process object.
<environment-variable>	An environment variable to remove.
[**,** ...]	A list of additional environment variables to remove. Multiple environment variables are separated by commas.

Description

The `remove environment` command removes the specified environment variables from the environment of the process object.

If the process object does not yet have its own environment, the `remove environment` command creates an environment for the process object. The new environment consists of the default environment minus the environment variables specified in the command.

Each new process will receive the modified environment. Existing processes are not affected.

Examples

The following examples remove environment variables from the environment of the current process object. These examples assume that the process object does not yet have an environment.

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

The above command creates an environment for the process object and then removes the environment variable `EDITOR` from the newly created environment. The `remove environment` command indicates to CXdb that you want to modify the environment for this process object. CXdb creates an environment for this process object consisting of the default environment with the `EDITOR` variable removed.

remove environment

You can remove multiple environment variables with a single command by separating them with a comma.

```
(CXdb) remove environment PAGER , PRINTER
```

The above command removes the variables `PAGER` and `PRINTER` from the environment of the current process object.

Related Commands

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

Related Concepts

default environment
environment
process object

Related Parameters

environment-variable
process-list

remove event

rem event

e-

Delete the specified eventpoints.

Syntax

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

Parameter	Meaning
<event-specifier>	An eventpoint to be removed. The asterisk (*) is used to specify all eventpoints.
[**,** ...]	An optional list of additional eventpoints. Multiple eventpoints are separated by commas.

Description

The `remove event` command removes all specified eventpoints from their process objects.

Removed eventpoints can no longer be referenced in CXdb commands. Removed eventpoints cannot be restored. The eventpoint numbers assigned to removed eventpoints are never reused during a CXdb session.

If you want to prevent an eventpoint from being reached but do not want to completely remove it from its process object, you can disable it with the `disable event` command or give it an ignore count with the `set ignore` command.

In X Windows mode, you can also enable, disable, or remove eventpoints through the Event Point dialog.

Examples

The following commands remove eventpoints.

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

The above command removes eventpoint 2 from its process object. Eventpoint 2 no longer exists and cannot be used in other CXdb commands.

```
(CXdb) remove event 1,3
Eventpoint 1 removed
Eventpoint 3 removed
```

The above command removes eventpoints 1 and 3. You can no longer reference either of these eventpoints.

```
(CXdb) remove event *
Eventpoint 0 removed
Eventpoint 4 removed
Eventpoint 5 removed
```

The above command removes all existing eventpoints. CXdb displays which eventpoints are removed.

Related Commands

`disable event`
`disable eventtype`
`enable event`
`enable eventtype`
`info event`
`info eventtype`
`remove eventtype`
`set default handler`
`set handler`
`set ignore`
`set typehandler`

Related Concepts

breakpoints
eventpoint handlers
eventpoints
tracepoints
watchpoints

Related Parameters

event-specifier

Related Windows

Event Point dialog
Events menu

remove eventtype

rem eventt
et-

Delete all eventpoints of the specified type.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<eventtype-specifier>	A list of eventpoint types whose eventpoints are to be removed. The asterisk (*) is used to specify all eventpoint types.
[**,** ...]	An optional list of additional eventpoint types. Multiple eventpoint types are separated by commas.

Description

The `remove eventtype` command removes all eventpoints of the specified eventpoint type.

The following is a list of eventpoint types:

```
break
trace
watch
exec
join
modify
reached
relation
signal
spawn
```

Removed eventpoints cannot be restored and can no longer be referenced in CXdb commands. The eventpoint numbers assigned to removed eventpoints are never reused during a CXdb session.

If you want to prevent the eventpoints of an eventpoint type from being reached but do not want to completely remove them from their process objects, you can disable them with the `disable eventtype` command or give them each an ignore count with the `set ignore` command.

Examples

The following examples illustrate how to remove the eventpoints of eventpoint types.

```
(CXdb) remove eventtype trace
Event 1 removed
```

The above command removes all tracepoints. In this case only eventpoint 1 is a tracepoint. Eventpoint 1 no longer exists and cannot be referenced in subsequent CXdb commands.

```
(CXdb) remove eventtype watch, break
Eventpoint 4 removed
Eventpoint 3 removed
Eventpoint 0 removed
```

The above command removes all watchpoints and breakpoints. Eventpoints 0, 4, and 3 no longer exist and cannot be referenced in subsequent CXdb commands.

```
(CXdb) remove eventtype *
Eventpoint 2 removed
Eventpoint 5 removed
```

The above command removes the existing eventpoints of all types. CXdb displays which eventpoints are removed.

Related Commands

`disable event`
`disable eventtype`
`enable event`
`enable eventtype`
`info event`
`info eventtype`
`remove event`
`set default handler`
`set handler`
`set ignore`
`set typehandler`

Related Concepts

breakpoints
eventpoint handlers
eventpoints
tracepoints
watchpoints

Related Parameters

eventtype-specifier
process-list

remove macro

rem m

Delete a macro.

Syntax

remove macro <*name*>

Parameter	Meaning
<*name*>	The full name of the macro to be deleted. Macro names are case sensitive.

Description

The `remove macro` command deletes the definition of the specified macro. You can remove only one macro at a time, and you must specify the full macro name.

Examples

The following example illustrates how to delete a macro.

```
(CXdb) remove macro slp
```

The above command deletes the macro named `slp`.

Related Commands

`alias`
`info alias`
`info macro`
`macro`
`remove alias`

remove path

rem p

p-

Delete directories from the process search path.

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process object.
<directory-specifier>	The directory to be removed.
[**,** ...]	An optional list of additional directories to be removed. Multiple directories are separated by commas.

Description

The `remove path` command removes the specified directories from the search path.

The next time CXdb searches for a source file it will not be able to search in the removed directories. Relative directory names use the console working directory as the base path name.

The current search path can be displayed using the `info path` command.

Examples

The following examples remove directories from the search path.

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

The above command removes the /usr/smith/programs directory from the search path for the current process.

```
(CXdb) remove path /usr/smith/data, /doc/cxdb/examples
```

The preceding example removes the listed directories from the search path of the current process. When CXdb searches for a source file, it no longer searches these two directories.

Related Commands

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

Related Concepts

console working directory
default search path
process object
process working directory
search path

Related Parameters

directory-specifier
process-list

remove variable

rem v

Delete a debugger variable.

Syntax

remove variable *<debugger-variable>*

Parameter	Meaning
<debugger-variable>	The debugger variable to be removed.

Description

The `remove variable` command removes the specified debugger variable.

Once a debugger variable has been removed, it may not be referenced in a CXdb command. You can, however, create a new debugger variable with the same name.

Examples

The following example illustrates how to remove debugger variables.

```
(CXdb) remove variable $break1
```

The above command removes the debugger variable `$break1`. You can no longer reference it in a CXdb command.

Related Commands

`break instruction`
`break line`
`break routine`
`break source`
`evaluate`
`event exec`
`event modify`
`event reached instruction`
`event reached line`
`event reached routine`
`event reached source`
`event relation`
`event signal`
`print`
`trace instruction`
`trace line`
`trace routine`
`trace source`
`watch`

Related Concepts

breakpoints
debugger variables
tracepoints
command files
eventpoints
watchpoints

Related Parameters

debugger-variable

rerun

rer

rr

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

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process object.
&	Runs the command in the background.

Description

The `rerun` command creates a process from the executable image of the process object and then begins execution of that process.

The process is run from the process working directory. The process working directory can be set with the `set directory` command.

The shell in which the process is run is called the process shell. It can be specified with the `set pshell` command. The arguments passed to the process shell are the same as those last specified with the `run` command. To run the process with a new set of arguments, or none at all, use the `run` command.

If a process already exists, CXdb asks you if you want to terminate the process and restart. If you answer yes, CXdb kills the current process and creates a new process. If you answer no, CXdb leaves the current process as is, and the CXdb prompt returns.

After the process is created, execution begins. Process execution continues until the process terminates or is stopped.

Examples

The following examples begin execution of a process.

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

The above command creates a new process from the executable file (docexample) of the current process object. The last arguments specified by the `run` command are passed to the process shell. If a `run` command has not yet been issued, no arguments are passed to the process shell.

```
(CXdb) rerun &
Command [#17] backgrounded

Process [#0] is already running with pid 21291.
Terminate existing process and restart? y
Starting process [#0]: docexample
(CXdb)
```

The above command creates a new process for the current process object. Because a process already exists, CXdb asks if you want to terminate the existing process. If you answer `yes`, CXdb terminates the existing process and creates the new process. The `&` on the command line runs the command in the background. This causes the CXdb command prompt to return, allowing you to enter other CXdb commands that do not require the process to be stopped.

Related Commands

`attach`	`continue`
`core`	`debug core`
`debug exec`	`debug proc`
`detach`	`executable`
`info cxdb`	`info process`
`kill process`	`run`
`set default pshell`	`set directory`
`set pshell`	`stop`

Related Concepts

background execution
process object
process working directory

Related Parameters

process-list

resume

res

Continue execution of the process from within an eventpoint handler.

Syntax

```
resume
```

Description

The `resume` command resumes process execution from within an eventpoint handler. It is the only process execution command you can use inside an eventpoint handler.

Process execution is resumed according to how it was before being interrupted. That is, the `resume` command continues the process with the same type of execution that was occurring before the eventpoint was triggered. The types of execution affected by the `resume` command are:

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

If a repetition count is specified with any of the above commands, the `resume` command continues execution with the appropriate count remaining.

Examples

The following examples illustrate the use of the `resume` command in an eventpoint handler.

```
(CXdb) break line 11 {echo 'J ='; print J; resume;}

#1: break line, on [#0/*], Enabled, ignore 0/0
        [0x800053e8] EXAMPLE in example.f line 11
    {
        echo 'J = ';
        print J;
        resume;
    }
```

The above command sets an eventpoint handler for breakpoint 1. The handler prints the value of `J`, then resumes process execution. The following example demonstrates the effect of this handler.

```
(CXdb) step block 3

Stepping process [#0/*] by 3 blocks
J =
(INTEGER*4) 1
J =
(INTEGER*4) 2
J =
(INTEGER*4) 3
Process [#0/0] stopped stepping at [0x800053e8] EXAMPLE in
       example.f line 11
```

The above command steps the process by 3 block source units. Breakpoint 1 is triggered by each iteration of the block, causing the eventpoint handler to be executed 3 times. The handler prints the value of `J` and then resumes process execution.

Related Commands

`attach`	`break instruction`
`break line`	`break routine`
`break source`	`continue`
`event exec`	`event modify`
`event reached instruction`	`event reached line`
`event reached routine`	`event reached source`
`event relation`	`event signal`
`finish`	`next`
`next instruction`	`next over`
`rerun`	`run`
`signal process`	`signal thread`
`step`	`step instruction`
`step over`	`stop`
`trace instruction`	`trace line`
`trace line`	`trace source`
`watch`	

Related Concepts

breakpoints	eventpoints
eventpoint handlers	stepping
tracepoints	watchpoints

return

ret

Return to the calling routine.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.
<language-expression>	Any expression that evaluates to a valid return value.

Description

The `return` command returns the specified value to the calling routine. This forced return pops the top frame from the process stack. The program counter (PC) is set to the next instruction after the call that has been returned.

When you issue the `return` command, CXdb prompts you to specify whether or not you want the function to return immediately. If you respond with a yes (`y`), the process returns the specified value to the calling routine and resets the PC. If you respond with a no (`n`), then CXdb aborts the `return` command. If you press the RETURN key without responding yes or no, the default is yes (`y`).

Examples

The following examples illustrate how to force a return to a calling routine.

```
(CXdb) return 5
Make function return with specified value now? y
```

The above command returns to the calling routine with a return value of 5. When CXdb prompts for confirmation of the command, the reply is `y` (yes) in this case.

```
(CXdb) return N+6
Make function return with specified value now?
```

The above command evaluates the expression `N+6` and returns the result to the calling routine. When CXdb prompts for confirmation of the command, the reply in this case is simply to press the RETURN key. This is equivalent to replying `y` (yes).

```
(CXdb) return N+6
Make function return with specified value now? n
(CXdb)
```

The above example shows that the `return` command was initiated but not completed. When CXdb prompts for confirmation of the command, the reply in this case is `n` (no). Therefore, CXdb aborts the command.

Related Commands

`backtrace`
`disassemble`
`info frame`
`info process`
`info stack`

Related Parameters

language-expression
process-list
thread-list

run

ru

r

Create and execute a new process.

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process object.
<argument-list>	A list of arguments to be passed to the process shell for interpretation.
&	Runs the command in the background.

Description

The `run` command creates a new process from the executable image of the process object and then begins execution of that process.

If the executable image was created from an executable file on the local host, the process is run from the process working directory on the local host. The process working directory can be set with the `set directory` command.

If the executable image was created from an executable file on a remote host, the process is run on the remote host in the process working directory. If the process working directory has not been set, the remote working directory is used. If the remote working directory for the process object has not been set (using the `set remotewd` command), the default remote working directory is used. If the default remote working directory has not been set (using the `set default remotewd` command), the console working directory on the local host is used as the remote working directory.

CXdb expands the arguments passed to the process through the `run` command. The expansion is done according to the rules of the shell specified as the process shell. This shell can be specified using the `set pshell` command. If the argument list is omitted, no arguments are passed to the process shell. To rerun the process using the previous argument list, use the `rerun` command. Redirection operators for the shell must be preceded by a backslash (\) to prevent interpretation by CXdb.

If a process already exists, CXdb asks if you want to terminate the process and restart. If you answer yes, CXdb kills the current process and creates a new process with any specified arguments. If you answer no, CXdb leaves the current process as is, and the CXdb prompt returns.

After the process is created, execution begins. Process execution continues until the process terminates or is stopped.

Examples

The following examples begin execution of a process.

```
(CXdb) run \< data
Starting process [#0]: docexample < data
```

The above command creates a new process from the executable file (docexample) of the current process object. The arguments passed to the shell are `< data`, which cause standard input to come from the file named data. The `<` is preceded by a backslash to prevent interpretation by CXdb.

Because the process is run from the process working directory, the process working directory is used as the base directory for relative path names. `< data` becomes the argument list for the current process object.

```
(CXdb) run &
Command [#17] backgrounded

Process [#0] is already running with pid 11540.
Terminate existing process and restart? y
Starting process [#0]: docexample
```

The above command creates a new process. Because a process already exists from the first `run` command, CXdb asks if you really want to kill the first process and restart execution with a new process. Because the argument list is omitted, no arguments are passed to the process.

The `&` on the command line runs the command in the background. This causes the CXdb command prompt to return, allowing you to enter other CXdb commands that do not require the process to be stopped.

```
(CXdb) run arg1 arg2 \< data > cmdout.log
```

The above command creates a new process. The process shell is passed `arg1 arg2 < data`. This string is interpreted by the process shell, and then the arguments `arg1` and `arg2` are passed to the process while standard input is redirected from the file named data.

The arguments `> cmdout.log` redirect the output of this command to the file cmdout.log. These arguments are not passed to the process shell because they are not preceded by a backslash. Because CXdb output is redirected to cmdout.log, no CXdb response appears after the command line.

Related Commands

`attach`
`clear default remotewd`
`continue`
`core`
`debug core`
`debug exec`
`debug proc`
`detach`
`executable`
`info cxdb`
`info process`
`kill process`
`rerun`
`set default pshell`
`set default remotewd`
`set directory`
`set pshell`
`set remotewd`
`stop`

Related Concepts

background execution
process object
process working directory
remote debugging

Related Parameters

process-list

set autocreate

se a

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

Syntax

```
set autocreate
```

Description

The `set autocreate` command enables dynamic creation of Source Code windows. When this setting is enabled, CXdb can automatically create a new Source Code window. Initially the autocreate option is enabled. You can disable it by using the `clear autocreate` command, or by using the `-ns` option when invoking CXdb with the `cxdb` command.

In the X Windows interface, you can enable or disable the autocreate setting by using the autocreate option of the CommandWindow menu in the Command window.

The new Source Code window is initially associated with all threads of the current process. You can set the threads associated with a Source Code window by using the `set threads` command. In X Windows, you can set the threads for a window using the Threads dialog. You can open this dialog by selecting the threads option under the SourceCodeWindow menu in the Source Code window.

If the `set autocreate` command is used in an initialization file, CXdb creates a Source Code window when it is invoked with the name of an executable file.

This command has no effect in line mode (when you invoke CXdb with the `-l` option).

Examples

The following example illustrates how to set the autocreate option.

```
(CXdb) set autocreate
```

The above command enables the autocreate option. If a Source Code window does not yet exist, and a thread is spawned, CXdb creates a new Source Code window.

Related Commands

`clear autocreate`	`cxdb`
`display source`	`list`
`set threads`	

Related Concepts

initialization files

Related Windows

CommandWindow menu

Source Code window

set cmderr

se cmde

Clear and redefine the list of files that log CXdb messages.

Syntax

`set cmderr` *<file-name>* **`[, ...]`**

Parameter	Meaning
<file name>	The name of a file that is to receive and store CXdb messages. Each file name is relative to the console working directory unless it is qualified by a path name.
`[, ...]`	A list of additional file names. Multiple file names in the list must be separated by commas. Spaces between the list items are optional.

Description

The `set cmderr` command clears the current list of viewport files for cmderr and replaces it with the specified list of file names.

Cmderr is the list of viewports (destinations) that receive all error and informational messages generated in response to CXdb commands. Cmderr is equivalent 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 includes the CXdb Command window (or stderr in line mode). You cannot remove the Command window (or stderr in line mode) from the viewport list.

CXdb creates the specified 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 reset the viewport list for cmderr.

```
(CXdb) set cmderr errmsgs
New cmderr: Window #[1], errmsgs
```

The above command deletes the current viewport list for cmderr and replaces it with a new list that contains the file errmsgs in the console working directory. Notice that the viewport list also contains Window #[1] (the Command window) by default.

```
(CXdb) set cmderr /tmp/errlog, myerrlog
New cmderr: Window #[1], /tmp/errlog, myerrlog
```

The above command deletes the current viewport list for cmderr and replaces it with a new list that contains the file errlog in the directory /tmp and the file myerrlog in the console working directory. The viewport list also includes Window #[1] (the Command window) by default.

