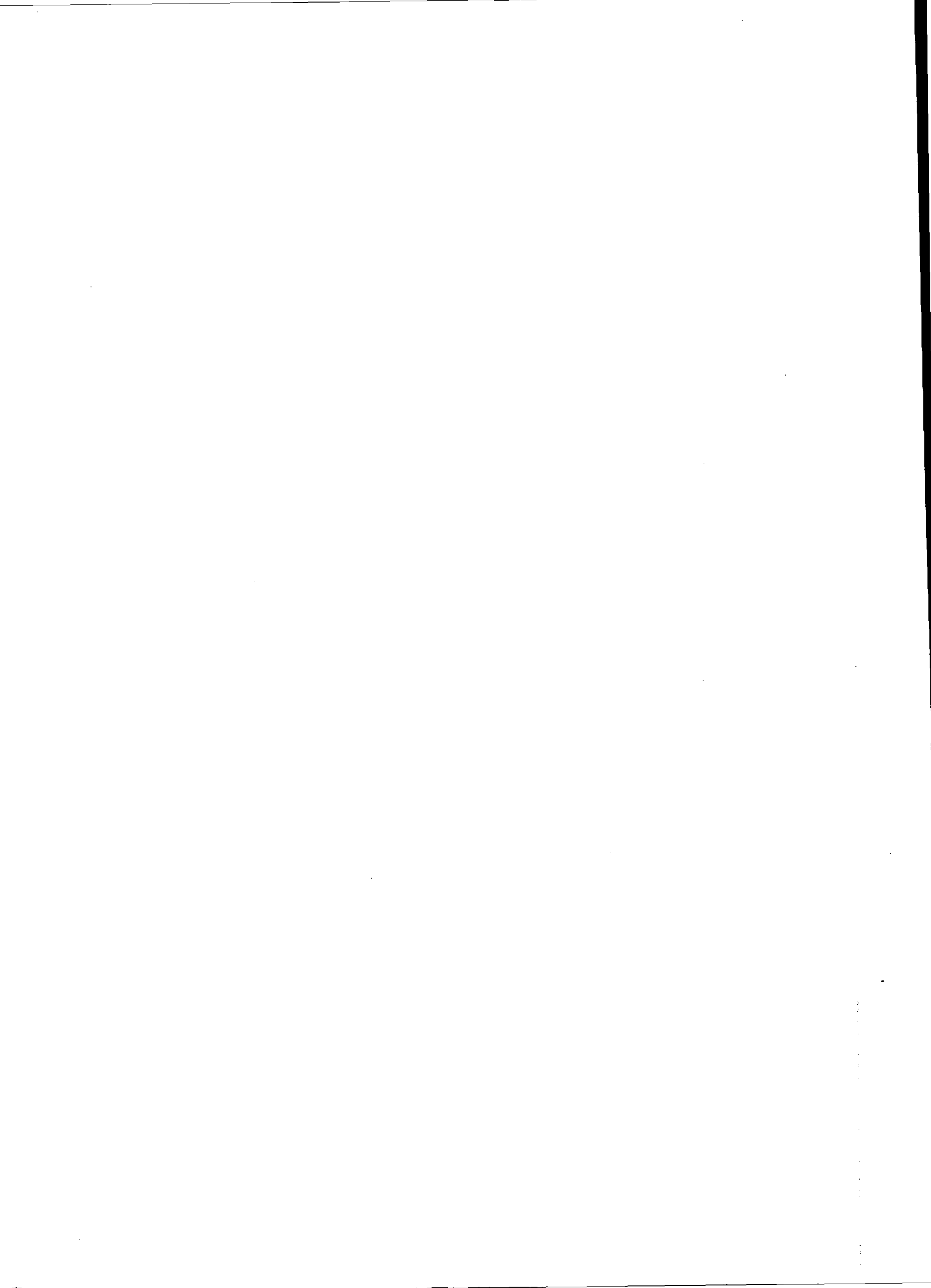
CONVEX CXdb Reference

Second Edition

CONVEX COMPUTER CORPORATION

609

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



CONVEX
CXdb Reference

Order No. DSW-472

Second Edition
December 1991

CONVEX Press
Richardson, Texas
United States of America

CONVEX
CXdb Reference

Order No. DSW-472

Copyright 1991 CONVEX Computer Corporation
All rights reserved.

This document is copyrighted. This document may not, in whole or part, be copied, duplicated, reproduced, translated, electronically stored, or reduced to machine readable form without prior written consent from CONVEX Computer Corporation.

Although the material contained herein has been carefully reviewed, CONVEX Computer Corporation does not warrant it to be free of errors or omissions. CONVEX reserves the right to make corrections, updates, revisions or changes to the information contained herein. CONVEX does not warrant the material described herein to be free of patent infringement.

UNLESS PROVIDED OTHERWISE IN WRITING WITH CONVEX COMPUTER CORPORATION (CONVEX), THE PROGRAM DESCRIBED HEREIN IS PROVIDED AS IS WITHOUT WARANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES. THE ABOVE EXCLUSION MAY NOT BE APPLICABLE TO ALL PURCHASERS BECAUSE WARRANTY RIGHTS CAN VARY FROM STATE TO STATE. IN NO EVENT WILL CONVEX BE LIABLE TO ANYONE FOR SPECIAL, COLLATERAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, INCLUDING ANY LOST PROFITS OR LOST SAVINGS, ARISING OUT OF THE USE OR INABILITY TO USE THIS PROGRAM. CONVEX WILL NOT BE LIABLE EVEN IF IT HAS BEEN NOTIFIED OF THE POSSIBILITY OF SUCH DAMAGE BY THE PURCHASER OF ANY THIRD PARTY.

CONVEX and the CONVEX logo ("C") are registered trademarks of CONVEX Computer Corporation.

ConvexOS is a trademark of CONVEX Computer Corporation

COVUE is a trademark of CONVEX Computer Corporation. COVUE products consist of COVUEbatch, COVUEbinary, COVUEedt, COVUElib, COVUEnet, and COVUEshell.

UNIX is a trademark of AT&T Bell Laboratories.

X Window System is a trademark of M.I.T.

Maryland Windows is copyrighted (c) 1983 University of Maryland Computer Science Department

Printed in the United States of America

Revision Information for

CONVEX CXdb Reference

Edition	Document No.	Description
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

Using this book xi
Purpose and audience xi
Organization xi
Notational conventions xii
Associated documents xiii
Ordering documentation xiv
Technical assistance xiv
Acknowledgments xv

1 Commands 1
add cmderr 3
add cmdlog 5
add cmdout 7
add default environment 9
add default path 11
add environment 13
add path 15
alias 17
attach 21
backtrace 23
bind 25
break instruction 29
break line 33
break routine 37
break source 41
cd 45
clear default environment 47
clear default fixed sched 49
clear default handler 51
clear echo 53
clear environment 55
clear fixed sched 57
clear handler 59
clear logging 61

clear noclobber 63
clear seq 65
clear sqs 67
clear step 69
clear typehandler 71
continue 73
copy 75
core 77
cxdb 79
debug core 85
debug exec 87
debug proc 89
detach 91
disable event 93
disable eventtype 95
disassemble 97
display file 101
display routine 103
echo 105
edit 107
enable event 109
enable eventtype 111
evaluate 115
event exec 117
event join 119
event modify 123
event reached instruction 129
event reached line 133
event reached routine 137
event reached source 141
event relation 145
event signal 149
event spawn 153
examine 157
executable 161
fill 163
find memory backward 167
find memory forward 171
find window backward 175
find window forward 177
finish 179
frame 183
goto address 185
goto line 187
goto source 189
help 191
if 193
info alias 197

info args . . . 199
info bind . . . 201
info break . . . 203
info cregisters . . . 205
info cxdb . . . 207
info default environment . . . 211
info dynamicobject . . . 213
info environment . . . 215
info errno . . . 217
info event . . . 219
info eventtype . . . 223
info expression . . . 227
info formatting . . . 231
info frame . . . 233
info frame at . . . 237
info history . . . 239
info line . . . 241
info locals . . . 245
info macro . . . 247
info objectmap . . . 249
info process . . . 251
info psw . . . 255
info registers . . . 257
info scope . . . 259
info signal . . . 261
info sourceunit . . . 265
info stack . . . 267
info symbols . . . 269
info threads . . . 271
info trace . . . 273
info type . . . 275
info vregisters . . . 279
info watch . . . 281
kill process . . . 283
load object . . . 285
macro . . . 287
next . . . 293
next instruction . . . 297
next over . . . 299
print . . . 303
pwd . . . 309
quit . . . 311
recall . . . 313
remove alias . . . 315
remove cmderr . . . 317
remove cmdlog . . . 319
remove cmdout . . . 321
remove default environment . . . 323

remove default path....325
remove environment....327
remove event....329
remove eventtype....331
remove macro....333
remove path....335
remove variable....337
rerun....339
resume....341
return....343
run....345
set cmderr....349
set cmdlog....351
set cmdout....353
set default environment....355
set default fixed sched....357
set default format....359
set default fpmode....361
set default handler....363
set default memory....365
set default path....367
set default pshell....369
set default step....371
set directory....373
set echo....375
set environment....377
set evalopts fpmode....379
set evalopts iprecision....381
set evalopts rprecision....383
set fixed sched....385
set format....387
set fpmode....389
set handler....391
set ignore....393
set logging....395
set memory....397
set noclobber....399
set path....401
set printopts maxarray....403
set printopts nopadding....405
set printopts padding....407
set printopts precision....409
set pshell....411
set seq....413
set shell....415
set signal....417
set sqs....419
set step....421

set typehandler....423
shell....425
signal process....427
signal thread....429
source....431
step....433
step instruction....437
step over....439
stop....443
trace instruction....445
trace line....449
trace routine....453
trace source....457
watch....461

2 Concepts....467

background execution....469
breakpoints....471
C language expressions....479
cmderr....487
cmdlog....489
cmdout....491
command files....493
Compiler-Debugger Interface....495
console working directory....497
csd....499
debugger variables....501
default environment....505
default search path....507
environment....511
eventpoint handlers....513
eventpoints....517
FORTRAN language expressions....521
gdb....529
initialization files....531
language expressions....535
logging....541
Maryland Windows....547
process object....549
process working directory....553
scope....555
search path....561
signals....563
source units....567
stepping....575
synthesized variables....581
tracepoints....585

viewports....593
watchpoints....597
windows....605
Xdefaults....609

3 Parameters....611

<array-slice>....613
<debugger-variable>....617
<directory-specifier>....619
<environment-variable>....621
<event-handler>....623
<event-specifier>....625
<eventtype-specifier>....627
<file-name>....631
<frame-specifier>....633
<function-name>....635
<granularity>....637
<key-name>....641
<language-expression>....643
<line-specifier>....645
<process-list>....647
<redirection-operator>....651
<regular-expression>....655
<signal-specifier>....659
<source-unit>....661
<string>....663
<synthesized-variable>....665
<thread-list>....669
<viewport>....671

4 CXdb messages....673

Using this book

Purpose and audience

The *CXdb Reference* manual describes each of the CXdb commands and error messages. It also includes descriptions of the major concepts and parameters associated with the commands.

This manual is intended as a reference source for users who are already familiar with CXdb. It assumes that you have read the *CONVEX CXdb Concepts* book and that you understand the basic concepts presented there.

The reference pages contained in this manual are also available through the CXdb online help system.

Organization

This manual is organized into four chapters:

- **Chapter 1, "Commands"**—Contains descriptions, syntax rules and examples for all CXdb commands.
- **Chapter 2, "Concepts"**—Explains the major concepts involved in using the CXdb commands.
- **Chapter 3, "Parameters"**—Describes the major parameters used in CXdb commands.
- **Chapter 4, "CXdb messages"**—Lists and describes all CXdb messages.

Notational conventions

This document uses the following notational conventions.

Command syntax

Consider this example:

```
(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 must be supplied.
4. The horizontal ellipsis in brackets [, ...] indicates that additional parameters may be specified.
5. Either **a** or **b** must be specified.
6. [<*param2*>] indicates an optional parameter.

General conventions

- **Bold constant-width font** identifies user input in examples.
- *italics*
  - Designate user-supplied variables in a command-line 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-X indicates that you must press and hold down the CTRL key and then press the X key.

- References to the *ConvexOS Programmer's Reference* 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.

Associated documents

This manual is not a complete explanation of CXdb. For more information, refer to:

- *CONVEX CXdb Concepts* (DSW-471)—An overview of CXdb and an explanation of how traditional and new debugging concepts are used in CXdb.
- *CONVEX CXdb User's Guide* (DSW-473)—A guide to using CXdb, including examples that you can enter as you read.
- *CONVEX CXdb Quick Reference* (DSW-474)—A quick reference card for using CXdb,containing command syntax and description.

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
- Outside the continental U.S., contact the local 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.

Acknowledgments

The authors wish to thank all the people at CONVEX and the users who contributed their ideas and time in the development of this book. In particular:

- The team who developed CXdb: Gary Brooks, Russell Buyse, Mark Chiarelli, Mike Garzione, Gil Hansen, David Lingle, Steve Simmons, Larry Streepy, and Jeff Woods.
- The team who created tests to help affirm the accuracy of CXdb and this book: Lloyd Tharel and Tim Powell.
- The people who provided vision and management for the product and documentation: Dave Holt, Larry Streepy, Gary Brooks, Kathy Harris, Mike Turner, and Frank Marshall.
- For his assistance in verifying the accuracy of this book: Keith Knox.
- For her editorial comments that help make this book more consistent and readable: Mary Clare Bernier.
- The field test sites who provided valuable feedback that improved CXdb and this book: The Computational Mechanics Co., Inc.; Western Geophysical, Inc.; Panasonic Advanced TV-Video Laboratories, Inc.; and Los Alamos National Laboratories.

—Raymond Cetrone and Kenneth S. Harward

Commands

1

This chapter contains reference pages for all CXdb commands. There is a separate reference page for each command. Each reference page is divided into 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 command being described.
- **Related Concepts** — A list of concepts related to the command, which are described in Chapter 2 of this manual.
- **Related Parameters** — A list of parameters related to the command. CXdb parameters are described more fully in Chapter 3 of this manual.

The heading at the top of the first page of each command description contains the following lines of information, in order, from top to bottom:

- Full command name
- Shortest possible abbreviation of the command name
- Default alias for the command (if one exists). If a command has two default aliases, they are separated by a comma.

You can invoke the command using any of these forms, or you can create your own aliases and macros.

add cmderr

ad cmde

Add viewports to cmderr.

Syntax

`add cmderr` *<viewport>* `[, ...]`

Parameter	Meaning
<viewport>	A file name or the object number of the 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

The `add cmderr` command adds viewports to cmderr.

Cmderr is the list of viewports, or destinations, that receive all error messages and informational messages generated in response to CXdb commands. A viewport may be either a file or the CXdb command window. The default viewport for cmderr is the command window.

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

To display the current 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/debug/errlog, myerrlog
New cmderr: Window #1, errmsgs, /tmp/debug/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/debug`. 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
viewports	windows

Related Parameters

redirection-operator	viewport

add cmdlog

ad cmdl

Add viewports to cmdlog.

Syntax

`add cmdlog` *<viewport>* `[, ...]`

Parameter	Meaning
<viewport>	A file name or the object number of the 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

The `add cmdlog` command adds viewports to cmdlog.

Cmdlog is the list of viewports, or destinations, that receive a log of every command entered in the CXdb command window. The `set logging` command enables logging to the viewports for cmdlog, and the `clear logging` command disables it. The default is logging disabled.

A viewport may be either a file or the CXdb command window. For a viewport that is a file, CXdb creates the file if it does not exist and overwrites the file if it does exist. To prevent overwriting of an existing file, use the `set noclobber` command.

Initially, the viewport list for cmdlog is empty. The commands you enter always display in the command window, regardless of the settings for cmdlog and logging. 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.

To display the setting for logging and the current viewports for cmdlog, use the command `info cxdb`.

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, /usr/local/Smith/inputlog
New cmdlog: logfile, logfile2, /usr/local/Smith/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 `/usr/local/Smith.` The file `logfile` is still included in the viewport list from the previous example.

Related Commands

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

Related Concepts

cmderr
cmdout
viewports
cmdlog
logging
windows

Related Parameters

viewport

add cmdout

ad cmdo

Add viewports to cmdout.

Syntax

add cmdout *<viewport>* [, ...]

Parameter	Meaning
<viewport>	A file name or the object number of the 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

The `add cmdout` command adds viewports to cmdout.

Cmdout is the list of viewports, or destinations, that receive the normal output generated in response to CXdb commands. A viewport may be either a file or the CXdb command window. The default viewport for cmdout is the command window.

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

To display 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/debug/outputlog, myoutputlog
New cmdout: Window #1, outputdata, /tmp/debug/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/debug`. 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
viewports	windows

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 string 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 contain a white space character (a blank) they must be delimited by either quotes or double quotes.

Related Commands

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

Related Concepts

default environment
process object
environment

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.

Examples

The following examples add directories to the default search path.

```
(CXdb) add default path /mnt/jones/project/source
Default search path:
        .
        /mnt/jones/project/source
```

The above command adds the `/mnt/jones/project/source` directory to the default search path. The `.` represents the console working directory, and is the initial setting of the default search path. If the console working directory changes, the new setting remains part of the default search path.

When a new process object is created it will inherit 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.

You can add multiple directories to the default search path by separating each directory with a comma.

```
(CXdb) add default path libraries , ~/math/libraries
Default search path:
        .
        /mnt/jones/project/source
        /mnt/jones/libraries
        /mnt/jones/math/libraries
```

The above example adds two directories to the default search path. The first directory added is relative to the console working directory, in this case the `/mnt/jones` directory. The second directory uses the tilde (~) to indicate that the directory is based from the home directory, which in this example is again the `/mnt/jones` directory. The tilde character is expanded to `/mnt/jones` before the directory is added.

Related Commands

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

Related Concepts

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

Related Commands

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

Related Concepts

default environment
process object
environment

Related Parameters

environment-variable
string
process-list

add path

ad p

p+

Add directories to the process 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.

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.

Examples

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

```
(CXdb) add path /mnt/jones/project/source
Search path:
        .
        /mnt/jones/project/source
```

The above command adds the `/mnt/jones/project/source` directory to the search path. The `.` is inherited from the default search path and represents the console working directory. If the console working directory changes, the new setting remains in the search path. You can set the search path automatically by using this command in a command file.

```
(CXdb) add path /mnt/jones/libraries , /mnt/jones/math/libraries
Search path:
        .
        /mnt/jones/project/source
        /mnt/jones/libraries
        /mnt/jones/math/libraries
```

The above example adds two more directories to the end of the search path.

Related Commands

`add default 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

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 may 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.

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 stepping at [0x80001534] BUILD_ARRAY in build.f line 9
(CXdb) info locals
Process [#0/0]
Frame : 0; [0x80001534] BUILD_ARRAY in build.f line 9
Number of locals : 1
     1 : I = (INTEGER*4) 3
```

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. Note that CXdb emits the `info locals` command while executing the alias.

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

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

```
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 a running process.

Syntax

[*<process-list>*] **attach** *<process-id>*

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

Description

The `attach` command lets you debug a process that is already running.

The image of the process is given to the process object, which must already exist. The process is brought under the control of CXdb and the process is stopped. If the process object has been given an executable file, CXdb associates the image of the running process with the information found in the executable file and any compiler-generated data files.

You can debug an attached process as you normally would any other process, but you cannot restart the attached process from its beginning. However, you can create a new process if you have specified an executable file. When you finish debugging an attached process you can kill the process or, using the `detach` command, let the process continue to run outside the control of CXdb.

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

Examples

The following command attaches CXdb to a running process.

```
(CXdb)  attach 12345
```

This command brings the running process `12345` under the control of CXdb. The process is stopped as soon as CXdb gains control of it.

Related Commands

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

Related Concepts

process object

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

To display information about stack frames, enter the following:

```
(CXdb) backtrace
Process [#0/0]
cf> 0 : 0x80001786 in BESTMV(PILE = (INTEGER*4(1:50)), ROUND = (INTEGER*4)2,
            NPLYRS = (INTEGER*4)3, MAXTK = (INTEGER*4)3) (pickup6.f line 69)
    1 : 0x800013fe in PICKUP() (pickup6.f line 19)
    2 : 0x8000245c in _main(1, 0xffffcc04, 0xffffcc0c)
    3 : 0x800010b8 in ___ap$envret()
```

The above command displays all of the stack frames for all of the threads of the current process. 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 : 0x80001786 in BESTMV(PILE = (INTEGER*4(1:50)), ROUND = (INTEGER*4)2,
              NPLYRS = (INTEGER*4)3, MAXTK = (INTEGER*4)3) (pickup6.f line 69)
    1 : 0x800013fe in PICKUP() (pickup6.f line 19)
More frames follow...
```

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

```
(CXdb) :T2 backtrace
Process [#0/2]
cf> 0 : 0x80001786 in BESTMV(PILE = (INTEGER*4(1:50)), ROUND = (INTEGER*4)2,
              NPLYRS = (INTEGER*4)3, MAXTK = (INTEGER*4)3) (pickup6.f line 69)
    1 : 0x800013fe in PICKUP() (pickup6.f line 19)
    2 : 0x8000245c in _main(1, 0xffffcc04, 0xffffcc0c)
    3 : 0x800010b8 in ___ap$envret()
```

The above command displays all stack frames for thread 2 of the current process.

Related Commands

`frame`	`info frame`
`info frame at`	`info stack`

Related Parameters

process-list	thread-list

bind

Define key bindings for Maryland Windows.

Syntax

bind *<function-name>* *<key-name>*

Parameter	Meaning
<function-name>	The name of one of the command-line editing functions for the command window of the Maryland Windows interface.
<key-name>	The keystroke sequence to bind to the specified function.

Description

The `bind` command binds a particular sequence of keystrokes to one of the command-line editing functions (such as character deletion) for the command window of the Maryland Windows interface.

Only the command-line editing functions are bindable; the window manipulation functions are not. The bindable functions and their default settings in the Maryland Windows interface are:

Keys	Function Name
^@	set-mark-command
^A	beginning-of-line
^B	backward-char
^C	undefined-key
^D	delete-char
^E	end-of-line
^F	forward-char
^G	keyboard-quit
^H	delete-backward-char
Tab	undefined-key
Lfd	newline
^K	kill-line
^L	redraw-display
Ret	newline
^N	down-history
^O	undefined-key
^P	up-history
^Q	undefined-key
^R	redraw-display

```
^S             undefined-key
^T             transpose-chars
^U             universal-argument
^W             kill-region
^X             exchange-point-and-mark
^Y             yank
^Z             undefined-key
Esc            prefix-meta
^\..^_         undefined-key
Spc../         self-insert
0..9           digit
:..~           self-insert
Del            delete-char
M-^@..M-^C     undefined-key
M-^D           kill-word
M-^E..M-^G     undefined-key
M-^H           backward-delete-word
M-Tab..M-/     undefined-key
M-0..M-9       digit-argument
M-:..M-;       undefined-key
M-=            undefined-key
M-?..M-@       undefined-key
M-E..M-L       undefined-key
M-Q            undefined-key
M-S..M-U       undefined-key
M-W            copy-region-as-kill
M-X..M-Y       undefined-key
M-[..M-a       undefined-key
M-b            backward-word
M-c            capitalize-word
M-d            kill-word
M-e            undefined-key
M-f            forward-word
M-g            undefined-key
M-h            backward-delete-word
M-i..M-k       undefined-key
M-l            downcase-word
M-q            undefined-key
M-s..M-t       undefined-key
M-u            upcase-word
M-w            copy-region-as-kill
M-x..M-y       undefined-key
M-{..M-~       undefined-key
M-Del          kill-word
```

Examples

The following examples illustrate how to define key bindings for the Maryland Windows interface.

```
(CXdb) bind transpose-chars c-t
```

The above command defines the keystroke sequence CTRL-t (represented as `c-t`) to be the function `transpose-chars,` which transposes the current character with the one preceding it. Any previous function for CTRL-t is overridden.

To enter the key name for the `bind` command, you literally type `c-t`. However, to use the `transpose-chars` function in the command window of Maryland Windows, hold down the CTRL key and then press t.

```
(CXdb) bind upcase-word m-U
```

The above command defines the keystroke sequence META-U (represented as `m-U`) to be the function `upcase-word,` which converts a word to all uppercase letters. Any previous function for META-U is overridden.

To enter the key name for the `bind` command, you literally type `m-U`. However, to use the `upcase-word` function in the command window of Maryland Windows, you press the META key and then press U.

Related Commands

`info bind`

Related Concepts

Maryland Windows

Related Parameters

function-name   key-name

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 semi-colon (;).
<*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 BESTMV

#0: break instruction, on [#0/*], Enabled, ignore 0/0
      [0x800015f0] BESTMV in pickup.f line 55
```

The above command sets a breakpoint at the first instruction of the routine `BESTMV`. The evaluation of the language expression `BESTMV` 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 executing the `info event` command on the new breakpoint. The output is explained below:

- `#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.
- `[0x800015f0]` — The hexadecimal address location of the breakpoint. In this case the address is `800015f0`.
- `BESTMV in pickup.f line 55` — The symbolic location of the breakpoint. In this case the breakpoint is in the routine `BESTMV` at line 55 of the source file `pickup.f`.

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 '800015f0'x

#1: break instruction, on [#0/*], Enabled, ignore 0/0
      [0x800015f0] BESTMV in pickup.f line 55.
```

The above command sets a breakpoint at the absolute address `800015f0`. The breakpoint number is 1, located at address `800015f0` in routine `BESTMV`, and corresponds to line 55 of the file `pickup.f`. The notation `'800015f0'x` is FORTRAN-specific and indicates the address is in hexadecimal notation.

Using C syntax:

```
(CXdb) break instruction 0x800015f0

#1: break instruction, on [#0/*], Enabled, ignore 0/0
      [0x800015f0] pickup`bestmv in pickup.c line 55.
```

The above command sets a breakpoint at the absolute address `800015f0`. The `0x` is the C notation for a hexadecimal number. The symbolic location uses the scope path of `pickup`bestmv` 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 BESTMV {echo 'routine BESTMV reached'; resume;}

#2: break instruction, on [#0/*], Enabled, ignore 0/0
      [0x800015f0] BESTMV in pickup.f line 55.
    {
       echo 'routine BESTMV reached';
       resume;
    }
```

The above command sets a breakpoint at address `800015f0`, the starting address of routine `BESTMV`. 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 '80001234'x \; $Break4

#4: break instruction, on [#0/*], Enabled, ignore 0/0
       [0x80001234] MAIN in pickup.f line 25.
```

The above command creates a new breakpoint at the absolute address `80001234`. 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 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 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

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 semi-colon (`;`).
<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
        [0x800013c4] PICKUP in pickup.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 executing the `info event` command on the new breakpoint. The output is explained below:

- `#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.
- `[0x800013c4]` — The hexadecimal address location of the breakpoint. In this case the address is `800013c4`.
- `PICKUP in pickup.f line 18` — The symbolic location of the breakpoint. In this case the breakpoint is in the routine `PICKUP` at line 18 of the source file `pickup.f`.