Related Commands

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

Related Concepts

cmderr
cmdlog
cmdout
logging
redirection
viewports

Related Parameters

redirection-operator
viewport

set cmdlog

se cmdl

Clear and redefine the list of files for logging CXdb input.

Syntax

`set cmdlog` *<file-name>* **[, ...]**

Parameter	Meaning
<file-name>	The name of a file that is to receive and store CXdb command-line input. Each file name is relative to the console working directory unless it is qualified by a path name.
[, ...]	A list of additional file names. Multiple file names in the list must be separated by commas. Spaces between the list items are optional.

Description

The `set cmdlog` command clears the current list of viewport files for cmdlog and replaces it with the specified list of file names.

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 equivalent 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. Therefore, 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.

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). Therefore, no input is sent to the cmdlog files unless you first use the `set logging` command to enable logging.

To display the current settings for logging and noclobber, as well as the current viewport list for cmdlog, use the `info cxdb` command.

Examples

The following examples illustrate how to reset the viewport list for cmdlog.

```
(CXdb) set cmdlog logfile
New cmdlog: logfile
```

The above command deletes the current viewport list for cmdlog and replaces it with a new list that contains only one entry, the file logfile in the console working directory.

```
(CXdb) set cmdlog logfile2, /tmp/inputlog
New cmdlog: logfile2, /tmp/inputlog
```

The above command deletes the current viewport list for cmdlog and replaces it with a new list that contains two entries, the file logfile2 in the console working directory and the file inputlog in the directory /tmp.

Related Commands

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

Related Concepts

cmderr
cmdlog
cmdout
logging
redirection
viewports

Related Parameters

viewport

set cmdout

se cmdo

Clear and redefine the list of files that log CXdb output.

Syntax

`set cmdout` *<file-name>* **`[, ...]`**

Parameter	Meaning
<file-name>	The name of a file that is to receive and store CXdb output. Each file name is relative to the console working directory unless it is qualified by a path name.
`[, ...]`	A list of additional file names. Multiple file names in the list must be separated by commas. Spaces between the list items are optional.

Description

The `set cmdout` command deletes the current list of viewport files for cmdout and replaces it with the specified list of file names.

Cmdout is the list of viewports (destinations) that receive the normal output generated in response to CXdb commands. Cmdout is equivalent 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 includes the CXdb Command window (or stdout in line mode). You cannot remove the Command window (or stderr in line mode) from the viewport list.

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 reset the viewport list for cmdout.

```
(CXdb) set cmdout outputdata
New cmdout: Window #[1], outputdata
```

The above command deletes the current viewport list for cmdout and replaces it with a new list that contains the file outputdata in the console working directory. Notice that the viewport list also contains Window #[1] (the Command window) by default.

```
(CXdb) set cmdout /tmp/outputlog, myoutputlog
New cmdout: Window #[1], /tmp/outputlog, myoutputlog
```

The above command deletes the current viewport list for cmdout and replaces it with a new list that contains the file outputlog in the directory /tmp and the file myoutputlog in the console working directory. The viewport list also includes Window #[1] (the Command window) by default.

Related Commands

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

Related Concepts

cmderr
cmdlog
cmdout
logging
redirection
viewports

Related Parameters

redirection-operator
viewport

set default environment

se de e
denv=

Clear and redefine the environment variables for the default environment.

Syntax

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

Parameter	Meaning
<environment-variable>	An environment variable to add after the default environment is cleared.
<string>	The value to be given to the environment variable.
[, ...]	An optional list of additional environment variable assignments. Multiple assignments must be separated by a comma (,).

Description

The `set default environment` command clears the default environment and adds the specified environment variables to the default environment.

The default environment is passed to a new process if the process object does not have its own environment. To display the default environment, use the `info default environment` command.

Examples

The following examples illustrate how to set the default environment.

```
(CXdb) set default environment EDITOR = vi
```

The above command clears the default environment and then adds the variable `EDITOR`, which is set to `vi`. This is the same as issuing a `clear default environment` command followed by an `add default environment` command.

You can set the default environment to multiple environment variables in a single command by separating each with a comma, as shown below.

```
(CXdb) set default environment PAGER = less , LESS = -MQce
```

The above command clears the default environment and then adds the environment variables `PAGER` and `LESS`.

Related Commands

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

Related Concepts

default environment	environment
process object	

Related Parameters

environment-variable	string

C Series only

set default fixed sched

se de fi s

Enable fixed scheduling in the default settings.

Syntax

```
set default fixed sched
```

Description

The `set default fixed sched` command enables fixed scheduling in the CXdb defaults. The CXdb default fixed scheduling is used by new process objects that have not explicitly had their fixed scheduling set with the `set fixed sched` or `clear fixed sched` commands.

Fixed scheduling means that the process requires the simultaneous use of all the processors on a given machine. When fixed scheduling is enabled, the process does not begin executing until all the processors become available. When fixed scheduling is disabled, the process executes on whichever processors are available during a given time slice.

The default is fixed scheduling disabled. To display the default setting for fixed scheduling, use the `info cxdb` command.

NOTE: Because of the additional system overhead involved with fixed scheduling, it is recommended for debugging multithreaded processes only.

Examples

The following example shows how to enable fixed scheduling.

```
(CXdb) set default fixed sched
```

The above command enables fixed scheduling in the CXdb defaults.

Related Commands

`clear default fixed sched`
`clear fixed sched`
`info cxdb`
`info process`
`set fixed sched`

Related Concepts

optimized code
threads

set default format

se de fo

Set the default formats for displaying memory.

Syntax

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

Parameter	Meaning

<memory-unit>	The type of memory unit displayed. The possible types are:
	`byte`
	`halfword`
	`word`
	`longword`
	`quadword`

<format>	The format for displaying a memory unit. The possible formats are:
	`binary`
	`character`
	`complex`
	`decimal`
	`eformat` — scientific notation
	`fformat` — floating point notation
	`hexadecimal`
	`logical`
	`octal`
	`unsigned` — unsigned decimal

Description

The `set default format` command sets the CXdb default formats for displaying the contents of memory. These formats become the default for any new process objects that have not explicitly had their default formats set with the `set format` command. The formats affect the appearance of output from the `examine` command.

Each format description consists of a memory unit type and its corresponding display format. For example, bytes of data can be displayed as characters, decimal numbers, binary numbers, or other formats.

set default format

The memory unit types and their available formats are:

- `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 display the default settings for memory unit formats, use the `info cxdb` command.

Examples

The following examples illustrate how to set default display formats for memory units.

```
(CXdb) set default format byte decimal
```

The above command selects decimal format as the default for displaying bytes of memory.

```
(CXdb) set default format word unsigned
```

The above command selects unsigned decimal format as the default for displaying words of memory.

Related Commands

`examine`
`info cxdb`
`info formatting`
`set default fpmode`
`set default memory`
`set format`
`set fpmode`
`set memory`

C Series only

set default fpmode

se de fp

Set the default floating point mode for new processes.

Syntax

set default fpmode { ieee | native | dual }

Parameter	Meaning
ieee	IEEE mode for floating point operations.
native	Native mode for floating point operations.
dual	Dual mode for floating point operations. In this mode, the process will use its own setting (IEEE or native) for floating point operations.

Description

The `set default fpmode` command sets the default mode for floating point operations to IEEE, native, or dual. To display the current setting of the default fpmode, use the `info cxdb` command.

The initial setting is dual. The default floating point mode is passed to new process objects. Existing process objects are not affected. Processes use the floating point mode of their process object. The floating point mode of a process can be set explicitly with the `set fpmode` command.

Examples

The following example sets the default floating point mode.

```
(CXdb) set default fpmode ieee
```

The above command sets the default floating point mode to IEEE. The floating point mode of a new process object will now be IEEE.

Related Commands

`info cxdb`
`info process`
`set default format`
`set format`
`set fpmode`

Related Concepts

process object

set default handler

se de h

Set the default handler for eventpoints.

Syntax

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

Parameter	Meaning
<event-handler>	The contents for the default eventpoint handler.

Description

The `set default handler` command sets the default eventpoint handler. The default eventpoint handler is used by all eventpoints that have not had an eventpoint handler explicitly defined for them.

You can define an eventpoint handler for an existing eventpoint with the `set handler` command. You can also change the default handler for specific types of eventpoints by using the `set typehandler` command.

Initially the default handler for eventpoints is set to display a message containing the eventpoint number, address, and symbolic location for the eventpoint. You can reset the default handler to its initial setting using the `clear default handler` command.

To display the current setting of the default handler, use the `info cxdb` command.

Examples

The following example shows how to set the default handler for eventpoints.

```
(CXdb) set default handler {echo "Reached eventpoint: "; print $self; }
```

The above command sets the default handler to echo the string "`Reached eventpoint: `" and then print the value of the debugger variable `$self`, which holds the eventpoint number of the currently executing eventpoint.

Related Commands

`clear default handler`
`clear handler`
`clear typehandler`
`info event`
`info eventtype`
`set handler`
`set typehandler`

Related Concepts

breakpoints
eventpoints
eventpoint handlers
tracepoints
watchpoints

Related Parameters

event-handler

set default memory

se de m

Set the default unit size for displaying memory.

Syntax

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

Parameter	Meaning

<memory-unit>	The type of memory unit displayed. The possible types are:
	`byte`
	`halfword`
	`word`
	`longword`
	`quadword`

Description

The `set default memory` command sets the CXdb default for the type of memory unit used to display the contents of memory. This memory unit becomes the default for any new process objects that have not explicitly had their default memory unit set with the `set memory` command. The memory unit affects the appearance of output from the `examine` command.

The types of memory units are:

- `byte` — 8 bits
- `halfword` — 16 bits
- `word` — 32 bits
- `longword` — 64 bits
- `quadword` — 128 bits

To display the current setting of the default memory unit, use the `info cxdb` command.

Each type of memory unit has its own default display format. This format can be set with the `set default format` command.

Examples

The following example illustrates how to set the default display size for memory units.

```
(CXdb) set default memory byte
```

The above command selects a byte as the default unit for displaying the contents of memory.

Related Commands

`examine`	`info cxdb`
`info formatting`	`set default format`
`set default fpmode`	`set format`
`set fpmode`	`set memory`

Related Concepts

displaying data

set default path

se de pa
dp=

Set the default search path.

Syntax

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

Parameter	Meaning
<directory-specifier>	A directory to use as the default search path.
`[, ...]`	An optional list of additional directories to include in the default search path. Multiple directories are separated by commas.

Description

The `set default path` command sets the default search path to the specified directories.

Relative directory names use the console working directory as the base path name. Each new process object that is created after this command receives the new default search path as part of its search path.

The `set default path` command can be used in initialization files to create default search paths automatically.

To display the default search path, use the `info path` command.

Examples

The following examples illustrate how to set the default search path.

```
(CXdb) set default path /usr/smith
```

The above command clears the current setting of the default search and then adds the /usr/smith directory to the empty default search path. This command can be placed in an initialization file to create a default search path automatically.

```
(CXdb) set default path /usr/smith , example2
```

The above command clears the default search path and then sets it to the two listed directories. Notice that the second directory does not start with the slash (/) character. This indicates to CXdb that it is a relative path name, and CXdb assumes the path name starts from the console working directory.

```
(CXdb) set default path .
```

The above command clears the default search path and then sets it to the current console working directory (represented by the dot). If the console working directory changes, the default search path reflects the new console working directory.

Related Commands

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

Related Concepts

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

Related Parameters

directory-specifier

set default pshell

se de ps

Set the default process shell.

Syntax

```
set default pshell {sh | csh}
```

Description

The `set default pshell` command sets the CXdb default process shell to either `sh` or `csh`. Initially, the default process shell is the `csh` if the shell in which CXdb is running is a `csh` or `tcsh`. Otherwise, the default process shell is initially `sh`.

The CXdb default process shell is used to set the process shell for all new process objects. Existing process objects are not affected. The process shell of a process object is the type of shell in which a new process begins execution. It is also the type of shell used to interpret the arguments passed with the `run` command.

To display the current setting of the default process shell, use the `info cxdb` command.

Examples

The following example shows how to set the default process shell.

```
(CXdb) set default pshell csh
```

The above command sets the default process shell to `csh`. A new process object would receive the new CXdb default process shell.

Related Commands

`info cxdb`
`info process`
`rerun`
`run`
`set pshell`
`set shell`

Related Concepts

process object

C Series only

set default remotewd

se de re

Set the default remote working directory.

Syntax

set default remotewd *<directory-specifier>*

Parameter	Meaning
<directory-specifier>	The directory to use as the default remote working directory.

Description

The `set default remotewd` command sets the default remote working directory. The default remote working directory is used if the remote working directory of the process object has not been explicitly set with the `set remotewd` command.

The remote working directory acts as the console working directory for a remote host when doing remote debugging. The remote working directory is used as:

- The base for relative path names specified on a remote host
- The directory for executing new processes on a remote host if a process working directory has not been specified with the `set directory` command

If the remote working directory is not set (by either the `set default remotewd` or `set remotewd` command), CXdb uses the console working directory as the remote working directory. For more information on the remote working directory, read the "remote debugging" concepts reference page.

You can clear the default remote working directory using the `clear default remotewd` command.

Examples

The following examples show how to set the default remote working directory.

```
(CXdb) set default remotewd /doc/cxdb/examples
```

The above command sets the default remote working directory to the /doc/cxdb/examples directory. The next process that is created on a remote machine will be run from this directory unless a directory is explicitly set using the `set remotewd` command.

```
(CXdb) set default remotewd REMOTEPROJ
```

The above command sets the default remote working directory to the value of the `REMOTEPROJ` environment variable. The environment variable is expanded before the default remote working directory is set.

Related Commands

`cd`
`clear default remotewd`
`core`
`debug core`
`debug exec`
`executable`
`info process`
`pwd`
`run`
`rerun`
`set remotewd`

Related Concepts

console working directory
process object
process working directory
remote debugging

Related Parameters

directory-specifier

set default step

se de s

Set the default stepping granularity.

Syntax

`set default step` *<granularity>*

Parameter	Meaning

<granularity>	The desired step size, which can be one of the following:
	`routine`
	`loop`
	`block`
	`statement`
	`expression`

Description

The `set default step` command sets the CXdb default granularity, or step size, for the stepping commands. This default granularity is used by new processes that have not explicitly had their default granularity set with the `set step` command. It is also used by existing processes that have had their default granularity reset with the `clear step` command.

Initially, the CXdb default granularity is statement. To display the current setting of the step size, use the `info cxdb` command.

Examples

The following example illustrates how to set the CXdb default step size (granularity).

```
(CXdb) set default step loop
```

The above command selects `loop` as the default granularity.

Related Commands

`clear step`
`finish`
`info cxdb`
`info process`
`next`
`next over`
`set step`
`step`
`step over`

Related Concepts

source units
stepping

Related Parameters

granularity

C Series only

set directory

se di

Set the process working directory.

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process object.
<directory-specifier>	The directory to use as the process working directory.

Description

The `set directory` command sets the process working directory.

The process working directory is the directory in which a new process (created by the `run` or `rerun` command) is run. If the process is created on a remote host, the process is run in the process working directory on the remote host.

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 be affected by changes to the console working directory.

To display the current process working directory, use the `info process` command.

Examples

The following examples show how to set the process working directory.

```
(CXdb) set directory /mnt/jones/project
```

The above command sets the process working directory to the /mnt/jones/project directory. The next process that is created will be run from the /mnt/jones/project directory.

```
(CXdb) set directory $FMHOME
```

The above command sets the process working directory to the value of the `FMHOME` environment variable. The variable `FMHOME` must be defined in your shell environment before you invoke CXdb. The environment variable is expanded before the process working directory is set.

Related Commands

`cd`
`info process`
`pwd`

Related Concepts

console working directory
default search path
process object
process working directory
remote debugging
search path

Related Parameters

directory-specifier
process-list

set echo

se ec

Enable echoing of input from command files and initialization files.

Syntax

```
set echo
```

Description

The `set echo` command enables echoing of input to CXdb. With echoing enabled, input coming from initialization files and command files is echoed at the CXdb command line.

By default echoing is disabled. You can also disable echoing by using the `clear echo` command. To display the current echo setting, use the `info cxdb` command.

Examples

The following example enables echoing.

```
(CXdb) set echo
```

The above command enables echoing of input. When the `source` command is used to execute a CXdb command file, input from that command file is echoed at the CXdb command line.

Related Commands

`clear echo`
`echo`
`info cxdb`

Related Concepts

cmdlog
command files
initialization files
logging

set environment

se en
env=

Clear and redefine the environment variables for the process environment.

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process object.
<environment-variable>	An environment variable to add after the environment has been cleared.
<string>	The value to be given to the environment variable.
[, ...]	An optional list of additional environment variable assignments. Multiple assignments must be separated by a comma (,).

Description

The `set environment` command clears the environment of the process object and then adds the specified environment variables to the environment.

If the process object does not yet have its own environment, the `set environment` command creates an environment for the process object consisting of the environment variables specified in the command.

Each new process receives the modified environment. An existing process will not be affected.

Examples

The following examples set the environment for the current process object. For these examples, assume that the process object does not yet have its own environment.

```
(CXdb) set environment EDITOR = vi
```

The above command sets the environment of the current process object to be the environment variable `EDITOR`. The `set environment` command indicates to CXdb that you want to modify the environment, so CXdb creates an environment for the current process object. The environment created is cleared, then the environment variable `EDITOR` is added to the empty environment.

You can set multiple environment variables using a single command by separating them with a comma.

```
(CXdb) set environment LESS = -MQ , LIBRARIES = "/usr/lib /mnt/jones/lib"
```

The above command clears the current environment and then adds the two environment variables `LESS` and `LIBRARIES`. In this case, the quotes are needed for the variable `LIBRARIES` because the string 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`

Related Concepts

default environment
environment
process object

Related Parameters

environment-variable
process-list
string

C Series only

set evalopts fpmode

se ev f

Set the floating point mode for evaluating expressions.

Syntax

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

Parameter	Meaning
`ieee`	IEEE floating point mode.
`native`	Native floating point mode.
`dual`	Dual mode. With this setting, CXdb evaluates language expressions by using the same floating point mode (IEEE or native) as the current process.

Description

The `set evalopts fpmode` command sets the floating point mode used by CXdb to evaluate language expressions. The available modes are:

- `ieee` — CXdb uses IEEE floating point mode to evaluate language expressions, regardless of the mode used by the current process.
- `native` — CXdb uses native floating point mode to evaluate language expressions, regardless of the mode used by the current process.
- `dual` — CXdb uses the same floating point mode (either IEEE or native) as the current process.

Dual is the default floating point mode for evaluating language expressions.

To display the current settings of the evalopts, use the `info cxdb` command.

Examples

The following examples illustrate how to select the floating point mode for language expression evaluations done by CXdb.

```
(CXdb) set evalopts fpmode native
```

The above command sets the CXdb floating point mode to native.

```
(CXdb) set evalopts fpmode ieee
```

The above command sets the CXdb floating point mode to IEEE.

```
(CXdb) set evalopts fpmode dual
```

The above command sets the CXdb floating point mode to dual. This means that CXdb will evaluate language expressions in the mode used by the current process.

Related Commands

```
evaluate                        info cxdb
print                           set default fpmode
set evalopts iprecision         set evalopts rprecision
set fpmode
```

Related Concepts

C language expressions
Fortran language expressions
language expressions

Related Parameters

language-expression

set evalopts iprecision

se ev i

Set the size of integer constants for CXdb.

Syntax

set evalopts iprecision {4 | 8}

Parameter	Meaning
4	4-byte integers.
8	8-byte integers.

Description

The `set evalopts iprecision` command sets the size for integer constants used by CXdb in evaluating language expressions. The available sizes are:

- 4-byte integers
- 8-byte integers

The default is 4-byte integers.

To display the current setting of `iprecision`, use the `info cxdb` command.

Examples

The following examples illustrate how to set the size for integer constants used by CXdb in evaluating language expressions.

```
(CXdb) set evalopts iprecision 8
```

The above command selects an integer size of 8 bytes.

```
(CXdb) set evalopts iprecision 4
```

The above command selects an integer size of 4 bytes.

Related Commands

`evaluate`
`print`
`set evalopts rprecision`
`info cxdb`
`set evalopts fpmode`

Related Concepts

C language expressions
Fortran language expressions
language expressions

Related Parameters

language-expression

set evalopts rprecision

se ev r

Set the size of real numbers for CXdb.

Syntax

`set evalopts rprecision {4 | 8}`

Parameter	Meaning
4	Single precision, or 4-byte real numbers.
8	Double precision, or 8-byte real numbers.

Description

The `set evalopts rprecision` command sets the level of precision for real number (floating point) constants used by CXdb in evaluating language expressions. The available precision levels are:

- Single precision (4-byte real)
- Double precision (8-byte real)

Double precision (8-byte real) is the default.

To display the current setting of `rprecision`, use the `info cxdb` command.

Examples

The following examples illustrate how to change the level of precision for floating point constants used by CXdb in evaluating language expressions.

```
(CXdb) set evalopts rprecision 8
```

The above command selects double precision for floating point constants.

```
(CXdb) set evalopts rprecision 4
```

The above command selects single precision for floating point constants.

Related Commands

```
evaluate                    info cxdb
print                       set default fpmode
set evalopts fpmode         set evalopts iprecision
set fpmode
```

Related Concepts

C language expressions
Fortran language expressions
language expressions

Related Parameters

language-expression

C Series only

set fixed sched

se fi s
sfs

Enable fixed scheduling in the process settings.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.

Description

The `set fixed sched` command enables fixed scheduling for the specified processes. Use this command to enable fixed scheduling before running a multithreaded process.

Fixed scheduling means that the process requires the simultaneous use of all the processors on a given machine. With fixed scheduling enabled, the process does not begin executing until all the processors become available.

The default is fixed scheduling disabled. To display the current setting for fixed scheduling, use the `info process` command.

NOTE: Because of the additional system overhead involved with fixed scheduling, it is recommended for debugging multithreaded processes only.

Examples

The following example shows how to enable fixed scheduling.

```
(CXdb) set fixed sched
```

The above command enables fixed scheduling for the current process.

set fixed sched

Related Commands

`clear default fixed sched`
`clear fixed sched`
`info cxdb`
`info process`
`set default fixed sched`

Related Concepts

optimized code
threads

Related Parameters

process-list

set format

se fo

Set the formats for displaying memory.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.

<memory-unit>	The type of memory unit displayed. The possible types are:
	`byte`
	`halfword`
	`word`
	`longword`
	`quadword`

<format>	The format for displaying a memory unit. The possible formats are:
	`binary`
	`character`
	`complex`
	`decimal`
	`eformat` — scientific notation
	`fformat` — floating point notation
	`hexadecimal`
	`logical`
	`octal`
	`unsigned` — unsigned decimal

Description

The `set format` command sets the default memory display formats for specified threads and processes. The formats affect the appearance of output from the `examine` command.

Each format description consists of a memory unit type and its corresponding display format. For example, bytes of data can be displayed as characters, decimal numbers, binary numbers, etc. The memory unit types and their available formats are:

- `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 display the current format settings, use the `info formatting` command.

Examples

The following examples illustrate how to set the memory display formats for all threads of the current process.

```
(CXdb) set format byte decimal
```

The above command selects the decimal format for displaying bytes of memory.

```
(CXdb) set format word unsigned
```

The above command selects the unsigned decimal format for displaying words of memory.

Related Commands

`examine`	`info cxdb`
`info formatting`	`set default format`
`set default fpmode`	`set default memory`
`set fpmode`	`set memory`

Related Parameters

process-list thread-list

C Series only

set fpmode

se fp

Set the floating point mode for the process.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
ieee	IEEE mode for floating point operations.
native	Native mode for floating point operations.

Description

The `set fpmode` command sets the floating point mode of the process to either IEEE or native. The floating point mode determines how a process handles floating point operations.

Initially, a new process uses the floating point mode of its process object. When a process object is created, it receives the default floating point mode of CXdb.

NOTE: If the process changes its floating point mode, CXdb will not automatically change the floating point mode back to the previous setting. You must issues another `set fpmode` command to reset the floating point mode.

Examples

The following example illustrates how to set the floating point mode.

```
(CXdb) set fpmode ieee
```

The above command sets the floating point mode for the current process to be IEEE. If the process later sets its floating point mode to native, CXdb does *not* reset it to IEEE.

Related Commands `set evalopts fpmode` `set default fpmode`

Related Concepts process object

Related Parameters process-list

set handler

se h

Set the handler for a specified eventpoint.

Syntax

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

Parameter	Meaning
<*event-specifier*>	An eventpoint to associate with the specified handler.
[**,** ...]	An optional list of additional eventpoints. Multiple eventpoints must be separated by a comma.
<*event-handler*>	The eventpoint handler defined for the specified eventpoints.

Description

The `set handler` command defines an eventpoint handler for the specified eventpoints.

The eventpoints must exist before the `set handler` command can be used. The handler is immediately associated with the eventpoints. The next time the eventpoint is triggered, the commands of the handler are executed. The handler can be removed with the `clear handler` command.

To display the current eventpoint handlers, use the `info event` or `info eventtype` commands.

Examples

The following examples set handlers for existing eventpoints.

```
(CXdb) set handler 1 {echo "Event 1 reached"; resume;}
```

The above command defines an eventpoint handler for eventpoint 1. The eventpoint handler echoes a message and then resumes execution of the process.

```
(CXdb) set handler 0,2 {echo "Routine 1 reached by: "; print $self;}
```

The above command defines an eventpoint handler for eventpoints 0 and 2. The handler uses the debugger variable `$self` to display the eventpoint number of the currently triggered eventpoint.

```
(CXdb) set handler * {print $pc; print $self; resume;}
```

The above command defines an eventpoint handler for all existing eventpoints. The handler displays the current value of the PC, stored in the debugger variable `$pc`, as well as the current eventpoint number stored in `$self`. The eventpoint handler also resumes execution, making all eventpoints behave like tracepoints.

Related Commands

`clear default handler`
`clear handler`
`clear typehandler`
`info event`
`info eventtype`
`set default handler`
`set typehandler`

Related Concepts

breakpoints
eventpoints
eventpoint handlers
tracepoints
watchpoints

Related Parameters

event-handler
event-specifier

set ignore

se i

Set an ignore count for an eventpoint.

Syntax

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

Parameter	Meaning
<ignore-count>	The number of times an eventpoint is to be ignored.
<event-specifier>	An eventpoint to be ignored.
[, ...]	An optional list of additional eventpoints to be ignored. Multiple eventpoints are separated by commas.

Description

The `set ignore` command creates an ignore count for each specified eventpoint.

An ignore count is the number of times an eventpoint is to be skipped after it is reached. When an eventpoint that has an ignore count is reached, its ignore counter is incremented. CXdb does not trigger the eventpoint, but instead checks to see if another enabled eventpoint exists at the same address.

When the ignore counter reaches the specified number, the next time the eventpoint is reached, its handler is executed. Each new specification of an ignore count for an eventpoint resets its ignore counter. Ignore counts are very useful for allowing locations to be executed numerous times before being stopped.

You can reset an ignore count to 0 by specifying an ignore count of 0 for the eventpoint. To display the current setting of an ignore count, use the `info event` or `info eventtype` commands.

Examples

The following examples illustrate how to set ignore counts.

```
(CXdb) set ignore 5 2
Event 2 will be ignored five times
```

The above command sets an ignore count of `5` for eventpoint 2. When eventpoint 2 is reached, its ignore counter is incremented, and CXdb continues process execution.

```
(CXdb) set ignore 0 0,1,2
Event 0 will be ignored 0 times
Event 1 will be ignored 0 times
Event 2 will be ignored 0 times
```

The above command sets an ignore count of `0` for eventpoints 0, 1, and 2. This command removes any existing ignore count for these eventpoints.

Related Commands

`disable event`
`disable eventtype`
`enable event`
`enable eventtype`
`info event`
`info eventtype`
`remove event`
`remove eventtype`
`set default handler`
`set handler`
`set typehandler`

Related Concepts

breakpoints
eventpoint handlers
eventpoints
tracepoints
watchpoints

Related Parameters

event-specifier

set logging
se l

Enable logging of CXdb input to the viewport files for cmdlog.

Syntax

```
set logging
```

Description

The `set logging` command enables logging to the cmdlog viewports.

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 equivalent to stdin in the shell.

When logging is enabled for cmdlog, everything you enter on the CXdb command line is also sent to the viewports of cmdlog. When logging is disabled, nothing is sent to the viewports of cmdlog. To disable logging, use the `clear logging` command.

The default is logging disabled. To display the current setting of the logging option, use the command `info cxdb`.

Examples

The following example illustrates how to enable logging.

```
(CXdb) set logging
```

The above command enables logging to all the viewports of cmdlog.

Related Commands

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

Related Concepts

cmdlog
logging
viewports

Related Parameters

viewport

set memory

se m

Set the unit size for displaying memory.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.

<memory-unit>	The type of memory unit displayed. The possible types are:
	`byte`
	`halfword`
	`word`
	`longword`
	`quadword`

Description

The `set memory` command selects the default type of memory unit used to display the contents of memory for specified threads and processes. The memory unit setting affects the appearance of output from the `examine` command.

The types of memory units are:

- `byte` — 8 bits
- `halfword` — 16 bits
- `word` — 32 bits
- `longword` — 64 bits
- `quadword` — 128 bits

Each type of memory unit has its own display format. This format can be set with the `set format` command.

To display the current memory unit and format settings, use the `info formatting` command.

Examples

The following examples illustrate how to set the default display size of memory units for specific threads of the current process.

```
(CXdb) set memory byte
```

The above command selects a byte as the default unit for displaying the contents of memory. This command applies to all threads of the current process.

```
(CXdb) :T2 set memory longword
```

The above command selects a longword as the default unit for displaying the contents of memory. This command applies only to thread 2 of the current process; the memory unit settings for all other threads would not change.

Related Commands

examine	info cxdb
info formatting	set default format
set default fpmode	set default memory
set format	set fpmode

Related Parameters

process-list thread-list

set noclobber

se n

Enable the noclobber option for all viewport files.

Syntax

```
set noclobber
```

Description

The `set noclobber` command enables the noclobber option, which protects viewport files that are used for logging or redirection operations.

When noclobber is enabled, an error results if CXdb tries to overwrite an existing viewport file or append to a viewport file that does not exist. When noclobber is disabled, CXdb may overwrite existing viewport files and create new files for appending.

The noclobber option applies to all files specified as viewports with the redirection operators or with the following commands:

```
add cmderr
add cmdlog
add cmdout
set cmderr
set cmdlog
set cmdout
```

The default is noclobber disabled (clear). To display the current setting of the noclobber option, use the `info cxdb` command.

Examples

The following example illustrates how to set the noclobber option.

```
(CXdb) set noclobber
```

The above command enables the noclobber option for all cmderr, cmdlog, and cmdout viewport files.

Related Commands

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

Related Concepts

cmderr	cmdlog
cmdout	logging
redirection	viewports

Related Parameters

redirection-operator	viewport

set path

se pa
p=

Set the search path for the process.

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process object.
<directory-specifier>	A directory to serve as the search path.
[**,** ...]	An optional list of additional directories to include in the search path. Multiple directories are separated by commas.

Description

The `set path` command sets the search path of the process object to the specified directories.

CXdb uses the updated search path the next time it searches for either a source file or the compiler-generated data files. Relative directory names use the console working directory as the base path name.

The `set path` command can be included in command files to create search paths automatically.

NOTE: If your source code was compiled using a version of the CONVEX Fortran compiler later than V7.0 or a version of the CONVEX C compiler later than V4.3, you may not need to specify a search path. These compilers embed the location of the source code and CTI data files for a program in the executable file itself. When using these newer compilers, you generally do not need the `set path` command unless you have changed the location of your source code.

To display the current search path, use the `info path` or `info process` commands.

Examples

The following examples set the search path for the current process object.

```
(CXdb) set path /usr/smith
```

The above command clears the current setting of the search path for the current process and then adds the /usr/smith directory to the empty search path. When CXdb now searches for a source file, it will look in the /usr/smith directory.

```
(CXdb) set path /usr/smith , example2
```

The above command clears the search path and then sets it to the two listed directories. The second directory does not start with the slash (/) character. This indicates to CXdb that it is a relative path name, and CXdb assumes the path name starts from the console working directory.

```
(CXdb) set path .
```

The above command clears the search path and then sets it to the current console working directory. If the console working directory changes, the search path reflects the new console working directory.

Related Commands

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

Related Concepts

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

Related Parameters

directory-specifier	process-list

set printopts maxarray

se prin m

Set the maximum number of array elements to print.

Syntax

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

Parameter	Meaning
<number-of-elements>	A positive integer that specifies the maximum number of array elements to print.

Description

The `set printopts maxarray` command sets the maximum number of array elements displayed by a single execution of the `print` command. The default for `maxarray` is 20.

To display the current print option settings, use the `info formatting` command.

Examples

The following example illustrates how to set the maximum number of array elements (`maxarray`) displayed at one time by the `print` command.

```
(CXdb) set printopts maxarray 25
```

The above command sets `maxarray` to 25.

Assume that your program contains an array called `table_A`, which has 1000 elements, and you issue the following command:

```
(CXdb) print table_A
```

The above command prints only the first 25 elements of `table_A` because `maxarray` limits the output to 25 elements.

Related Commands

`info formatting`		`print`
`set printopts nopadding`		`set printopts padding`
`set printopts precision`		

Related Concepts

displaying data

set printopts nopadding

se prin n

Disable padding with leading zeros when printing.

Syntax

```
set printopts nopadding
```

Description

The `set printopts nopadding` command disables padding with leading zeros for values displayed with the `print` command. Padding helps to align array elements in the printed output.

The default is padding disabled. Use the `set printopts padding` command to enable padding with leading zeros. To display the current print option settings, use the `info formatting` command.

Examples

The following examples illustrate the effects of padding on the `print` command.

```
(CXdb) set printopts nopadding
```

The above command disables padding with leading zeros.

Assume that your program contains an array of integers, called `ARRAY`. With padding disabled, printing the array produces the following result:

```
(CXdb) print ARRAY
(1..4,1) :  1 2 3 4
(1..4,2) :  1 4 9 16
(1..4,3) :  1 8 27 64
(1..4,4) :  1 16 81 256
```

In the above example, the array elements are not properly aligned because padding is disabled.

If padding is enabled, the same array prints as follows:

```
(CXdb) print ARRAY
INTEGER*4(1:4, 1:4)
(1..4,1) :   000000001  000000002  000000003  000000004
(1..4,2) :   000000001  000000004  000000009  000000016
(1..4,3) :   000000001  000000008  000000027  000000064
(1..4,4) :   000000001  000000016  000000081  000000256
```

In the above example, the array elements line up properly because padding is enabled.

Related Commands

`info formatting`
`print`
`set printopts maxarray`
`set printopts padding`
`set printopts precision`

set printopts padding

se prin pa

Enable padding with leading zeros when printing.

Syntax

```
set printopts padding
```

Description

The `set printopts padding` command enables padding with leading zeros for values displayed with the `print` command. Padding helps to align array elements in the printed output.

The default is padding disabled. To display the current print option settings, use the `info formatting` command.

Examples

The following examples illustrate the effects of padding on the `print` command.

```
(CXdb) set printopts padding
```

The above command enables padding with leading zeros.

Assume that your program contains an array of integers, called `ARRAY`. With padding enabled, printing the array produces the following result:

```
(CXdb) print ARRAY
INTEGER*4(1:4, 1:4)
(1..4,1) :   000000001  000000002  000000003  000000004
(1..4,2) :   000000001  000000004  000000009  000000016
(1..4,3) :   000000001  000000008  000000027  000000064
(1..4,4) :   000000001  000000016  000000081  000000256
```

In the above example, the array elements line up properly because padding is enabled.

If padding is disabled, the same array prints as follows:

```
(CXdb) print ARRAY
(1..4,1) :  1 2 3 4
(1..4,2) :  1 4 9 16
(1..4,3) :  1 8 27 64
(1..4,4) :  1 16 81 256
```

In the above example, the array elements are not properly aligned because padding is disabled.