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

```
(CXdb) break line pickup.f:30

#1: break line, on [#0/*], Enabled, ignore 0/0
        [0x80001234] SUB1 in pickup.f line 30
```

The above command sets a breakpoint at the starting address of line 30 of the source file `pickup.f`. This source file must have been part of the compilation of the current executable file, compiled with the `-cxdb` option, and be included in the search path of the process object.

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

#2: break line, on [#0/*], Enabled, ignore 0/0
        [0x800013c4] PICKUP in pickup.f line 18
    {
       echo 'Line 18 reached';
       resume;
    }
```

The above command sets a breakpoint at the starting address of line 18 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 18 $Break3

#3: break line, on [#0/*], Enabled, ignore 0/0
        [0x800013c4] PICKUP in pickup.f line 18
```

The above command creates a new breakpoint at line `18`. 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) (CXdb) break line 20

ERROR: 97
No source statements found.

Line 21 is the next line with object code. Use it? y

#0: break line, on [#0/*], Enabled, ignore 0/0
[0x800013e0] PICKUP in pickup5.f line 21
```

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 `21`.

Related Commands

`break instruction`
`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 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

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 semi-colon (;).
<*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 BESTMV

#0: break routine, on [#0/*], Enabled, ignore 0/0
        [0x800015f2] BESTMV in pickup.f line 59
```

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

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

- `#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.
- `[0x800015f2]` — The hexadecimal address location of the breakpoint. In this case the address is `800015f2`.
- `BESTMV in pickup.f line 59` — The symbolic location of the breakpoint. In this case the breakpoint is in the routine `BESTMV` at line 59 of the source file `pickup.f`.

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 '800015f8'x

#1: break routine, on [#0/*], Enabled, ignore 0/0
       [0x800015f2] BESTMV in pickup.f line 59
```

The above command sets a breakpoint at the starting address of the routine that contains the absolute address `800015f8`. The breakpoint number is 1, located at address `800015f2` in routine `BESTMV` at line 59 of the file `pickup.f`. The notation `'800015f8'x` is FORTRAN-specific and indicates that the address is in hexadecimal notation.

Using C syntax:

```
(CXdb) break routine 0x800015f8

#1: break routine, on [#0/*], Enabled, ignore 0/0
       [0x800015f2] pickup`bestmv in pickup.c line 59
```

The above command sets a breakpoint at the starting address that contains the absolute address `800015f8`. The breakpoint number is 1, located at address `800015f2` in routine `bestmv` in the source file `pickup.c` at line 59. The notation `0x` is C-specific and indicates that the address is in hexadecimal notation.

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

#2: break routine, on [#0/*], Enabled, ignore 0/0
       [0x800015f2] BESTMV in pickup.f line 59
    {
       echo 'routine BESTMV reached';
       resume;
    }
```

The above command sets a breakpoint at the address of the first executable source unit of the routine `BESTMV`. 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 '80001234'x \; $Break4

#4: break routine, on [#0/*], Enabled, ignore 0/0
        [0x80001234] MAIN in pickup.f line 25.
```

The above command creates a new breakpoint at the first executable source unit of the routine containing the absolute address `80001234`. 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 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 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	

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 semi-colon (;).
<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 30

#0: break source, on [#0/*], Enabled, ignore 0/0
        [0x80001394] PICKUP in pickup.f line 14
```

The above command sets a breakpoint at the start of source unit 30 of the current source file.

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

- `#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.
- `[0x80001394]` — The hexadecimal address location of the breakpoint. In this case the address is `80001394`.
- `PICKUP in pickup.f line 14` — The symbolic location of the breakpoint. In this case the breakpoint is in the routine `PICKUP` at line 14 of the source file `pickup.f`.

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

```
(CXdb) break source pickup.f:300

#1: break source, on [#0/*], Enabled, ignore 0/0
        [0x80001234] SUB1 in pickup.f line 30
```

The above command sets a breakpoint at the starting address of source unit 300 of the source file `pickup.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 30 {echo 'Source unit 30 reached'; resume;}

#2: break source, on [#0/*], Enabled, ignore 0/0
        [0x80001394] PICKUP in pickup.f line 14
    {
       echo 'Source unit 30 reached';
       resume;
    }
```

The above command sets a breakpoint at the starting address of source unit 30 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 30 $Break3

#3: break source, on [#0/*], Enabled, ignore 0/0
        [0x80001394] PICKUP in pickup.f line 14
```

The above command sets a breakpoint at the starting address of source unit 30 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 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
trace source	watch

Related Concepts	breakpoints eventpoints tracepoints	debugger variables eventpoint handlers watchpoints
Related Parameters	debugger-variable source-unit thread-list	event-handler process-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 /mnt/jones/project
```

The above command changes the console working directory to be the `/mnt/jones/project` 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 `/mnt/jones/project` directory. Now the console working directory is set to the `/mnt/jones` directory.

Related Commands	`info cxdb` `pwd`	`info process` `set directory`
Related Concepts	console working directory process working directory	process object
Related Parameters	directory-specifier	

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

clear default fixed sched

cl d f s

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 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. 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.

NOTE: Because of the additional system overhead involved with fixed scheduling, it is recommended for debugging multi-threaded 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
```

clear default fixed sched

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.

Examples

The following example clears the default handler.

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

The above command removes the default handler. The default handler returns to being the initial setting, which displays a message.

Related Commands

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

Related Concepts

breakpoints
eventpoint handlers
watchpoints
eventpoints
tracepoints

Related Parameters

event-handler
event-specifier

clear echo

cl ec

Disable echoing of input from initialization files and command files.

Syntax

clear echo

Description

The `clear echo` command disables echoing.

With echoing disabled, commands executed from initialization files and command files are not echoed in the command window. You can enable echoing using the `set echo` command.

By default echoing is disabled.

Examples

The following example disables echoing.

```
(CXdb) clear echo
```

The above command disables echoing. If the `source` command is used, the commands executed are not echoed in the command window.

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 which is empty.

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

clear environment

Related Commands

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

Related Concepts

default environment
process object
environment

Related Parameters

process-list

clear fixed sched

cl f s

cfs

Disable fixed scheduling in the process settings.

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 specified processes. With fixed scheduling disabled, the process executes on any processors that are available during a given time slice. The default is fixed scheduling disabled.

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.

NOTE: Because of the additional system overhead involved with fixed scheduling, it is recommended for debugging multi-threaded 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

clear fixed sched

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 its type. If a handler has not been defined for its 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 for eventpoints executes.

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

The above command clears the handler for eventpoints 0 and 2.

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-handler
event-specifier

clear logging

cl l

Disable logging for cmdlog.

Syntax

```
clear logging
```

Description

The `clear logging` command disables logging to the viewports for cmdlog.

Cmdlog is a list of viewports, or destinations, that receive a log of everything entered in the CXdb command window. When logging is enabled, everything entered in the command window 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 display the current setting of logging, use the command `info cxdb`.

Examples

The following example illustrates how to disable logging.

```
(CXdb) clear logging
```

The above command disables logging to all the viewports of cmdlog.

Related Commands

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

Related Concepts

cmdlog
logging
viewports

Related Parameters

viewport

clear noclobber

cl n

Disable the noclobber option for all viewports.

Syntax

```
clear noclobber
```

Description

The `clear noclobber` command disables the noclobber option.

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
```

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 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 viewports.

clear noclobber

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
	viewports	

Related Parameters		
	redirection-operator	viewport

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 the *CONVEX Architecture Reference*, Chapter 3, "Register Sets."

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 fixed sched        set seq
set sqs
```

Related Parameters

process-list

thread-list

clear sqs

cl sq

Clear the 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 the *CONVEX Architecture Reference*, Chapter 3, "Register Sets."

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`
`set fixed sched`
`set sqs`
`info psw`
`set seq`

Related Parameters

process-list
thread-list

clear step

cl st

Reset the stepping granularity to the default setting.

Syntax

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

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 step` command resets the default granularity (or step size) of the specified process to match the current setting of the CXdb default 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 granularity of a particular process, use the `info process` command. To display the CXdb default granularity, use the `info cxdb` command.

Examples

The following examples illustrate how to reset the default granularity for the stepping commands.

```
(CXdb) clear step
```

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

clear step

```
(CXdb) :T2 clear step
```

The above command resets the step size to be the current value of the CXdb default granularity. This command applies only to thread 2 of the current process.

Related Commands

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

Related Concepts

source units
stepping

Related Parameters

granularity

clear typehandler

cl t

Clear the handler for a specified type of eventpoint.

Syntax

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

Parameter	Meaning
<eventtype-specifier>	An eventtype to clear the handler of. Possible eventtypes are: `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 eventpoints 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

event-handler
event-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 only thread 1 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	windows

Related Parameters

process-list	thread-list

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.
<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_A \; array_B
```

The above command copies the contents of `array_A` into `array_B`. Both arrays are in the current process. Since the command does not specify the number of bytes to copy, all bytes of `array_A` are copied. If `array_B` 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_A:40 \; array_B
```

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

```
(CXdb) copy '800015da'x..'8000161a'x \; '80002000'x
```

The above command copies everything in the region from address `800015da` to `8000161a` into the region starting at address `80002000`.

Related Commands

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

Related Parameters

language-expression
process-list
thread-list

core

cor

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

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process object.
<file-name>	The name of the core file. Relative path names 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 image of the core file is given to the process object, which must already exist. If the process object has been given an executable file, CXdb associates the image of the core file with the information found in the executable file and any compiler-generated data files.

You can use the image of the core file to examine the state of a process that was abnormally terminated when an exception occurred. If the corresponding executable file was specified using the `debug exec` command, or is specified using the `executable` command, you can debug symbolically. Otherwise, CXdb lets you examine the contents of the core file image using absolute addresses and global symbols. Because the image is from a terminated process, you cannot use any process execution commands on the image.

You cannot retrieve the image from a core file if a process already exists for the process object. To kill an existing process, use the `kill process` command.

Examples

The following example retrieves an image from a core file.

```
(CXdb) core core
```

The above command retrieves the image from the core file. The image is associated with the current process object. You can now examine the state of the process that created the core file.

Related Commands

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

Related Concepts

process object

Related Parameters

process-list   file-name

cxdb

Invoke CXdb from the shell.

Syntax

cxdb [**-a** *<process-id>*] [**-b**] [**-D** *<directory-specifier>*[, ...]]
[**-f** *<file-name>*] [**-F**] [**-nw**] [**-nx**] [**-sw** *<geometry>*]
[**-cw** *<geometry>*] [**-hw** *<geometry>*] [**-pw** *<geometry>*]
[**-fw** *<geometry>*] [**-x** *<command-list>*] [[**-e**] *<file-name>*]
[[**-c**] *<file-name>*] [*<x-toolkit-options>*]

Parameter	Meaning
-a *<process-id>*	Attaches CXdb to the process with the specified process ID. 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.
-b	Runs CXdb in batch mode. Batch mode is not interactive. You can use the `-x` and `-f` options in conjunction with the `-b` option to specify what commands are to be executed. You can use shell redirection to redirect standard output and standard error.
-D *<directory-specifier>*	A directory to be added to the default search path. Use a comma (,) to separate multiple directory names in the list. You may use the `-D` option multiple times on the command line. This option performs the `add default path` command.

-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.
-F	Enables fixed scheduling. This option performs the `set default fixed sched` command.
-nw	Forces CXdb to use the Maryland Windows interface, even it is being run in a CXwindows environment. This enables you to use the Maryland Windows interface on an X terminal. It is not necessary to specify this option when running CXdb on a CRT.
-nx	Prevents CXdb from executing initialization files.
-x *<command-list>*	A list of CXdb commands that are executed upon start-up. A semicolon (;) separates 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.
-e *<file-name>*	Specifies an executable file to debug. The executable file, and any associated compiler-generated 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 that is not preceded by an option flag is assumed to be the executable file.
-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 that is not preceded by an option flag is assumed to be the core file.
-sw *<geometry>*	Specifies a geometry for the source window with the Maryland Windows interface. A geometry specification is given as (width) x (height) +(x-origin) +(y-origin) (parentheses for clarity only).
-cw *<geometry>*	Specifies a geometry for the command window with the Maryland Windows interface.
-hw *<geometry>*	Specifies a geometry for the help window with the Maryland Windows interface.
-pw *<geometry>*	Specifies a geometry for the process interface window with the Maryland Windows interface.
-fw *<geometry>*	Specifies a geometry for the display file window with the Maryland Windows interface.
<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 batch mode
- Add directories to the default search path
- Run CXdb using the Maryland Windows interface
- 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 the geometry of windows in Maryland Windows
- Specify X flags (such as window geometries)

When CXdb is invoked, initialization files are executed (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. The `-a` option performs the `debug proc` command and thus creates a process object, because it was specified first. The `-e` option performs the `executable` command and specifies an executable file for the newly created process object.

```
% cxdb -b -x "set cmdlog cxdb.log; set logging" -f batchcmds > cxdb.out
```

The above command invokes CXdb in batch mode. CXdb first executes any initialization files and then executes the `set cmdlog` command. The `set cmdlog` command causes all command logging to go to the `cxdb.log` file, and then the `set logging` command turns logging on. After these command are executed, CXdb sources the `batchcmds` command file. The output from all of the commands executed in batch mode goes to the `cxdb.out` file. CXdb does not exit from batch mode automatically. In the above example, the command file `batchcmds` invokes the `quit` command to exit from CXdb.

```
% cxdb -D /mnt/jones, /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 -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 also need to 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 were used in the alias definitions.

```
% cxdb a.out
```

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

```
% cxdb -c core
```

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` file. Because no executable file was specified, the process object does not yet have an executable file.

```
% cxdb a.out core
```

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 and the image from the `core` file.

```
% cxdb -nw -cw 80x15+0+0
```

The above command invokes CXdb. The `-nw` option cause CXdb to use the Maryland Windows interface. The `-cw` option specifies a geometry for the command window. The command window will have 80 columns and 15 rows, and its upper left corner will be in the upper left corner of the screen.

Related Commands

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

Related Concepts

command files
default search path
initialization files
Maryland Windows
process object
windows
Xdefaults

Related Parameters

directory-specifier
file-name

debug core

deb c

dbgc

Debug a core file or a checkpoint file.

Syntax

debug core *<file-name>* [*<debugger-variable>*]

Parameter	Meaning
<file-name>	The name of the core file. Relative path names use the console working directory as a base.
<debugger-variable>	The debugger variable for this process object.

Description

The `debug core` command lets you debug a core file or a checkpoint file.

When you specify the core file, CXdb creates a process object. The process image is retrieved from the core file and brought into the process object. The process object does not yet have an executable file. Because CXdb V1.0 uses only one process object, the `debug core` command can be used only once in a CXdb session. To examine a different core file image you must use the `core` command.

You can use the image of the core file to examine the state of a process that was abnormally terminated when an exeception occurred. If you specify the corresponding executable file using the `executable` command you can then debug the core file symbolically. Otherwise, CXdb lets you examine the contents of the core file image using absolute addresses or global symbols. If an executable file is later specified, you can debug the image symbolically. Because the image is from a terminated process, you cannot use any process execution commands on the image.

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 command creates a process object with the image of a core file.

```
(CXdb) debug core core
```

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 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

Related Parameters

debugger-variable

file-name

debug exec

deb e

dbg

Debug an executable file.

Syntax

debug exec *<file-name>* [*<debugger-variable>*]

Parameter	Meaning
<file-name>	The name of the executable file. Relative path names use the console working directory as a base.
<debugger-variable>	The debugger variable for the process object.

Description

The `debug exec` command lets you debug an executable file.

When you specify the executable file, CXdb creates a process object. Information from the executable file is brought into the process object. The directory in which CXdb finds the executable file is added to the search path of the process object. If compiler-generated data files for this executable file are found in a `.cxdb` directory along the search path, CXdb maps the information in them to the executable file. These data files exist only if the executable file was compiled with the `-cxdb` option of the latest release of the CONVEX FORTRAN or CONVEX C compilers.

The process object does not yet have an image associated with it. The image to be associated with the executable can be specified using either the `attach` or `core` commands. The image can also be created using the `run` or `rerun` commands. Because CXdb V1.0 uses only one process object, the `debug exec` command can only be used once in a CXdb session. To specify a different executable file you must use the `executable` 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 exec` command has the same effect as using the `-e` parameter by itself when invoking CXdb from the shell prompt.

debug exec

Examples

The following example creates a process object with an executable file.

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

This command creates a process object in CXdb. The process object consists of information found in the executable file named `a.out,` which was located in the console working directory. If data files for this executable file exist in the `.CXdb` directory, then these files are mapped to the executable file. The `.CXdb` directory can be located in any directory on the search path.

Related Commands

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

Related Concepts

debugger variables — process object

Related Parameters

debugger-variable — file-name

debug proc

deb p

dbgp

Debug a running process.

Syntax

debug proc *<process-id>* [*<debugger-variable>*]

Parameter	Meaning
<process-id>	The process ID of the process.
<debugger-variable>	The debugger variable for the process object.

Description

The `debug proc` command lets you debug a running process.

When you specify the process ID of the process, CXdb creates a process object. The process object attaches to the image of the process. The process object does not yet have an executable file or compiler-generated data files. Because CXdb V1.0 uses only one process object, the `debug proc` command can only be used once in a CXdb session. To attach to a different process you must use the `attach` command.

CXdb takes control of the process and the process is stopped. You can debug the process as you normally would, but you cannot restart the attached process from its beginning. However, you can create a new process if you have specified an executable file. When you finish debugging an attached process, you can either kill the process or detach from the process and let it continue to run outside the control of CXdb.

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 by itself when invoking CXdb from the shell prompt.

Examples

The following example creates a process object with the image from a process.

```
(CXdb) debug proc 12345
```

This command creates a process object in CXdb. The process object is given the image of process `12345.` The process object does not have an executable file or any compiler-generated data files associated with it. However, you can debug the process using absolute addresses.

You can associate an executable file with the current process object by using the `executable` command.

Related Commands

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

Related Concepts

debugger variables
process object

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 of 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

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.

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

INFO: 373
Event 3 is already disabled

INFO: 373
Event 2 is already disabled

INFO: 373
Event 1 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 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

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 (*) is used to specify all eventpoint types.
[, ...]	An optional list of additional eventpoint types. Multiple eventpoint types are separated by commas.

Description

The `disable eventtype` command disables all of the 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 of an eventpoint type can be enabled again with the `enable event` or `enable eventtype` command. All eventpoint types can be disabled.

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 only disables 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 1 disabled
Eventpoint 2 disabled
```

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

```
(CXdb) disable eventtype *
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

disassemble

disas

Display the disassembled 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.

disassemble

Description

The `disassemble` command displays the assembly language code for the specified region of memory.

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.

```
(CXdb) disassemble
Disassemble Process [#0/0] from 0x80001766 to 0x80001856
   0x80001766  BESTMV:   sub.w    #0,a0
   0x80001768  BESTMV+(0x2):    ld.w    #0,s0
   0x8000176c  BESTMV+(0x6):    st.w    s0,mth$hwtype+(0x4dc)
   0x80001772  BESTMV+(0xc):    ld.w    @12(ap),s0                  ; MAXTK
   0x80001776  BESTMV+(0x10):   ld.w    @8(ap),s1                   ; NPLYRS
        .
        .
        .
   0x8000184a  BESTMV+(0xe4):   st.w    s0,mth$hwtype+(0x4e4)
   0x80001850  BESTMV+(0xea):   ld.w    mth$hwtype+(0x4e4),s0
   0x80001856  BESTMV+(0xf0):   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 `BESTMV`. Because a count is not specified, the response displays all instructions for routine `BESTMV`. (The vertical ellipsis indicates that part of the response has been omitted.)

```
(CXdb) disassemble '80001800'x .. '80001808'x
Disassemble Process [#0/0] from 0x80001800 to 0x80001808
   0x80001800  BESTMV+(0x9a):   st.b    a0,708706316(a5)
   0x80001806  BESTMV+(0xa0):   ld.w    @12(ap),s1                  ; MAXTK
```

The above command displays the disassembled code starting at address `80001800` of the current process and continuing through address `80001808`. The format used here for the addresses is the FORTRAN syntax for hexadecimal numbers.

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

```
(CXdb) disassemble 0x80001800 .. 0x80001808
Disassemble Process [#0/0] from 0x80001800 to 0x80001808
   0x80001800  BESTMV+(0x9a):   st.b    a0,708706316(a5)
   0x80001806  BESTMV+(0xa0):   ld.w    @12(ap),s1                 ; MAXTK
```

The above command also displays the disassembled code starting at address `80001800` and continuing through address `80001808`. The output is the same as shown in the previous example.

```
(CXdb) disassemble $pc..'80001770'x
Disassemble Process [#0/0] from 0x80001766 to 0x80001770
   0x80001766  BESTMV:  sub.w   #0,a0
   0x80001768  BESTMV+(0x2):    ld.w    #0,s0
   0x8000176c  BESTMV+(0x6):    st.w    s0,mth$hwtype+(0x4dc)
```

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 `80001770`.

```
(CXdb) disassemble INIT
Disassemble Process [#0/0] from 0x800015ba for 40 machine instructions
   0x800015ba  INIT:    sub.w   #8,a0
   0x800015be  INIT+(0x4):      ld.w    @16(ap),s0                 ; NPLYRS
   0x800015c2  INIT+(0x8):      st.w    s0,-4(fp)                  ; <TEMP0>
          .
          .
          .
   0x80001658  INIT+(0x9e):     mul.w   #4,a5
   0x8000165a  INIT+(0xa0):     ld.w    #0,s0
   0x8000165e  INIT+(0xa4):     add.w   a5,a1
```

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

```
(CXdb) disassemble INIT:4
Disassemble Process [#0/0] from 0x800015ba for 4 machine instructions
   0x800015ba  INIT:     sub.w   #8,a0
   0x800015be  INIT+(0x4):       ld.w    @16(ap),s0             ; NPLYRS
   0x800015c2  INIT+(0x8):       st.w    s0,-4(fp)              ; <TEMP0>
   0x800015c6  INIT+(0xc):       ld.w    #1,s0
```

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

Related Commands

`examine`

Related Concepts

windows

Related Parameters

language-expression

process-list

thread-list

display file

disp f

Display the contents of a file.

Syntax

[<*process-list*>] **display file** <*file-name*>

Parameter	Meaning
<*process-list*>	A list of processes affected by this command. The default is the current process.
<*file-name*>	The name of the file to display. The file name is relative to the console working directory, unless the path name is also specified.

Description

The `display file` command opens a new display file window, and displays the contents of the specified file in that window. The file must be an ASCII text file.

In CXwindows, the display file window is an xterm window. In Maryland Windows, it is a subdivision of the screen.

Examples

The following examples illustrates how to display the contents of a text file.

```
(CXdb) display file myfile.txt
```

The above command opens a new display file window, then it displays the contents of `myfile.txt` in that window. The file `myfile.txt` is in the console working directory in this case.

```
(CXdb) display file /tmp/smith/output.log
```

The above command opens a new display window. Then it displays the contents of the file `output.log`, which is in the `/tmp/smith` directory.

display file

Related Commands		
	`cd`	`display routine`
	`edit`	`pwd`

Related Parameters **file-name**

display routine

disp r

Display the source code for a routine.

Syntax

[<*process-list*>] **display routine** <*language-expression*>

Parameter	Meaning
<*process-list*>	A list of processes affected by this command. The default is the current process.
<*language-expression*>	An expression that evaluates to a valid address within the bounds of the specified process.

Description

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

Examples

The following examples illustrate how to display a routine.

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

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

Related Commands

`disassemble`
`display file`
`examine`

Related Parameters

language-expression
process-list
thread-list

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`
`set echo`
`print`

Related Concepts

command files
eventpoint handlers
eventpoints
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

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.

The `edit` command is not available in batch mode.

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.

When you exit from the editor, the editing window closes.

Examples

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

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

The above command invokes the default editor, opens a new window for that editor, and opens the file `myfile.c` for editing in the new window. 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 1 enabled
Event 3 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

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` — 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 Z=3
```

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

```
(CXdb) evaluate Z=X+Y
```

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

```
(CXdb) evaluate $X=PILE(3)+DELTA/2
```

The above command evaluates the language expression PILE(3)+DELTA/2 in the context of the current process. It then assigns the result of this evaluation to the debugger variable X.

```
(CXdb) evaluate $B=BESTMV(PILE,3,4,4)
```

The above command evaluates the function BESTMV in the context of current process. The value returned by BESTMV is stored in the debugger variable B. This command executes BESTMV 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 BESTMV may be triggered by this independent execution from the evaluate command.

Related Commands

info cxdb
set evalopts fpmode
set evalopts rprecision
print
set evalopts iprecision

Related Concepts

C language expressions
FORTRAN language expressions
scope
debugger variables
language expressions

Related Parameters

debugger-variable
process-list
language-expression
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.
<*thread-list*>	A list of threads affected by this command. The default is all threads of the specified process.
<*event-handler*>	A sequence of CXdb commands enclosed in curly-braces ({ }). Each command must be terminated with a semi-colon (;).
<*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

#2: 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	thread-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 semi-colon (;).
<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 *CONVEX CXdb User's Guide* and *CONVEX CXdb Concepts*.

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

`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`
`event spawn`	`info event`
`info eventtype`	`resume`
`set default handler`	`set handler`
`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
process-list	

event modify

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 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 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 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 semi-colon (;).
<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 address being watched 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.

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 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(A)
#1: modify 0x80051008..0x80051197, on [#0/*], Enabled, ignore 0/0
```

The above command sets an eventpoint to monitor the address of the FORTRAN array `A`. The FORTRAN function `loc()` provides the address of the variable.

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 used to identify this particular eventpoint in other CXdb commands. In this case the eventpoint number is 1.
- `modify` — The type of eventpoint.
- `0x80051008..0x80051197` — The address range that the eventpoint monitors.
- `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.