Related Commands

`info formatting`
`print`
`set printopts maxarray`
`set printopts nopadding`
`set printopts precision`

set printopts precision

se prin pr

Set the precision used to print floating point numbers.

Syntax

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

Parameter	Meaning
<width>	The total field width (or maximum number of characters) to display, including the decimal point. This value must be a positive integer.
<precision>	The maximum number of digits to display to the right of the decimal point. This value must be a positive integer.

Description

The `set printopts precision` command sets the precision used for displaying floating point numbers with the `print` command. The default precision is 10.4.

To display the current print option settings, use the `info formatting` command.

Examples

The following example illustrates how to set the precision for floating point values displayed with the `print` command.

```
(CXdb) set printopts precision 8.2
```

The above command sets the precision to 8.2 for the `print` command. The maximum number of characters displayed is 8, and the number of digits displayed to the left of the decimal point is 2.

Assume that your program contains the variable `AVG`, which has a current value of 118.84616, and you issue the following command:

```
(CXdb) print AVG
REAL*4 118.85
```

The above command prints the value of AVG with a precision of 8.2. CXdb rounds the value of AVG to compensate for the digits that are not displayed.

Related Commands

`info formatting`
`print`
`set printopts maxarray`
`set printopts nopadding`
`set printopts padding`

Related Concepts

displaying data

set pshell

se ps

Set the process shell.

Syntax

[*<process-list>*] **set pshell** {**sh** | **csh**}

Parameter	Meaning
<process-list>	A list of processes affected by this command.

Description

The `set pshell` command sets the process shell of a process object to either `sh` or `csh`.

The process shell is the type of shell from which CXdb begins execution of a new process. The process shell is also the type of shell used to interpret arguments passed using the `run` command.

Initially, the process shell for a process object is the setting of the CXdb default process shell when the process object is created. To display the current setting of the process shell, use the `info process` command.

Examples

The following example sets the process shell.

```
(CXdb) set pshell csh
```

The above command sets the process shell for the current process object to be the C shell. The next time a process is created, execution begins from a C shell. The arguments passed using the `run` command are interpreted by this shell.

Related Commands

`info cxdb`
`info process`
`rerun`
`run`
`set default pshell`
`set shell`

Related Concepts

process object

Related Parameters

process-list

C Series only

set remotewd

se r

Set the remote working directory.

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process object.
<directory-specifier>	The directory to become the remote working directory.

Description

The `set remotewd` command sets the remote working directory.

The remote working directory acts as the console working directory for the remote host during a remote debugging session. The remote working directory is used as:

- The base for relative path names specified on a remote host
- The directory for executing new processes on a remote host if a process working directory has not been specified for the process object using the `set directory` command

If the remote working directory is not set by the `set remotewd` command, CXdb uses the directory specified by the `set default remotewd` command. If neither of these is set, CXdb uses the console working directory as the remote working directory. For more information on the remote working directory, read the "remote debugging" concepts reference page.

Examples

The following examples show how to set the remote working directory.

```
(CXdb) set remotewd /mnt/jones/project
```

The above command sets the remote working directory to the /mnt/jones/project directory. The next process that is created on a remote host will be run from this directory.

```
(CXdb) set remotewd $REMOTEPROJ
```

The above command sets the remote working directory to the value of the `$REMOTEPROJ` environment variable. The environment variable is expanded before the remote working directory is set.

Related Commands

`cd`
`clear default remotewd`
`core`
`debug core`
`debug exec`
`executable`
`info process`
`pwd`
`run`
`rerun`
`set default remotewd`

Related Concepts

console working directory
process object
process working directory
remote debugging

Related Parameters

directory-specifier
process-list

C Series only

set seq

se se

Set the sequential mode (SEQ) bit.

Syntax

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

Parameter	Meaning
<*process-list*>	A list of processes affected by this command. The default is the current process.
<*thread-list*>	A list of threads affected by this command. The default is all threads of the current process.

Description

The `set seq` command sets the sequential mode (SEQ) bit of the processor status word (PSW).

The SEQ bit controls pipelining within the processor. If this bit is set, the processor executes all instructions sequentially; that is, the execution of the next instruction is initiated only after the previous instruction has been executed. If this bit is clear, the processor operates with maximum pipelining and overlap.

The default is SEQ set. To display the current setting of the SEQ bit, use the `info psw` command.

For more information about the PSW and the SEQ bit on C Series machines, refer to *Convex C-Series Architecture* (Order No. DSW-300).

Examples

The following example illustrates how to set the SEQ bit.

```
(CXdb) set seq
```

The above command sets the SEQ bit for all threads of the current process.

Related Commands

`clear seq`
`clear sqs`
`info psw`
`set fixed sched`
`set sqs`

Related Parameters process-list thread-list

set shell

se sh

Set the type of shell invoked from within CXdb.

Syntax

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

Description

The `set shell` command sets the shell type to be used when a `shell` command is executed.

The possible shell types are:

`sh` — The Bourne shell.
`ksh` — The Korn shell.
`csh` — The C shell.
`tcsh` — The `tc` shell.

Initially the shell type is the type from which CXdb was invoked. To display the current shell setting, use the `info cxdb` command.

Examples

The following example sets the shell type.

```
(CXdb) set shell csh
```

The above command sets the shell type to `csh`. The next `shell` command that does not specify a shell type will use the C shell.

Related Commands

`info cxdb`
`set pshell`
`info process`
`shell`

set signal

se si

Set the actions for the specified signal.

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process object.
<signal-specifier>	The signal whose actions you want to set. You can use either the signal name or signal number as the specifier. Signal names are not case sensitive.

Description

The `set signal` command sets the actions for the specified signal.

Three actions can occur when CXdb catches a signal sent to the process. The only signals sent to the process that CXdb does not catch are those sent from CXdb. The possible actions are described below.

- `stop` — Stop process execution when CXdb catches the signal.
- `pass` — Pass the signal to the process when process execution resumes.
- `print` — Print a message when CXdb catches the signal.

Each of the actions can be set by using the name of the action. The actions can be unset by prefixing the name with `no` (for example, `nopass`). When specifying more than one action in a single command, use a comma between action names. Actions not specified on the command line are left unchanged.

Signals names and numbers vary between C Series and SPP Series machines.To display the names, numbers, and default actions for signals, use the `info signal` command.

Examples

The following commands set the different actions for the signal `SIGINT`.

```
(CXdb) set signal SIGINT stop
```

The above command sets the `stop` action for the `SIGINT` signal. When CXdb catches a `SIGINT` signal, process execution is stopped. The other actions, `pass` and `print`, are left unchanged.

```
(CXdb) set signal SIGINT print
```

The above command sets the `print` action for the `SIGINT` signal. When CXdb catches the `SIGINT` signal, a message is printed. This occurs even if process execution is not stopped.

```
(CXdb) set signal SIGINT pass
```

The above command sets the `pass` action for the `SIGINT` signal. When CXdb catches the `SIGINT` signal, the signal is passed to the process. If process execution is stopped, the signal is sent when process execution resumes. If process execution is not stopped, the process receives the signal immediately.

```
(CXdb) set signal 2 stop, nopass, noprint
```

The above command sets the `stop` action and unsets the `pass` and `print` actions for the `SIGINT` signal. (Comma separators are required.) In this case the signal number was used rather than the signal name. When CXdb catches the signal, process execution is stopped. No message is printed. No signal is passed when process execution resumes.

Related Commands

`evaluate`
`info signal`
`print`
`signal process`
`signal thread`

Related Concepts

signals

Related Parameters

process-list
signal-specifier

C Series only

set sqs

se sq

Set the sequential store enable (SQS) bit.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the current process.

Description

The `set sqs` command sets the sequential store enable (SQS) bit of the processor status word (PSW).

If the SQS bit is set, all stores to memory occur in instruction execution order. If this bit is clear, stores to memory can occur in nonsequential order.

The default is SQS set. To display the current setting of the SQS bit, use the `info psw` command.

For more information about the PSW and the SQS bit, refer to *Convex C-Series Architecture* (DSW-300).

Examples

The following example illustrates how to set the SQS bit.

```
(CXdb) set sqs
```

The above command sets the SQS bit for all threads of the current process.

Related Commands

`clear seq`
`clear sqs`
`info psw`
`set fixed sched`
`set seq`

Related Parameters

process-list
thread-list

set step

se st

Set the stepping granularity.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.

<granularity>	The desired step size, which may be one of the following:
	`routine`
	`block`
	`loop`
	`statement`
	`expression`

Description

The command `set step` sets the default granularity, or step size, for all threads of the specified process. This default granularity is used by stepping commands that do not explicitly specify a different granularity.

To display the current default granularity for a particular process, use the `info process` command.

Examples

The following examples illustrate how to set the default granularity for the current process.

```
(CXdb) set step expression
```

The above command selects `expression` as the default granularity for all threads of the current process.

```
(CXdb) set step block
```

The above command selects `block` as the default granularity for all threads of the current process.

Related Commands

`clear step`	`finish`
`info cxdb`	`info process`
`next`	`next over`
`set default step`	`step`
`step over`	

Related Concepts

source units

stepping

Related Parameters

granularity

process-list

set threads

se th

In X Windows mode, associate windows with particular threads.

Syntax

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

Parameter	Meaning
<*window*>	The object number of a CXdb window. The window must be able to be made thread-specific.
[, ...]	A list of additional windows. Multiple windows are separated by commas.
<*thread-number*>	The number of the thread to associate with the window.
[, ...]	A list of additional threads. Multiple threads in the list must be separated by commas.

Description

The `set threads` command sets the threads associated with any of the following windows:

- Source Code window
- Assembly Code window
- Memory Display window
- Stack Trace window
- Process Status Word window
- Various register windows

The `set threads` command has no effect in line mode (when you invoke CXdb with the `-nw` option).

In the X Windows interface, you can also set the threads for a window by using the Threads dialog. You can open this dialog for a specific window by selecting the threads option from the first (leftmost) menu of any window that can be associated with specific threads.

Examples

The following examples illustrate how to set threads for various windows.

```
(CXdb) set threads 2 0
```

The above command associates window 2 with thread 0 of the current process.

```
(CXdb) set threads 3,4 0,1
```

The above command associates windows 3 and 4 with threads 0 and 1.

Related Commands

`display disassembly`
`display examine`
`display routine`
`display source`
`display stack`
`info threads`

Related Concepts

optimized code
threads

Related Windows

Threads dialog

set typehandler

se t

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

Syntax

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

Parameter	Meaning

<eventtype-specifier>	An eventtype to associate with the specified eventpoint handler. Possible eventtypes are:
	`break`
	`trace`
	`watch`
	`exec`
	`join`
	`modify`
	`reached`
	`relation`
	`signal`
	`spawn`
	`*` (all)

`[, ...]`	An optional list of additional eventtypes. Multiple eventtypes are separated by commas.
<event-handler>	The eventpoint handler to assign to the specified eventtypes.

Description

The `set typehandler` command defines the default handler for the specified eventtypes.

If an eventpoint does not have its own handler, then it uses the default handler for its eventtype. If the eventtype does not have its own handler, then the default handler for general eventpoints is used.

When a default handler for a given type is changed, the new setting can be displayed using the `info eventtype` command. The handler can be removed using the `clear typehandler` command.

Examples

The following examples illustrate how to define a default handler for various eventtypes.

```
(CXdb) set typehandler trace {echo "Reached tracepoint: "; print $self;
       resume; }
```

The above command sets the default handler for tracepoints. The default handler echoes a message, displays the current eventpoint number stored in the debugger variable `$self`, and resumes process execution. All tracepoints that do not have a specified handler will use the new default tracepoint handler when they are next triggered.

```
(CXdb) set typehandler break, reached {echo "Reached eventpoint: "; print
       $self; }
```

The above command sets the default handler for breakpoints and event reached eventpoints. The handler echoes a message and displays the current eventpoint number stored in the debugger variable `$self`. Process execution does not resume.

```
(CXdb) set typehandler * {echo "Using default handler"; print $self;}
```

The above command sets the default handler for all eventtypes. The new handler echoes a message and displays the current value of the `$self` debugger variable. Process execution does not resume. This command sets all eventpoints, no matter what the type, to perform the same function, unless the eventpoint has its own eventpoint handler.

Related Commands

`clear default handler`
`clear handler`
`clear typehandler`
`info event`
`info eventtype`
`set default handler`
`set handler`

Related Concepts

breakpoints
debugger variables
eventpoints
eventpoint handlers
tracepoints
watchpoints

Related Parameters

event-handler
eventtype-specifier

shell
sh

Invoke a shell.

Syntax

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

Parameter	Meaning
<shell-specifier>	The type of shell to use. The possible shell types are: `sh` — The Bourne shell `ksh` — The Korn shell `csh` — The C shell `tcsh` — The `tc` shell
<shell-commands>	A *<string>* of shell commands to be executed in the opened shell.

Description

The `shell` command opens a shell outside of CXdb.

If a shell command string is included on the command line, it is passed to the shell as a command. Upon completion of the command, control returns to CXdb. If a shell command string is not included on the command line, the shell is interactive.

The shell operates in the same way as if you invoked it from another shell. A specific type of shell can be opened by specifying a shell type on the command line. If a shell type is not specified, the current setting for the shell type is used. Initially the shell type is the same as the shell from which CXdb was invoked. The shell type can be set with the `set shell` command and displayed with the `info cxdb` command.

In line mode, only one shell can be open at a time. With the X Windows interface, multiple shells can be opened at once.

Examples

The following examples illustrate how to open shells from within CXdb.

```
(CXdb) shell
```

The above command opens an interactive shell of the current shell type.

```
(CXdb) shell /ksh
```

The above command opens an interactive Korn shell. This does *not* change the current shell type in CXdb.

```
(CXdb) shell ls
```

The above command opens a shell of the current shell type. The string `ls` is passed to the shell as a command. When the command finishes, the shell is exited.

```
(CXdb) shell /ksh "cd /usr/smith; ls"
```

The above command opens a Korn shell. The string "`cd /usr/smith; ls`" is passed to the shell as a command. The string must be delimited by quotes or double quotes because it contains white space characters (blanks) and a semi-colon (`;`).

Related Commands

`info cxdb`
`info process`
`set pshell`
`set shell`

Related Parameters

string

signal process

sig p

Send a signal to the process and continue process execution.

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process object.
<signal-specifier>	The signal to be sent to the process. The signal specifier can be either the signal name or the signal number. Signal names are not case sensitive.
`&`	Runs the command in the background.

Description

The `signal process` command sends the specified signal to the process and automatically continues process execution. This command can be used to control which signal is sent to your process.

Process execution resumes in the same manner as if a `continue` command had been issued, except that the signal is immediately sent to the process. The process must be stopped before you can issue the `signal process` command. The signal can be received by any existing thread of the process. Because the signal is generated by CXdb, CXdb does not catch the signal.

NOTE: Signal names and numbers vary between C Series and SPP Series architectures. To list signal names, numbers, and default actions, use the `info signal` command.

Examples

The following examples send the `SIGINT` signal to the process.

```
(CXdb) signal process SIGINT
```

The above command sends the `SIGINT` signal to the current process. Process execution resumes and one thread of the process receives the signal. Process execution continues until the process terminates or is stopped.

```
(CXdb) signal process 2 &
```

The above command again sends the `SIGINT` signal to the current process. The signal number is used instead of the signal name. By using the `&` flag, this command is placed in the background. Thus process execution continues, but the CXdb command prompt returns, allowing you to enter other CXdb commands.

Related Commands

`continue`
`event signal`
`info signal`
`set signal`
`signal thread`

Related Concepts

background execution
signals

Related Parameters

process-list
signal-specifier

signal thread

sig t

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

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process object.
<thread>	A single thread to which the signal is sent.
<signal-specifier>	The signal to be sent to the process. The signal specifier can be either the signal name or the signal number. Signal names are not case sensitive.
&	Runs the command in the background.

Description

The `signal thread` command sends the specified signal to the specified thread of the process; then it continues execution of that thread. Only one thread can be specified with this command.

The thread resumes execution in the same manner as if a `continue` command had been issued for that thread, except that the signal is immediately sent to the thread. The thread must be stopped before the `signal thread` command is issued. Because the signal is generated by CXdb, CXdb does not catch the signal.

The `signal thread` command can be used to control what signal is sent to which thread of your process. If a thread is not specified and the process has only one thread, that thread receives the signal. If a thread is not specified and multiple threads exist, it is an error.

NOTE: Signal names and numbers vary between C Series and SPP Series architectures. To list signal names, numbers, and default actions, use the `info signal` command.

Examples

The following examples illustrate how to send the `SIGINT` signal to a particular thread.

```
(CXdb) :T0 signal thread SIGINT
```

The above command sends the `SIGINT` signal to thread `0` of the current process. Thread `0` resumes execution and receives the signal. Thread `0` continues to execute until it is finished or it is stopped.

```
(CXdb) :T1 signal thread 2 &
```

The above command sends the `SIGINT` signal to thread 1 of the current process. The signal number is used instead of the signal name. Thread `1` receives the signal and continues execution. By using the `&` flag, this command is placed in the background. Thus, thread execution continues, but the CXdb command prompt returns, allowing you to enter other CXdb commands.

Related Commands

`continue`
`event signal`
`info signal`
`set signal`
`signal process`

Related Concepts

background execution
signals
threads

Related Parameters

process-list
signal-specifier
thread-list

source

sou

Execute a CXdb command file.

Syntax

source *<file-name>*

Parameter	Meaning
<file-name>	The name of a CXdb command file or initialization file.

Description

The `source` command executes a CXdb command file.

A command file is any file that contains a sequence of CXdb commands. You can create the command file outside of CXdb by using a standard editor such as `emacs` or `vi`. The lines of the command file must conform to the grammar and syntax rules of the CXdb command language. You can also use the `source` command within a command file.

When you use `source` on a command file, CXdb reads one line of the file at a time and executes it just as if you had typed that line in the Command window yourself. If echo is enabled, each line of the command file is echoed in the Command window as it is executed. If logging is enabled, the input lines from the command file also go to the cmdlog viewports. Any output goes to the cmdout viewports.

If any line of the command file causes an error, the error message goes to the cmderr viewports. CXdb ignores the particular line in which the error occurred, and it continues to execute the other lines of the command file in sequence.