When any value stored in the array `A` changes, the eventpoint is triggered.

```
(CXdb) event modify &count

#1: modify 0xffffc6d4..0xffffc6d7, 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 `count`. The C operator `&` provides the address of the variable. CXdb responds with the same type of information as shown in the FORTRAN example above. The `INFO` message explains that, because the address region is part of the current frame on the stack, the eventpoint will be disabled when this frame is popped from the stack.

When the value stored in `count` 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

#2: 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

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

The above command watches four bytes starting with address `80002345.` The notation `0x80002345` 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

#3: 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;}

#4: 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 is executed, then process execution resumes.

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

#5: modify 0x80051008..0x80051197, on [#0/0], Enabled, ignore 0/0
```

The above command watches the array `A`. The `\;` is needed to separate the language expression from the debugger variable. A debugger variable has been assigned to the eventpoint. In future CXdb commands, you can use the debugger variable `$w1` 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 handler`
`watch`
`set default handler`
`set typehandler`

Related Concepts

breakpoints
eventpoints
tracepoints
debugger variables
eventpoint handlers
watchpoints

Related Parameters

debugger-variable
language-expression
thread-list
event-handler
process-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 semi-colon (;).
<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 BESTMV

#0: reached instruction, on [#0/*], Enabled, ignore 0/0
       [0x800015f0] BESTMV in pickup.f line 55
```

The above command sets an eventpoint at the first instruction of the routine `BESTMV`. The evaluation of the language expression `BESTMV` 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.
- `[0x800015f0]` — The hexadecimal address location of the eventpoint. In this case, the address is `800015f0`.
- `BESTMV in pickup.f line 55` — The symbolic location of the eventpoint. In this case, the eventpoint is in the routine `BESTMV` at line 55 of the source file `pickup.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] BESTMV in pickup.f line 55.
```

The above command sets an eventpoint at the absolute address `800015f0`. The eventpoint number is 1, located at address `800015f0` in routine `BESTMV` corresponding to line 55 of the file `pickup.f`. The notation `'800015f0'x` is FORTRAN-specific and indicates the address is in hexadecimal notation.

Using C syntax:

```
(CXdb) event reached instruction 0x800015f0

#1: reached instruction, on [#0/*], Enabled, ignore 0/0
       [0x800015f0] pickup`bestmv in pickup.c line 55.
```

The above command sets an eventpoint at the absolute address `800015f0`. The `0x` is the C notation for a hexadecimal number. The symbolic location uses the scope path of `pickup`bestmv` 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 BESTMV {echo 'routine BESTMV reached';}

#2: reached instruction, on [#0/*], Enabled, ignore 0/0
       [0x800015f0] BESTMV in pickup.f line 55.
    {
       echo 'routine BESTMV reached';
    }
```

The above command sets an eventpoint at address `800015f0`, the starting address of routine `BESTMV`. 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 '80001234'x \; $Event4

#4: reached instruction, on [#0/*], Enabled, ignore 0/0
      [0x80001234] MAIN in pickup.f line 25.
```

The above command creates a new eventpoint at the absolute address `80001234`. 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          break line
break routine              break source
event exec                 event modify
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 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 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 semi-colon (`;`).
<*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
        [0x800013c4] PICKUP in pickup.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.
- `[0x800013c4]` — The hexadecimal address location of the eventpoint. In this case the address is `800013c4`.
- `PICKUP in pickup.f line 18` — The symbolic location of the eventpoint. In this case the eventpoint is in the routine `PICKUP` at line 18 of the source file `pickup.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 pickup2.f:30

#1: reached line, on [#0/*], Enabled, ignore 0/0
        [0x80001234] SUB1 in pickup2.f line 30
```

The above command sets an eventpoint at the starting address of line 30 of the source file `pickup2.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 line 18 {echo 'Line 18 reached'; resume;}

#2: reached line, on [#0/*], Enabled, ignore 0/0
        [0x800013c4] PICKUP in pickup.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
        [0x800013c4] PICKUP in pickup.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 instruction                break line
break routine                    break source
event exec                       event modify
event reached instruction        event reached routine
event reached source             event relation
event signal                     info event
info eventtype                   resume
set default handler              set handler
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
line-specifier
process-list
thread-list

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 semi-colon (`;`).
<*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 BESTMV

#0: reached routine, on [#0/*], Enabled, ignore 0/0
        [0x800015f2] BESTMV in pickup.f line 59
```

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

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.
- `[0x800015f2]` — The hexadecimal address location of the eventpoint. In this case the address is `800015f2`.
- `BESTMV in pickup.f line 59` — The symbolic location of the eventpoint. In this case the eventpoint is in the routine `BESTMV` at line 59 of the source file `pickup.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 '800015f8'x

#1: reached routine, on [#0/*], Enabled, ignore 0/0
      [0x800015f2] BESTMV in pickup.f line 59
```

The above command sets an eventpoint at the starting address of the routine that contains the absolute address `800015f8`. The eventpoint number is 1, located at address `800015f2` in routine `BESTMV` at line 59 of the file `pickup.f`. The notation `'800015f8'x` is FORTRAN-specific and indicates that the address is in hexadecimal notation.

Using C syntax:

```
(CXdb) event reached routine 0x800015f8

#1: reached routine, on [#0/*], Enabled, ignore 0/0
      [0x800015f2] pickup'bestmv in pickup.c line 59
```

The above command sets an eventpoint at the starting address of the routine that contains the absolute address `800015f8`. The `0x` is the C notation for a hexadecimal number. The symbolic location uses the scope path of `pickup'bestmv` to indicate the source file and routine in which the eventpoint is located.

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

#2: reached routine, on [#0/*], Enabled, ignore 0/0
      [0x800015f2] BESTMV in pickup.f line 59
    {
      echo 'routine BESTMV reached';
    }
```

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

event reached routine

```
(CXdb) event reached routine '80001234'x \; $Event4

#4: reached routine, on [#0/*], Enabled, ignore 0/0
        [0x80001234] MAIN in pickup.f line 25.
```

The above command creates a new eventpoint at the first executable source unit of the routine containing the absolute address `80001234`. The `\;` is needed to separate the language expression from the debugger variable. The debugger variable `$Event4` is created and set equal to the number of this eventpoint. In subsequent commands you could use `$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              break line
break routine                  break source
event exec                     event modify
event reached instruction      event reached line
event reached source           event relation
event signal                   resume
set default handler            set handler
set typehandler                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 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 semi-colon (;).
<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 30

#0: reached source, on [#0/*], Enabled, ignore 0/0
        [0x80001394] PICKUP in pickup.f line 14
```

The above command sets an eventpoint at the starting address of source unit 30 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.
- `[0x80001394]` — The hexadecimal address location of the eventpoint. In this case the address is `80001394.`
- `PICKUP in pickup.f line 14` — The symbolic location of the eventpoint. In this case the eventpoint is in the routine `PICKUP` at line 14 of the source file `pickup.f.`

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

```
(CXdb) event reached source pickup2.f:300

#1: reached source, on [#0/*], Enabled, ignore 0/0
        [0x80001234] SUB1 in pickup2.f line 30
```

The above command sets an eventpoint at the starting address of source unit 300 of the source file `pickup2.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 30 {echo 'Source unit 30 reached'; resume;}

#2: reached source, on [#0/*], Enabled, ignore 0/0
        [0x80001394] PICKUP in pickup.f line 14
    {
       echo 'Source unit 30 reached';
       resume;
    }
```

The above command sets an eventpoint at the starting address of source unit 30 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 30 $Event3

#3: reached source, on [#0/*], Enabled, ignore 0/0
        [0x80001394] PICKUP in pickup.f line 14
```

The above command sets an eventpoint at the starting address of source unit 30 in the current source file. A debugger variable has been assigned to this eventpoint. In subsequent 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

- break instruction
- break line
- break routine
- break source
- event exec
- event modify
- event reached instruction
- event reached line
- event reached routine
- event relation
- event signal
- info line
- info sourceunit
- resume
- set default handler
- set handler
- 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
- source-unit
- process-list
- thread-list

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 semi-colon (`;`).
<*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 a+i
#0: relation (a+i), on [#0/0], Enabled, ignore 0/0
Eventpoint 0 will be disabled when current stack frame returns
```

The above command sets an eventpoint to watch for the relation `a+i` 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:

- `#0:` —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.
- `(a + i)` —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 i .EQ. 4
#1: relation (i .EQ. 4), on [#0/0], Enabled, ignore 0/0
Eventpoint 1 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 FORTRAN, so the expression is in FORTRAN syntax. When `i` equals `4`, the eventpoint is triggered, and the default handler for relation eventpoints is executed.

Using C syntax:

```
(CXdb) event relation i==4
#2: relation (i==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 `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;}
#3: relation (i), on [#0/0], Enabled, ignore 0/0
    {
        print i;
    }
Eventpoint 3 will be disabled when current stack frame returns
```

The above command sets an eventpoint to be triggered when the value of `i` is non-zero. The eventpoint has been given its own handler. The eventpoint handler prints the value of `i`. A semi-colon 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 a(i) .EQ. b(i+5) \; $Checkarrays
#4: relation (a(i) .EQ. b(i+5)), 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 the two arrays 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

```
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 signal                  info event
info eventtype                resume
set default handler           set handler
set ignore                    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.
<*thread-list*>	A list of threads affected by this command. The default is all threads of the specified 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 semi-colon (;).
<*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.

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

```
break instruction          break line
break routine              break source
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
resume                     set default handler
set handler                set signal
trace instruction          trace line
trace routine              trace source
watch
```

Related Concepts

breakpoints
debugger variables
eventpoints
eventpoint handlers
signals
tracepoints
watchpoints

Related Parameters

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

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 semi-colon (;).
<*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 *CONVEX CXdb User's Guide* and *CONVEX CXdb Concepts*.

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

`break instruction`	`break line`
`break routine`	`break source`
`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`	`resume`
`set default handler`	`set handler`
`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
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
- **I** – IEEE mode floating point
- **N** – native mode floating point

<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`)

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 0x80057404 to 0x80057450
 80057404:     00000032 0000002f 00000028 00000021
 80057414:     0000001b 00000018 00000011 0000000c
 80057424:     00000006 00000001 00000001 00000000
 80057434:     00000000 00000000 00000000 00000000
 80057444:     00000000 00000000 00000000 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 0x80057404 to 0x8005740b
 80057404:             0       0       0      50       0       0       0      47
```

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`
`info formatting`
`set default fpmode`
`set format`
`set memory`
`info cxdb`
`set default format`
`set default memory`
`set fpmode`

Related Concepts

cmdout
windows
viewports

Related Parameters

language-expression
thread-list
process-list

executable

exe

Load an executable file for debugging.

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process object.
<file-name>	The name of the executable file. Relative path names use the console working directory as a base.

Description

The `executable` command brings an executable file into a process object.

The executable file is associated with the process object, which must already exist. If the process object has an image, CXdb associates the image with the information found in the executable file and any compiler-generated data files.

The directory in which CXdb finds the executable file is added to the search path of the process object. If compiler-generated data files for this executable file are found in a `.cxdb` directory along the search path, CXdb maps the information in them to the executable file. These data files only exist if the executable file was compiled with the `-cxdb` option of the latest release of the CONVEX FORTRAN or CONVEX C compilers.

Examples

The following example brings a new executable file into a process object.

```
(CXdb) executable projects/a.out
```

The above command brings the executable file named `a.out` located in the `projects` directory into the current process object. The directory where the file was found is added to the search path of the process object. If data files for this executable file exist in the `.CXdb` directory, then these files are mapped to to the executable file. The `.CXdb` directory can be located in any directory on the search path.

Related Commands

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

Related Concepts

process object

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.

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_A \; 3
```

The above command fills `array_A` with the value `3`. Since the command does not specify the number of memory units to fill, all elements of `array_A` 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_A` is a word, then the value `3` is written to each word of the array. On the other hand, if each element of `array_A` 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_A:40 \; X+Y
```

The above command fills the first `40` words (specified by `/w`) of `array_A` with the value of the language expression `X+Y`. The fill takes place by words of memory, and each word is four bytes (32 bits). If each element of `array_A` is also a word, then the above command fills the first 40 elements of `array_A` with the value of `X+Y`.

```
(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

Related Parameters

language-expression
process-list
thread-list

find memory backward

find m b

fmb

Find 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 object.
<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.

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 '800573d4'x:100
Data found at 0x80057404
```

The above command searches for the byte pattern `ff`. It searches backward through the 100-byte memory region at address `800573d4`. In other words, the search starts at address `80057448 (800573d4+64` hex) and proceeds toward address `800573d4`. The first location where it finds the byte pattern is at address `80057404`.

```
(CXdb) find memory backward/w 2b09 INIT..STATUS
Data not found within memory range [0x800015da:0x80001880]
```

The above command searches for the byte pattern `2b09`. It searches backward through the memory region that extends from the routine called `INIT` to the routine called `STATUS`. 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 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 object.
<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.

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 '800573d4'x:100
Data found at 0x80057404
```

The above command searches for the byte pattern `ff` (1111 1111 binary). It searches through the 100-byte memory region that starts at address `800573d4` in the current process. The first location where it finds the byte pattern is at address `80057404`.

```
(CXdb) find memory forward 2b09 INIT..STATUS
Data not found within memory range [0x800015da:0x80001880]
```

The above command searches for the byte pattern `2b09` (0010 1011 0000 1001 binary). It searches through the memory region that starts at the routine called `INIT` and ends at the routine called `STATUS` 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` `find memory backward`	`examine` `print`
Related Concepts	process object	
Related Parameters	language-expression thread-list	process-list

find window backward

find w b

fwb

Find a character string in a source window.

Syntax

`find window backward` *<string>*
[{ *<window-number>* | *<debugger-variable>* }]

Parameter	Meaning
<string>	The character string to search for. If the string contains white spaces, it must be enclosed in quotation marks.
<window-number>	The number of the source window to search. The default is the current source window. (The window number appears in square brackets at the top right of the window title bar.)
<debugger-variable>	A debugger variable that contains the source window number.

Description

The `find window backward` command searches backward through the source file displayed in the specified source window to find the specified character string.

The search starts at the position of the last search (or at the end of the source file, if it is the first search) and continues backward. When it encounters the next occurrence of the specified string, it highlights that string in the specified window. If the search reaches the beginning of the source file before finding the specified string, it reports that the string was not found.

Examples

The following examples illustrate how to search for character strings in source windows.

```
(CXdb) find window backward INIT 2
```

The above command searches backward for the next occurrence of the string `INIT` in window number 2. It then highlights that occurrence.

find window backward

```
(CXdb) find window backward "IF (A" 3
The pattern 'IF (A' was not found.
```

The above command searches backward through window number 3 for the string `IF (A`. The response indicates that the string was not found anywhere in that window.

Related Commands

`display routine` `find window forward`

Related Concepts

windows

Related Parameters

debugger-variable string

find window forward

find w f

fwf

Find a character string in a source window.

Syntax

find window forward *<string>*
[{ *<window-number>* | *<debugger-variable>* }]

Parameter	Meaning
<string>	The character string to search for. If the string contains white spaces, it must be enclosed in quotation marks.
<window-number>	The number of the source window to search. The default is the current source window. (The window number appears in square brackets at the top right of the window title bar.)
<debugger-variable>	A debugger variable that contains the source window number.

Description

The `find window forward` command searches forward for the specified character string in the source file displayed in the specified source window.

The search starts at the position of the last search (or at the beginning of the source file, if it is the first search) and continues forward. When it encounters the next occurrence of the specified string, it highlights that string in the specified window. If the search reaches the end of the source file before finding the specified string, it reports that the string was not found.

Examples

The following examples illustrate how to search forward for character strings in source windows.

```
(CXdb) find window forward INIT 2
```

The above command searches for the next occurrence of the string `INIT` in window number 2. It then highlights that occurrence.

```
(CXdb) find window forward "IF (A" 3
The pattern 'IF (A' was not found.
```

The above command searches forward through window number 3 for the string `IF (A`. The response indicates that the string was not found anywhere in that window.

Related Commands

`display routine` `find window backward`

Related Concepts

windows

Related Parameters

debugger-variable string

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).

Examples

The examples shown below relate to the following FORTRAN source code:

```
1        PROGRAM EXAMPLE
2        PRINT *, "The example program has started."
3        CALL SUBA(10)
4        PRINT *, "The example program is done."
5        END
6
7        SUBROUTINE SUBA(N)
8        INTEGER N
9        PRINT 98, "Subroutine SUBA has started. The value of N is ", N
10       DO K = 1, N
11         PRINT 98, "K = ", K
12         DO L = 1, N
13           PRINT 98, "L = ", L
14         ENDDO
15         PRINT 98, "The loop for L is done, with L = ", L
16         DO M = 1, N
17           I = M + N
18           PRINT 99, "M = ", M, "I= ", I
19         ENDDO
20         PRINT 98, "The loop for M is done, with M = ", M
21       ENDDO
22       PRINT 98, "Subroutine SUBA is done. The value of K is ", K
23       RETURN
24    98 FORMAT (A,I2)
25    99 FORMAT (A,I2,4X,A,I2)
26       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 17.

```
(CXdb) finish
Finishing innermost statement in Process [#0/*]
Process [#0/0] stopped stepping at [0x8000152c] SUBA in example.f line 18
```

Because `statement` is the default granularity, the above command finishes the innermost active statement (line 17). Also because of the default granularity, execution stops at the next statement (line 18). The PC now points to the beginning of line 18.

```
(CXdb) finish loop
Finishing innermost loop in Process [#0/*]
Process [#0/0] stopped stepping at [0x800015aa] SUBA in example.f line 20
```

The above command finishes the innermost active loop that contains the current PC (line 18). That loop begins on line 16 and ends on line 19. Because `statement` is the default granularity, execution stops at the next statement after line 19, which is on line 20.

```
(CXdb) finish loop
Finishing innermost loop in Process [#0/*]
Process [#0/0] stopped stepping at [0x8000160c] SUBA in example.f line 22
```

The above command finishes the innermost active loop that contains the current PC (line 20). That loop actually begins on line 10 and ends on line 21. Because `statement` is the default granularity, execution stops at the next statement after line 21, which is on line 22.

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 13. Enter the following command:

```
(CXdb) finish loop
Finishing innermost loop in Process [#0/*]
Process [#0/0] stopped stepping at [0x800014fa] SUBA in example.f line 16
```

The above command finishes the innermost active loop at line 13. This loop ends at line 14. However, because the default granularity is now `loop`, execution does not stop until the process reaches the next loop after line 14. Thus, when the process stops, the PC points to the beginning of the loop on line 16.

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 to be 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.

Examples

The following examples illustrate how to change the current scope by using the `frame` command.

```
(CXdb) frame 1
     1 : 0x800013fe  in PICKUP()  (pickup6.f line 19)
```

The above command selects frame 1 as the current frame. The response indicates that this frame has an execution address of `800013fe`, which is at line 19 in the file called `pickup.f`. The frame represents the routine called `PICKUP`. 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 : 0x80001786  in BESTMV()  (pickup6.f line 69)
```

The above command uses a relative frame number of `-1`. This sets the current frame to frame 0.

```
(CXdb) up
     1 : 0x800013fe  in PICKUP()  (pickup6.f line 19)
```

The above command uses the alias for `frame +1`. This sets the current frame to frame 1.

Related Commands

```
backtrace               info frame
info frame at           info locals
info process            info stack
```

Related Concepts

scope

Related Parameters

frame-specifier

goto address

ga

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 an address in the default language of the specified process. The 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.

Caution

This 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 '800017d2'x
```

The above command sets the PC to the address `800017d2` (expressed in FORTRAN syntax). It affects all threads of the current process.

goto address

```
(CXdb) goto address 0x800017d2
```

The above command sets the PC to the address `800017d2` (expressed in C syntax). It affects all threads of the current process.

```
(CXdb) goto address SUBX
```

The above command sets the PC to the starting address of the routine `SUBX`. 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

gl

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 by CXdb and are displayed in the source window.) The 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.

Caution

This 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 92
```

The above command sets the PC to the starting address of line `92` of the current source file. It affects all threads of the current process.

```
(CXdb) goto line otherfile.c:92
```

The above command sets the PC to the starting address of line `92` in the source file `otherfile.c.`

Related Commands

`disassemble`
`display file`
`display routine`
`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 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.

Caution

This 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 otherfile.c:92
```

The above command sets the PC to the starting address of source unit `92` in the source file `otherfile.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 topics containing the string are displayed. 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.
- **CXdb messages** — Text and descriptions of messages that are generated as responses to CXdb commands. The messages are listed by number.

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. For a further explanation of the help system as well as a tutorial on the subject, refer to the *CONVEX CXdb User's Guide* or the *CXdb Online Guide.*

Examples

The following examples illustrate different methods of requesting help from CXdb.

```
(CXdb) help
```

The above command invokes the help system and brings up the help window.

```
(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 21
```

The above command invokes the help system and brings up the text that describes CXdb message `21`.

Related Concepts

windows

Related Parameters

string

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 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; [0x80001786] SUBR2 in myfile.f line 74
Number of arguments : 4
     1 : ARRAY= INTEGER*4(1:50) 0x80057404
     2 : VAR1= (INTEGER*4) 2
     3 : VAR2= (INTEGER*4) 3
     4 : VAR3= (INTEGER*4) 5
```

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 `SUBR2`, which is in the source file called `myfile.f`. The current value of the program counter (PC) is `80001786`, which is the address of the current source unit in line 74 of `myfile.f`. Frame 0 is the current frame, and it has four arguments. The names of the arguments are `ARRAY`, `VAR1`, `VAR2`, and `VAR3`. `ARRAY` is an array of 50 elements, each of which is a four-byte integer, and the starting address of the array is `80057404`. Each of the other arguments is a single four-byte integer. The current value of `VAR1` is 2, `VAR2` is 3, and `VAR3` is 5.

Related Commands

`backtrace`
`info frame`
`info symbols`
`info scope`
`print`
`info expression`
`info frame at`
`info locals`
`info stack`

Related Parameters

process-list
thread-list

info bind

in bi

i bi

Display key bindings for Maryland Windows.

Syntax

`info bind` [*<key-name>*]

Parameter	Meaning
<key-name>	The keystroke sequence whose function you want to display.

Description

The `info bind` command displays the key bindings for the command window of the Maryland Windows interface. The key bindings define a particular sequence of keystrokes used to invoke each of the Maryland Windows functions such as cursor movement or scrolling.

Examples

The following examples illustrate how to display the key bindings for the Maryland Windows interface.

```
(CXdb) info bind c-v
 ^V            down-screen
```

The above command displays the function represented by the keystroke sequence CTRL-V. The response from CXdb indicates that this key sequence represents the function down-screen, which scrolls the display down by one screen. To enter the key name for the `info bind` command, you literally type `c-v`. However, to use the down-screen function in Maryland Windows, you hold down the CTRL key and press V.

info bind

To list all current key bindings, you can enter the following command:

```
(CXdb) info bind
 ^@              set-mark-command
 ^A              beginning-of-line
 ^B              backward-char
 ^C              undefined-key
 ^D              delete-char
                  .
                  .
                  .
 M-z             resize-window
 M-{..M-~        undefined-key
 M-Del           kill-word
```

The above command displays all the current key bindings for the command window of the Maryland Windows. In this example, the key bindings shown are the defaults. The vertical ellipsis has been added to indicate that some of the output is omitted from the example.

Related Commands

bind

Related Concepts

Maryland Windows

Related Parameters

function-name   key-name

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/*     [0x80001344] PICKUP in pickup.f line 7
```

The above command displays a table of the settings of all the breakpoints. 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 is displayed.

```
(CXdb) info break

Event   Enabled Ignore  proc/td Address         Where
#0         y     0/0     0/*     [0x80001344] PICKUP in pickup.f line 7
#1         y     0/0     0/*     [0x80001348] PICKUP in pickup.f line 8
    {
        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

thread-list

info cregisters

in cr
i cr

Display the communication registers.

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 use different configurations for the communication registers. For more information about these registers, refer to the *CONVEX Architecture Reference* manual, Chapter 5, "Multiprocessor Management."

Examples

The following example illustrates how to display the contents of the communication registers.

```
(CXdb) info cregisters

Process [#0/0]
 C[00] 0000000000000000 (0)
        ...
 C[63] 0000000000000000 (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[00]`. The value in parentheses `(0)` at the end of each line is the lock bit for that register.

Related Commands		
	`info frame`	`info frame at`
	`info registers`	`info stack`
	`info vregisters`	`print`
Related Concepts	debugger variables	windows
Related Parameters	process-list	thread-list

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: 26947
             cwd: /devel/smith/proj3
   command modes: echo on, logging off, noclobber off
          cmdout: Window #1
          cmderr: Window #1
          cmdlog:
        evalopts: fpmode = dual, iprecision = 4, rprecision = 4
           shell: tcsh

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


       PROCESSES:
    process [#0]: created pid 27370, state = running, executable = a.out
                  shell = csh
```

```
 ACTIVE COMMANDS:

command [#93] - 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
windows
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=/mnt/jones
SHELL=csh
EDITOR=emacs
NEWS=rn
MORE=-c
```

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
```

Related Concepts

default environment

environment

Related Parameters regular-expression

info dynamicobject

i 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, nor 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 illustrate 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  0x00000000  0x80013000  0x00000000  obj1.o
```

The above command displays a table showing the addresses for the `obj1.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.

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
SHELL=/bin/csh
USER=/jones
TERM=xterm
EDITOR=vi
PAGER=less
LESS=-MQce
```

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.

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 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 error message associated with the current value of `errno`.

Examples

The following example illustrates how to display error messages received by the process.

```
(CXdb) info errno
thread 0: errno = 2 - No such file or directory
```

The above command displays the current 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
           [0x8000135a] PICKUP in pickup5.f line 9
```

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:

- `[0x8000135a]` — The hexadecimal address of the eventpoint.
- `PICKUP in pickup5.f line 9` — The symbolic location of the eventpoint. The eventpoint is located in the `PICKUP` routine of the source file `pickup5.f` at line 9.