Examples

Assume that you have a file called example_aliases in the console working directory, and the file contains the following lines:

```
alias ic 'info cxdb'
alias ig 'info globals'
alias il 'info locals'
alias ip 'info process'
```

In the Command window, you enter the following command:

```
(CXdb) source example_aliases
(CXdb) alias ic 'info cxdb'
(CXdb) alias ig 'info globals'
(CXdb) alias il 'info locals'
(CXdb) alias ip 'info process'
```

The above command executes the command file example_aliases. If echoing is enabled, the lines of the command file are echoed in the Command window, as shown above.

In this case, the command file defines a set of aliases that can be used during the rest of the debugging session.

Related Commands

`clear logging`
`clear echo`
`info cxdb`
`set echo`
`set logging`

Related Concepts

command files
cmderr
cmdlog
cmdout
initialization files
logging
viewports

Related Parameters

file-name

step

ste

s

Step to the next source unit.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.

<granularity>	The type of source unit, or step size. Available granularities are:
	`routine`
	`block`
	`loop`
	`statement`
	`expression`
	If you do not specify a granularity, CXdb uses the default granularity of the specified process.

<count>	The number of times to repeat this command. The default is 1.
&	Runs the command in the background.

Description

The `step` command continues execution of your process until it reaches the next source unit of the specified granularity. If the current routine does not contain another source unit of the specified granularity, then the process continues executing until it reaches the end of the current routine.

Examples

The 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) points 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 the default granularity is statement, the above command steps the current process by one statement. When execution stops, the PC points to the beginning of line 5.

```
(CXdb) step 4
Stepping process [#0/*] by 4 statements
Process [#0/0] stopped stepping at [0x80001a88] SUB5A in chapter5.f line 23
```

The above command steps the current process by four statements because the default granularity is statement. The first statement source unit executed is the assignment `I=1` on line 5. The second statement source unit executed is the print statement on line 6. The third statement source unit executed is line 7, which is a call to subroutine `SUB5A`. Therefore, the fourth statement source unit executed is line 22 in `SUB5A`. When the process stops, the PC points to the beginning of line 23 in `SUBA`.

```
(CXdb) step loop
Stepping process [#0/*] by 1 loop
Process [#0/0] stopped stepping at [0x80001af8] SUB5A in chapter5.f line 26
```

The above command steps the process to the beginning of the next loop. When the process stops, the PC points to the beginning of line 26.

Related Commands

`finish`	`info cxdb`
`info line`	`info process`
`info sourceunit`	`next`
`next instruction`	`next over`
`set default step`	`set step`
`step instruction`	`step over`

Related Concepts

process object
source units
stepping

Related Parameters

granularity
process-list
thread-list

step instruction

ste i

si, stepi

Step to the next instruction.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.
<count>	The number of times to repeat this command. The default is 1.
&	Runs the command in the background.

Description

The `step instruction` command steps the process by the specified number of machine instructions.

To display the machine instructions for the process, use the `disassemble` command or open the Assembly Code window.

Examples

The following examples illustrate how to step a process by machine instructions.

```
(CXdb) step instruction
Stepping process [#0/*] by 1 instruction
Process [#0/0] stopped stepping at [0x800053b6] EXAMPLE in example.f line 7
```

The above command steps the current process by one machine instruction.

```
(CXdb) step instruction 5
Stepping process [#0/*] by 5 instructions
Process [#0/0] stopped stepping at [0x800208d8] _for$s_wsle+0x12
```

The above command steps the current process by five machine instructions.

step instruction

Related Commands	`disassemble`	`finish`
	`next`	`next instruction`
	`next over`	`step`
	`step over`	
Related Concepts	process object	stepping
Related Parameters	process-list	thread-list

step over

ste o

so

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

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.

<granularity>	The type of source unit, or step size. Available granularities are:
	`routine`
	`block`
	`loop`
	`statement`
	`expression`
	If you do not specify a granularity, CXdb uses the default granularity of the specified process.

<count>	The number of times to repeat this command. The default is 1.
`&`	Runs the command in the background.

Description

The `step over` command continues execution of your process until it reaches the next source unit of default granularity. In searching for the target source unit, the `step over` command ignores the current source unit of specified granularity.

The current source unit is one that 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, and all of these source units can be current at the same time. However, the only one of interest here is the current source unit that has the granularity specified in the `step over` command. If none of the current source units are of the specified granularity, then the `step over` command uses the current source unit of default granularity.

Examples

The 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 over
Stepping process [#0/*] by 1 statement outside current statement
Process [#0/0] stopped stepping at [0x800017ec] CHAPTER5 in chapter5.f line 5
```

Because the default granularity is statement, the above command steps the process over the current statement and stops execution at the beginning of the next statement. Before this command was executed, line 4 was the current source unit of statement granularity. When execution stops, the PC is pointing to the beginning of line 5, which is the next source unit of statement granularity.

```
(CXdb) step over 3
Stepping process [#0/*] by 3 statements outside current statement
Process [#0/0] stopped stepping at [0x80001a36] SUB5A in chapter5.f line 22
```

The above command steps the process over the current statement and stops execution at the beginning of the next statement after that. Again, this is because the default granularity is statement. A repetition factor is specified, so the command executes three times. Notice that line 7 is a call to subroutine `SUB5A`. This call is executed as one of the three repetitions of the command. Therefore, when the process stops, the PC points to the beginning of line 22, which is the first executable source unit in `SUB5A`.

```
(CXdb) step over routine
Stepping process [#0/*] by 1 statement outside current routine
Process [#0/0] stopped stepping at [0x80001a88] SUB5A in chapter5.f line 23
```

The above command steps the process over the current routine and stops execution at the next statement after that. Before this command was executed, the PC pointed to line 22 in subroutine `SUB5A`. There is no current *routine* source unit at line 22. Subroutine `SUB5A` is active because it contains the current point of execution, but it is not the current routine because the starting address of `SUB5A` is not indicated by the current value of the PC. Therefore, the `step over` command ignores the specified granularity of routine and reverts to the default granularity of statement. The net result is that the process steps by only one statement, and the PC now points to line 23.

```
(CXdb) step over loop
Stepping process [#0/*] by 1 statement outside current loop
Process [#0/0] stopped stepping at [0x80001bd8] SUB5A in chapter5.f line 32
```

The above command steps the process over the current loop and stops execution at the next statement after that. The current loop begins on line 23 and ends on line 31. 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`

Related Concepts

process object
source units
stepping

Related Parameters

granularity
process-list
thread-list

stop
sto

Stop execution of the process.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.

Description

The `stop` command stops execution of a process.

The process must currently be running for the `stop` command to have any effect. The `stop` command stops all threads in the process.

The `stop` command is used to stop process execution controlled by a command that is running in the background. If the process execution command is not running in the background, the CXdb command prompt is not available for you to enter the `stop` command.

Even if the process execution command is not running in background, you can stop the process by typing **CTRL-c** in the Command window. This kills the command that started process execution. Typing **CTRL-c** in the process interface window sends the interrupt signal to the process, which is caught by CXdb before the process receives it. Unless the handler for the interrupt signal has been redefined, this will stop the process.

Examples

The following example stops the process.

```
(CXdb) stop
```

The above command stops execution of all threads of the current process. For this command to work, the command that began process execution must be running in background.

Related Commands

`continue`	`rerun`
`run`	`quit`
`signal process`	`signal thread`

Related Concepts

background execution	process object

Related Parameters

process-list

trace instruction

t i
ti

Set a tracepoint at an instruction.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.
<language-expression>	A language expression that evaluates to a valid instruction address.
<event-handler>	A sequence of CXdb commands enclosed within curly braces ({ }). Each command in the handler must be terminated by a semicolon (;).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

The `trace instruction` command sets a tracepoint at the specified instruction address. The address can be specified as any valid language expression.

When the tracepoint is triggered, process execution stops and the commands of the tracepoint's handler are executed. If the tracepoint does not have its own handler, CXdb executes the default handler for tracepoints, which displays a message and then resumes process execution.

Examples

The following examples set tracepoints at specific instruction addresses.

```
(CXdb) trace instruction CLEAR_ARRAY

#0: trace instruction, on [#0/*], Enabled, ignore 0/0
        [0x80005694] CLEAR_ARRAY in example.f line 50
```

The above command sets a tracepoint at the first instruction of the routine **CLEAR_ARRAY**. The evaluation of the language expression **CLEAR_ARRAY** is used as the address for this tracepoint. When a routine name is used with a `trace instruction` command, the tracepoint is placed before the preamble (which manages the stack) of the routine. In contrast, a routine name used with a `trace routine` command places the tracepoint at the first executable source unit of the routine.

When you create a tracepoint, CXdb responds by displaying the following information:

- `#0:` — The eventpoint number used to identify this particular eventpoint in other CXdb commands. In this case, the tracepoint number is 0.
- `trace instruction` — The type of tracepoint.
- `on [#0/*], Enabled, ignore 0/0` — The tracepoint is set on process object 0, for all threads (*). It is enabled, and does not have an ignore count.
- `[0x80005694]` — The hexadecimal address for the location of the tracepoint. In this case the address is `80005694`.
- `CLEAR_ARRAY in example.f line 50` — The symbolic location of the tracepoint. In this case the tracepoint is in the routine `CLEAR_ARRAY` at line 50 of the source file example.f.

You can also display the above information by using the `info event` command.

When the tracepoint is triggered, execution is stopped before the instruction at the specified address is executed. A message is displayed to tell you that this tracepoint was reached, then process execution is resumed.

The syntax for specifying an absolute address is different between Fortran and C. The next two examples demonstrate this difference.

Using Fortran syntax:

```
(CXdb) trace instruction '80005694'x

#1: trace instruction, on [#0/*], Enabled, ignore 0/0
        [0x80005694] CLEAR_ARRAY in example.f line 50
```

The above command sets a tracepoint at the absolute address `80005694`. The tracepoint number is `1`, located at address `80005694` in routine `CLEAR_ARRAY` at line `50` of the file example.f. The notation `'80005694'x` is Fortran-specific and indicates that the address is in hexadecimal notation.

Using C syntax:

```
(CXdb) trace instruction 0x80004718

#2: trace instruction, on [#0/*], Enabled, ignore 0/0
        [0x80004718] chapter7C`isqr in chapter7C.c line 23
```

The above command sets a tracepoint at the absolute address `80004718`. The `0x` is the C notation for a hexadecimal number. The symbolic location uses the scope path of `chapter7C`isqr` to indicate the source file (chapter7C) and routine (`isqr`) in which the tracepoint is located.

When you specify an absolute address, the tracepoint is set at the closest even boundary. Because of this, you must be sure that the address is actually the starting address for the instruction. If the tracepoint is placed at an address in the middle of an instruction, the tracepoint will be interpreted as a portion of the instruction, which can cause unpredictable results.

```
(CXdb) trace instruction CLEAR_ARRAY {echo 'routine CLEAR_ARRAY
       reached';}

#3: trace instruction, on [#0/*], Enabled, ignore 0/0
        [0x80005694] CLEAR_ARRAY in example.f line 50
    {
        echo 'routine CLEAR_ARRAY reached';
    }
```

The above command sets a tracepoint at address `80005694`, the starting address of routine `CLEAR_ARRAY`. An eventpoint handler is defined for the tracepoint. When the tracepoint is triggered, execution is stopped and then the `echo` command is executed. Even though the eventpoint is a tracepoint, execution is not resumed because the specified handler overrides the default handler for tracepoints.

```
(CXdb) trace instruction '80005694'x \; $Trace4

#4: trace instruction, on [#0/*], Enabled, ignore 0/0
        [0x80005694] CLEAR_ARRAY in example.f line 50
```

The above command creates a new tracepoint at the absolute address `80005694`. The language expression terminator `\;` is needed to separate the language expression from the debugger variable. The debugger variable `$Trace4` is created and set equal to the number of this eventpoint. In subsequent commands you can use `$Trace4` to refer to this tracepoint. Debugger variables allow you to refer to eventpoints without having to remember their eventpoint numbers.

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`
`resume`	`set default handler`
`set handler`	`set typehandler`
`trace line`	`trace routine`
`trace source`	`watch`

Related Concepts

breakpoints	debugger variables
eventpoints	eventpoint handlers
tracepoints	watchpoints

Related Parameters

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

trace line

t l
tl

Set a tracepoint at a source line.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.
<line-specifier>	The line number where the tracepoint is set. The line number must be an integer, and may be preceded by a source file name.
<event-handler>	A sequence of CXdb commands enclosed within curly braces (`{ }`). Each command in the handler must be terminated by a semicolon (`;`).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

The `trace line` command sets a tracepoint before the first statement on the specified line.

If the line number does not map to a source line (whether due to optimizations or the line being a comment line), CXdb asks if you want the tracepoint set at the next highest line number that maps to a source line.

When the tracepoint is triggered, process execution stops and the commands of the tracepoint's handler are executed. If the tracepoint does not have its own handler, the default handler for tracepoints, which displays a message and resumes process execution, is executed.

Examples

The following examples set tracepoints at specific source lines.

```
(CXdb) trace line 18

#0: trace line, on [#0/*], Enabled, ignore 0/0
        [0x8000545e] EXAMPLE in example.f line 18
```

The above command sets a tracepoint at the starting address that corresponds to line 18 of the current source file.

When you create a tracepoint, CXdb responds by displaying the following information:

- `#0:` — The eventpoint number used to identify this particular eventpoint in other CXdb commands. In this case, the tracepoint number is 0.
- `trace line` — The type of tracepoint.
- `on [#0/*], Enabled, ignore 0/0` — The tracepoint is set on process object 0, for all threads (*). It is enabled, and does not have an ignore count.
- `[0x8000545e]` — The hexadecimal address for the location of the tracepoint. In this case the address is `8000545e`.
- `EXAMPLE in example.f line 18` — The symbolic location of the tracepoint. In this case the tracepoint is in the routine `EXAMPLE` at line 18 of the source file example.f.

You can also display the above information by using the `info event` command.

When the tracepoint is triggered, execution stops before the first instruction of the first statement on that line is executed. A message is displayed, telling you that this tracepoint has been reached. Process execution then resumes.

```
(CXdb) trace line example.f:31

#1: trace line, on [#0/*], Enabled, ignore 0/0
        [0x800054d8] EXAMPLE in example.f line 31
```

The above command sets a tracepoint at the starting address of line `31` of the source file example.f. This source file must be part of the compilation of the current executable file and be included in the search path of the process object.

```
(CXdb) trace line 18 {echo 'Line 18 reached';}

#2: trace line, on [#0/*], Enabled, ignore 0/0
        [0x8000545e] EXAMPLE in example.f line 18
    {
        echo 'Line 18 reached';
    }
```

The above command sets a tracepoint at the starting address of line `18` of the current source file. An eventpoint handler is defined for the tracepoint. When the tracepoint is triggered, process execution stops, and the commands of the eventpoint handler are executed. The eventpoint handler displays a message. Even though the eventpoint is a tracepoint, execution is not resumed because the specified handler overrides the default handler for tracepoints.

```
(CXdb) trace line 18 $Trace3

#3: trace line, on [#0/*], Enabled, ignore 0/0
        [0x8000545e] EXAMPLE in example.f line 18
```

The above command creates a new tracepoint at line `18`. The debugger variable `$Trace3` is created and set equal to the number of this eventpoint. In subsequent commands you can use `$Trace3` to refer to this tracepoint. Debugger variables allow you to refer to eventpoints without having to remember their eventpoint numbers.

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`
`resume`	`set default handler`
`set handler`	`set typehandler`
`trace instruction`	`trace routine`
`trace source`	`watch`

Related Concepts

breakpoints	debugger variables
eventpoints	eventpoint handlers
tracepoints	watchpoints

Related Parameters

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

trace routine

t r
tr

Set a tracepoint at the beginning of a routine.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.
<language-expression>	A language expression that evaluates to a valid instruction address.
<event-handler>	A sequence of CXdb commands enclosed within curly braces (`{ }`). Each command in the handler must be terminated by a semicolon (`;`).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

The `trace routine` command sets a tracepoint at the first executable source unit of the routine containing the specified instruction address. If there are multiple entry points into the routine, a tracepoint is set at each entry point.

The specified address must be a valid language expression that evaluates to an address. CXdb finds the routine that contains this address and places the tracepoint at its first executable source unit. The first executable source unit is usually the first statement of a routine, unless there are local variable intializations.

When the tracepoint is triggered, process execution stops and the commands of the tracepoint's handler are executed. If the tracepoint does not have its own handler, the default handler for tracepoints, which displays a message and then resumes process execution, is executed.

Examples

The following examples set tracepoints at the first executable source unit of a routine.

```
(CXdb) trace routine CLEAR_ARRAY

#0: trace routine, on [#0/*], Enabled, ignore 0/0
        [0x80005696] CLEAR_ARRAY in example.f line 53
```

The above command sets a tracepoint at the first executable source unit of the routine `CLEAR_ARRAY`.

When you create a tracepoint, CXdb responds by displaying the following information:

- `#0:` — The eventpoint number used to identify this particular eventpoint in other CXdb commands. In this case, the tracepoint number is 0.
- `trace routine` — The type of tracepoint.
- `on [#0/*], Enabled, ignore 0/0` — The tracepoint is set on process object 0, for all threads (`*`). It is enabled, and does not have an ignore count.
- `[0x80005696]` — The hexadecimal address for the location of the tracepoint. In this case the address is `80005696`.
- `CLEAR_ARRAY in example.f line 53` — The symbolic location of the tracepoint. In this case the tracepoint is in the routine `CLEAR_ARRAY` at line 53 of the source file example.f.

You can also display the above information by using the `info event` command.

When the tracepoint is triggered, execution stops before the first source unit in the routine is executed. A message is displayed, telling you that this tracepoint has been reached, then process execution resumes.

The following two examples set a tracepoint at the start of a routine by specifying an absolute address inside of that routine. CXdb finds the routine containing the absolute address and places the tracepoint at the first source unit. The syntax for specifying an absolute address is different between Fortran and C.

Using Fortran syntax:

```
(CXdb) trace routine '80005700'x

#1: trace routine, on [#0/*], Enabled, ignore 0/0
        [0x80005696] CLEAR_ARRAY in example.f line 53
```

The above command sets a tracepoint at the starting address of the routine that contains the absolute address `80005700`. The tracepoint number is `1`, located at address `80005696` in routine `CLEAR_ARRAY` at line `53` of the file example.f. The notation `'80005700'x` is Fortran-specific and indicates that the address is in hexadecimal notation.

Using C syntax:

```
(CXdb) trace routine 0x80004800

#2: trace routine, on [#0/*], Enabled, ignore 0/0
      [0x8000472c] chapter7C`bld_matrix in chapter7C.c line 33
```

The above command sets a tracepoint at the starting address of the routine that contains the absolute address `80004800`. The `0x` is the C notation for a hexadecimal number. The symbolic location uses the scope path of `chapter7C`bld_matrix` to indicate the program (chapter7C) and routine (`bld_matrix`) in which the tracepoint is located.