```
(CXdb) info event 1,2

#1: trace routine, on [#0/*], Enabled, ignore 0/0
         [0x800013c8] PRIME in primes.f line 18

#2: event reached routine, on [#0/*], Enabled, ignore 0/0
         [0x800013c8] PRIME in primes.f line 18
    {
        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
          [0x8000135a] PICKUP in pickup5.f line 9

#1: trace routine, on [#0/*], Enabled, ignore 0/0
        [0x800013c8] PRIME in primes.f line 18

#2: event reached routine, on [#0/*], Enabled, ignore 0/0
        [0x800013c8] PRIME in primes.f line 18
    {
        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
          [0x8000135a] PICKUP in pickup5.f line 9
```

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
eventpoints
eventpoint handlers
tracepoints
watchpoints

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

#0: break line, on [#0/*], Enabled, ignore 0/0
          [0x8000135a] PICKUP in pickup.f line 9
```

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:

- `[0x8000135a]` — The hexadecimal address of the eventpoint.
- `PICKUP in pickup5.f line 9` — The symbolic location of the eventpoint. The eventpoint is located in the `PICKUP` routine of the source file `pickup5.f` at line 9.

```
(CXdb) info eventtype trace, reached

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
          [0x80001394] PICKUP in pickup.f line 14

Status of eventpoints of type Reached:

#2: event reached routine, on [#0/*], Enabled, ignore 0/0
       [0x800013c8] PRIME in pickup.f line 18
    {
        print I;
        resume ;
    }
```

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: event reached routine, on [#0/*], Enabled, ignore 0/0
       [0x800013c8] PRIME in pickup.f line 18
    {
        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
            [0x80001394] PICKUP in pickup.f line 14

Status of eventpoints of type Breakpoint:

#0: break line, on [#0/*], Enabled, ignore 0/0
           [0x8000135a] PICKUP in pickup.f line 9
```

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 X
object type: Fortran identifier
   location: 0x8005e8a4
       size: 4 bytes
       type: REAL*4
      value: 3.0000
       3 liveness ranges:
               Start      End          Location
         1. 0x80001560:0x80001568 - register s1
         2. 0x80001594:0x800015a2 - register s0
         3. 0x8004a000:0x8004b000 - 0x8005e8a4
```

The above command displays information about the variable `X` from the current process. The liveness ranges indicate where `X` 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 `X` is not available and cannot be displayed.

```
(CXdb) info expression (X .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 `(X .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 PILE
object type: Fortran array
orientation: column
     bounds: REAL*4(1:<TEMP0>, 1:<TEMP1>)
  base type: REAL*4
  base size: 4 bytes
 total size: 40000 bytes
  base addr: 0x80001840
```

The above command displays information about the array `PILE` 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 J
object type: Fortran identifier
   location: <none>
       size: 4 bytes
       type: INTEGER*4
      value: 3
       used to create 1 synthesized variable(s):
         1. <INDV> ?i7 = ?i1+((4*N)*(J-1))
```

The above command displays information about the program variable `J`, which is a loop induction variable. Because the program that contains `J` has been compiled with the `-O1` optimization option, the compiler replaces the use of `J` with the synthesized variable `?i7`. The equation that the compiler generates to calculate `J` is also displayed. Because `J` is not used, it is not stored. Therefore, no storage location or liveness ranges are listed for `J`.

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 SUBB
object type: Fortran function
entry point: 0x80001550
return type: void
  arg count: 4
  prototype: void SUBB( INTEGER*4, INTEGER*4, REAL*4, REAL*4 )
```

The above command displays information about the FORTRAN subroutine `SUBB` in the current process. The prototype shows that `SUBB` does not return a value (void), but it accepts four arguments as input.

```
(CXdb) info expression sub7c
object type: C function
entry point: 0x800029ae
return type: void
  arg count: 5
  prototype: void sub7c( int, int, float, float, int* )
```

The above command displays information about the C function `sub7c` in the current process. The prototype shows that `sub7c` does not return a value (void), but it accepts five arguments as input.

```
(CXdb) info expression $B5
object type: debugger variable
   writable: yes
   var type: reference to eventpoint [#5]
```

The above command displays information about the debugger variable `$B5`. 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

language expressions
synthesized variables

Related Parameters

language-expression
process-list
synthesized-variable
thread-list

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:

- `maxarray` — The maximum number of array elements printed in a single execution of the print command.
- `precision` — The format precision used for printing floating point numbers.

Each memory 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. 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:
    max array elements: 20
 floating point format: 10.4

Process [#0] format settings:

Format settings for Thread 0
           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`.

The memory format settings show that the default memory unit is a `byte`, and the default format for displaying bytes is a `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 provides the ability to override these defaults.

Related Commands

`examine`
`set default format`
`set default memory`
`set fpmode`
`set printopts maxarray`
`info cxdb`
`set default fpmode`
`set format`
`set memory`
`set printopts precision`

Related Parameters

process-list
thread-list

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.

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, Chapter 4, "Memory Management."

Examples

The following examples illustrate how to display stack frames.

```
(CXdb) info frame
Process [#0/0]
Frame : 0; [0x80001786] BESTMV in pickup6.f line 69
Frame address : 0xffffccb0
Saved registers : pc=0x800013fe   psw=0x790b600   fp=0xffffccd0   ap=0x80001c20
Floating point mode : IEEE;  Language : FORTRAN
Routine return type : INTEGER*4
Number of arguments : 4
```

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 than the frame for the current point of execution.

The displayed information includes 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)
  - The arguments pointer (ap)
- Floating point mode — The mode for performing floating point operations.
- Language — The source language for the routine represented by the frame.
- Routine return type — The data type for the value returned by this routine.
- Number of arguments — The number of arguments passed to this routine.

```
(CXdb) info frame 1
Process [#0/0]
Frame : 1; [0x800013fe] PICKUP in pickup6.f line 19
Frame address : 0xffffccb0
Saved registers : pc=0x800013fe   psw=0x790b600   fp=0xffffccd0   ap=0x80001c20
Floating point mode : IEEE;  Language : FORTRAN
Number of arguments : 0
```

The above command displays information for frame `1` of the current process.

Assume that frame 2 is the current frame, and you enter the following command:

```
(CXdb) info frame -1
Process [#0/0]
Frame : 1; [0x800013fe] PICKUP in pickup6.f line 19
Frame address : 0xffffccb0
Saved registers : pc=0x800013fe   psw=0x790b600   fp=0xffffccd0   ap=0x80001c20
Floating point mode : IEEE;  Language : FORTRAN
Number of arguments : 0
```

The above command uses a relative frame number of `-1`. Because the current frame is frame 2, the command displays frame 1.

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

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.

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, Chapter 4, "Memory Management."

Examples

The following examples illustrate how to display stack frames.

```
(CXdb) info frame at 'ffffca98'x
Process [#0/0]
Frame : 2; [0x8000135e] EXAMPLE in arrays.f line 4
Frame address : 0xffffca98
Saved registers : pc=0x8000135e   psw=0x87109400   fp=0xffffcaa8   ap=0x80001604
Floating point mode : NATIVE;  Language   : FORTRAN
Number of arguments : 0
```

The above command displays information about the frame stored at address `ffffca98`.

The displayed information includes 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)
  - The arguments pointer (ap)
- Floating point mode — The mode for performing floating point operations.
- Language — The source language for the routine represented by the frame.
- Number of arguments — The number of arguments passed to the routine represented by the frame.

```
(CXdb) info frame at $fp
Process [#0/0]
Frame : 1; [0x800013d6] SUBA in arrays.f line 12
Frame address : 0xffffca74
Saved registers : pc=0x800013d6   psw=0x7109400   fp=0xffffca98   ap=0xffffca88
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`).

Related Commands

backtrace	frame
info args	info frame
info locals	info process
info scope	info stack

Related Concepts

scope

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 in the command window.

Examples

The following examples illustrate how to display the command history.

```
(CXdb) info history

debug exec a.out
break line 11
run
step
step
step
break line 17
continue
continue
continue
next
next
info process
print N
info history
```

The above command displays all commands currently stored in the command history. In this case, there are 15 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

next
info process
print N
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 on 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 on the selected line. The information displayed includes the source unit number, the address location, the source window location, 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.

```
20      SUBROUTINE PRIME(N)
21      INTEGER N,A(1000)
22      DO J = 2, N
23        IF (A(J) .EQ. 1)
24          K = J*2
25          DO WHILE (K .LE. N)
26            A(K) = 0
27            K = K + J
28          ENDDO
39        ENDIF
30      ENDDO
31      RETURN
32      END
```

The above FORTRAN code is used in the following examples.

info line

```
(CXdb) info line 20
     Id    Address Boundaries      Start     End    Kind
1. ( 28)   800013f0:8000149a      20 x 7   32 x 9 <ROUT> SUBROUTINE PRIME(N) <...>
```

The above example displays information about the source units on line 20. The information displayed is broken into the following units:

- `Id` — The source unit number. This number can be used in commands which take 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.

```
(CXdb) info line 25
     Id  Address Boundaries       Start    End    Kind
1. ( 45) 80001438:8000147a       25 x 11 28 x 15 <LOOP> DO WHILE (K .LE. N)
2. ( 46) 8000146c:80001478       25 x 21 25 x 28 <EXPR> K .LE. N
         80001438:80001444
3. ( 47) 80001470:80001476       25 x 21 25 x 21 <EXPR> K
         8000143c:80001442
4. ( 48) 8000146c:80001470       25 x 28 25 x 28 <EXPR> N
         80001438:8000143c
```

The above command displays the source units on line 25. There are four source units. Source units 46, 47, and 48 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.

```
(CXdb) info line 27
     Id  Address Boundaries      Start    End    Kind
1. ( 53) 80001458:8000146c     27 x 13 27 x 21 <STMT> K = K + J
2. ( 54) 80001458:80001466     27 x 17 27 x 21 <EXPR> K + J
3. ( 56) 80001458:8000145e     27 x 21 27 x 21 <EXPR> J
4. ( 55) 8000145e:80001464     27 x 17 27 x 17 <EXPR> K
```

The above example displays the source units on line 27. There are four source units.

Related Commands

`break source`
`finish`
`info sourceunit`
`next over`
`set step`
`step over`
`event reached source`
`goto source`
`next`
`set default step`
`step`
`trace source`

Related Concepts

breakpoints
source units
tracepoints
eventpoints
stepping

Related Parameters

granularity
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; [0x80001324] PICKUP in pickup.f line 7
Number of locals : 5
 1 : I = Value is not available.
 2 : TURN = (INTEGER*4) 0
 3 : ROUND = (INTEGER*4) 0
 4 : AGAIN = Value is not available.
 5 : PLAYER = INTEGER*4(1:50, 1:2) 0x80055018
```

The above command displays the local variables of the routine for the current frame. The current frame is frame 0. For each local variable the name, size, and value is displayed, if applicable. The variables `I` and `AGAIN` are never referenced. The variable `PLAYER` is a two-dimensional array of integers. The hexadecimal address shown is the starting address of the array.

Related Commands

`backtrace`
`frame`
`info frame`
`evaluate`
`info args`
`info symbols`

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

Related Parameters **regular-expression**

info objectmap

in o

i ob

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 includes address ranges and object file names for each of the following segments of memory, when applicable:

- Text — Object code.
- Data — Initialized data.
- Tdata — Initialized thread-specific data.
- Bss — Uninitialized data.
- Tbss — Uninitialized thread-specific data.

Examples

The following example shows how to display object map information.

```
(CXdb) info objectmap

Object Map for Process [#0]:
Executable: a.out

   Address Range                    Section
low              high               Type     Object

0x80001300       0x800022b0         Text     myfile.o
0x80044070       0x80044070         Data     myfile.o
0x80056000       0x80056000         Tbss     myfile.o
0x80057008       0x80057500         Bss      myfile.o
```

The above command displays the object map information for the current process. The executable file is a.out, and the object file is `myfile.o`. The text segment of the object file starts at address `80001300` and continues through `800022b0`, the data segment is at `80044070` to `80044070`, the tbss segment is at `80056000` to `80056000`, and the bss segment is at `80057008` to `80057500`.

Related Commands

`disassemble` `examine`
`info process`

Related Concepts

process object

Related Parameters

process-list

info process

in pr

p?

Display the status of the current 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 current process object.

```
(CXdb) info process

status of process [#0]:

      executable: a.out
       arguments: (none)
fixed scheduling: off
          pshell: csh
    image status: created pid 6270, state = stopped
     working dir: /mnt/jones/project
    default step: statement
default language: Fortran
         threads: 1
  current thread: 0

 thread 0 status: stopped at [0x8000139c] TEST in test.f line 12

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`.
- `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.
- `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 which 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`
`remove path`	`rerun`
`run`	`set fixed sched`
`set pshell`	`set path`

Related Concepts

console working directory	default search path
process object	process working directory
search path	stepping

Related Parameters

process-list

info psw

in ps

i ps

Display the processor status word.

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, refer to the *CONVEX Architecture Reference* manual, Chapter 3, "Register Sets."

Examples

The following example illustrates how to display the contents of the PSW.

```
(CXdb) info psw
Contents of Process Status Word for thread 0

PSW = 0x390b680
80000000  no   A carry
40000000  no   A integer overflow
20000000  no   A zero divide
10000000  no   Integer overflow enable
08000000  no   Trace
06000000  1    Frame length
01000000  yes  Sequential
00800000  yes  S carry
00400000  no   S integer overflow
00200000  no   S 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  yes  IEEE
00001000  yes  Sequential stores
00000800  no   Intrinsic error
00000400  yes  Intrinsic error enable
00000200  yes  Trace thread creates
00000100  no   Thread init trap
000000e0  4    Reserved
0000001f  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 `0390b680`.

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 0, as indicated by the word `no` 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.

Related Commands

`clear seq`	`clear sqs`
`frame`	`info frame`
`info frame at`	`print`
`set seq`	`set sqs`

Related Concepts

debugger variables

Related Parameters

process-list	thread-list

info registers

in r

i r

Display the scalar and address registers.

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 the registers for the specified process. The display is in hexadecimal format.

The registers displayed are:

- Program counter (PC)
- Processor status word (PSW)
- Address registers (A0 to A7)
- Scalar registers (S0 to S7)

For more information about these registers, refer to the *CONVEX Architecture Reference* manual, Chapter 3, "Register Sets."

Examples

The following example illustrates how to display the registers.

```
(CXdb) info registers
Process [#0/0]
pc : 8000151ax
psw: 03909480x
fp : ffffcaa0x
ap : 800016b8x
a5 : 800c9024x
a4 : 00000000x
a3 : 00000090x
a2 : 8004db54x
a1 : 800c8100x
sp : ffffca90x
s7 : 6f66204e00000000
s6 : 652076616c756520
s5 : 77697468204c203d
s4 : 7320646f00000000
s3 : 20666f7200000000
s2 : 5468652000000000
s1 : 204c203d00000001
s0 : 000000000000000a
```

The above command displays the registers for all threads of the current process. There are eight scalar registers, designated as `s0` through `s7`, and eight address registers, designated as `a0` through `a7`. Register `a0` is called the stack pointer (`sp`), register `a6` is called the argument pointer (`ap`), and register `a7` is called the frame pointer (`fp`).

Related Commands

`info cregisters`
`info frame at`
`info stack`
`print`
`info frame`
`info psw`
`info vregisters`

Related Concepts

debugger variables
windows

Related Parameters

process-list
thread-list

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$PICKUP
```

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.
  `PICKUP` — 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 2 scope: c$pickup`main`1
```

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 the settings of the 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.

Examples

The following examples display the settings for signals.

```
(CXdb) info signal

The current signal actions are:

Signal                        Signal
number  Stop  Pass  Print     name
------  ----  ----  -----     ---------------------
0        Yes   Yes   Yes      Signal 0
1        Yes   Yes   Yes      Hangup
2        Yes   No    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       Yes   Yes   Yes      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. 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 to the related concept, "signals", in Chapter 2 of this manual.

The settings displayed above are the default settings for all 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:

#1: 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.

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>*] [*<thread-list>*] **`info sourceunit`** *<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 specified process.
<source-unit>	The identification number of the desired source unit.

Description

The `info sourceunit` command displays information about the specified source unit. 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.

To obtain the identification numbers of all source units on a given line of source code, use the `info line` command.

NOTE: Source unit numbers are random and can change between different compilations of the same source file.

Examples

The following examples illustrate how to display information about source units.

```
(CXdb) info sourceunit 25
 Id     Address Boundaries    Start      End     Kind
( 25)   800013a6:800013b0    11 x 10    11 x 18  <STMT>  ROUND = 2
```

The above command displays information about source unit `25` from the current process. The address range for this source unit is `800013a6` to `800013b0`. The source unit starts on line `10`, column `11` of the source file and ends on line `11`, column `18`. The source unit is a statement (`STMT`), and the text of the statement is `ROUND=2`.

Related Commands

`break source`
`goto source`
`trace source`
`event reached source`
`info line`

Related Concepts

source units

Related Parameters

granularity
source-unit
process-list
thread-list

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
- 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: 3
        current: 0
        extent: 0xffffcbc0 - 0xffffcb58
```

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 three frames on this stack, and frame 0 is the current frame. The memory extent for the stack is `ffffcbc0` (highest address) to `ffffcb58` (lowest address).

info stack

```
(CXdb) :T1 info stack
Process [#0/1] stack:
        frames: 3
        current: 2 at 0xffffcb78
        extent: 0xffffcbc0 - 0xffffcb58
```

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 `ffffcb78`.

Related Commands

`backtrace`
`frame`
`info frame`
`info frame at`
`info process`

Related Parameters

process-list
thread-list

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
PICKUP
 1. SUBROUTINE PICKUP()
 2. INTEGER*4 I
 3. INTEGER*4 TURN
 4. INTEGER*4 ROUND
 5. INTEGER*4 AGAIN
 6. INTEGER*4 PLAYER(1:50, 1:2)
 7. INTEGER*4 PILE(1:50)
 8. INTEGER*4 FUNCTION BESTMV(...)
 9. SUBROUTINE INIT(...)
10. SUBROUTINE STATUS(...)
11. SUBROUTINE PBOARD(...)
BESTMV
12. REAL*4 FUNCTION RAN(...)
```

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.*
PICKUP
 1. SUBROUTINE PICKUP()
 2. INTEGER*4 PLAYER(1:50, 1:2)
 3. INTEGER*4 PILE(1:50)
 4. SUBROUTINE PBOARD(...)
```

The above command displays all program symbols matching the regular expression.

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

Parameter	Meaning
<process-list>	A list of process objects 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 object.

Description

The `info threads` command displays the status of the specified 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:

 thread count: 1
current thread: 0

 Thread 0: stopped at [0x80001324] PICKUP in pickup.f line 7
 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.

- `current thread` — The thread that is considered to be current by CXdb.
- `thread 0 status` — The status of each active thread. 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 `80001324`.
  - `PICKUP in pickup.f line 7` — The source code location where the thread was stopped. In this case, the thread was stopped in the routine `PICKUP` on `line 7` of the source file `pickup.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: 2
active threads: 0,1
current thread: 1

 Thread 0: stopped at [0x80001378] mthread`doit in mthread.c line 21
 by general process stop

 Thread 1: running
```

In the above example, both thread 0 and thread 1 are active. Thread 0 is stopped, and thread 1 is running.

Related Commands

`event join`
`info cxdb`
`event spawn`
`info process`

Related Concepts

eventpoints

Related Parameters

process-list
thread-list

info trace

Display all tracepoints.

Syntax

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

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 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/*     [0x80001344] PICKUP in pickup.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/*     [0x80001344] PICKUP in pickup.f line 7
#1          y     0/0     0/*     [0x80001348] PICKUP in pickup.f line 8
    {
        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	thread-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

info vregisters

in v

i vr

Display the vector registers.

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 four types of registers in the vector register set:

- Vector merge (VM)
- Vector length (VL)
- Vector stride (VS)
- Vector accumulators (V0 to V7)

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, refer to the *CONVEX Architecture Reference* manual, Chapter 3, "Register Sets."

Examples

The following example illustrates how to display the contents of the vector registers.

```
(CXdb) info vregisters
Process [#0/0]
Vector Merge : 0000000000000000 0000000000000000
Vector Length: 0x8
Vector Stride: 0x40

v0: (000-003) 0000000000000000 0000000000000000 0000000000000000 0000000000000000
                    ...
v0: (124-127) 0000000000000000 0000000000000000 0000000000000000 0000000000000000

v1: (000-003) 0000000000000000 0000000000000000 0000000000000000 0000000000000000
                    ...
v1: (124-127) 0000000000000000 0000000000000000 0000000000000000 0000000000000000
       .
       .
       .
v6: (000-003) 0000000000000000 0000000000000000 0000000000000000 0000000000000000
                    ...
v6: (124-127) 0000000000000000 0000000000000000 0000000000000000 0000000000000000

v7: (000-003) 0000000000000000 0000000000000000 0000000000000000 0000000000000000
                    ...
v7: (124-127) 0000000000000000 0000000000000000 0000000000000000 0000000000000000
```

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 `0000000000000000`. 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 frame`
`info frame at`	`info registers`
`info stack`	`print`

Related Concepts

debugger variables	windows

Related Parameters

process-list	thread-list

info watch

in w

i w

Display all watchpoints.

Syntax

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

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 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
#0         y     0/0    0/0     0x80029010   0x80029013
```

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
#0          y     0/0    0/0      0x80029010   0x80029013
   {
      print $pc;
      resume ;
   }
```

The above command displays information about all existing watchpoints. The commands of the eventpoint handler for watchpoint 0 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
tracepoints

Related Parameters

process-list
thread-list

kill process

ki p
k

Terminate the process.

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 an existing process. It is also used to discard images of core files.

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. No other aspect of the process object is affected. 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`

kill process

Related Concepts process object

Related Parameters process-list

load object

lo

Load CDI 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 CDI data for an object file that has been dynamically loaded into memory. Once the CDI 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, nor 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 CDI 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 CDI 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 CDI 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 obj1.o 0xc0000110 0xc0000558 0x00000000 0x80013000
       00000000
```

The above command loads into memory the CDI data of the object file named `obj1.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

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 after return at [0x800013fe] PICKUP in pickup6.f line 19
Process [#0/0]
Frame : 0; [0x800013fe] PICKUP in pickup6.f line 19
Number of locals : 4
     1 :  TURN = (INTEGER*4) 2
     2 :  ROUND = (INTEGER*4) 7
     3 :  NPLYRS = (INTEGER*4) 3
     4 :  MAXTK = (INTEGER*4) 3
```

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()` 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 after return at [0x80001454] PICKUP in pickup6.f line 23
Process [#0/0]
Frame : 0; [0x800013fe] PICKUP in pickup6.f line 19
Number of locals : 4
     1 :  TURN = (INTEGER*4) 4
     2 :  ROUND = (INTEGER*4) 7
     3 :  NPLYRS = (INTEGER*4) 3
     4 :  MAXTK = (INTEGER*4) 3
```

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:", A, B, C)
CHARACTER*15 'The values are:'
(INTEGER*4) 15
(INTEGER*4) 26
(INTEGER*4) 8
```

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,VAR4,I)
Stepping process [#0/*] by 2 loops
Process [#0/0] stopped after return at [0x80001786] BESTMV in pickup6.f line 69
(INTEGER*4) 32
(INTEGER*4) 7
```

The above example invokes the macro `slp` to step the current process by 2 loops. It then prints the values of the process variables `VAR4` and `I`, which are `32` and `7` respectively.

```
(CXdb) @slp(,VAR4,I)
Stepping process [#0/*] by 1 loop
Process [#0/0] stopped stepping at [0x800019ce] STATUS in pickup6.f line 98
(INTEGER*4) 35
(INTEGER*4) 8
```

The above example invokes the macro `slp` to step the process and print the values of the process variables `VAR4` and `I`. 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) "my_fortran_source_file.f:##LN"
```

The above command creates the macro `FL`. This macro accepts the integer parameter `LN` and concatenates it to the character string `my_fortran_source_file.f:`. The default value for `LN` is `1`. You can use this macro to set breakpoints at various line numbers of the file `my_fortran_source_file.f`, as shown in the following example.

```
(CXdb) break line @FL(22)
#2: break line, on [#0/*], Enabled, ignore 0/0
        [0x800014da] BUILD_ARRAY in my_fortran_source_file.f line 22
```

The above example sets a breakpoint at line 22 of the file `my_fortran_source_file.f`.

Related Commands

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

Related Parameters

string

next

Step to the next source unit, ignoring 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 consider any of the source units inside called subroutines.

Examples

The examples shown below relate to the following FORTRAN source code:

```
1       PROGRAM EXAMPLE
2       PRINT *, "The example program has started."
3       DO I = 1, 10
4         PRINT 99, "I = ", I
5         CALL SUBA(I)
6       ENDDO
7       PRINT *, "The loop for M is next."
8       DO M = 1, 5
9         PRINT 99, "M = ", M
10      ENDDO
11      PRINT 99, "The loop for M is done, with M = ", M
12      PRINT *, "The example program is done."
13   99 FORMAT (A,I2)
14      END
15
16      SUBROUTINE SUBA(N)
17      INTEGER N
18      PRINT 98, "Subroutine SUBA has started. The value of N is ", N
19      DO K = 1, N
20        PRINT 98, "K = ", K
21        IF (K .LE. 5) THEN
22          DO L = 1, N
23            PRINT 98, "L = ", L
24          ENDDO
25          PRINT 98, "The loop for L is done, with L = ", L
26        ENDIF
27      ENDDO
28      PRINT 98, "Subroutine SUBA is done. The value of K is ", K
29      RETURN
30   98 FORMAT (A,I2)
31      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 2.

```
(CXdb) next
Nexting process [#0/*] by 1 statement
Process [#0/0] stopped stepping at [0x80001364] EXAMPLE in example.f line 3
```

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 3.

```
(CXdb) next 2
Nexting process [#0/*] by 2 statements
Process [#0/0] stopped stepping at [0x8000139e] EXAMPLE in example.f line 5
```

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 5, which is a call to a subroutine.

```
(CXdb) next loop
Nexting process [#0/*] by 1 loop
Process [#0/0] stopped stepping at [0x800013ea] EXAMPLE in example.f line 8
```

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 5, which is a subroutine call. The `next` command ignores all source units in a called subroutine. Therefore, the process continues executing until it reaches the next loop after returning from subroutine `SUBA`. When the process stops, the PC points to the beginning of line 8.

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 stepping at [0x80001494] EXAMPLE in example.f line 14
```

The above command steps the process to the next loop. Since there are no new loops after line 8, the process does not stop executing until it reaches the `END` statement on line 14.

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

ni

ni, nexti

Step to the next instruction, ignoring 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 consider any machine instructions in a called routine.

To display the machine instructions for the process, use the `disassemble` command.

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 [0x80001374] EXAMPLE in ex.f line 4
```

The above command steps the current process by one machine instruction.

next instruction

```
(CXdb) next instruction 5
Nexting process [#0/*] by 5 instructions
Process [#0/0] stopped nexting at [0x800013aa] EXAMPLE in ex.f line 6
```

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

no

no

Step from the current source unit of specified granularity to the next source unit of default granularity, ignoring 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 consider the current source unit of specified granularity. It also does not consider 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. Therefore, 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 current source unit of default granularity is used.

Examples

The examples shown below relate to the following FORTRAN source code:

```
1       PROGRAM EXAMPLE
2       PRINT *, "The example program has started."
3       DO I = 1, 10
4         PRINT 99, "I = ", I
5         CALL SUBA(I)
6         PRINT *, "Subroutine SUBA has returned."
7       ENDDO
8       PRINT *, "The loop for M is next."
9       DO M = 1, 5
10        PRINT 99, "M = ", M
11      ENDDO
12      PRINT 99, "The loop for M is done, with M = ", M
13      PRINT *, "The example program is done."
14   99 FORMAT (A,I2)
15      END
16
17      SUBROUTINE SUBA(N)
18      INTEGER N
19      PRINT 98, "Subroutine SUBA has started. The value of N is ", N
20      DO K = 1, N
21        PRINT 98, "K = ", K
22        IF (K .LE. 5) THEN
23          DO L = 1, N
24            PRINT 98, "L = ", L
25          ENDDO
26          PRINT 98, "The loop for L is done, with L = ", L
27        ENDIF
28      ENDDO
29      PRINT 98, "Subroutine SUBA is done. The value of K is ", K
30      RETURN
31   98 FORMAT (A,I2)
32      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 2.

```
(CXdb) next over
Nexting process [#0/*] by 1 statement
Process [#0/0] stopped stepping at [0x80001364] EXAMPLE in example.f line 3
```

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 2 was the current source unit of granularity statement. When execution stops, the PC points to the beginning of line 3, which is the next source unit of statement granularity.

```
(CXdb) next over 3
Nexting process [#0/*] by 3 statements
Process [#0/0] stopped stepping at [0x800013aa] EXAMPLE in example.f line 6
```

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 5 is a call to subroutine `SUBA`. The call was executed, but none of the statements in `SUBA` were counted toward the specified number of 3 because the `next over` command ignores all source units inside called routines. Therefore, when the process stops, the PC points to the beginning of line 6.

```
(CXdb) next over loop
Nexting process [#0/*] by 1 loop
Process [#0/0] stopped stepping at [0x8000136e] EXAMPLE in example.f line 4
```

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 6. There is no current loop source unit at line 6. The `DO` loop that begins on line 3 is active because it contains the current point of execution, but it is not a 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 4.

```
(CXdb) next over block 4
Nexting process [#0/*] by 4 blocks
Process [#0/0] stopped stepping at [0x8000136e] EXAMPLE in example.f line 4
```

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 4, which is the beginning of the block for the `DO` loop on line 3. Thus, line 4 is the current block. Because this block is inside a loop, it repeats. Therefore, the `next over` command keeps encountering this same loop during all four repetitions. The net result is that the above command executes four cycles of the `DO` loop on line 3. When the process stops, the PC points to line 4, 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 4, and the value of `I` is 6:

```
(CXdb) next over block 5
Nexting process [#0/*] by 5 blocks
Process [#0/0] stopped stepping at [0x8000140e] EXAMPLE in example.f line 9
```

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 4, and it is contained within the `DO` loop that starts on line 3. Because the above command repeats five times, it takes 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 9.

Related Commands

`finish`
`info cxdb`
`info line`
`info process`
`info sourceunit`
`next`
`next instruction`
`set default step`
`set step`
`step`
`step instruction`
`step over`

Related Concepts

process object
source units
stepping

Related Parameters

granularity
process-list
thread-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 • `I` — IEEE floating point mode • `N` — Native floating point mode
<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 Z
(INTEGER*4) 5
```

The above command prints the current value of the variable `Z` from the current process. The particular instance of variable `Z` that is referenced here is the one in the current scope. The default format of decimal is used to print the value.

```
(CXdb) print Z+2
(INTEGER*4) 7
```

The above command evaluates the expression `Z+2` and prints the result. The current value of `Z` is not modified.

```
(CXdb) print/B Z+2
(INTEGER*4) 0000 0000 0000 0000 0000 0000 0000 0101
```

The above command evaluates the expression `Z+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$SUB1'Y
(INTEGER*4) 51
```

The above command prints the value of the variable `Y`. Since `Y` is not in the current scope, its scope path is specified.

```
(CXdb) print Z=3
(INTEGER*4) 3
```

The above command assigns the value `3` to the process variable `Z`, then it prints the result. The process uses this new value for `Z` when it resumes execution.

```
(CXdb) print AR
REAL*4(1:5, 1:5)
(1..5,1) :       72.6101     102.2203     131.8304     161.4405     191.0506
(1..5,2) :       65.6101      95.2203     124.8304     154.4405     184.0506
(1..5,3) :       58.6101      88.2203     117.8304     147.4405     177.0506
(1..5,4) :       51.6101      81.2203     110.8304     140.4405     170.0506
(1..5,5) :       44.6101      74.2203       0.0000       0.0000       0.0000
```

The above command prints the elements of array `AR`. The array has 25 elements and is five rows by five columns. (The command `set printopts maxarray` determines how many array elements are printed at one time.)

```
(CXdb) print AR(1..5,2)
REAL*4(1:5, 2:2)
(1..5,2) :       65.6101      95.2203     124.8304     154.4405     184.0506
```

The above command prints the second row of the array `AR`. The subscripts of `AR` in this example follow the syntax for specifying array slices in CXdb.

```
(CXdb) print/e6.3 AR
REAL*4(1:5, 1:5)
(1..5,1) :  0.726E+002 0.102E+003 0.132E+003 0.161E+003 0.191E+003
(1..5,2) :  0.656E+002 0.952E+002 0.125E+003 0.154E+003 0.184E+003
(1..5,3) :  0.586E+002 0.882E+002 0.118E+003 0.147E+003 0.177E+003
(1..5,4) :  0.516E+002 0.812E+002 0.111E+003 0.140E+003 0.170E+003
(1..5,5) :  0.446E+002 0.742E+002 0.000E+000 0.000E+000 0.000E+000
```

The above command also prints array `AR`. 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 AR
REAL*4(1:5, 1:5)
(1..5,1) :   72.610 102.220 131.830 161.441 191.051
(1..5,2) :   65.610 95.220 124.830 154.441 184.051
(1..5,3) :   58.610 88.220 117.830 147.441 177.051
(1..5,4) :   51.610 81.220 110.830 140.441 170.051
(1..5,5) :   44.610 74.220  0.000  0.000  0.000
```

The above command also prints array `AR`. 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=PILE(2)
(INTEGER*4) 43
```

The above command assigns the value of the process variable `PILE(2)` to the debugger variable `X`. It also prints that value.

```
(CXdb) print BESTMV(PILE,3,4,4)
(INTEGER*4) 1
```

The above command evaluates the function `BESTMV` in the current process and prints the value returned by that function. This command executes `BESTMV` 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 `BESTMV` may be triggered by this independent execution from the `print` command.

Related Commands

`evaluate`
`set default format`
`set format`
`set printopts maxarray`
`info formatting`
`set default memory`
`set memory`
`set printopts precision`

Related Concepts

C language expressions
FORTRAN language expressions
scope
debugger variables
language expressions

Related Parameters		
	array-slice	debugger-variable
	language-expression	process-list
	redirection-operator	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.

Examples

The following example shows how to display the setting of the console working directory.

```
(CXdb) pwd
/mnt/jones/project/source
```

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 buffers the last 100 commands entered in the command window.

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.

Examples

The following examples illustrate how to search through the command history and re-execute a previous command.

```
(CXdb) recall print
(CXdb) print I
(INTEGER*4) 12
```

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 I from the current process. Note that CXdb does not ask for confirmation before executing the recalled command.

```
(CXdb) recall 'print $'
(CXdb) print $A=2.5
(REAL*4) 2.50000000
```

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 2.5 to the debugger variable `A`, and then it prints this value.

```
(CXdb) recall ?$A
(CXdb) print $A=2.5
(REAL*4) 2.50000000
```

The above command recalls the most recent command that contains the string `$A` anywhere on the command line. In this case, the recalled command references the debugger variable `A`.

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.

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 viewports from cmderr.

Syntax

remove cmderr *<viewport>* [, ...]

Parameter	Meaning
<viewport>	A file name or the object number of the 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

The `remove cmderr` command removes viewports from cmderr.

Cmderr is the list of viewports, or destinations, that receive all error messages and informational messages generated in response to CXdb commands. A viewport may be either a file or the CXdb command window. The default viewport for cmderr is the command window.

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
errmsgs
/tmp/debug/err_log
myerr.log
```

To remove the file `errmsgs` from the list, enter the following command:

```
(CXdb) remove cmderr errmsgs
New cmderr: Window #1, /tmp/debug/err_log, myerr.log
```

The response to the above command reflects the updated viewport list.

To remove the other files from the viewport list, enter the following command:

```
(CXdb) remove cmderr /tmp/debug/err_log, myerr.log
New cmderr: Window #1
```

The response to the above command indicates that Window #1 (the command window) is the only remaining viewport for cmderr.

Related Commands

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

Related Concepts

cmderr
cmdout
viewports
cmdlog
logging
windows

Related Parameters

redirection-operator
viewport

remove cmdlog

rem cmdl

Delete viewports from cmdlog.

Syntax

`remove cmdlog` *<viewport>* `[, ...]`

Parameter	Meaning
<viewport>	A file name or the object number of the 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

The `remove cmdlog` command removes viewports from cmdlog.

Cmdlog is the list of viewports, or destinations, that receive a log of everything entered in the CXdb command window. A viewport can be either a file or the CXdb command window.

To display 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/local/Smith/input.log
```

remove cmdlog

To remove the file `cxdb_input` from the list, you could enter the following command:

```
(CXdb) remove cmdlog cxdb_input
New cmdlog: log_file, /usr/local/Smith/input.log
```

The response to the above command reflects the updated viewport list.

To remove the other files from the viewport list, enter the following command:

```
(CXdb) remove cmdlog log_file, cxdb_input
New cmdlog:
```

The above response indicates that the cmdlog viewport list is now empty.

Your entries in the command window 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
viewports
windows

Related Parameters

viewport

remove cmdout

rem cmdo

Delete viewports from cmdout.

Syntax

remove cmdout *<viewport>* [, ...]

Parameter	Meaning
<viewport>	A file name or the object number of the 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

The `remove cmdout` command removes viewports from cmdout.

Cmdout is the list of viewports, or destinations, that receive all output generated in response to CXdb commands. A viewport can be either a file or the CXdb command window. The default viewport for cmdout is the command window.

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
/tmp/debug/output_log
myoutput.log
```

To remove the file `output_data` from the list, enter the following command:

```
(CXdb) remove cmdout output_data
New cmdout: Window #1, /tmp/debug/output_log, myoutput.log
```

The response to the above command reflects the updated viewport list.

To remove the other files from the list, enter the following command.

```
(CXdb) remove cmdout /tmp/debug/output_log, myoutput.log
New cmdout: Window #1
```

The response to the above command indicates that Window #1 (the 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
viewports	windows

Related Parameters

redirection-operator

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 LESS , PAGER
```

In the above command, the two environment variables `LESS` and `PAGER` are removed from the 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.

Examples

The following examples remove directories from the default search path.

```
(CXdb) remove default path /mnt/jones/project/libraries
Default search path:
        .
        /mnt/jones/libraries
        /mnt/jones/math/libraries
```

The above command removes the `/mnt/jones/project/libraries` directory from the default search path. When CXdb creates a new search path for a new process object, it will not include the `/mnt/jones/project/libraries` directory.

You can remove multiple directories with a single command by separating them with a comma.

remove default path

```
(CXdb) remove default path /mnt/jones/libraries,/mnt/jones/math/libraries
Default search path:
        .
```

The above command removes the `/mnt/jones/libraries` and `/mnt/jones/math/libraries` directories from the default search path.

Related Commands

`add default path`
`add path`
`info cxdb`
`info process`
`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 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.

You can remove multiple environment variables with a single command by separating them with a comma.

```
(CXdb) remove environment PAGER , LESS
```

The above command removes the variables `PAGER` and `LESS` 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.

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 4 removed
Eventpoint 5 removed
```

The above command removes all existing eventpoints. CXdb displays which eventpoints are removed.

Related Commands

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

Related Concepts

breakpoints
eventpoint handlers
watchpoints
eventpoints
tracepoints

Related Parameters

event-specifier

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.
<event-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 2 removed
```

The above command removes all tracepoints. In this case only eventpoint 2 is a tracepoint. Eventpoint 2 no longer exists and cannot be referenced in subsequent CXdb commands.

```
(CXdb) remove eventtype watch, break
Eventpoint 1 removed
Eventpoint 3 removed
```

The above command removes all watchpoints and breakpoints. Neither eventpoint 1 or 3 can be referenced in subsequent CXdb commands.

```
(CXdb) remove eventtype *
Eventpoint 0 removed
Eventpoint 4 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
eventpoints
eventpoint handlers
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 SS
```

The above command deletes the macro named `SS`.

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

Examples

The following examples remove directories from the search path.

```
(CXdb) remove path /mnt/jones/project/libraries
Search path:
        .
        /mnt/jones/libraries
        /mnt/jones/math/libraries
```

The above command removes the `/mnt/jones/project/libraries` directory from the search path for the current process.

You can specify multiple directories in a single command by separating them with a comma.

```
(CXdb) remove path /mnt/jones/libraries , /mnt/jones/math/libraries
Search path:
        .
```

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 cxdb`
`info process`
`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

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 trace1
```

The above command removes the debugger variable `trace1.` 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
command files
debugger variables
eventpoints
tracepoints
watchpoints

Related Parameters debugger-variable

rerun

rer

rr

Start execution of the 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 file of the process object and then begins process 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
Beginning execution of Process [#0]
```

The above command creates a new process from the executable file 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 [#7] backgrounded

Process [#0] is already running with pid 27660.
Terminate existing process and restart? y

Beginning execution of Process [#0]
(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	windows

Related Parameters

process-list

Continue execution of the process from within an eventpoint handler.

Syntax

```
resume
```

Description

The `resume` command resumes process execution from within an eventpoint handler. You can only use the `resume` command from within 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 `resume` command is the only process execution command that can be used inside an eventpoint handler. It replaces the following process execution commands:

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

If a count is specified with a stepping command, the `resume` command continues stepping with the appropriate count remaining.

Examples

The following examples illustrate the use of the `resume` command in an eventpoint handler.

```
(CXdb) break line 35 {print i; resume;}
Breakpoint 0, [0x80001234] PRIME in prime.f line 35
```

The above command sets an eventpoint handler for breakpoint 0. The handler prints the value of `i` then resumes process execution. The following example demonstrates the effect of this handler.

```
(CXdb) step expression 100
Stepping process [#0] by 100 expressions
INTEGER*4 25
Resuming execution of process [#0]
```

The above command begins a new process without any arguments passed to it. Breakpoint 0 is triggered, causing the eventpoint handler to be executed. The handler prints the value of `i`, 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 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 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 X+Y
Make function return now?
```

The above command evaluates the expression `X+Y` 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 array_A
Make function return now? n
```

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

Start execution of the process.

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process object.
<string>	A string to be passed as an argument list to the process shell.
`&`	Runs the command in the background.

Description

The `run` command creates a process from the executable file 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.

A specified string, called an argument list, contains the arguments to be passed to the process shell. 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.

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"
Beginning execution of Process [#0]
```

The above command creates a new process from the executable file of the current process object. The string "< data" is passed to the process shell, causing standard input to come from the data file. Because the process is run from the process working directory, the process working directory is used as the base directory for the relative path names. The string "< data" also becomes the argument list for the current process object.

```
(CXdb) run &
Process [#0] is already running with pid 17540.
Terminate existing process and restart? y

Beginning execution of Process [#0]
```

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 string 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
Beginning execution of Process [#0]
```

The above command creates a new process. The process shell is passed the string "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 data file.

The > cmdout.log redirects the output of this command to be sent to the file cmdout.log. This is not passed to the process shell because it is not enclosed in quotes.

Related Commands

`attach`	`continue`
`core`	`debug core`
`debug exec`	`debug proc`
`detach`	`executable`
`info cxdb`	`info process`
`kill process`	`rerun`
`set default pshell`	`set directory`
`set pshell`	`stop`

Related Concepts

background execution	process object
process working directory	windows

Related Parameters

process-list	string

set cmderr

se cmde

Clear and redefine the viewport list for cmderr.

Syntax

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

Parameter	Meaning
<viewport>	A file name or the object number of the 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

The `set cmderr` command deletes the current list of viewports for cmderr and replaces it with the specified list.

Cmderr is the list of viewports, or destinations, that receive all error messages and informational messages generated in response to CXdb commands. A viewport may be either a file or the CXdb command window. The default viewport for cmderr is the command window.

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

To display the current 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: errmsgs
```

The above command deletes the current viewport list for cmderr and replaces it with a new list that contains only one entry. That entry is the file `errmsgs,` which is in the console working directory. This command also removes Window #1 (the command window) from the viewport list, so CXdb messages will not be sent to the command window.

```
(CXdb) set cmderr 1, /tmp/debug/err_log, myerr.log
New cmderr: Window #1, /tmp/debug/err_log, myerr.log
```

The above command deletes the current viewport list for cmderr and replaces it with a new list that contains three entries. The entries are `Window #1` (the command window), the file `err_log` in the directory `/tmp/debug,` and the file `myerr.log` in the console working directory.

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
viewports
windows

Related Parameters

redirection-operator
viewport

set cmdlog

se cmdl

Clear and redefine the viewport list for cmdlog.

Syntax

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

Parameter	Meaning
<viewport>	A file name or the object number of the 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

The `set cmdlog` command deletes the current list of viewports for cmdlog and replaces it with the specified list.

Cmdlog is the list of viewports, or destinations, that receive a log of everything entered in the CXdb command window. The `set logging` command enables logging to the viewports for cmdlog, and the `clear logging` command disables it. The default is logging disabled (clear).

A viewport can be either a file or the CXdb command window. For a viewport that is a file, CXdb creates the file if it does not exist and overwrites the file if it does exist. To prevent overwriting of an existing file, use the `set noclobber` command.

To display the setting for log and the current viewports for cmdlog, use the command `info cxdb`.

Your entries in the command window are always automatically echoed, regardless of the settings for cmdlog and log. Therefore, there is no need to add the command window to the viewport list for cmdlog. In fact, doing so causes each of your entries to appear twice in the command window.

Examples

The following examples illustrate how to reset the viewport list for cmdlog.

```
(CXdb) set cmdlog log_file
New cmdlog: log_file
```

The above command deletes the current viewport list for cmdlog and replaces it with a new list that contains only one entry. That entry is the file named `log_file`, which is in the console working directory in this case.

```
(CXdb) set cmdlog cxdb_log, /usr/local/Smith/input.log
New cmdlog: cxdb_log, /usr/local/Smith/input.log
```

The above command deletes the current viewport list for cmdlog and replaces it with a new list that contains two entries. The new entries are the file `cxdb_log` in the console working directory and the file `input.log` in the directory `/usr/local/Smith`.

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
viewports
windows

Related Parameters

viewport

set cmdout

se cmdo

Clear and redefine the viewport list for cmdout.

Syntax

set cmdout *<viewport>* [, ...]

Parameter	Meaning
<viewport>	A file name or the object number of the 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

The `set cmdout` command deletes the current list of viewports for cmdout and replaces it with the specified list.

Cmdout is the list of viewports, or destinations, that receive the normal output generated in response to CXdb commands. A viewport can be either a file or the CXdb command window. The default viewport for cmdout is the command window.

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

To display 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 output_data
New cmdout: output_data
```

The above command deletes the current viewport list for cmdout and replaces it with a new list that contains only one entry. That entry is the file `output_data`, which is in the console working directory. This command also removes `Window #1` (the command window) from the viewport list, so CXdb output will not be sent to the command window in this case.

```
(CXdb) set cmdout 1, /tmp/debug/output_log, myoutput.log
New cmdout: Window #1, /tmp/debug/output_log, myoutput.log
```

The above command deletes the current viewport list for cmdout and replaces it with a new list that contains three entries. The entries are `Window #1` (the command window), the file `output_log` in the directory `/tmp/debug`, and the file `myoutput.log` in the console working directory.

Related Commands

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

Related Concepts

cmderr
cmdout
viewports
cmdlog
logging
windows

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.

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.

set default environment

```
(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

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.

Because of the additional system overhead involved with fixed scheduling, it is recommended for debugging multi-threaded 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
```

set default fixed sched

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.

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 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
```

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` sets the default mode for floating point operations to IEEE, native, or dual.

The initial setting is dual. The default floating point mode is given 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 explicitly set 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 eventpoint handler to become the default handler.

Description

The `set default handler` sets the default eventpoint handler.

The default eventpoint handler is used by all eventpoints that have not had an eventpoint handler 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.

Examples

The following example shows how to set the default handler.

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

Related Concepts

breakpoints
eventpoint handlers
watchpoints
eventpoints
tracepoints

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

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
```

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 become 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.

Examples

The following examples illustrate how to set the default search path.

```
(CXdb) set default path /mnt/jones/project
Default search path:
       /mnt/jones/project
```

The above command clears the current setting of the default search and then adds the `/mnt/jones/project` directory to the empty default search path. This command can be placed in an initialization file to create a default search path automatically.

You can set the default search path to multiple directories with a single command by separating them with a comma.

set default path

```
(CXdb) set default path /mnt/jones/libraries , math/libraries
Default search path:
       /mnt/jones/libraries
       /mnt/jones/math/libraries
```

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
Default search path:
```

The above command clears the default search path and then sets it to reflect the current console working directory. If the console working directory changes, the default search path reflects the new console working directory.

Related Commands

add default path
add path
info cxdb
info process
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.

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

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, 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
```

set default step

Related Concepts	source units	stepping
Related Parameters	granularity	

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 become the process working directory.

Description

The `set directory` command sets the process working directory.

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

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 be the `/mnt/jones/project` directory. The next process that is created will be run from the `/mnt/jones/project` directory.

```
(CXdb) set directory $PROJ2DIR
```

The above command sets the process working directory to the value of the `$PROJ2DIR` environment variable. The environment variable is expanded before the process working directory is set.

set directory

Related Commands	`cd` `pwd`	`info process`
Related Concepts	console working directory process object search path	default search path process working directory
Related Parameters	directory-specifier	process-list

set echo

se ec

Enable echoing of input from command files.

Syntax

`set echo`

Description

The `set echo` command enables echoing.

With echoing enabled, commands executed from command files are echoed in the command window. You can disable echoing by using the `clear echo` command.

By default echoing is disabled.

Examples

The following example turns echoing on.

```
(CXdb) set echo
Echoing is now turned on.
```

The above command enables echoing. When the `source` command is used, the commands executed are echoed in the command window.

Related Commands

`clear echo` `echo`

Related Concepts

cmdlog
logging
command files

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

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.

set evalopts fpmode

```
(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
language expressions
FORTRAN 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)

Single precision (4-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
```

set evalopts rprecision

Related Concepts	C language expressions language expressions	FORTRAN language expressions
Related Parameters	language-expression	

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.

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.

Because of the additional system overhead involved with fixed scheduling, it is recommended for debugging multi-threaded processes only. The default is fixed scheduling disabled.

Examples

The following example shows how to enable fixed scheduling.

```
(CXdb) set fixed sched
```

The above command enables fixed scheduling for the current process.

Related Commands

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

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 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

Examples

The following examples illustrate how to set the memory display formats for specific threads of the current process.

```
(CXdb) set format byte decimal
```

The above command selects the decimal format for displaying bytes of memory. This command applies the format to all threads of the current process.

```
(CXdb) set format word unsigned
```

The above command selects the unsigned decimal format for displaying words of memory. This command applies the format to all threads of the current process.

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

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. Another `set fpmode` command must be issued 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

```
info cxdb                    info process
set evalopts fpmode          set default fpmode
```

set 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 to be 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.

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 is reached that has an ignore count, 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.

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 3,4,5
Eventpoint 3 will be ignored 0 times
Eventpoint 4 will be ignored 0 times
Eventpoint 5 will be ignored 0 times
```

The above command sets an ignore count of `0` for eventpoints 3, 4, and 5. 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
eventpoints
eventpoint handlers
tracepoints
watchpoints

Related Parameters

event-specifier

set logging

se l

Enable logging for cmdlog.

Syntax

```
set logging
```

Description

The `set logging` command enables logging to the viewports for cmdlog.

Cmdlog is a list of viewports, or destinations, that receive a log of everything entered in the CXdb command window. When logging is enabled, everything entered in the command window 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 (off).

A viewport can be either a file or the command window. For a viewport that is a file, CXdb creates the file if it does not exist and overwrites the file if it does exist. To prevent overwriting of an existing file, use the `set noclobber` command.

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 sets the default type of memory unit used to display the contents of memory for specified threads and processes. 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

Each type of memory unit has its own display format. This format can be set with the `set format` 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 to thread 2 of the current process.

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

Enable the noclobber option for all viewports.

Syntax

```
set noclobber
```

Description

The `set noclobber` command enables the noclobber option.

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
```

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 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 set the noclobber option.

```
(CXdb) set noclobber
```

The above command enables the noclobber option for all cmderr, cmdlog, and cmdout viewports.

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
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.

Examples

The following examples set the search path for the current process object.