```
(CXdb) trace routine CLEAR_ARRAY {echo 'routine CLEAR_ARRAY reached';}

#3: trace routine, on [#0/*], Enabled, ignore 0/0
        [0x80005696] CLEAR_ARRAY in example.f line 53
    {
        echo 'routine CLEAR_ARRAY reached';
    }
```

The above command sets a tracepoint at the address of the first executable source unit of the routine `CLEAR_ARRAY`. An eventpoint handler is defined for the tracepoint. When the tracepoint is triggered, execution stops and the `echo` command is executed. Even though the eventpoint is a tracepoint, execution does not resume because this eventpoint's handler overrides the default handler for tracepoints.

```
(CXdb) trace routine '80005696'x \; $Trace4

#4: trace routine, on [#0/*], Enabled, ignore 0/0
        [0x80005696] CLEAR_ARRAY in example.f line 53
```

The above command creates a new tracepoint at the first executable source unit of the routine containing the absolute address `80005696`. The language expression terminator `\;` is needed to separate the language expression from the debugger variable. The debugger variable `$Trace4` is created and set equal to the number of this eventpoint. In subsequent commands you can use `$Trace4` to refer to this tracepoint. Debugger variables allow you to refer to eventpoints without having to remember their eventpoint numbers.

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`
`resume`	`set default handler`
`set handler`	`set typehandler`
`trace instruction`	`trace line`
`trace source`	`watch`

Related Concepts

breakpoints	debugger variables
eventpoints	eventpoint handlers
tracepoints	watchpoints

Related Parameters

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

trace source

t s
ts

Set a tracepoint at a source unit.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.
<source-unit>	The source unit number where the tracepoint is set. The source unit number must be an integer, and may be preceded by a source file name.
<event-handler>	A sequence of CXdb commands enclosed within curly braces ({ }). Each command in the handler must be terminated by a semicolon (;).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

The `trace source` command sets a tracepoint at the specified source unit number.

A source unit is a syntactical element of your source code. Source units are generated when you compile your program with the `-cxdb` option, and they are numbered sequentially. To determine the number and type of a particular source unit, use the `info line` command. If you already know the source unit number, you can gather more information about the source unit by using the `info sourceunit` command.

When the tracepoint is triggered, process execution stops and the commands of the tracepoint's handler are executed. If the tracepoint does not have its own handler, the default handler for tracepoints, which displays a message and resumes process execution, is executed.

Examples

The following examples set tracepoints at specific source units.

```
(CXdb) trace source 25

#0: trace source, on [#0/*], Enabled, ignore 0/0
        [0x80005452] EXAMPLE in example.f line 17
```

The above command sets a tracepoint at the starting address of source unit 25 of the current source file.

When you create a tracepoint, CXdb responds by displaying the following information:

- `#0:` — The eventpoint number used to identify this particular eventpoint in other CXdb commands. In this case, the tracepoint number is 0.
- `trace source` — The type of tracepoint.
- `on [#0/*], Enabled, ignore 0/0` — The tracepoint is set on process object 0, for all threads (*). It is enabled, and does not have an ignore count.
- `[0x80005452]` — The hexadecimal address location of the tracepoint. In this case the address is `80005452`.
- `EXAMPLE in example.f line 17` — The symbolic location of the tracepoint. In this case the tracepoint is in the routine `EXAMPLE` at line 17 of the source file example.f.

You can also display the above information by using the `info event` command.

When the tracepoint is triggered, execution stops before the first instruction of the source unit is executed. A message is displayed, telling you that this tracepoint has been reached, then process execution resumes.

```
(CXdb) trace source example.f:100

#1: trace source, on [#0/*], Enabled, ignore 0/0
        [0x800056b0] CLEAR_ARRAY in example.f line 55
```

The above command sets a tracepoint at the starting address of source unit 100 of the source file example.f. This source file must be part of the compilation of the current executable file and be included in the search path of the process object.

```
(CXdb) trace source 25 {echo 'Source unit 25 reached';}

#2: trace source, on [#0/*], Enabled, ignore 0/0
        [0x80005452] EXAMPLE in example.f line 17
    {
        echo 'Source unit 25 reached';
    }
```

The above command sets a tracepoint at the starting address of source unit 25 of the current source file. An eventpoint handler is defined for the tracepoint. When the tracepoint is triggered, process execution stops and the commands of the event handler are executed. The eventpoint handler displays a message. Even though the eventpoint is a tracepoint, execution is not resumed because the specified handler overrides the default handler for tracepoints.

```
(CXdb) trace source 25 $Trace3

#3: trace source, on [#0/*], Enabled, ignore 0/0
        [0x80005452] EXAMPLE in example.f line 17
```

The above command sets a tracepoint at the starting address of source unit 25 in the current source file. The debugger variable `$Trace3` has been assigned to this tracepoint. In subsequent commands you could use the debugger variable `$Trace3` to refer to this eventpoint. Debugger variables allow you to refer to eventpoints without having to remember their eventpoint numbers.

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`
`info line`	`info sourceunit`
`resume`	`set default handler`
`set handler`	`set typehandler`
`trace instruction`	`trace line`
`trace routine`	`watch`

Related Concepts

breakpoints
eventpoints
tracepoints
debugger variables
eventpoint handlers
watchpoints

Related Parameters

debugger-variable
process-list
thread-list
event-handler
source-unit

watch

W

Set a watchpoint to monitor an address range.

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process object.
<thread-list>	A list of threads affected by this process. The default is all threads of the specified process object.
<starting-address>	Any valid language expression whose evaluation is used as the starting address of the address range.
<ending-address>	Any valid language expression whose evaluation is used as the ending address of the address range.
<byte-count>	The total number of bytes to watch, including the start of the address range. The language expression use for this count must evaluate to a positive integer.
<event-handler>	A sequence of CXdb commands enclosed in curly-braces ({ }). Each command in the handler must be terminated by a semicolon (;).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

The `watch` command sets a watchpoint to watch for a change to occur at the specified address range. A process image must exist for a watchpoint to be created.

After the execution of each instruction, CXdb tests to see if the value stored at the watched address has changed. If the value has changed, the watchpoint is triggered.

When the watchpoint is triggered, process execution stops, and then the commands of the watchpoint's handler are executed. If the watchpoint does not have its own handler, then the default handler for watchpoints, which displays a message, is executed. Execution of the process does not resume unless you include the `resume` command in the watchpoint handler.

Watchpoints are associated with memory addresses rather than with particular lines of executing code. Eventpoints of this type are known as asynchronous eventpoints. Multiple asynchronous eventpoints can be triggered at the same time. In such cases, only the eventpoint handler of the lowest-numbered asynchronous eventpoint is executed.

Specifying an address range

The address range can be specified using one of the following three methods:

- Specify a starting address and ending address. Both addresses are language expressions whose evaluations are used as the address.
- Specify a starting address and a number of bytes to watch. The bytes begin at the starting address. The number of bytes to watch is a language expression that must evaluate to a positive integer.
- Specify only a starting address. The starting address is a language expression. If the address of a variable is specified, the entire region of the variable is watched. If an absolute address is specified, only one byte at that address is watched.

Examples

The following examples set watchpoints. The syntax for retrieving a variable's address is different between Fortran and C. The next two examples demonstrate this difference.

```
(CXdb) watch loc(TABLE)
#1: watch 0x80077058..0x80077097, on [#0/0], Enabled, ignore 0/0
```

The above command sets a watchpoint to watch the address of the Fortran array `TABLE`. The Fortran function `loc()` provides the address of the variable.

When you create a watchpoint, CXdb responds by displaying the following information:

- `#1:` — The eventpoint number used to identify this particular eventpoint in other CXdb commands. In this case the eventpoint number is 1.
- `watch` — The type of eventpoint.
- `0x80077058..0x80077097` — The address range that the watchpoint monitors.
- `on [#0/0], Enabled, ignore 0/0` — The eventpoint is set on process object 0, for all threads (*). It is enabled and does not have an ignore count.

You can also display the above information by using the `info event` command.

When any value stored in the array `TABLE` changes, the watchpoint is triggered.

```
(CXdb) watch &slice

#3: watch 0xffffc968..0xffffc96b, on [#0/0], Enabled, ignore 0/0

INFO 175: Data region lies on stack. Eventpoint will be disabled when frame is popped.
```

The above command watches the address of the C variable `slice`. The C operator `&` is used to provide the address of the variable. CXdb responds with the same information as with the Fortran example above. The informative message explains that because the address region is part of the current frame on the stack, the watchpoint will be disabled when this frame is popped from the stack.

When the value stored in `slice` changes, the watchpoint is triggered.

The syntax for specifying an absolute address is different between Fortran and C. The next two examples demonstrate this difference.

```
(CXdb) watch '80001234'x:4

#4: watch 0x80001234..0x80001237, on [#0/0], Enabled, ignore 0/0
```

The above command uses the : notation to specify a byte count. The watchpoint monitors four bytes, starting with the address `80001234` and ending with the address `80001237`. The notation `'80001234'x` is Fortran-specific and indicates the address is in hexadecimal notation. When the value stored in this address range changes, the watchpoint is triggered.

```
(CXdb) watch 0x80001234:4

#5: watch 0x80001234..0x80001237, on [#0/0], Enabled, ignore 0/0
```

The above command watches four bytes starting with address `80001234`. The notation `0x80001234` is C-specific and indicates the address is in hexadecimal notation. When the value stored in this range changes, the watchpoint is triggered.

```
(CXdb) watch '80001234'x..'80001237'x

#6: watch 0x80001234..0x80001237, on [#0/0], Enabled, ignore 0/0
```

The above command uses the .. notation to specify an address range, starting with the address `80001234` and ending with `80001237`. When the value stored in this address range changes, the watchpoint is triggered.

```
(CXdb) watch '80002345'x:8 {echo "region B modified"; resume;}

#7: watch 0x80002345..0x8000234c, on [#0/0], Enabled, ignore 0/0
    {
        echo "region B modified";
        resume;
    }
```

The above command sets a watchpoint to watch the eight bytes starting from the specified address. A handler is defined for the watchpoint. When the watchpoint is triggered, the `echo` command is executed, and process execution resumes.

```
(CXdb) watch loc(TABLE) \; $w1

#8: watch 0x80077058..0x80077097, on [#0/0], Enabled, ignore 0/0
```

The above command watches the array `TABLE`. The language expression terminator `\;` is needed to separate the language expression from the debugger variable. A debugger variable has been assigned to the watchpoint. In subsequent CXdb commands, you can use the debugger variable `$w1` to refer to this watchpoint. Debugger variables allow you to refer to eventpoints without having to remember their eventpoint numbers.

Related Commands

`event modify`
`event relation`
`set default handler`
`set handler`
`set typehandler`

Related Concepts

breakpoints
debugger variables
eventpoints
eventpoint handlers
tracepoints
watchpoints

Related Parameters

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

Parameters

2

This chapter contains reference pages that explain the parameters used with CXdb commands. There is a separate reference page for each parameter. The reference pages can contain the following sections:

- **Description** — Text explaining the purpose and functionality of the parameter.
- **Syntax** — Format rules for the parameter.
- **Examples** — One or more examples illustrating the use of the parameter.
- **Related Commands** — A list of CXdb commands that use the parameter. The commands are described in more detail in Chapter 1.
- **Related Concepts** — A list of major concepts related to the parameter. Related concepts are described in Chapter 3.
- **Related Parameters** — A list of other parameters related to the parameter being described. The related parameters are also described in this chapter.
- **Related Windows** — A list of CXdb windows that relate to the parameter being described. You can use these windows only if you are running CXdb in X Windows mode. The windows are described in Chapter 4.

<array-slice>

A subset of an array.

Syntax

<starting-subscript> [. . *<ending-subscript>*]

Parameter	Meaning
<starting-subscript>	The subscript of the first element in the array slice. The subscript can be a *<language-expression>*.
<ending-subscript>	The subscript of the last element in the array slice. The subscript can be a *<language-expression>*. If you do not specify an ending subscript, the default array slice is the single element specified by the starting subscript.

Description

An *<array-slice>* is a subset of an array. Because arrays are often too large to work with all of their elements at once, it often is convenient to divide the array into segments, or slices.

You can use array slices in language expressions that work with arrays. For example, one use of array slices in CXdb is with the `print` command.

Examples

The following examples illustrate how to specify array slices. All of the examples use a C array called `matrix`. The array is three-dimensional (5x5x5) and contains four-byte floating point values. For all examples, assume that `printopts maxarray` is set to 150.

```
(CXdb) print matrix
float[5][5][5]
[0][0][0..4] :     -7.5000     -4.6250     -2.7400     -1.3500     -0.2429
[0][1][0..4] :     -1.0250      0.8600      2.2500      3.3571      4.2875
[0][2][0..4] :      6.4600      7.8500      8.9571      9.8875     10.7000
[0][3][0..4] :     15.4500     16.5571     17.4875     18.3000     19.0300
[0][4][0..4] :  0.0000E+000 0.0000E+000 0.0000E+000 0.0000E+000 0.0000E+000
             ***
[1][0][0..4] :      4.9750      6.8600      8.2500      9.3571     10.2875
[1][1][0..4] :     21.4600     22.8500     23.9571     24.8875     25.7000
[1][2][0..4] :     43.4500     44.5571     45.4875     46.3000     47.0300
[1][3][0..4] :      0.1571      1.0875      1.9000      2.6300      3.3000
[1][4][0..4] :  0.0000E+000 0.0000E+000 0.0000E+000 0.0000E+000 0.0000E+000
             ***
[2][0][0..4] :     48.4600     49.8500     50.9571     51.8875     52.7000
[2][1][0..4] :    107.4500    108.5571    109.4875    110.3000    111.0300
[2][2][0..4] :      1.1571      2.0875      2.9000      3.6300      4.3000
[2][3][0..4] :     28.6875     29.5000     30.2300     30.9000     31.5250
[2][4][0..4] :  0.0000E+000 0.0000E+000 0.0000E+000 0.0000E+000 0.0000E+000
             ***
[3][0][0..4] :    315.4500    316.5571    317.4875    318.3000    319.0300
[3][1][0..4] :     -3.8429     -2.9125     -2.1000     -1.3700     -0.7000
[3][2][0..4] :     -3.3125     -2.5000     -1.7700     -1.1000     -0.4750
[3][3][0..4] :     -2.9000     -2.1700     -1.5000     -0.8750     -0.2846
[3][4][0..4] :  0.0000E+000 0.0000E+000 0.0000E+000 0.0000E+000 0.0000E+000
             ***
[4][0][0..4] :  0.0000E+000 0.0000E+000 0.0000E+000 0.0000E+000 0.0000E+000
[4][1][0..4] :  0.0000E+000 0.0000E+000 0.0000E+000 0.0000E+000 0.0000E+000
[4][2][0..4] :  0.0000E+000 0.0000E+000 0.0000E+000 0.0000E+000 0.0000E+000
[4][3][0..4] :  0.0000E+000 0.0000E+000 0.0000E+000 0.0000E+000 0.0000E+000
[4][4][0..4] :  0.0000E+000 0.0000E+000 0.0000E+000 0.0000E+000 0.0000E+000
             ***
```

The above command prints the entire array because an array slice was not specified.

```
(CXdb) print matrix[0][0][0..4]
float[1][1][5]
[0][0][0..4] :      -7.5000     -4.6250     -2.7400     -1.3500     -0.2429
```

The above command prints the first row of the array.

```
(CXdb) print matrix[0][0..4][0]
float[1][5][1]
[0][0..4][0] :      -7.5000     -1.0250      6.4600     15.4500 0.0000E+000
```

The above command prints the first column of the array.

```
(CXdb) print matrix[0..4][0][0]
float[5][1][1]
[0..4][0][0] :      -7.5000      4.9750     48.4600    315.4500 0.0000E+000
```

The above command prints the first element from each level (third dimension) of the array.

```
(CXdb) print matrix[0][0..4][0..4]
float[1][5][5]
[0][0][0..4] :      -7.5000     -4.6250     -2.7400     -1.3500     -0.2429
[0][1][0..4] :      -1.0250      0.8600      2.2500      3.3571      4.2875
[0][2][0..4] :       6.4600      7.8500      8.9571      9.8875     10.7000
[0][3][0..4] :      15.4500     16.5571     17.4875     18.3000     19.0300
[0][4][0..4] :  0.0000E+000 0.0000E+000 0.0000E+000 0.0000E+000 0.0000E+000
```

The above command prints all rows and columns in the first level of the array.