```
(CXdb) set path /mnt/jones/project
Search path:
        /mnt/jones/project
```

The above command clears the current setting of the search path for the current process and then adds the `/mnt/jones/project` directory to the empty search path. When CXdb now searches for a source file, it will look in the `/mnt/jones/project` directory. This command can be placed in a command file to automate the setting of the search path.

You can set the search path to multiple directories with a single command by separating them with a comma.

set path

```
(CXdb) set path /mnt/jones/libraries , math/libraries
Search path:
       /mnt/jones/libraries
       /mnt/jones/math/libraries
```

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 .
Search 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 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 pr 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.

Related Commands

`info formatting`
`print`
`set printopts nopadding`
`set printopts padding`
`set printopts precision`

set printopts nopadding

se pr 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. 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 `AR`. With padding disabled, printing the array produces the following result:

```
(CXdb) print AR
INTEGER*4(1:5, 1:5)
(1..5,1) :  1 2 3 4 5
(1..5,2) :  2 4 6 8 10
(1..5,3) :  3 6 9 12 15
(1..5,4) :  4 8 12 16 20
...
```

In the above example, the array elements are not properly aligned because padding is disabled.

set printopts nopadding

If padding is enabled, the same array prints as follows:

```
(CXdb) print AR
INTEGER*4(1:5, 1:5)
(1..5,1) :    000000001 000000002 000000003 000000004 000000005
(1..5,2) :    000000002 000000004 000000006 000000008 000000010
(1..5,3) :    000000003 000000006 000000009 000000012 000000015
(1..5,4) :    000000004 000000008 000000012 000000016 000000020
...
```

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 pr 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 `AR`. With padding enabled, printing the array produces the following result:

```
(CXdb) print AR
INTEGER*4(1:5, 1:5)
(1..5,1) :    000000001 000000002 000000003 000000004 000000005
(1..5,2) :    000000002 000000004 000000006 000000008 000000010
(1..5,3) :    000000003 000000006 000000009 000000012 000000015
(1..5,4) :    000000004 000000008 000000012 000000016 000000020
...
```

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 AR
INTEGER*4(1:5, 1:5)
(1..5,1) :  1 2 3 4 5
(1..5,2) :  2 4 6 8 10
(1..5,3) :  3 6 9 12 15
(1..5,4) :  4 8 12 16 20
...
```

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 pr 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.

Assume that your program contains the variable `AVG`, which has a current value of 123.45678, and you issue the following command:

```
(CXdb) print AVG
REAL*4 123.46
```

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
```

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.

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

set seq

se se

Set the 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.

For more information about the PSW and the SEQ bit, refer to the *CONVEX Architecture Reference*, Chapter 3, "Register Sets."

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`
`info psw`
`set sqs`
`clear sqs`
`set fixed sched`

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 | COVUE}

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.
`COVUE` — The CONVEX COVUE shell.

Initially the shell type is the type from which CXdb was invoked.

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`	`info process`
`set pshell`	`shell`

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.

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`". 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.

The default actions for all signals can be displayed by using the `info signal` command before changing any signal's actions.

Signals names are not case sensitive.

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`
`print`
`signal thread`
`info signal`
`signal process`

Related Concepts

signals

Related Parameters

process-list
signal-specifier

set sqs

se sq

Set the 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.

For more information about the PSW and the SQS bit, refer to the *CONVEX Architecture Reference,* Chapter 3, "Register Sets."

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*>] [<*thread-list*>] **set step** <*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 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 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
thread-list	

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.

set typehandler

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. 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
eventpoints
tracepoints
debugger variables
eventpoint handlers
watchpoints

Related Parameters

event-handler
eventtype-specifier

shell

sh

Invoke a shell.

Syntax

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

Parameter	Meaning
<shell-specifier>	A shell type to be used rather than the current shell type. The possible shell types are: sh — The Bourne shell ksh — The Korn shell csh — The c shell tcsh — The tc shell COVUE — The CONVEX COVUE 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.

With the CXwindows 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.

shell

```
(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 project/source; ls"
```

The above command opens a Korn shell. The string `"cd project/source; 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.

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.
&	Runs the command in the background.

Description

The `signal process` command sends the specified signal to the 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 the `signal process` command is issued. The signal may be received by any existing thread of the process. Because the signal is generated by CXdb, CXdb does not catch the signal.

The `signal process` command can be used to control which signal is sent to your process.

Signal names are not case sensitive.

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 can be used instead of the signal name. By using the `&` flag, this command is placed into 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.

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.
&	Runs the command in the background.

Description

The `signal thread` command sends the specified signal to the specified thread of the process.

This 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. If no thread is specified, the signal may be received by any of the threads of the process. 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.

Signal names are not case sensitive.

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 may be used instead of the signal name. Thread `1` receives the signal and continues execution. By using the `&` flag, this command is placed into 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

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.

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 `aliases.cxdb` 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, enter the following command:

```
(CXdb) source aliases.cxdb
(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 `aliases.cxdb.` 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`
`info cxdb`
`set logging`
`clear echo`
`set echo`

Related Concepts

command files
cmdlog
initialization files
viewports
cmderr
cmdout
logging

Related Parameters

file-name

step

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:

```
1       PROGRAM EXAMPLE
2       PRINT *, "The example program has started."
3       DO I = 1, 10
4         PRINT 99, "I = ", I
5         CALL SUBA(I)
6         PRINT *, "Subroutine SUBA has returned."
7       ENDDO
8       PRINT *, "The loop for M is next."
9       DO M = 1, 5
10        PRINT 99, "M = ", M
11      ENDDO
12      PRINT 99, "The loop for M is done, with M = ", M
13      PRINT *, "The example program is done."
14   99 FORMAT (A,I2)
15      END
16
17      SUBROUTINE SUBA(N)
18      INTEGER N
19      PRINT 98, "Subroutine SUBA has started. The value of N is ", N
20      DO K = 1, N
21        PRINT 98, "K = ", K
22        IF (K .LE. 5) THEN
23          DO L = 1, N
24            PRINT 98, "L = ", L
25          ENDDO
26          PRINT 98, "The loop for L is done, with L = ", L
27        ENDIF
28      ENDDO
29      PRINT 98, "Subroutine SUBA is done. The value of K is ", K
30      RETURN
31   98 FORMAT (A,I2)
32      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 2.

```
(CXdb) step
Stepping process [#0/*] by 1 statement
Process [#0/0] stopped stepping at [0x80001364] EXAMPLE in example.f line 3
```

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 3.

```
(CXdb) step 4
Stepping process [#0/*] by 4 statements
Process [#0/0] stopped stepping at [0x80001518] SUBA in example.f line 20
```

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 3. The second statement source unit executed is the print statement on line 4. The third statement source unit executed is line 5, which is a call to subroutine `SUBA`. Therefore, the fourth statement source unit executed is line 19 in `SUBA`. When the process stops, the PC points to the beginning of line 20 in `SUBA`.

```
(CXdb) step loop
Stepping process [#0/*] by 1 loop
Process [#0/0] stopped stepping at [0x80001588] SUBA in example.f line 23
```

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 23.

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

step 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 disassembly 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 [0x800014cc] SUBA in ex.f line 19
```

The above command steps the current process by one machine instruction.

step instruction

```
(CXdb) step instruction 5
Stepping process [#0/*] by 5 instructions
Process [#0/0] stopped stepping at [0x8000ad24] _for$s_wsfe+0x12
```

The above command steps the current process by five machine instructions.

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 does not look at 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. Therefore, 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 current source unit of default granularity is used.

Examples

The examples shown below relate to the following FORTRAN source code:

```
1       PROGRAM EXAMPLE
2       PRINT *, "The example program has started."
3       DO I = 1, 10
4         PRINT 99, "I = ", I
5         CALL SUBA(I)
6         PRINT *, "Subroutine SUBA has returned."
7       ENDDO
8       PRINT *, "The loop for M is next."
9       DO M = 1, 5
10        PRINT 99, "M = ", M
11      ENDDO
12      PRINT 99, "The loop for M is done, with M = ", M
13      PRINT *, "The example program is done."
14   99 FORMAT (A,I2)
15      END
16
17      SUBROUTINE SUBA(N)
18      INTEGER N
19      PRINT 98, "Subroutine SUBA has started. The value of N is ", N
20      DO K = 1, N
21        PRINT 98, "K = ", K
22        IF (K .LE. 5) THEN
23          DO L = 1, N
24            PRINT 98, "L = ", L
25          ENDDO
26          PRINT 98, "The loop for L is done, with L = ", L
27        ENDIF
28      ENDDO
29      PRINT 98, "Subroutine SUBA is done. The value of K is ", K
30      RETURN
31   98 FORMAT (A,I2)
32      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 2.

```
(CXdb) step over
Stepping process [#0/*] by 1 statement
Process [#0/0] stopped stepping at [0x80001364] EXAMPLE in example.f line 3
```

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 2 was the current source unit of statement granularity. When execution stops, the PC is pointing to the beginning of line 3, which is the next source unit of statement granularity.

```
(CXdb) step over 3
Stepping process [#0/*] by 3 statements
Process [#0/0] stopped stepping at [0x800014c6] SUBA in example.f line 19
```

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 5 is a call to subroutine `SUBA`. This call is executed as one of the three repetitions of the command. Therefore, when the process stops, the PC is pointing to the beginning of line 19 in `SUBA`.

```
(CXdb) step over routine
Stepping process [#0/*] by 1 loop
Process [#0/0] stopped stepping at [0x80001518] SUBA in example.f line 20
```

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 19 in subroutine `SUBA`. There is no current routine source unit at line 19. Subroutine `SUBA` is active because it contains the current point of execution, but it is not the current routine because the starting address of `SUBA` 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 only one statement, and the PC now points to line 20.

```
(CXdb) step over loop
Stepping process [#0/*] by 1 loop
Process [#0/0] stopped stepping at [0x80001668] SUBA in example.f line 29
```

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 19 and ends on line 28. Therefore, when the process stops, the PC points to the beginning of line 29.

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 be currently 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.

To stop process execution, type 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 the background.

Related Commands

`continue`
`rerun`
`run`
`quit`
`signal process`
`signal thread`

Related Concepts

background execution
process object
windows

Related Parameters

process-list

trace instruction

`ti`

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 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 semi-colon (`;`).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

The `trace instruction` command sets a tracepoint at the start of the specified instruction address.

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

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 BESTMV

#0: trace instruction, on [#0/*], Enabled, ignore 0/0
      [0x800015f0] BESTMV in pickup.f line 55
```

The above command sets a tracepoint at the first instruction of the routine `BESTMV`. The evaluation of the language expression `BESTMV` 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 executing the `info event` command on the new tracepoint. The output is explained below:

- `#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.
- `[0x800015f0]` — The hexadecimal address location of the tracepoint. In this case the address is `800015f0`.
- `BESTMV in pickup.f line 55` — The symbolic location of the tracepoint. In this case the tracepoint is in the routine `BESTMV` at line 55 of the source file `pickup.f`.

When the tracepoint is triggered, execution is stopped before the instruction at this 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 '800015f0'x

#1: trace instruction, on [#0/*], Enabled, ignore 0/0
        [0x800015f0] BESTMV in pickup.f line 55.
```

The above command sets a tracepoint at the absolute address `800015f0`. The tracepoint number is `1`, located at address `800015f0` in routine `BESTMV` at line `55` of the file `pickup.f`. The notation `'800015c8'x` is FORTRAN-specific and indicates that the address is in hexadecimal notation.

Using C syntax:

```
(CXdb) trace instruction 0x800015f0

#1: trace instruction, on [#0/*], Enabled, ignore 0/0
        [0x800015f0] pickup`bestmv in pickup.c line 55.
```

The above command sets a tracepoint at the absolute address `800015f0`. The `0x` is the C notation for a hexadecimal number. The symbolic location uses the scope path of `` pickup`bestmv `` to indicate the source file and routine 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 need to 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 BESTMV {echo 'routine BESTMV reached';}

#2: trace instruction, on [#0/*], Enabled, ignore 0/0
        [0x800015f0] BESTMV in pickup.f line 55.
    {
       echo 'routine BESTMV reached';
    }
```

The above command sets a tracepoint at address `800015f0`, the starting address of routine `BESTMV`. 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 '80001234'x \; $Trace4

#4: trace instruction, on [#0/*], Enabled, ignore 0/0
        [0x80001234] MAIN in pickup.f line 25.
```

The above command creates a new tracepoint at the absolute address `80001234`. The `\;` 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

`tl`

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 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 semi-colon (`;`).
<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 a specific source lines.

```
(CXdb) trace line 18

#0: trace line, on [#0/*], Enabled, ignore 0/0
       [0x800013c4] PICKUP in pickup.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 executing the `info event` command on the new tracepoint. The output is explained below:

- `#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.
- `[0x800013c4]` — The hexadecimal address location of the tracepoint. In this case the address is `800013c4`.
- `PICKUP in pickup.f line 18` — The symbolic location of the tracepoint. In this case the tracepoint is in the routine `PICKUP` at line 18 of the source file `pickup.f`.

When the tracepoint is triggered, execution is stopped 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 is then resumed.

```
(CXdb) trace line pickup2.f:30

#1: trace line, on [#0/*], Enabled, ignore 0/0
       [0x80001234] SUB1 in pickup2.f line 30
```

The above command sets a tracepoint at the starting address of line `30` of the source file `pickup2.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) trace line 18 {echo 'Line 18 reached';}

#2: trace line, on [#0/*], Enabled, ignore 0/0
        [0x800013c4] PICKUP in pickup.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
        [0x800013c4] PICKUP in pickup.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

tr

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 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 semi-colon (`;`).
<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 units of routines.

```
(CXdb) trace routine BESTMV

#0: trace routine, on [#0/*], Enabled, ignore 0/0
        [0x800015f2] BESTMV in pickup.f line 59
```

The above command sets a tracepoint at the first executable source unit of the routine `BESTMV`.

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

- `#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.
- `[0x800015f2]` —The hexadecimal address location of the tracepoint. In this case the address is `800015f2`.
- `BESTMV in pickup.f line 59` —The symbolic location of the tracepoint. In this case the tracepoint is in the routine `BESTMV` at line 59 of the source file `pickup.f`.

When the tracepoint is triggered, execution is stopped 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 is resumed.

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 '800015f8'x

#1: trace routine, on [#0/*], Enabled, ignore 0/0
       [0x800015f2] BESTMV in pickup.f line 59
```

The above command sets a tracepoint at the starting address of the routine that contains the absolute address `800015f8`. The tracepoint number is `1`, located at address `800015f2` in routine `BESTMV` at line `59` of the file `pickup.f`. The notation `'800015f8'x` is FORTRAN specific and indicates that the address is in hexadecimal notation.

Using C syntax:

```
(CXdb) trace routine 0x800015f8

#1: trace routine, on [#0/*], Enabled, ignore 0/0
       [0x800015f2] pickup`bestmv in pickup.c line 59
```

The above command sets a tracepoint at the starting address that contains the absolute address `800015f8`. The `0x` is the C notation for a hexadecimal number. The symbolic location uses the scope path of `pickup`bestmv` to indicate the program and routine in which the tracepoint is located.

```
(CXdb) trace routine BESTMV {echo 'routine BESTMV reached';}

#2: trace routine, on [#0/*], Enabled, ignore 0/0
       [0x800015f2] BESTMV in pickup.f line 59
    {
       echo 'routine BESTMV reached';
    }
```

The above command sets a tracepoint at the address of the first executable source unit of the routine `BESTMV`. An eventpoint handler is defined for the tracepoint. When the tracepoint is triggered, execution is stopped and the `echo` command is executed. Even though the eventpoint is a tracepoint, execution is not resumed because this eventpoint's handler overrides the default handler for tracepoints.

```
(CXdb) trace routine '80001234'x \; $Trace4

#4: trace routine, on [#0/*], Enabled, ignore 0/0
       [0x80001234] MAIN in pickup.f line 25.
```

The above command creates a new tracepoint at the first executable source unit of the routine containing the absolute address `80001234`. The `\;` 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 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 semi-colon (;).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

The `trace source` command sets a tracepoint 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 more information about the source unit that a source unit number corresponds to 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 30

#0: trace source, on [#0/*], Enabled, ignore 0/0
       [0x80001394] PICKUP in pickup.f line 14
```

The above command sets a tracepoint at the starting address of source unit `30` of the current source file.

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

- `#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.
- `[0x80001394]` — The hexadecimal address location of the tracepoint. In this case the address is `80001394`.
- `PICKUP in pickup.f line 14` — The symbolic location of the tracepoint. In this case the tracepoint is in the routine `PICKUP` at line 14 of the source file `pickup.f`.

When the tracepoint is triggered, execution is stopped 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 is resumed.

```
(CXdb) trace source pickup2.f:300

#1: trace source, on [#0/*], Enabled, ignore 0/0
       [0x80001234] SUB1 in pickup2.f line 30
```

The above command sets a tracepoint at the starting address of source unit `300` of the source file `pickup2.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) trace source 30 {echo 'Source unit 30 reached';}

#2: trace source, on [#0/*], Enabled, ignore 0/0
        [0x80001394] PICKUP in pickup.f line 14
    {
       echo 'Source unit 30 reached';
    }
```

The above command sets a tracepoint at the starting address of source unit 30 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 30 $Trace3

#3: trace source, on [#0/*], Enabled, ignore 0/0
        [0x80001394] PICKUP in pickup.f line 14
```

The above command sets a tracepoint at the starting address of source unit 30 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 describing 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 semi-colon (`;`).
<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 statement, CXdb tests to see if the value stored at the watched address has a value different from the value stored prior to the execution of that source unit. 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. Unless the watchpoint handler includes the `resume` command, execution is not restarted.

Watchpoints are triggered when the address being watched 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.

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 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 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(A)
#1: watch 0x80051008..0x80051197, on [#0/*], Enabled, ignore 0/0
```

The above command sets a watchpoint to watch the address of the FORTRAN array `A.` The FORTRAN function `loc()` provides the address of the variable.

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

- `#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.
- `0x80051008..0x80051197` — 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.

When any value stored in the array `A` changes, the watchpoint is triggered.

```
(CXdb) watch &count

#1: watch 0xffffc6d4..0xffffc6d7, 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 `count`. 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 `count` 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

#2: watch 0x80001234..0x80001237, on [#0/0], Enabled, ignore 0/0
```

The above command uses the `:` notation to specify an address range. 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

#2: watch 0x80001234..0x80001237, on [#0/0], Enabled, ignore 0/0
```

The above command watches four bytes starting with address `80002345`. The notation `0x80002345` 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

#3: 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;}

#4: 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, then process execution resumes.

```
(CXdb) watch loc(A) \; $w1

#5: watch 0x80051008..0x80051197, on [#0/0], Enabled, ignore 0/0
```

The above command watches the array `A`. The `\;` is needed to separate the language expression from the debugger variable. A debugger variable has been assigned to the watchpoint. In future 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

Concepts

2

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

- **Description** — Text explaining the concept.
- **Examples** — One or more examples illustrating the concept, where applicable.
- **Related Commands** — A list of CXdb commands related to the concept. The commands are described more fully in Chapter 1 of this manual.
- **Related Concepts** — A list of other concepts related to the concept being described. The related concepts are also described in this chapter.
- **Related Parameters** — A list of command parameters related to the concept. CXdb parameters are described more fully in Chapter 3 of this manual.

For more details about the concepts described in this chapter, refer to the *CONVEX CXdb Concepts* manual and the *CONVEX CXdb User's Guide.*

background execution

Description

CXdb process execution commands can be run in the background.

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

With background execution, your process executes while you continue to use CXdb. To run a command in the background, follow the command with the ampersand (`&`) on the command line. When a command is run in the background, the CXdb command prompt returns, allowing you to enter other CXdb commands. While a command is running in the background you cannot use any CXdb commands that require the process to be stopped. The command runs in the background until the command is completed or process execution is stopped. Only one command may run in the background at a time.

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

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

To move one of the above commands to the background after the command has begun execution you can type CTRL-C in the command window. This does not stop the process.

Once a process execution command is running in the background you can stop the process with the `stop` command. When the process is stopped (or terminates) the command in the background is completed. CXdb displays a message when the command finishes executing.

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

Examples

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

```
(CXdb) run &
Command [#3] backgrounded

Beginning execution of process [#0]
(CXdb)
```

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

```
(CXdb) step statement 1000 &
Command [#45] backgrounded

Stepping process [#0] by statement 1000 times
Command [#45] completed.
(CXdb)
```

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

Related Commands

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

Related Concepts

stepping
windows

breakpoints

Description

A breakpoint is a pre-defined eventpoint, or trap, that you place in your executable code. When process execution reaches the location of an enabled breakpoint, the set of actions associated with the breakpoint, called the eventpoint handler, are taken. Breakpoints enable you to stop the execution of your process at key locations in your program.

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

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

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

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

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

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

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

There are several different ways to set a breakpoint. The different commands are described below:

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

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

Several commands allow you to interact with existing breakpoints. These commands are described below:

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

Examples

The following series of examples create and manipulate several different breakpoints in one process object.

```
(CXdb) break line 60

#0: break line, on [#0/*], Enabled, ignore 0/0
        [0x80001620] BESTMV in pickup.f line 60
```

The above command sets a breakpoint at line 60 in the current source file. The eventpoint number for this breakpoint is 0 and the address of the breakpoint is `80001620` in routine `BESTMV` at line 60 in the source file `pickup.f`.

```
(CXdb) break routine BESTMV

#1: break routine, on [#0/*], Enabled, ignore 0/0
        [0x800015f2] BESTMV in pickup.f line 59
```

The above command sets a breakpoint at the first executable source unit in the routine containing the address of the routine `BESTMV`. Obviously the routine `BESTMV` contains its own address, so the breakpoint is set at the first executable source unit of the routine `BESTMV`. The first executable source unit of a routine is usually the first statement of a routine after local variables have been declared unless there are local initializations.

```
(CXdb) break instruction BESTMV

#2: break instruction, on [#0/*], Enabled, ignore 0/0
        [0x800015f0] BESTMV in pickup.f line 55
```

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

```
(CXdb) break source 135

#3: break source, on [#0/*], Enabled, ignore 0/0
        [0x800016f0] BESTMV in pickup.f line 65
```

The above command sets a breakpoint at the starting address of source unit 135. The source unit numbers for a given line can be displayed using the `info line` command.

```
(CXdb) info break
Event   Enabled     Ignore     proc/td      Address        Where
#0        y          0/0        0/*      [0x80001620] BESTMV in pickup.f line 60
#1        y          0/0        0/*      [0x800015f2] BESTMV in pickup.f line 59
#2        y          0/0        0/*      [0x800015f0] BESTMV in pickup.f line 55
#3        y          0/0        0/*      [0x800016f0] BESTMV in pickup.f line 65
```

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

```
(CXdb) run
Beginning execution of Process [#0]
Process [#0/0] stopped by Bkpt 2, at [0x800015f0] BESTMV in pickup.f line 55
```

The above example begins process execution without passing any arguments to the process. When execution reaches the address of `800015f0`, breakpoint 2 is reached because it is enabled. Because it does not have an ignore count, the breakpoint is triggered. Because the breakpoint did not have its own eventpoint handler, the default handler for breakpoints is executed, which prints the message telling you what caused process execution to stop.

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

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

```
(CXdb) break line 60 $Middle

#4: break line, on [#0/*], Enabled, ignore 1/2
        [0x80001620] BESTMV in pickup.f line 60

INFO: Eventpoint 0 also has a breakpoint at address 0x80001620.
```

The above command sets another breakpoint at line 60 of the current source file. The debugger variable `$Middle` was created and assigned to this eventpoint. CXdb informs you that two eventpoints now reside at the same address location.

```
(CXdb) break line 60 {echo "Breakpoint 5 reached" ; resume;}

#5: break line, on [#0/*], Disabled, ignore 0/0
        [0x80001620] BESTMV in pickup.f line 60
 {
       echo "Breakpoint 5 reached" ;
       resume;
 }

INFO: Eventpoint 4 also has a breakpoint at address 0x80001620.
```

The above command sets a third breakpoint at line 60. An eventpoint handler is specified for this eventpoint. The handler prints a message and then resumes execution of the process.

```
(CXdb) continue
Resuming execution of process [#0]
Breakpoint 5 reached
Resuming execution of process [#0]
Process [#0/0] stopped by Bkpt 3, at [0x800016f0] BESTMV in pickup.f line 65
```

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

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

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

```
(CXdb) run
Process [#0] is already running with pid 16139.
Terminate existing process and restart? y
Beginning execution of Process [#0]
Process [#0/0] stopped by Bkpt 0, at [0x80001620] BESTMV in pickup.f line 60
```

When process execution reaches address `80001620,` breakpoint 4 is reached because eventpoint 5 is disabled. Breakpoint 4 has an ignore count, so its counter is incremented by one. Because an eventpoint has still not been triggered, breakpoint 0 is reached. Because it is enabled and does not have an ignore count, breakpoint 0 is triggered.

```
(CXdb) info event *
#0: break line, on [#0/*], Enabled, ignore 0/0
        [0x80001620] BESTMV in pickup.f line 60
#3: break source, on [#0/*], Enabled, ignore 0/0
        [0x800016f0] BESTMV in pickup.f line 65
#4: break line, on [#0/*], Enabled, ignore 1/2
        [0x80001620] BESTMV in pickup.f line 60
#5: break line, on [#0/*], Disabled, ignore 0/0
        [0x80001620] BESTMV in pickup.f line 60
 {
       echo "Breakpoint 5 reached" ;
       resume;
 }
```

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

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

The above command enables breakpoint 5. This will cause breakpoint 5 to be triggered the next time address `80001620` is reached because it is the highest-numbered, enabled breakpoint.

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

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

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

Related Commands

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

Related Concepts

eventpoints
eventpoint handlers
tracepoints
watchpoints

Related Parameters

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

C language expressions

Description

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

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

1. Identifiers:

- Restrictions:
  - Identifiers are case sensitive.
  - If filename appears in the scope path prefix to the identifier, and if the filename contains a special character such as dot (.) that is a lexical element in the alphabet of the language, then precede the special character with a backslash (\).
  - If a dollar sign (`$`) is part of the identifier name, precede the dollar sign with a backslash (\).
  - An identifier qualified by "`const`" is tracked, and a warning is issued if it is modified.
- Program variables can be either:
  - Unqualified
  - Qualified by a scope path prefix
- CXdb debugger variables:
  - Debugger variables must begin with the CXdb scope prefix `cxdb$` or `$`.
  - Debugger variable names are case sensitive.
  - An assignment to a debugger variable modifies it to have the value, type, and precision of the right-hand side of the assignment.
- Hardware registers:
  - Register names must be qualified with the CXdb scope prefix `cxdb$` or `$`.
  - Register names are not case sensitive, except for the scalar, vector, and communication registers.
  - An assignment to a register modifies its value only; its type and precision remain the same.

2. Constants:

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

    `long int` (32 bits)
    `unsigned long`
    `long long`
    `unsigned long long`
  - A suffix of `ll` or `LL` designates one of the following, depending on the magnitude of the constant:

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

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

    `\a` — alert (audible or visual)
    `\b` — backspace
    `\f` — formfeed
    `\n` — newline
    `\r` — carriage return
    `\t` — horizontal tab
    `\v` — vertical tab
  - The following trigraph sequences are accepted:

    `??=` yields `#`
    `??(` yields `[`
    `??)` yields `]`
    `??/` yields `\`
    `??'` yields `^`
    `??<` yields `{`
    `??>` yields `}`
    `??!` yields `|`
    `??-` yields `~`
- Enumeration — Variables that are assigned an enumeration type by the program are internally changed to type `int` by the C compiler. Therefore, for consistency, CXdb also converts any enumeration type to type `int` within cast expressions.
- String literals — The ANSI "wide" string type is not supported and not recognized.