```
(CXdb) print matrix[i-1][j-1][k-4]
(float)     29.5000
```

The above command prints the single element referenced by the subscripts. In this example, the subscripts are expressions that evaluate to positive integers.

<array-slice>

Related Commands

`evaluate`	`examine`
`print`	`set printopts maxarray`

Related Concepts

C language expressions	displaying data
Fortran language expressions	language expressions

Related Parameters

language-expression

<debugger-variable>

A variable used in CXdb commands.

Syntax

[cxdb$ | $]<*variable-name*>

Parameter	Meaning
cxdb$	One of the symbols to delimit the debugger variable name and distinguish it from the name of a process variable.
$	One of the symbols to delimit the debugger variable name and distinguish it from the name of a process variable.
<*variable-name*>	An alphanumeric string that is the name of the debugger variable.

Description

A <*debugger-variable*> is a variable that you can define in CXdb. A debugger variable can store the following types of data:

- A CXdb object number
- The result of a language expression
- A signal number
- The contents of a register

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

For more information, read the "debugger variables" concepts reference page.

Examples

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

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

#2: 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 2, and this object number is stored in the debugger variable `$D`. The language expression terminator `\;` is needed to separate the language expression from the debugger variable.

Once the debugger variable has been created, it can be referenced in a subsequent command, as follows:

```
(CXdb) set ignore 3 $D
Event 2 will be ignored 3 times
```

The above command sets an ignore count for eventpoint 2, which is represented in this command by the debugger variable `$D`.

Related Commands

`evaluate` `print`

Related Concepts

command files
debugger variables
eventpoint handlers
initialization files

Related Parameters

event-handler

<directory-specifier>

A directory path name.

Syntax

<directory-specifier>

Description

A *<directory-specifier>* is a relative or absolute directory path name.

In addition to the names of directories and the slash (/), a path name can begin with the following special items:

- *$variable* — An environment variable defined in your shell environment before you invoked CXdb. It is expanded to its value before the CXdb command is executed.
- tilde (*~<user>*) — A user's home directory. If *<user>* is omitted, it is your home directory. It is expanded to its value.
- dot (.) — The current directory. It is not expanded.
- dot-dot (..) — The parent directory. It is not expanded.

After the expansion is done, if the path name begins with a slash (/), it is an absolute path name. If not, it is a relative path name.

Examples

The following examples show the use of a directory specifier with the `add path` command.

```
(CXdb) add path /mnt/jones/projects
```

The above command adds the /mnt/jones/projects directory to the search path. Because the directory path name begins with the slash character (/), it is an absolute path name.

```
(CXdb) add path libraries
```

The above command adds the /mnt/jones/libraries directory to the search path. Because it is a relative path name, the console working directory is used as the base path name. (The console working directory is the base directory for all relative path names used in commands that affect CXdb.)

```
(CXdb) add path .. , .
```

The above command adds the parent directory and the current directory to the search path. Because neither of these path names are expanded, the search path now consists of directories that are always relative to the current console working directory.

```
(CXdb) add path ~smith/projects, $MYDIR/libraries
```

The above command adds two more directories to the search path. The first directory added is the projects directory found under the home directory of the user `smith`. The second directory added is the libraries directory, found under the path represented by the environment variable `MYDIR`. The variable `MYDIR` must be defined in the shell environment before CXdb is invoked.

Related Commands

`add default path`	`add path`
`cd`	`remove default path`
`remove path`	`set default path`
`set directory`	`set path`

Related Concepts

default search path	process object
search path	

Related Parameters

environment-variable	file-name

<environment-variable>

An environment variable.

Syntax

<environment-variable>

Description

An *<environment-variable>* is a variable that holds a string value and is passed as part of the environment to each new process.

Environment variables can be referenced in CXdb commands or referenced by the process to which they were passed.

When an environment variable is being created or changed, the variable name is used by itself. When the value of an environment variable is needed, the variable name is preceded by a dollar sign ($).

Examples

The following examples use environment variables.

```
(CXdb) add environment MYDIR = project/program1
```

The above command adds the environment variable `MYDIR` to the environment of the current process object, if the variable does not already exist. If it does exist, the value of the variable is changed.

```
(CXdb) add default environment LIBS = "/mnt/jones/lib /mnt/jones/math"
```

The above command adds the environment variable `LIBS` to the default environment. Because the string contains a white space character (a blank), the string is delimited by double quotes.

```
(CXdb) cd $PROJ2
```

The above command changes the console working directory to the value stored in the default environment variable named `PROJ2`. (The console working directory is the base directory for all relative path names used in commands that affect CXdb.)

<environment-variable>

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 — environment

Related Parameters

string

<event-handler>

A handler for an eventpoint.

Syntax

{ *<command>* **;** [...] **}**

Parameter	Meaning
<command>	A CXdb command. The `resume` command must be used as the last command in a handler when process execution is to be continued.
[...]	An optional list of additional CXdb commands. Each command must be terminated with a semicolon (`;`).

Description

An *<event-handler>* is a collection of CXdb commands that are executed when the corresponding event is triggered.

The commands are enclosed in curly braces (`{}`). Each statement inside of an event-handler must end with a semicolon (`;`).

The one exception to the above rule is in the use of the `if` command. The `if` command itself does not need to be terminated with a semicolon. Each command in the body of the `if` command must terminate with a semicolon. Multiple commands in the body can be enclosed in curly braces.

Examples

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

```
(CXdb) break routine BLD_MATRIX {print M;}
```

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

```
(CXdb) break routine BLD_MATRIX {print M; resume;}
```

When the breakpoint in the above command is triggered, execution stops, and the value of the variable `M` 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 routine BLD_MATRIX {if (M .LT. 5) echo "M is less than 5";
       else {print M; resume;}}
```

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

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`
`resume`
`set default handler`
`set handler`
`trace instruction`
`trace line`
`trace routine`
`trace source`
`watch`

Related Concepts

breakpoints
C language expressions
eventpoints
eventpoint handlers
Fortran language expressions
language expressions
tracepoints
watchpoints

<event-specifier>

An eventpoint identifier.

Syntax

{ *<eventpoint-number>* | *<debugger-variable>* | * }

Parameter	Meaning
<eventpoint-number>	The eventpoint number assigned by CXdb to the eventpoint when it is created.
<debugger-variable>	A debugger variable that you assigned to the eventpoint.
*	All eventpoints in the current process.

Description

An *<event-specifier>* identifies an eventpoint to be affected by a CXdb command. The eventpoint may be specified by its eventpoint number or by a debugger variable if a debugger variable has been assigned to the eventpoint. To specify all eventpoints, use the asterisk (*).

In a list, multiple event specifiers are separated by commas.

Examples

The following examples use an event specifier with the `info event` command.

```
(CXdb) info event 0,2

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

#2: break line, on [#0/*], Enabled, ignore 0/0
        [0x8000568a] PRINT_ARRAY in example.f line 47
```

The above command displays information about eventpoints 0 and 2. The event specifiers are separated on the command line by a comma.

```
(CXdb) info event $TRACE_1

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

The above command displays information about the eventpoint that has been assigned to the debugger variable `$TRACE_1`. The debugger variable can be assigned when the eventpoint is created.

```
(CXdb) info event *

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

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

#2: break line, on [#0/*], Enabled, ignore 0/0
        [0x8000568a] PRINT_ARRAY in example.f line 47
```

The above command displays information about all eventpoints in the current process.

Related Commands

`disable event`	`disable eventtype`
`info event`	`info eventtype`
`remove event`	`remove eventtype`
`set handler`	`set typehandler`
`set ignore`	

Related Concepts

breakpoints	eventpoints
eventpoint handlers	tracepoints
watchpoints	

Related Parameters

debugger-variable	eventtype-specifier

<eventtype-specifier>

An eventpoint type.

Syntax

```
{ break | trace | watch | exec | join | modify |
   reached | relation | signal | spawn | * }
```

Parameter	Meaning
break	All breakpoints (includes instruction, line, routine, and source).
trace	All tracepoints (includes instruction, line, routine, and source).
watch	All watchpoints.
exec (C Series only)	All event exec eventpoints.
join (C Series only)	All event join eventpoints.
modify	All event modify eventpoints.
reached (C Series only)	All event reached eventpoints (includes instruction, line, routine, and source).
relation	All event relation eventpoints.
signal	All event signal eventpoints.
spawn	All event spawn eventpoints.
*	All eventpoints in the current process.

<eventtype-specifier>

Description

An *<eventtype-specifier>* identifies the eventtype to be affected by a CXdb command.

The available eventtypes are listed below:

```
break
trace
watch
exec (C Series only)
join
modify (C Series only)
reached (C Series only)
relation
signal
spawn
```

To specify all eventtypes, the asterisk (*) is used.

In a list, multiple eventtype specifiers are separated by commas.

Examples

The following examples use an eventtype specifier with the `info eventtype` command.

```
(CXdb) info eventtype trace, break
Status of eventpoints of type Breakpoint:

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

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

Status of eventpoints of type Tracepoint:

#2: trace line, on [#0/*], Enabled, ignore 0/0
        [0x800054b2] EXAMPLE in example.f line 25
```

The above command displays information about all tracepoints and breakpoints.

```
(CXdb) info eventtype *
Status of eventpoints of type Signal:

Status of eventpoints of type Relation:

Status of eventpoints of type Modify:

Status of eventpoints of type Reached:

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

Status of eventpoints of type Join:

Status of eventpoints of type Spawn:

Status of eventpoints of type Exec:

Status of eventpoints of type Watchpoint:

Status of eventpoints of type Tracepoint:

#2: trace line, on [#0/*], Enabled, ignore 0/0
        [0x800054b2] EXAMPLE in example.f line 25

Status of eventpoints of type Breakpoint:

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

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

The above command displays information about all the eventpoints of all eventtypes.

Related Commands

`disable event`	`disable eventtype`
`info event`	`info eventtype`
`remove event`	`remove eventtype`
`set handler`	`set typehandler`
`set ignore`	

<eventtype-specifier>

Related Concepts	breakpoints eventpoint handlers watchpoints	eventpoints tracepoints
Related Parameters	debugger-variable	event-specifier

<file-name>

The name of a file.

Syntax

[*<directory-specifier>* /] *<file-name>*

Parameter	Meaning
<directory-specifier>	The path name to the specified file.

Description

A *<file-name>* is the name of a file.

The file name can be preceded by a directory path name.

Examples

The following examples show the use of a file name with the `executable` command.

```
(CXdb) executable para
```

The above command specifies `para` as the new executable file. The file is in the console working directory in this case. (The console working directory is the base directory for all relative path names used in commands that affect CXdb.)

```
(CXdb) executable $MYDIR/para
```

The above command specifies `para` as the new executable file. This file is found under the directory represented by the environment variable `MYDIR`.

Related Commands

`add cmderr`	`add cmdlog`
`add cmdout`	`core`
`debug core`	`debug exec`
`edit`	`remove cmderr`
`remove cmdlog`	`remove cmdout`
`set cmderr`	`set cmdlog`
`set cmdout`	`source`

<file-name>

Related Concepts

cmderr	cmdlog
cmdout	command files
logging	initialization files

Related Parameters

directory-specifier	viewport

<frame-specifier>

A stack frame identifier.

Syntax

[{+ | -}] *<integer>*

Parameter	Meaning
+	An operator indicating that the specified integer is to be added to the current frame number.
-	An operator indicating that the specified integer is to be subtracted from the current frame number.
<integer>	A positive integer. If no operator (+ or –) is specified, then the integer represents an absolute frame number.

Description

The *<frame-specifier>* identifies a particular frame on the process stack. It can be expressed as an absolute frame number or as a displacement relative to the current frame number. The topmost frame is frame 0.

The current frame can be selected with the `frame` command.

Examples

The following examples illustrate the use of frame specifiers in the `info frame` command. For these examples, assume that the process stack contains five frames, and that the current frame is frame 4.

```
(CXdb) info frame 1
Process [#0/0]
Frame : 1; [0x80002948] CHAPTER7 in chapter7F.f line 10
Frame address : 0xffffca38
Saved Registers : pc=0x80002948   psw=0x7909400   fp=0xffffca48   ap=0x80002d70
Floating point mode : NATIVE;  Language  : FORTRAN
Number of arguments : 1
```

The above command displays frame number 1 of the stack. The integer 1 is used here as an absolute frame number.

```
(CXdb) info frame -2
Process [#0/0]
Frame : 0; [0x800029bc] BLD_MATRIX in chapter7F.f line 26
Floating point mode : NATIVE;  Language  : FORTRAN
Number of arguments : 4
```

In the above command, the frame specifier is `-2`. The current frame is 2, so the command displays frame 0 (or `2 + -2`).

```
(CXdb) info frame +1
Process [#0/0]
Frame : 3; [0x8000e928] _main+0x1cc
Frame address : 0xffffca5c
Saved Registers : pc=0x8000e928 psw=0x87109400 fp=0xffffcab0
      ap=0xffffca70
Floating point mode : NATIVE;  Language  : FORTRAN
Number of arguments : 3
```

In the above command, the frame specifier is `+1`. The current frame is 2, so the command displays frame 3 (or `2 + 1`).

Related Commands

`backtrace`
`display stack`
`frame`
`info frame`
`info locals`
`info process`
`info stack`

Related Concepts

scope

<granularity>

The source unit granularity.

Syntax

{expression | statement | block | loop | routine}

Parameter	Meaning
expression	Any combination of constants, variables, and operators that is valid in the current source language.
statement	A combination of expressions that constitutes a complete instruction in the current source language.
block	The statements that make up the body of a loop or conditional construct. In Fortran, each block is contained within one of the following constructs: • DO — ENDDO • DO WHILE — ENDDO • IF ( ) THEN — ENDIF • IF ( ) THEN — ELSE • IF ( ) THEN — ELSE IF • ELSE — ENDIF • ELSE IF ( ) THEN — ELSE • ELSE IF ( ) THEN — ELSE IF • ELSE IF ( ) THEN — ENDIF In C, a block is any group of statements enclosed in curly braces ({}).

loop

A special type of statement that delimits a block of repeating code.

In Fortran, the looping structures are:

- DO — ENDDO
- DO WHILE — ENDDO

In C, the looping structures are:

- do-while
- for
- while

routine

A main routine, subroutine, or function.

In Fortran, the main routine may begin with the `PROGRAM` statement, and it terminates with an `END` statement. Subroutines start with the `SUBROUTINE` statement and terminate with an `END` statement. Functions start with the `FUNCTION` statement and terminate with an `END` statement.

In C, the main routine begins with the symbol `main()`. All other routines in a C program are called functions. Each function starts with a name, and the body of the function is enclosed in curly braces (`{}`). The function may or may not include an argument list.

Description

The *<granularity>* is a particular size or type of source unit. Granularity is used with stepping commands to specify how big each step should be.

When you invoke CXdb, the default granularity is `statement`. The commands `set default step` and `set step` let you change this default.

Examples

The following examples illustrate how to specify granularity with some of the stepping commands.

```
(CXdb) step routine
```

The above command steps the process to the beginning of the next subroutine or function.

```
(CXdb) set step block
```

The above command changes the default granularity to `block`.

```
(CXdb) next loop
```

The above command steps to the beginning of the next loop in the current routine or function.

```
(CXdb) step statement 3
```

The above command steps through the next three statements and halts execution at the beginning of the fourth statement in sequence.

```
(CXdb) next expression
```

The above command steps to the beginning of the next expression in the current routine or function.

<granularity>

Related Commands

`info cxdb`	`info line`
`info sourceunit`	`next`
`next over`	`set step`
`step`	`step over`

Related Concepts

source units

stepping

Related Parameters

source-unit

<language-expression>

An expression in the source language.

Syntax

<language-expression> [\;]

Parameter	Meaning
<language-expression>	A valid expression in the current source language. The exact syntax of an expression depends on the language being used to evaluate the expression. (Refer to a Fortran or C reference manual for more details.)
`\;`	A delimiter that indicates the end of the language expression. This delimiter is necessary only when an additional part of a CXdb command follows the expression.

Description

A *<language-expression>* is any expression that is valid in the current source language. The expression may contain a combination of the following:

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

CXdb evaluates the language expression according to the rules of the current source language. The current source language is the language of the source file associated with the currently selected stack frame.

Evaluation of a language expression depends on the particular CXdb command in which the expression appears. Some CXdb commands evaluate the expression to an address, while others evaluate it to a numerical value.

Examples

The following examples illustrate the use of language expressions as addresses and numerical values.

```
(CXdb) break routine PRINT_ARRAY \; $Break1

#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 routine called `PRINT_ARRAY`, creates the debugger variable `$Break1`, and sets it equal to the number of that eventpoint. In this case, CXdb evaluates the expression `PRINT_ARRAY` to determine the address of the routine. The `\;` is needed to separate the language expression from the debugger variable.

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

The above command evaluates the expression `I+J` and prints the resulting value, which is an integer.

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
language expressions
Fortran language expressions
scope
source units

Related Parameters

array-slice
debugger-variable
string

<line-specifier>

A source line identifier.

Syntax

[*<file-name>*:] *<integer>*

Parameter	Meaning
<file-name>	The absolute or relative path name of a source file. The default is the source file of the current process object.
<integer>	A positive integer.

Description

A *<line-specifier>* identifies a particular line in the specified source file.

The line specified must contain an executable source unit. Blank lines, comment lines, and lines that have been eliminated by optimization do *not* contain executable source units. Therefore, specifying such a source line results in an error.

Examples

The following examples illustrate the use of line specifiers with the `break line` command.

```
(CXdb) break line 7

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

The above command sets a breakpoint at the machine instruction corresponding to line 7 of the current source file.

```
(CXdb) break line chapter7F.f:26

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

The above command sets a breakpoint at the machine instruction corresponding to line 26 of the file chapter7F.f.

<line-specifier>

Related Commands

`break line`
`event reached line`
`goto line`
`info line`
`trace line`

Related Concepts

source units

Related Parameters

file-name

<process-list>

A list of processes.

Syntax

:p {*<process-number>* [**,** ...] | *****}

Parameter	Meaning
<process-number>	The number of a CXdb process object. The number can be expressed as an integer or as a debugger variable that contains the value of the process object number.
[**,** ...]	Additional process numbers in the list. Commas must separate the entries in the list. Spaces between the entries are optional.
*	The wildcard symbol denoting all processes.

Description

A *<process-list>* is a list of processes that are affected by a command.

NOTE: The current version of CXdb maintains only one process object (process object 0) at any given time. Because of this, you may omit the process list from commands in this release of CXdb.

If you are debugging only one process at a time, you can omit the process list from the command. If you are debugging multiple processes, the process list is required to specify which processes you want to affect. If you have multiple processes but do not specify a process list, CXdb assumes that you want to affect all processes.

Examples

The following examples show the use of process lists with the `continue` command.

```
(CXdb) :p0 continue
```

The above command continues execution of process 0.

```
(CXdb) :p1,2 continue
```

The above command continues execution of both processes 1 and 2 simultaneously.

```
(CXdb) :p$X continue
```

The above command continues execution of the process whose number is stored in a debugger variable called `X`. Prior to its use here, the variable `X` must be set equal to a valid process number.