3. Expressions:

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

applied to the pointer expression.

- Function calls
- Structure and union members

  Selection
  Remote selection
  Bit fields

- Postfix increment and decrement operators

  `++` postfix increment
  `--` postfix decrement

- Unary operators include:
  - Prefix operators

    `++` prefix increment
    `--` prefix decrement

  - Address and indirection operators

    `&` address of
    `*` indirection

  - Unary arithmetic operators

    `+` unary plus
    `-` unary minus
    `~` bitwise complement
    `!` logical negation

  - The `sizeof` operator — Computes the size of an object in bytes.

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

- Multiplicative operators include:

  `*` multiply
  `/` divide
  `%` mod left operand by right

- Additive operators include:

  `+` add
  `-` subtract

- Shift operators include:
  - << shift left
  - >> shift right
- Relational operators include:
  - < test for less than
  - > test for greater than
  - <= test for less than or equal to
  - >= test for greater than or equal to
- Equality operators include:
  - == test for equal to
  - != test for not equal to
- Bitwise operators include:
  - & AND
  - ^ exclusive OR
  - | inclusive OR
- Logical operators include:
  - && test for AND
  - || test for OR
- The conditional operator (? :):
  - If the test condition is true, evaluate the operand to the left of the colon (:).
  - Otherwise, evaluate the operand to the right of the colon.
- Assignment operators include:
  - = simple assignment
  - *= multiply and assign
  - /= divide and assign
  - %= mod and assign
  - += add and assign
  - -= subtract and assign
  - &= AND and assign
  - ^= exclusive OR and assign
  - |= inclusive OR and assign
  - <<= shift left and assign
  - >>= shift right and assign
- The comma operator (,):
  - Evaluate the left operand, then the right.
  - Return the result of the right operand.

Examples

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

```
(CXdb) break routine 0x800013d4
#1: break routine, on [#0/0], Enabled, ignore 0/0
        [0x800013d4] numbers'suba in numbers.c line 19
```

The above command sets a breakpoint at the beginning of the routine that contains the hexadecimal address `800013d4`. The expression `0x800013d4` is C language notation for a hexadecimal address.

```
(CXdb) print ++k
(int) 5
```

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

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

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

```
(CXdb) print/x &y
(int*) 0x8000c268
```

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

```
(CXdb) print matrix[0][3][0..4]
float[1][3..3][5]
[0][3][0..4] :     28.5585    17.6754    6.7924    -4.0906    -14.9737
```

The above command prints an array slice, or subset. The slice consists of elements `[0][3][0]` through `[0][3][4]` of `matrix`.

```
(CXdb) find memory forward ff &matrix:k+50
Data found at 0x8000d438
```

The above command searches for the hexadecimal byte pattern `ff` in a region of process memory. The memory region is specified by the language expression `&matrix:k+50`. The first part of the expression uses the operator & to return the starting address of `matrix`. The second part of the expression is `k+50`, and it evaluates to the number of bytes of memory to search.

Related Commands

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

Related Concepts

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

Related Parameters

array-slice
debugger-variable
language-expression
string

cmderr

Description

`Cmderr` is the list of viewports, or destinations, that receive all error messages and informational messages generated in response to CXdb commands. A viewport can be either a file or the CXdb command window. The default viewport for cmderr is the command window.

To direct the CXdb messages to different destinations, you create a list of viewports for `cmderr`. The commands for maintaining this list are:

- `add cmderr` — Add new viewports to the viewport list for cmderr.
- `remove cmderr` — Remove viewports from the viewport list for cmderr.
- `set cmderr` — Delete the current viewport list for cmderr and replace it with the specified list.

For viewports that are files, you might want to control whether or not the new data will overwrite an existing file. The following commands give you this control:

- `clear noclobber` — Allow existing files to be overwritten.
- `set noclobber` — Respond with an error message if the specified viewport file already exists.

The default viewport for `cmderr` is Window #1 (the command window), and the default for noclobber is clear (off).

Examples

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

```
(CXdb) set noclobber
```

The above command prevents overwriting of existing files.

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

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

Related Commands

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

Related Concepts

cmdlog
logging
windows
cmdout
viewports

Related Parameters

redirection-operator
viewport

cmdlog

Description

Cmdlog refers to any entries, or input, made in the CXdb command window. It includes input read from a command file or initialization file as well as entries you make directly in the command window.

Entries that you make directly are always echoed in the command window. Input read from command files or initialization files is not echoed in the command window unless the echo option is turned on (either by default or by means of the `set echo` command).

In addition to echoing input in the command window, you can store it in any number of files. You do this by specifying a viewport list that contains the names of the destination files for cmdlog. The commands to modify the viewport list for cmdlog are:

- `add cmdlog` — Add new viewports to the viewport list for cmdlog.
- `remove cmdlog` — Remove viewports from the viewport list for cmdlog.
- `set cmdlog` — Delete the current viewport list for cmdlog and replace it with the specified list.

Once you have specified a viewport list for cmdlog, you can enable or disable logging to all the viewports in that list by using the following commands:

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

If you are using files to log input data, you can use the following commands to specify whether or not the new data will overwrite an existing file:

- `clear noclobber` — Allow existing files to be overwritten.
- `set noclobber` — Respond with an error message if the specified viewport file already exists.

Examples

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

```
(CXdb) set noclobber
```

The above command prevents overwriting of existing files.

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

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

```
(CXdb) set logging
```

The above command enables logging to all viewports of cmdlog. In this case, any further input to the command window is stored in the file `my_input.log.`

Related Commands

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

Related Concepts

cmderr
logging
windows
cmdout
viewports

Related Parameters

viewport

cmdout

Description

Cmdout is the list of viewports, or destinations, that receive the normal output generated in response to CXdb commands. A viewport can be either a file or the CXdb command window. The default viewport for cmdout is the command window.

By default, the CXdb output appears in the command window. However, you can also send this output to other destinations, called viewports. A viewport may be either a file or the CXdb command window.

To direct the CXdb output to different destinations, you create a list of viewports for cmdout. The commands for maintaining this list are:

- `add cmdout` — Add new viewports to the viewport list for cmdout.
- `remove cmdout` — Remove viewports from the viewport list for cmdout.
- `set cmdout` — Delete the current viewport list for cmdout and replace it with the specified list.

For viewports that are files, you might want to control whether or not the new data will overwrite an existing file. The following commands give you this control:

- `clear noclobber` — Allow existing files to be overwritten.
- `set noclobber` — Respond with an error message if the specified viewport file already exists.

The default viewport for cmdout is Window #1 (the command window), and the default for noclobber is clear (off).

Examples

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

```
(CXdb) set noclobber
```

The above command prevents overwriting of existing files.

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

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

Related Commands

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

Related Concepts

cmderr
logging
windows
cmdlog
viewports

Related Parameters

redirection-operator
viewport

command files

Description

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

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

There are four primary uses for command files:

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

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

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

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

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

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

Examples

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

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

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

Also assume that echoing is enabled (on).

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

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

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

Related Commands

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

Related Concepts

cmdlog
initialization files
windows
console working directory
logging

Related Parameters

file-name

Compiler-Debugger Interface

CDI

Description

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

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

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

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

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

- `.ns` —Name space; provides information about external program symbols and scope blocks.
- `.tsi` —Type and scope information; provides language specific data and lexical scope information about all program identifiers.
- `.sut` —Source unit table; provides information about individual source units as well as the interrelationships between source units. The source units reflect the syntax of the source code.

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

- `.lrt` — Location range table; specifies the storage location of each variable for different ranges of the program counter (PC).
- `.srt` — Source range table; specifies which source units are active over different ranges of the program counter (PC).
- `.vt` — Variable table; specifies the attributes of all variables, including synthesized variables. It is analogous to the symbol table used by the compiler.
- `.xpt` — Expression table; provides information about all synthesized expressions used by the compiler.

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

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

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

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

Caution

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

console working directory

Description

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

The commands that use the console working directory are listed below:

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

The commands that affect the console working directory are described below:

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

The console working directory is initially set to the directory from which you invoked CXdb. The process working directory is initially set to the console working directory.

console working directory

Examples

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

```
(CXdb) source cxdb/commfile.1
```

The above command executes the CXdb commands found in the `/mnt/jones/cxdb/commfile.1` file. Because the path name to the file is relative, the console working directory is used as the base path name.

```
(CXdb) cd cxdb
```

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

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

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

Related Commands

```
add cmderr             add cmdlog
add cmdout             add default path
add path               cd
core                   debug core
debug exec             executable
info cxdb              pwd
remove cmderr          remove cmdlog
remove cmdout          remove default path
remove path            set cmderr
set cmdlog             set cmdout
set default path       set directory
set path               source
```

Related Concepts

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

Related Parameters

directory-specifier
file-name

csd

Description

The CONVEX csd debugger is a symbolic debugger. Many of the most frequently used csd commands are available in CXdb through aliases. By including the `source /usr/lib/cxdb/csd_aliases` command in your `.cxdbinit` file, you can incorporate these predefined aliases automatically each time CXdb is invoked.

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

For a full list of csd command supported by CXdb, refer to Appendix C, of the *CONVEX CXdb User's Guide.*

Related Commands

`info alias` `source`

Related Concepts

gdb

debugger variables

Description

Debugger variables are variables that you can define in CXdb.

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

In some cases, it is necessary to distinguish a debugger variable from another type of symbol with the same name. To distinguish the symbols, you can precede the debugger variable with a dollar sign (`$`).

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

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

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

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

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

```
evaluate
print
```

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

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

In addition to debugger variables that you define, there are a number of special, predefined debugger variables that enable you to reference the process registers. These debugger variables are dynamic. When your process is running, they are updated to contain the current register values. The names of these special variables, and the registers they represent, are as follows:

- `$a0` to `$a7` (or `$A0` to `$A7`) — The 32-bit address registers A0 to A7.
- `$ap` (or `$AP`) — The argument pointer (AP or A6).
- `$c0` to `$c63` — The 32-bit communication registers from ring 4, regardless of which set is allocated.
- `$C0` to `$C63` — The 64-bit communication registers from ring 4, regardless of which set is allocated.
- `$cl0` to `$cl63` (or `$CL0` to `$CL63`) — The lock bits for the communication registers (one bit per register).
- `$cir` (or `$CIR`)— The 3-bit communication index register (CIR).
- `$fp` (or `$FP`) — The frame pointer (FP or A7).

- `$pc` (or `$PC`) — The program counter (PC).
- `$psw` (or `$PSW`) — The processor status word (PSW).
- `$s0` to `$s7` — The 32-bit scalar registers S0 to S7.
- `$S0` to `$S7` — The 64-bit scalar registers S0 to S7.
- `$sp` (or `$SP`) — The stack pointer (SP, or A0).
- `$v0` to `$v7` — The 32-bit vector registers V0 to V7. Each vector register may contain up to 128 elements (numbered 0 to 127), and each element is 32 bits long. To access a particular element, use array notation. For example, to access the 123rd element of vector register V4, use the notation `$v4(123)` in FORTRAN or `$v[122]` in C.
- `$V0` to `$V7` — The 64-bit vector registers V0 to V7. Each vector register may contain up to 128 elements (numbered 0 to 127), and each element is 64 bits long. To access a particular element, use array notation. For example, to access the 123rd element of vector register V4, use the notation `$V4(123)` in FORTRAN or `$V[122]` in C.
- `$vl` (or `$VL`) — The vector length register (VL).
- `$vml` (or `$VML`) — The lower half of the vector merge register (VM).
- `$vmu` (or `$VMU`) — The upper half of the vector merge register (VM).
- `$vs` (or `$VS`) — The vector stride register (VS).

Finally, there are two other predefined debugger variables that are updated dynamically:

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

Examples

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

```
(CXdb) break routine SUB_D $D
Breakpoint 2, [0x80001882] SUB_D in myprog.f line 93
```

The above command sets a breakpoint at the routine called `SUB_D`. The object number for this breakpoint is 2, and this object number is stored in the debugger variable `$D`.

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`.

```
(CXdb) evaluate $A=x+y
```

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

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

The above command stores the value `1234` in the scalar register S0. The contents of the other predefined debugger variables, except `$self`, can be modified in a similar way.

Related Commands

```
break instruction              break line
break routine                  break source
debug core                     debug exec
debug proc                     evaluate
event exec                     event modify
event reached instruction      event reached line
event reached routine          event reached source
event relation                 event signal
macro                          print
trace instruction              trace line
trace routine                  trace source
watch
```

Related Concepts

command files | initialization files
eventpoint handlers

Related Parameters

debugger-variable

default environment

Description

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

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

Modifications to the default environment do not affect existing processes that were passed the default environment. You can modify the default environment with the following commands:

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

Examples

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

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

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

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

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

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

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

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

If a new process is created, and its process object does not have its own environment, it is passed these three environment variables of the default environment.

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

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

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

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

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

Related Commands

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

Related Concepts

environment	process object

Related Parameters

environment-variable	string

default search path

Description

The default search path is used as the basis for the search path for each new process object.

The search path of a process object is used to find program source files as well as CXdb compiler-generated data files.

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

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

- `add default path` — Adds directories to the end of the default search path.
- `info cxdb` — Displays information about the current state of CXdb as well as the directories in the default search path.
- `remove default path` — Removes directories from the default search path.
- `set default path` — Sets the default search path to the specified directories.

Examples

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

```
(CXdb) info cxdb
       .
       .
       .
Process [#0], No image, executable = a.out
       shell = csh

Default search path:
        .
```

The above example displays the state of CXdb. At the bottom of the example you can see the default search path setting. This is the current console working directory. Now that you know what directory the search path is set to, you can change it.

```
(CXdb) add default path /mnt/jones/libraries , /mnt/jones/math/libraries
Default search path:
        .
        /mnt/jones/libraries
        /mnt/jones/math/libraries
```

The above example adds two directories to the default search path. The next process object that is created will receive the new default search path which now includes the `/mnt/jones/libraries` and `/mnt/jones/math/libraries` directories.

```
(CXdb) remove default path /mnt/jones/math/libraries
Default search path:
        .
        /mnt/jones/libraries
```

The above command removes the `/mnt/jones/math/libraries` directory from the default search path. The other directories in the default search path remain.

```
(CXdb) set default path /mnt/jones/project2 , /mnt/jones/project2/source
       /mnt/jones/project2
       /mnt/jones/project2/source
```

The above command removes all the existing directories from the default search path and sets the default search path to the two listed directories. Now each new process object will include the `/mnt/jones/project2` and `/mnt/jones/project2/source` directories, as well as the process working directory as its search path.

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

Related Commands

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

Related Concepts

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

Related Parameters

directory-specifier

default search path

environment

Description

The environment is the set of environment variables of a process object. A process object does not initially have an environment. An environment is created for a process object the first time you change its environment. The environment then becomes a copy of the default environment modified by the effects of the environment command issued.

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

You can modify the environment of a process object with the following commands. Only the `info environment` command does not create an environment for a process object.

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

Examples

The following examples illustrate how to use the environment commands.

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

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

```
(CXdb) clear environment
```

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

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

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

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

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

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

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

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

Related Commands

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

Related Concepts

default environment   process object

Related Parameters

environment-variable   string

eventpoint handlers

Description

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

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

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

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

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

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

eventpoint handlers

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

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

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

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

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

Examples

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

```
(CXdb) break line 56 {print x;}
```

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

```
(CXdb) break line 56 {print x; resume;}
```

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

```
(CXdb) break line 56 {if (x==49) {echo "x is 49";} else {resume;} }
```

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

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

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

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

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

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

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

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

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

eventpoint handlers

Related Commands

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

Related Concepts

breakpoints	**debugger variables**
eventpoints	**tracepoints**
watchpoints	

Related Parameters

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

eventpoints

Description

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

You can trap the following types of events:

- instruction is reached
- line is reached
- first source unit of a routine is reached
- source unit is reached
- address range is modified
- signal is caught
- relational expression evaluates to TRUE
- process exec's
- thread is spawned
- a thread is joined with another thread

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

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

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

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

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

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

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

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

There are many different ways to set a eventpoint. The different commands are divided into their eventpoint types.

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

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

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

The following command sets a *signal* eventpoint.

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

The following command sets a *modify* eventpoint.

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

The following command sets a *relational* eventpoint.

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

The following command sets an *exec* eventpoint.

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

The following command sets a *join* eventpoint.

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

The following command sets a *spawn* eventpoint.

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

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

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

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

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

Examples

For examples of how to set, manipulate, and remove eventpoints, please refer to the individual reference pages for the above commands.

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

Related Commands

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

Related Concepts

eventpoints
eventpoint handlers
language expressions
tracepoints
watchpoints

Related Parameters

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

FORTRAN language expressions

Description

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

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

1. Identifiers:

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

2. Constants:

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

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

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

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

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

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

3. Arithmetic expressions:

- Additive operators:

  + (binary)
  – (binary)
  + (unary)
  – (unary)

- Multiplicative operators (binary):

  *
  /

- Exponential operators (binary):

  **

4. Relational expressions:

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

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

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

     `.AND.` (binary)
     `.OR.` (binary)
     `.EQV.` (binary)
     `.NEQV.` (binary)
     `.NOT.` (unary)
6. Character expressions:
   - White space is significant.
   - Concatenation is done with the binary concatenation operator `//`.
7. Assignment statements:
   - Assignments can be made to either:
     - Program variables
     - CXdb debugger variables
   - The assignment operator is `=`.
8. Arrays:
   - Refer to the *CONVEX FORTRAN Language Reference Manual* for specifications on array subscript expressions.
   - Array slices:
     - An array slice is a subscript expression that contains a slice range.
     - The data type of the slice is "array of ...", where the upper and lower bounds of the resulting array are defined by the slice range.
     - The syntax for an array slice is `(lower..upper)`, where lower is the first element and upper is the last element, inclusive.
     - Restrictions — All subscripts must be specified in the slice range. For example, if the array definition is `INTEGER Y(10,2)` and the slice specification is `Y(2..4,2)`, then the slice contains data from columns 2, 3, and 4 of row 2 (or rows 2, 3, and4 of column 2 in the presence of the row-wise compiler directive).
9. Character substring expressions:
   - Refer to the *CONVEX FORTRAN Language Reference Manual* for specifications on character substring expressions.
10. VAX records:
    - Refer to the *CONVEX FORTRAN Language Reference Manual* for the structure and field selection syntax of VAX records.

11. Cray pointers:

- Refer to "Cray FORTRAN compatibility" in the *CONVEX FORTRAN Language Reference Manual* for specifications on pointers.

12. Intrinsic functions:

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

- Intrinsic functions for converting to integer:

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

- Intrinsic functions for converting to real:

```
REAL
DBLE
QEXT
```

- Intrinsic functions for converting to complex:

```
CMPLX
DCMPLX
```

- Intrinsic functions for character conversions:

```
CHAR
ICHAR
```

- Intrinsic functions for string comparisons:

```
LLT
LLE
LGT
LGE
```

- Intrinsic functions for conversion to nearest integer:

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

- Intrinsic functions for address acquisition:

```
LOC
%LOC
```

- Miscellaneous intrinsic functions:

```
ABS
MOD
```

Examples

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

```
(CXdb) break routine '8000139a'x
#1: break routine, on [#0/0], Enabled, ignore 0/0
        [0x8000139a] SUBA in numbers.f line 16
```

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

```
(CXdb) print K=K+1
(INTEGER*4) 5
```

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

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

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

```
(CXdb) print/x loc(Y)
(INTEGER*4) 0x8004a00c
```

The above command evaluates the expression `loc(Y)` and prints the result in hexadecimal (/x) format. The FORTRAN function `loc()` returns the address of the program variable `Y`. Thus, `8004a00c` is the hexadecimal address of `Y`. (The `0x` in front of the address indicates that it is a hexadecimal number.)

```
(CXdb) print MATRIX(1,4,1..5)
REAL*4(1:1, 4:4, 1:5)
(1,4,1..5) :     28.5585    17.6754    6.7924    -4.0906    -14.9737
```

The above command prints an array slice, or subset. The slice consists of elements `(1,4,1)` through `(1,4,5)` of `MATRIX`.

```
(CXdb) find memory forward ff loc(ARRAY):J+50
Data found at 0x800ac438
```

The above command searches for the hexadecimal byte pattern `ff` in a region of process memory. The memory region is specified by the language expression `loc(ARRAY):J+50`. The first part of the expression uses the FORTRAN function `loc()` to return the starting address of `ARRAY`. The second part of the expression is `J+50`, and it evaluates to the number of bytes of memory to search.

Related Commands

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

FORTRAN language expressions

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

Related Parameters		
	array-slice	debugger-variable
	language-expression	string

gdb

Description

The CONVEX gdb debugger is a symbolic debugger. Many of the most frequently used gdb commands are available in CXdb through aliases. By including the `source /usr/lib/cxdb/gdb_aliases` command in your `.cxdbinit` file, you can incorporate these predefined aliases automatically each time CXdb is invoked.

With the predefined aliases incorporated, you can type in a gdb command while using CXdb. If the command has an alias, the alias is substituted, and the equivalent CXdb command is executed. If the gdb 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, CXdb suggests a CXdb command with the closest functionality to the gdb command.

For a full list of gdb command supported by CXdb, refer to Appendix D, of the *CONVEX CXdb User's Guide*.

Related Commands

`info alias` `source`

Related Concepts

csd

initialization files

Description

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

The primary uses for initialization files are:

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

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

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

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

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

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

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

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

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

Examples

The following example illustrates several uses of initialization files.

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

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

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

```
debug exec a.out
break routine 1$_MAIN__
run without
disassemble
alias s print
set default step loop
```

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

```
(CXdb)
Default source file: ./pickup6.f
Default source language: Fortran

Process [#0] created
Breakpoint 0, [0x80001334] PICKUP in pickup6.f line 1
Beginning execution of Process [#0]
Process [#0/0] stopped by Bkpt 0, at [0x80001334] PICKUP in pickup6.f line 1
Disassemble Process [#0/0] from 0x80001334 to 0x80001460

(CXdb)
```

First CXdb executes the file `/usr/lib/cxdb/.cxdbinit`, which establishes some aliases and sets the default step granularity to statement. Next CXdb executes the `.cxdbinit` file in the console working directory, and that file actually starts a process as well as defining an alias and setting a default step granularity.

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

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

Related Commands

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

Related Concepts

cmdlog
command files
console working directory
logging
windows

Related Parameters

file-name

language expressions

Description

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

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

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

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

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

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

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

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

- As a data value
- As an address

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

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

Examples

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

```
(CXdb) break routine SUBA \; $BreakA
#1: break routine, on [#0/0], Enabled, ignore 0/0
        [0x8000139a] SUBA in numbers.f line 16
```

The above command sets a breakpoint at the beginning of the routine called `SUBA`. In this case, CXdb interprets the language expression `SUBA` to be the starting address of the routine `SUBA`. The debugger variable `$BreakA` stores the number of this breakpoint, which is 1. The backslash-semicolon (`\;`) delimiter is required to separate the language expression `SUBA` from the debugger variable `$BreakA`.

```
(CXdb) break routine '8000139a'x
#1: break routine, on [#0/0], Enabled, ignore 0/0
        [0x8000139a] SUBA in numbers.f line 16
```

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

```
(CXdb) break routine 0x800013d4
#1: break routine, on [#0/0], Enabled, ignore 0/0
        [0x800013d4] numbers`suba in numbers.c line 19
```

The above command sets a breakpoint at the beginning of the routine that contains the hexadecimal address `800013d4`. The expression `0x800013d4` is C language notation for a hexadecimal address.

```
(CXdb) print Y+2
(REAL*4) 4.5000
```

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

```
(CXdb) print Y=Y+2
(REAL*4) 6.5000
```

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

```
(CXdb) print/x loc(Y)
(INTEGER*4) 0x8004a00c
```

The above command evaluates the expression `loc(Y)` and prints the result in hexadecimal (`/x`) format. The FORTRAN function `loc()` returns the address of the program variable `Y`. Thus, `8004a00c` is the hexadecimal address of `Y`. (The `0x` in front of the address indicates that it is a hexadecimal number.)

```
(CXdb) print/x &y
(int*) 0x8000c268
```

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

```
(CXdb) print "This is a test"
(CHARACTER*14) "This is a test"
```

The above command prints a literal string of characters.

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

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

```
(CXdb) print MATRIX(1,4,1..5)
REAL*4(1:1, 4:4, 1:5)
(1,4,1..5) :      28.5585     17.6754     6.7924     -4.0906     -14.9737
```

The above command prints an array slice, or subset. The slice consists of elements (1,4,1) through (1,4,5) of MATRIX. The subscripts in this example are in FORTRAN notation.

```
(CXdb) print matrix[0][3][0..4]
float[1][3..3][5]
[0][3][0..4] :      28.5585     17.6754     6.7924     -4.0906     -14.9737
```

The above command prints an array slice, or subset. The slice consists of elements [0][3][0] through [0][3][4] of matrix. The subscripts in this example are in C notation.

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

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

```
(CXdb) find memory forward ff loc(ARRAY):J+50
Data found at 0x800ac438
```

The above command searches for the hexadecimal byte pattern ff in a region of process memory. The memory region is specified by the language expression loc(ARRAY):J+50. The first part of the expression uses the FORTRAN function loc() to return the starting address of ARRAY. (The equivalent function for returning the address of a variable in C syntax is &.) The second part of the expression is J+50, and it evaluates to the number of bytes of memory to search.

Related Commands

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

Related Concepts

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

Related Parameters

array-slice	debugger-variable
language-expression	string

logging

Description

Logging is the process of recording the activity that takes place during a debugging session. Three types of information can be logged, and each type has a particular name. The types are:

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

The method for logging the information is to direct it to a viewport. A viewport is either a file or the CXdb command window. A file is a permanent log because it can be saved, but the command window is obviously a temporary log for display purposes only.