```
(CXdb) continue
```

The above command continues execution of all active processes.

```
(CXdb) :p* continue
```

The above command continues execution of all active processes. It uses the wildcard symbol (*) to specify all processes.

Related Commands

add environment	add path
attach	backtrace
break instruction	break line
break routine	break source
clear environment	clear fixed sched
clear seq	clear sqs
clear step	continue
copy	core
detach	disable eventtype
disassemble	display disassembly
display examine	display routine
display source	display stack
enable eventtype	evaluate
event exec	event join
event modify	event reached instruction
event reached line	event reached routine
event reached source	event relation
event signal	event spawn
examine	executable
fill	find memory backward

```
find memory forward          finish
frame                        get
goto address                 goto line
goto source                  info args
info break                   info cregisters
info dynamicobject           info environment
info errno                   info eventtype
info expression              info formatting
info frame                   info frame at
info line                    info locals
info objectmap               info process
info psw                     info registers
info scope                   info signal
info sourceunit              info stack
info symbols                 info threads
info trace                   info type
info vregisters              info watch
kill process                 load object
next                         next instruction
next over                    print
put                          remove environment
remove eventtype             remove path
rerun                        return
run                          set directory
set environment              set fixed sched
set format                   set fpmode
set memory                   set path
set pshell                   set remotewd
set seq                      set signal
set sqs                      set step
signal process               signal thread
step                         step instruction
step over                    stop
trace instruction            trace line
trace routine                trace source
watch
```

Related Concepts

process object

Related Parameters

thread-list

<process-list>

<redirection-operator>

An operator that redirects cmdout or cmderr.

Syntax

{> | >! | >> | >>! | >& | >&! | >>& | >>&!}
<viewport> [, ...]

Parameter	Meaning
>	Redirect cmdout to the specified viewports. Create specified files if they do not already exist. Overwrite existing files if noclobber is off. Generate an error message if noclobber is on and a specified file already exists.
>!	Redirect cmdout to the specified viewports. Create specified files if they do not already exist. Overwrite existing files regardless of the noclobber setting.
>>	Redirect cmdout to the specified viewports. Create specified files if they do not already exist. Append new data to the end of existing files if noclobber is off. Generate an error message if noclobber is on and a specified file does not exist.
>>!	Redirect cmdout to the specified viewports. Create specified files if they do not already exist. Append new data to the end of existing files regardless of the noclobber setting.
>&	Redirect cmderr to the specified viewports. Create specified files if they do not already exist. Overwrite existing files if noclobber is off. Generate an error message if noclobber is on and a specified file already exists.
>&!	Redirect cmderr to the specified viewports. Create specified files if they do not already exist. Overwrite existing files regardless of the noclobber setting.

<redirection-operator>

>>&	Redirect cmderr to the specified viewports. Create specified files if they do not already exist. Append new data to the end of existing files if noclobber is off. Generate an error message if noclobber is on and a specified file does not exist.
>>&!	Redirect cmderr to the specified viewports. Create specified files if they do not already exist. Append new data to the end of existing files regardless of the noclobber setting.
<viewport>	A file name or the object number of the CXdb Command window. Each file name is relative to the console working directory unless it is qualified by a path name.
[**,** ...]	A list of additional viewports. Multiple viewports in the list must be separated by commas. Spaces between the list items are optional.

Description

Redirection operators send CXdb output and error messages to the specified viewports, or destinations. A viewport can be either a file or the CXdb Command window.

Redirection operators override the default viewport lists for cmderr and cmdout. They can be used with any CXdb command or within aliases and macros. You can use any number of redirection operators with a single command; the operators affect only the command with which they appear.

The redirection operators must be placed at the end of the command line, after all other command parameters except the background directive (&). If a redirection operator follows a language expression, use the language expression terminator (\;) after the language expression.

The noclobber flag controls writing to the viewport files for cmderr and cmdout. When noclobber is enabled, CXdb responds with an error message if it tries to overwrite an existing viewport file or append to a viewport file that does not exist. When noclobber is disabled, CXdb can overwrite existing viewport files and create new files for appending. The commands to toggle the noclobber flag are:

- `clear noclobber` — Disable noclobber.
- `set noclobber` — Enable noclobber.

By default, noclobber is disabled (clear).

Examples

The following examples illustrate the use of redirection operators. For all these examples, assume that noclobber is enabled (set).

```
(CXdb) info cxdb > tempfile
```

The above example redirects the output of the `info cxdb` command to the file called tempfile. CXdb creates this file in the console working directory. The console working directory is the base directory for all relative path names used in commands that affect CXdb and is initially set to the directory from which you invoke CXdb. (If a file by this name already exists in the console working directory, an error message results because noclobber is on.) Note that only `tempfile` receives the output of this particular command; the output does not appear in the Command window or in any other viewports. However, there is no effect on cmderr and cmdlog for this command or on cmdout for other commands.


```
(CXdb) print I+J\; > tempfile
```

The above example redirects the output of the `print` command to the file tempfile in the console working directory. (If a file by this name already exists in the console working directory, an error message results because noclobber is on.) Note that the language expression terminator (`\;`) is needed to delimit the end of the language expression `I+J`.

```
(CXdb) print I+J\; > tempfile, 1
(INTEGER*4) 7
```

The above example redirects the output of the `print` command to both tempfile and the Command window (Window #1). (If tempfile already exists in the console working directory, an error message results because noclobber is on.)

```
(CXdb) print I+J\; >>! cxdbdata >>&! errlog
```

The above example redirects the output of the `print` command to the file cxdbdata, and it redirects any error messages from this command to the file errlog. The new information is appended to these files, regardless of the setting of noclobber and regardless of whether cxdbdata and errlog already exist.

```
(CXdb) print I+J\; >>! cxdbdata >>&! errlog >tempfile,1
(INTEGER*4) 7
```

The above command redirects the output of the `print` command to the files cxdbdata and tempfile as well as to the Command window (Window #1). Any error messages are directed to the file errlog. The new information is appended to the end of cxdbdata and errlog, but tempfile is overwritten only if it does not already exist (because noclobber is enabled).

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
viewports	

Related Parameters

viewport

<regular-expression>

A character pattern for searches.

Syntax

A regular expression can contain the following search patterns:

Parameter	Meaning
<character>	A single literal character. Characters that are not strictly alphanumeric may have special meaning. To use one of these special characters as part of the search pattern, precede the character with a backslash (\).
.	A special pattern that matches any single character.
[ ]	A set of characters. The set may include single characters or ranges of characters. To specify a character range, use a dash (-).
[^]	The complement of a character set.
*	An operator that repeats the preceding regular expression as many times as possible in order to find a match.
+	An operator that requires at least one match of the preceding regular expression.
?	An operator that requires zero or one match of the preceding regular expression.
|	A logical OR operator that finds matches for the regular expressions on either side of the operator.
\(\)	A group of regular expressions.
<digit>	A count used after a group of regular expressions to indicate which previously matched pattern must be matched again. The allowed digits are 1 through 9.
\b	An empty string that terminates the regular expression.

<regular-expression>

Description

A *<regular-expression>* is a character pattern used to search for a string that matches the pattern. Regular expressions in CXdb are a subset of the regular expressions used with the search function `egrep`.

The CXdb commands that accept regular expressions are:

- `info alias` — List the alias names and their definitions.
- `info macro` — List the macro names and their definitions.
- `info symbols` — List the process symbols from the current scope.
- `info type` — List the type definitions from the current scope.

The above commands return all occurrences that match the specified regular expression.

Examples

The following examples illustrate the use of regular expressions with the `info alias` command.

```
(CXdb) info alias d

d         "disassemble"
dbg       "debug exec"
dbgc      "debug core"
dbgp      "debug proc"
denv+     "add default environment"
denv-     "remove default environment"
denv=     "set default environment"
dis       "disable event"
down      "frame -1"
dp+       "add default path"
dp-       "remove default path"
dp=       "set default path"
```

The above command displays all aliases whose names start with the character `d`.

```
(CXdb) info alias dbg

dbg       "debug exec"
dbgc      "debug core"
dbgp      "debug proc"
```

The above command displays all aliases whose names start with the characters `dbg`.

```
(CXdb) info alias dbg\b

dbg       "debug exec"
```

The above command displays the one alias whose name is `dbg`.

```
(CXdb) info alias d.n

denv+     "add default environment"
denv-     "remove default environment"
denv=     "set default environment"
```

The above command displays all aliases whose names start with a three-character pattern. The pattern is d followed by any character followed by `n`.

```
(CXdb) info alias [h-m]
i         "info"
info application         "info process"
info tasks       "info threads"
k         "kill process"
locals    "info locals"
```

The above command displays all aliases whose names start with any of the characters in the range `h-m`.

```
(CXdb) info alias [aklm]

a         "info args"
k         "kill process"
locals    "info locals"
```

The above command displays all aliases whose names start with either `a`, `k`, `l`, or `m`.

```
(CXdb) info alias [^a-z]

!         "recall"
.         "source"
?         "help"
```

The above command displays all aliases whose names do not start with one of the characters in the range `a-z`.

```
(CXdb) info alias .*\?

?        "help"
b?       "info break"
e?       "info event"
env?     "info environment"
et?      "info eventtype"
p?       "info process"
t?       "info trace"
```

The above command displays all aliases whose names end with the question mark (?) character.

```
(CXdb) info alias [a-z]+\?

b?       "info break"
e?       "info event"
env?     "info environment"
et?      "info eventtype"
p?       "info process"
t?       "info trace"
```

The above command displays all aliases whose names start with one of the characters in the range `a-z` and end with the question mark (?) character.

```
(CXdb) info alias [a-z]?\?

?        "help"
b?       "info break"
e?       "info event"
p?       "info process"
t?       "info trace"
```

The above command displays all aliases whose names end with the question mark (?) character and contain no more than one other character in the range `a-z`.

Related Commands

`info alias` `info macro`
`info symbols` `info type`

<signal-specifier>

A signal identifier.

Syntax

<signal-specifier>

Description

A *<signal-specifier>* indicates the signal to be used in a command. The signal can be referenced by:

- Full name (including the SIG prefix)
- Abbreviation (without the SIG prefix)
- Number

Signal names are *not* case sensitive.

Signal names and numbers are different on C Series and SPP Series machines. To display the signal names, numbers, and signal actions for your system, use the `info signal` command. For more information about signals, refer to the "signals" reference page.

Signal 0 indicates that no signal should be sent to the process.

Examples

The following examples use signals with the `signal process` command.

```
(CXdb) signal process SIGINT
```

The above command sends the `SIGINT` signal to the current process.

```
(CXdb) signal process int
```

The above command sends the `SIGINT` signal to the current process. The signal name can be abbreviated by dropping the `SIG` prefix. Signal names are not case sensitive.

<signal-specifier>

```
(CXdb) signal process 2
```

The above command also sends the `SIGINT` signal (signal number 2) to the current process. The signal number can be used in place of the signal name.

Related Commands

`info signal`
`set signal`
`signal thread`
`event signal`
`signal process`

Related Concepts

signals

<source-unit>

A source unit identifier.

Syntax

[*<file-name>*:] *<integer>*

Parameter	Meaning
<file-name>	The name of the file that contains the source unit of interest. The default is the source file for the current process object.
<integer>	The source unit number that uniquely identifies the source unit of interest.

Description

The *<source-unit>* is the unique identifier of a particular source unit.

Each source unit in a given source file is assigned a unique identification number when you compile the source code with the `-cxdb` option. To display the source unit numbers for all source units on a given line of source code, use the `info line` command.

Examples

The following examples illustrate the use of source unit numbers with several CXdb commands.

```
(CXdb) break source 25

#0: break source, on [#0/*], Enabled, ignore 0/0
 [0x80005452] EXAMPLE in example.f line 17
```

The above command sets a breakpoint at source unit 25 in the current source file of the current process object. The current source file is the source file that contains the current point of execution.

```
(CXdb) goto source example.f:71
```

The above command sets the program counter (PC) to the starting address of source unit 71 in the file example.f.

<source-unit>

Related Commands

```
break source                event reached source
goto source                 info line
info sourceunit             trace source
```

Related Concepts

source units

Related Parameters

file-name

<string>

A character string.

Syntax

<string>

Description

A *<string>* is any sequence of characters taken together as a whole unit. A string is delimited by any of the following:

- White space (blanks or tabs)
- Quotes (')
- Double quotes (")

To include a white space character in a string, either delimit the string with quotes, or precede the space character with a backslash (\). To include one of the quote delimiters (double or single), either delimit the string with the other quote character or precede the quote with the backslash (\).

Examples

The following are examples of valid strings:

```
data1
"data1 data2"
'data1 data2'
data1\ data2\ data3
'echo "routine reached"'
'echo \'routine reached\''
```

The above examples demonstrate different methods for delimiting a string. The backslash character is used to include a delimiting character in the string.

Related Commands

`add default environment`	`add environment`
`echo`	`print`
`set default environment`	`set environment`

Related Parameters

language-expression	regular-expression

<synthesized-variable>

A variable created by the compiler at optimization level `-O1` and above.

Syntax

[**s$** | **s$**][*<object-file>*`]\`*<identifier>*

Parameter	Meaning
s$	One of the delimiters used to distinguish the synthesized variable name from other symbolic names associated with the process.
s$	One of the delimiters used to distinguish the synthesized variable name from other symbolic names associated with the process.
<object-file>	The name of the object file that uses the synthesized variable. The `.o` suffix on the object file name is not required. The default is the object file associated with the current PC (program counter).
`	The delimiter that separates the object file name from the synthesized variable name.
\	The escape character, used to delimit the identifier. It is required when the identifier begins with a special character (such as `?` or `#`).
<identifier>	The name of the synthesized variable. This is the same name that appears in the assembler listing generated by the compiler. The name usually begins with a special character (such as `?` or `#`) that generally is not allowed as part of an identifier in the source language.

<synthesized-variable>

Description

A *<synthesized-variable>* is a variable 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. It also lists the reason for the use of each synthesized variable.

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 only need to display 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. Liveness ranges are regions of memory where the value of the variable has meaning. Outside the liveness ranges, the value of the variable is not available. 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
optimized code
synthesized variables

Related Parameters

language-expression

<thread-list>

A list of process threads.

Syntax

:t {*<thread-number>* [**,** ...] | *****}

Parameter	Meaning
<thread-number>	A thread number. The number must be expressed as an integer.
[**,** ...]	Additional thread numbers in the list. Commas must separate the entries in the list. Spaces between the entries are optional.
*	The wildcard symbol denoting all threads.

Description

A *<thread-list>* is a list of process threads that are affected by a command.

If your process has only one thread, you can omit the thread list from the command. If the process has multiple threads but you do not specify a thread list, CXdb assumes that you want to affect all threads.

NOTE: If you continue process execution on multiple threads, CXdb stops process execution on all threads as soon as one thread has stopped.

Examples

The following examples show the use of thread lists with the `continue` command.

```
(CXdb) :t0 continue
```

The above command continues execution of thread 0 for the current process.

```
(CXdb) :t0,1 continue
```

The above command continues execution of threads 0 and 1 of the current process.

```
(CXdb) :t* continue
```

The above command continues execution of all threads for the current process. It uses the wildcard symbol (*) to specify all threads.

Related Commands

```
backtrace                       break instruction
break line                      break routine
break source                    clear seq
clear sqs                       continue
copy                            disassemble
display disassembly             display examine
display routine                 evaluate
event modify                    event reached instruction
event reached line              event reached routine
event reached source            event relation
examine                         fill
find memory backward            find memory forward
finish                          frame
goto address                    goto line
goto source                     info args
info break                      info errno
info expression                 info frame
info frame at                   info locals
info psw                        info registers
info scope                      info stack
info threads                    info type
info vregisters                 next
next instruction                next over
print                           put
return                          set format
set memory                      set seq
set sqs                         signal thread
step                            step instruction
step over                       trace instruction
trace line                      trace routine
trace source                    watch
```

Related Parameters

process-list

<viewport>

A destination for CXdb input, output, or messages.

Syntax

{*<file-name>* | **1** | **stderr** | **stdout** | **$***<debugger-variable>*}

Parameter	Meaning
<file-name>	The name of a file to be used as a viewport.
1	The window number of the CXdb Command window (in X Windows mode only).
stderr	Standard error (in line mode only).
stdout	Standard output (in line mode only).
$*<debugger-variable>*	A debugger variable that contains the number of the CXdb Command window (in X Windows mode only). The delimiter (**$**) is required in this case.

Description

A *<viewport>* is a destination for receiving CXdb input, output, or messages. You can use any of the following as a viewport:

- A file
- The CXdb Command window (in X Windows mode only)
- stderr (in line mode only)
- stdout (in line mode only)

CXdb maintains three different lists of viewports, based on the type of information sent to the viewports. The names of the lists, and the type of information sent to the viewports in each list, are:

- `cmderr` — Error messages and informational messages generated in response to CXdb commands.
- `cmdlog` — Input entered on the CXdb command line.
- `cmdout` — Output generated in response to CXdb commands.

Examples

The following examples illustrate how to use viewports in several different types of commands.

```
(CXdb) add cmdlog input_log
New cmdlog: input_log
```

The above command adds the file input_log to the viewport list for cmdlog.

```
(CXdb) remove cmderr mycxdb.err, /usr/smith/data/errlog
New cmderr: Window #1, save_errors
```

The above command removes the file names mycxdb.err and errlog from the viewport list for cmderr. The file mycxdb.err is in the console working directory, and errlog is in the directory /usr/smith/data.

```
(CXdb) set cmdout save_cxdbout
New cmdout: Window #1, save_cxdbout
```

The above command establishes a new viewport list for cmdout. This list includes Window #1 (the Command window) by default, along with the newly specified file save_cxdbout.

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

file-name	redirection-operator

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

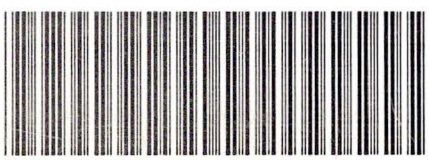
V

W

X

ORDER NUMBER
DSW-472

DOCUMENT NUMBER
710-015022-000



CONVEX PRESS