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

- `add cmderr` — Add new viewports to the list of cmderr viewports.
- `add cmdlog` — Add new viewports to the list of cmdlog viewports.
- `add cmdout` — Add new viewports to the list of cmdout viewports.
- `remove cmderr` — Remove viewports from the list of cmderr viewports.
- `remove cmdlog` — Remove viewports from the list of cmdlog viewports.
- `remove cmdout` — Remove viewports from the list of cmdout viewports.
- `set cmderr` — Delete the current list of cmderr viewports and replace it with a new list.
- `set cmdlog` — Delete the current list of cmdlog viewports and replace it with a new list.
- `set cmdout` — Delete the current list of cmdout viewports and replace it with a new list.

The default viewport for cmderr and cmdout is the CXdb command window (Window #1). There is no default viewport for cmdlog because your entries in the command window are automatically echoed there.

For logging, most of the viewports you specify will be files. If the specified file does not exist, CXdb creates it. If the specified file already exists, then you can use the following commands to control whether or not CXdb writes (either overwrites or appends) to the existing file:

- `clear noclobber` — Allow overwriting of existing files and creation of new files for appending.
- `set noclobber` — Respond with an error message if attempting to overwrite an existing file or attempting to append to a file that does not exist.

The default for noclobber is clear (off).

The viewport lists for cmderr and cmdout apply to all commands executed by CXdb, regardless of whether you enter the commands directly yourself or read them in from a command file. However, for each individual command, you can override the viewport lists for cmdout and cmderr by using redirection operators with the command. The redirection operators enable you to specify a special viewport list that applies only to the one command with which it appears.

For cmdlog, the following commands enable you to control whether or not input from command files and initialization files is echoed to the cmdlog viewports:

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

The default for echo is set (on).

Also for cmdlog, you can enable or disable logging of all input with the following commands:

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

The default is logging disabled (clear).

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

Examples

The following examples illustrate one way to modify the viewport lists for cmderr, cmdlog, and cmdout. They also illustrate how to use redirection operators to override the viewport lists.

```
(CXdb) set noclobber
```

The above command prevents writing to existing files.

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

The above command adds the file `error_log` to the viewport list for cmderr. Note that the list already contains Window #1 (the command window) because it is the default viewport for cmderr. All CXdb messages are now stored in `error_log` as well as being displayed in the command window.

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

The above command adds the file `output_log` to the viewport list for cmdout. This viewport list already contains Window #1 (the command window) as the default. All output from CXdb commands is now stored in `output_log` as well as being displayed in the command window.

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

The above command adds the file `input_log` to the viewport list for cmdlog. This viewport list does not contain Window #1 as the default because everything you enter in the command window is automatically echoed there. Adding Window #1 to the viewport list for cmdlog actually causes each input line to appear twice in the command window.

At this point, nothing is sent to the file `input_log` because logging has not been enabled for cmdlog. To enable it, enter the following:

```
(CXdb) set logging
```

Now any commands you enter in the command window are stored in `input_log` as well as being displayed in the command window. This is a convenient way to create a command file. After exiting from CXdb, you can edit `input_log` so that it contains only the commands you want to use again. The next time you invoke CXdb, you can execute this file with the `source` command.

To display the all current settings, enter the following:

```
(CXdb) info cxdb

Current CXdb state:

     Environment:
             pid: 2130
             cwd: /doc/Smith/ug
   Command Modes: Echo On, Logging On, Noclobber On
          cmdout: Window #1, output_log
          cmderr: Window #1, error_log
          cmdlog: input_log
        EvalOpts: fpmode = dual, iprecision = 4, rprecision = 4
           Shell: csh (tcsh is busted for now)

Process Defaults:
Fixed Scheduling: Off
       Step size: statement
   Process shell: csh (tcsh is busted for now)
          fpmode: dual
     Memory size: (none)
  Memory Formats: byte=(none), halfword=(none), word=(none)
                  longword=(none), quadword=(none)
     Search path:
        .

       Processes:
```

The above response indicates that echo, logging, and noclobber are all enabled (on). It also list the current viewports for cmdout, cmderr, and cmdlog.

You can override the cmdout and cmderr viewport lists for individual commands by using redirection operators. For example, you could enter the following:

```
(CXdb) info cxdb >temp_out >&temp_err
```

Output from the above command is redirected to the file `temp_out`, and any CXdb messages associated with this command are redirected to `temp_err`. No other viewports receive the response from this command, but the command line itself is still logged in `input_log`. (Because the noclobber option is enabled, an error results if `temp_out` or `temp_err` exists.)

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	viewports
windows	

Related Parameters

redirection-operator	viewport

Maryland Windows

Description

The Maryland Windows interface provides a windowing environment for CXdb when run on a non-graphics terminal. The windows in the Maryland Windows interface perform much like they do in the CXwindows interface. However, keystrokes are used to manipulate the windows and edit the text inside the windows.

Functions are activated by specific key bindings. In the key bindings below, the META key (sometimes labeled ESC) is represented as `M-` and the CTRL key by the caret (^).

The function names are grouped together in the list below. Before each function name is a default key binding that performs that function.

- Window movement functions:
  - `M-o` `lower-window`
  - `M-m` `move-window`
  - `M-n` `next-window`
  - `M-p` `previous-window`
  - `M-r` `raise-window`
  - `M-z` `resize-window`
  - `M-k` `close-window` (except command window)
- Cursor movement and scrolling functions:
  - `M-<` `beginning-of-buffer`
  - `M->` `end-of-buffer`
  - `^A` `beginning-of-line`
  - `^E` `end-of-line`
  - `^B` `backward-char` (except command window)
  - `^F` `forward-char` (except command window)
  - `M-b` `backward-word`
  - `M-f` `forward-word`
  - `^N` `down-line` (except command window)
  - `^P` `up-line` (except command window)
  - `^V` `down-screen`
  - `M-v` `up-screen`

To move the cursor around the command window, one character at a time, you must use the arrow keys.

In the command window you can perform the functions described below:

- Editing functions:

`M-h`	`backward-delete-word`
`M-^D,M-d,M-delete`	`kill-word`
`backspace,^h`	`delete-backward-char`
`delete,^D`	`delete-char`
`^K`	`kill-line`
`^W`	`kill-region`
`M-w,M-W`	`copy-region-as-kill`
`^@`	`set-mark-command`
`^X`	`exchange-point-and-mark`
`^T`	`transpose-characters`
`^Y`	`yank`

- History functions:

`^N`	`down-history`
`^P`	`up-history`

- Miscellaneous functions:

`M-c`	`capitalize-word`
`M-0 to M-9`	`digit-argument`
`^U`	`universal-argument`
`M-l`	`downcase-word`
`M-u`	`upcase-word`
`^L`	`redraw-display`

For a more complete description on the performance of these functions refer to the *CONVEX CXdb User's Guide* or the reference page for the `bind` command.

Related Commands

`bind` `info bind`

Related Concepts

windows

Related Parameters

function-name key-name

process object

Description

A process object contains all of the information about the program currently being debugged in CXdb. A process object is created when you use any of the following `debug` commands:

```
debug core
debug exec
debug proc
```

Once the process object has been created, you can modify the information it holds to meet your debugging needs. The process object holds references to the following pieces of information:

- executable file and compiler-generated data files
- image of the process
- process working directory in which to run the process
- environment in which to run the process
- search path for source files and compiler-generated data files.
- eventpoints created in the process and their handlers
- display formats
- memory formats
- process shell in which the process executes
- settings of seq and sqs bits

Once a process object has been created, you can change any of the existing information in the process object. You can introduce a new executable file or a different image without having to leave CXdb or create a new process object.

Examples

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

```
(CXdb) debug exec a.out

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

Process object [#0] created.
```

This first example creates a new process object in CXdb. Because a process object did not yet exist, the `debug exec` command created one to hold the executable data file and the information in the compiler-generated data files (if they are available). The process object does not yet have an image of the program.

```
(CXdb) core corefile
```

The above example brings the image from the core file into the process object. At this point you could use the image to examine the state of your process when it dumped core.

```
(CXdb) attach 12345
Attaching process #0 to pid 12345
```

The above command performs several actions. The image of the core file is removed from the process object. CXdb attaches to the process with a process ID of `12345`. This process is stopped and brought under the control of CXdb. The image of this process is associated with the process object. The existing windows are updated to reflect the new image of the process object.

```
(CXdb) detach
Detaching from process #0
(CXdb) executable b.out

Default source file: program2.c
Default source language: C
```

The above example performs two actions. First, CXdb detaches from the current running process. This leaves the process running as it was prior to being attached. Second, a new executable file is brought into the process object. The source and disassembly windows associated with the old executable file are removed.

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

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

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

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

```
(CXdb) run &
Command [#16] backgrounded

Beginning execution of Process [#0]
```

The above command creates a new process running under CXdb. The image of this process is associated with the process object. The & runs the command in the background. Running a process execution command in the background causes the CXdb command prompt to return, allowing you to enter other CXdb commands.

```
(CXdb) info process

status of process [#0]:

       executable: b.out
        arguments: (none)
fixed scheduling: off
           pshell: csh
     image status: created pid 17981, state = running
      working dir: /mnt/jones/project
     default step: statement
default language: Fortran
          threads: 1
   current thread: 0

source file search path:
```

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

Related Commands

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

Related Concepts

breakpoints
environment
eventpoints
eventpoint handlers
search path
tracepoints
watchpoints

Related Parameters

process-list
thread-list

process working directory

Description

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

The following commands work with the process working directory:

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

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

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

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

Examples

The following commands use the process working directory.

```
(CXdb) set directory /mnt/jones/projects
```

This command sets the process working directory to be the `/mnt/jones/projects` directory.

```
(CXdb) info process

status of process [#0]:

       executable: a.out
        arguments: (none)
fixed scheduling: off
           pshell: csh
     image status: no image
      working dir: (default to current CXdb directory)
     default step: statement
default language: Fortran

source file search path:
        .
```

The above command displays information about the current process object. The setting of the process working directory is shown to be the current console working directory.

Related Commands

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

Related Concepts

console working directory
default search path
search path

Related Parameters

directory-specifier

scope

Description

The scope of a program identifier determines where that identifier is visible. There are two concepts involving scope in CXdb.

- **Current scope** — The current scope is used to find identifiers that do not have a complete scope path. The current scope is determined by the current program counter (PC). Usually the PC is in the current frame, but by using the `frame` command you can change the current frame, thus also changing the current scope.
- **Scope path** — A scope path is a complete path to the declaration of a program identifier. Scope paths enable you to reference program identifiers that are not currently visible, such as static variables, shadowed variables, and global variables. Scope paths also enable you to reference variables in other namespaces, such as loader symbols or debugger variables.

A scope path can be used to access several different namespaces. Namespaces are the storage allocations given to the different types of language identifiers CXdb can recognize. The possible namespaces are:

- debugger variables (scope path is `cxdb$ or $`)
- loader symbols (scope path is `l$, or asm$`)
- FORTRAN (scope path is explained below)
- C (scope path is explained below)

In FORTRAN, scope paths enable you to reference common block identifiers from any point in the program. This allows you to view the data of a common block from different perspectives.

In C, scope paths enable you to reference static identifiers and identifiers that are shadowed. An identifier is shadowed when another identifier with the same name is declared in an inner block of the same scope.

Because the scope rules are different for FORTRAN and C, so are their scope paths.

In FORTRAN:

`f$`*<routine-name>*`` ` ``*<identifier>*

- *<routine-name>* — The routine in which the identifier is declared. If the routine name is omitted, the current scope is used to find the identifier.

In C:

c$<*file-name*>'<*routine-name*>'<*block-list*>'<*identifier*>

- <*file-name*> — The name of the file in which the identifier is declared. If this is omitted, the current source file is used.
- <*routine-name*> — The name of the routine in which the identifier is declared. If this is omitted, the current routine is used or, if a file-name has been specified, the global declarations of that file.
- <*block-list*> — A list of block numbers specifying the block in which the identifier is declared. A block is anything inside of braces. Unless you have compiled your program using the -pcc option, block numbers are not assigned to blocks in which variables are not declared. There can be one or more blocks in a block list, corresponding to the nesting of the blocks.

The first block, inside the routine, is block 1. Subsequent blocks have incremental block numbers. Each new level of blocks begins over at 1. The following C pseudocode uses identifiers named block to help demonstrate block numbers.

```
              main()
 |----------{
 |              static float block = 0;
 |       |------{
 |       |          static float block = 1;
 |       |    |-----{
 |       |    1         static float block = 1.1;
 |       1    |-----}
 |       |    |-----{
 |       |    2         static float block = 1.2;
 |       |    |-----}
main     |------}
 |              {    /* block ignored unless -pcc option used */
 |              }
 |       |------{
 |       2          static float block = 2;
 |       |------}
 |       |------{
 |       3          static float block = 3;
 |       |------}
 |              execution_stopped_here();
 |----------}
```

Assuming execution was stopped at the end of the routine, the following scope paths print the different identifiers named block:

```
print block – 0
print `1`block – 1
print `1`1`block – 1.1
print `1`2`block – 1.2
print `2`block – 2
print `3`block – 3
```

The current scope from the end of the routine is the main routine. Thus, no scope path is needed to reach the identifier with a value of zero.

By starting a scope path with a backquote, you are asking CXdb to find the identifier within the current routine. In the above commands, the current scope was the main routine. Thus, all of the scope paths that start with a backquote begin with an implied `c$prog`main`.

Examples

The following examples demonstrate the use of scope paths; first for FORTRAN and then for C.

Consider the following FORTRAN program.

```
PROGRAM PROG
INTEGER I,A,B
COMMON /B1/ A,B

DO I=1,10
   A=1111
   B=8888
   CALL SUB1(I)
ENDDO
END

SUBROUTINE SUB1(I)
INTEGER I
INTEGER C,D
COMMON /B1/ C,D

C=.2222
D=.9999
END
```

Assume that program execution has been stopped at the call to the function `SUB1` after five iterations of the `DO` loop. The current scope is based from frame 0, the current point of execution.

```
(CXdb) info scope
Process [#0/0], frame 0 scope: f$PROG
(CXdb) print A
INTEGER*4 1111
(CXdb) print B
INTEGER*4 8888
(CXdb) print SUB1'I
Variable has not been assigned storage: line: 1 col: 7
(CXdb) print SUB1'C
INTEGER*4 1111
(CXdb) print SUB1'D
INTEGER*4 8888
```

The above example prints out the values of the identifiers. Because `C` and `D` are in the same `COMMON` block as `A` and `B`, they really represent the same identifier location. Thus, they can be referenced in the subroutine because they share the same storage as `A` and `B`. The parameter to the subroutine cannot be referenced because it is only assigned storage when the subroutine is executing.

For the next series of examples, assume that process execution is stopped just before the end of `SUB1`.

```
(CXdb) info scope
Process [#0/0], frame 0 scope: f$SUB1
(CXdb) print C
INTEGER*4 2222
(CXdb) print D
INTEGER*4 9999
(CXdb) print PROG'A
INTEGER*4 2222
(CXdb) print PROG'B
INTEGER84 9999
```

The above examples print out the values of the various identifiers. Again, because `A` and `B` represent the same memory storage as `C` and `D`, they can be referenced from the subroutine.

The next series of examples use the following C program, made up of two source files. The first source file is `prog.c`:

```
#include <stdio.h>
extern void sub1();
int x = 0;

main()
{
   int i ;
   int x = 1111;

   for (i=1; i<=20; i++)
   {
       int x = 2222 ;
       if (i < 10)
           if (i < 6)
           {
              int x = 3333 ;
           }
           else
           {
              int x = 4444 ;
              sub1() ;
           }
   }
   printf("Finished\n");
}
```

The second source file is `sub.c`:

```
extern int x;

void sub1()
{
   static int x = 5555 ;
}
```

CXdb scope paths allow you to access the different identifiers all named `a`. Assume that execution has stopped just before the call to the function `sub1`.

```
(CXdb) info scope
Process [#0/0], frame 0 scope: c$prog'prog'main'1'2
(CXdb) print x
int 4444
(CXdb) print prog'x
int 0
(CXdb) print prog'main'x
int 1111
(CXdb) print prog'main'1'x
int 2222
(CXdb) print prog'main'1'1'x
int 3333
```

The above examples reference the different identifiers named `x`. The current scope is found using the `info scope` command.

The current scope is shown to be the block in which `x` is set to 4444. The first command does not use a scope path, so `x` is found in the current scope.

Now assume that execution has continued to just before the `printf` statement at the end of the program.

```
(CXdb) info scope
Process [#0/0], frame 0 scope: c$prog'prog'main
(CXdb) print x
int 1111
(CXdb) print c$prog'main'1'2'x
int 4444
(CXdb) print sub'x
int 0
(CXdb) print sub'sub1'x
int 5555
```

The above example again print the values of the identifiers named `x`. The current scope is the main routine. The global identifier `x` is referenced in the file `sub.c`. The local identifier of the subroutine `sub1`, located in `sub.c`, can be referenced because it was declared as a static identifier.

Related Commands

`evaluate`	`frame`
`info scope`	`print`

Related Concepts

source units

search path

Description

The search path is a list of directories CXdb searches when it is looking for source files and compiler-generated data files. Initially, the search path for each process object is set to the default search path plus the process working directory.

The following commands manipulate the search path of a process object.

- `add path` — Adds directories to the search path.
- `info process` — Displays information about the current state of the process object, including the search path.
- `remove path` — Remove directories from the search path.
- `set path` — Sets the search path to the specified directories.

Examples

The following examples use the search path commands.

```
(CXdb) info process
Current status of Process [#0]:
        .
        .
        .
Source file search path:
         .
        /mnt/jones
        /mnt/jones/project
```

The above example displays the status of the current process object. The response shows that the search path is initially set to the default search path directory and the process working directory.

```
(CXdb) add path /mnt/jones/libraries , /mnt/jones/math/libraries
Search path:
        .
        /mnt/jones
        /mnt/jones/project
        /mnt/jones/libraries
        /mnt/jones/math/libraries
```

The above example adds two directories to the search path. The next time CXdb is searching for a source file it will use the new search path which now includes the `/mnt/jones/libraries` and `/mnt/jones/math/libraries` directories.

```
(CXdb) remove path /mnt/jones/math/libraries
Search path:
        .
        /mnt/jones
        /mnt/jones/project
        /mnt/jones/libraries
```

The above command removes the `/mnt/jones/math/libraries` directory from the search path. The other directories in the search path remain.

```
(CXdb) set path /mnt/jones/project2 , /mnt/jones/project2/source
Search path:
        /mnt/jones/project2
        /mnt/jones/project2/source
```

The above command removes all the existing directories from the search path and sets it to the two listed directories.

Related Commands

`add default path`
`add path`
`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

Related Parameters

directory-specifier

signals

Description

Signals sent to an executing process can be controlled through several CXdb commands. Eventpoints can be set to watch for the receipt of a particular signal. The actions to be taken when CXdb catches a signal can be set for each different signal. The signal received by a process can be specified.

CXdb catches all signals sent to the process before the process ever receives them, unless the signal was sent by CXdb itself. When CXdb catches a signal, the signal number for that signal is placed in the debugger variable `$signal`.

If an eventpoint is triggered by a signal, the commands of the eventpoint handler for that eventpoint are executed. If an eventpoint is not triggered, the actions CXdb takes after a signal is caught are described below.

- **stop** —Stop process execution. If stop is not set, process execution is not stopped.
- **print** — Print a message telling what signal was caught. If print is not set, no message is printed.
- **pass** — Pass the value of the pre-defined debugger variable `$signal` to the process when process execution resumes. By modifying the value stored in `$signal`, you can change what signal is passed to the process. If pass is not set, no signal is passed to the process.

The above actions are independent of one another, and can be set or unset for each signal. The signals are briefly described below. The number in parentheses is the signal number for that signal.

- `SIGHUP(1)` — Hangup
- `SIGINT(2)` — Interrupt
- `SIGQUIT(3)` — Quit
- `SIGILL(4)` — Illegal instruction
- `SIGTRAP(5)` — Trace/Breakpoint trap
- `SIGIOT(6)` — IOT trap
- `SIGEMT(7)` — EMT trap
- `SIGFPE(8)` — Floating point exception
- `SIGKILL(9)` — Killed

- `SIGBUS(10)` — Bus error
- `SIGSEGV(11)` — Segmentation violation
- `SIGSYS(12)` — Bad system call
- `SIGPIPE(13)` — Broken pipe
- `SIGALRM(14)` — Alarm clock
- `SIGTERM(15)` — Terminated
- `SIGURG(16)` — Urgent I/O condition
- `SIGTSTP(17)` — Stopped (terminal)
- `SIGSTOP(18)` — Stopped
- `SIGCONT(19)` — Continued
- `SIGCHLD(20)` — Child exited
- `SIGTTIN(21)` — Stopped (tty input)
- `SIGTTOU(22)` — Stopped (tty output)
- `SIGIO(23)` — I/O possible
- `SIGXCPU(24)` — CPU time limit exceeded
- `SIGXFSZ(25)` — Filesize limit exceeded
- `SIGVTALRM(26)` — Virtual timer expired
- `SIGPROF(27)` — Profiling timer expired
- `SIGWINCH(28)` — Window size change
- `SIGLOST(29)` — Resource lost
- `SIGUSR1(30)` — User-defined signal 1
- `SIGUSR2(31)` — User-defined signal 2

Signal 0 is used to indicate that no signal should be sent to the process.

Signal names are not case sensitive.

The following commands work with signals:

- `info signal` — Displays the current actions for a signal or all signals.
- `event signal` — Sets an eventpoint for a signal.
- `set signal` — Sets the actions for a signal.
- `signal process` — Sends a signal to the specified process.
- `signal thread` — Sends a signal to the specified thread.

Examples

The following commands control the effect of the `SIGINT` signal on the process. For these examples, assume that the process is currently stopped.

```
(CXdb) info signal SIGINT

The current signal actions are:

Signal                          Signal
number  Stop  Pass  Print  name
------  ----  ----  -----  ----------------------
2        Yes    No   Yes    Interrupt
```

The above command displays the current settings for the actions to take when CXdb catches the `SIGINT` signal. Initially CXdb will stop the process, not pass the signal, and print a message when the signal is caught.

```
(CXdb) event signal SIGINT {eval $signal = 0; resume;}

#1: signal 2 on [#0], Enabled, ignore 0/0
  {
      eval $signal=0;
      resume;
  }
```

The above command sets an eventpoint watching for the `SIGINT` signal. The eventpoint handler sets the debugger variable `$signal` to zero and then resumes execution. Because the value of `$signal` is zero, when execution resumes the signal is not sent to the process. In effect, this handler causes the `SIGINT` signal to be ignored.

```
(CXdb) set signal INT nostop print nopass
```

The above command changes the default settings for the `SIGINT` signal. When CXdb catches the signal, a message is printed saying the signal has been caught. Because the stop action is not set, process execution continues. However, the signal is not passed to the process. This has the same effect as the previously set eventpoint; the `SIGINT` signal is ignored.

```
(CXdb) signal process 2
Resuming execution of Process [#0] with signal 2
```

The above command causes process execution to continue with the `SIGINT` signal (signal number 2) immediately being sent to the process. Because this signal is generated by CXdb, the signal is not caught by CXdb.

Related Commands

`info signal`
`set signal`
`signal thread`
`event signal`
`signal process`

Related Parameters

process-list
thread-list
signal-specifier

source units

Description

Source units are syntactic units of source code. There are five granularities of source units:

- `expression` — Any valid combination of constants, operators, and operands.
- `statement` — A combination of expressions that constitutes a complete instruction in the source language.
- `block` — The statements that make up the body of a routine, a loop, or a conditional construct.
- `loop` — A special type of statement that encloses a block.
- `routine` — A subroutine or function.

Source units let you control the granularity used in analyzing your source code. You can specify source units for stepping as well as for setting breakpoints, eventpoints, and tracepoints. Source units are essential for debugging optimized code because optimized code often does not execute in the same sequence as the original source statements.

The compiler assigns an identification number to each source unit when you compile your program with the `-cxdb` option. These source unit numbers are unique within a single source file. To list the identification numbers of all source units on a given line of source code, use the `info line` command.

Examples

The following two examples present the same section of source code written in both FORTRAN and C. This will help you compare source units in the two languages.

First consider the following section of FORTRAN source code:

```
1       SUBROUTINE PRIME(N)
2       INTEGER A(1000)
3
4       DO J = 2, N
5         IF (A(J) .EQ. 1)
6           K = J*2
7           DO WHILE ((K .LE. N).AND.(K .LE. 1000))
8             A(K) = 0
9             K = K + J
10          ENDDO
11        ENDIF
12      ENDDO
13      RETURN
14      END
```

In the above code, there is only one routine source unit, and `PRIME` is the name of the routine represented by this source unit. This source unit consists of everything from the `SUBROUTINE` statement on line 1 up to and including the `END` statement on line 14.

There are two loop source units in the above example: a `DO` loop and a `DO WHILE` loop. They consist of the following:

- `DO` loop — Lines 4 through 12, inclusive.
- `DO WHILE` loop — Lines 7 through 10, inclusive.

There are four block source units in the above example. The first is a routine block, the second is a `DO` block, the third is an `IF` block, and the fourth is a `DO WHILE` block. These blocks consist of the following statements:

- Routine block — Lines 2 through 14, inclusive.
- `DO` block — Lines 5 through 11, inclusive.
- `IF` block — Lines 6 through 10, inclusive.
- `DO WHILE` block — Lines 8 and 9.

The statement source units in the above example are:

- Line 4 — `DO J = 2, N`
- Line 4 — `J = 2`
- Line 5 — `IF (A(J) .EQ. 1) THEN`
- Line 6 — `K = J*2`
- Line 7 — `DO WHILE((K .LE. N).AND.(K .LE. 1000))`
- Line 8 — `A(K) = 0`
- Line 9 — `K = K + J`
- Line 13 — `RETURN`
- Line 14 — `END`

Notice that a statement must contain some executable code in order to be counted as a source unit. Thus, statements such as `SUBROUTINE`, `INTEGER`, `ENDIF`, and `ENDDO` are not considered to be source units. Neither are comment lines or blank lines.

Also notice that the `DO` and `DO WHILE` statements are both statement source units and loop source units at the same time.

The expression source units in the above example are:

- Line 4:
  ```
  N
  2
  ```
- Line 5:
  ```
  A(J) .EQ. 1
  A(J)
  J
  1
  ```
- Line 6:
  ```
  J*2
  J
  2
  ```

- Line 7:

```
(K .LE. N).and.(K .LE. 1000)
(K .LE. 1000)
K .LE. 1000
(K .LE. N)
K .LE. N
1000
```

  `K` (first occurrence)

  `N`

  `K` (second occurrence)

- Line 8:

```
K
0
```

- Line 9:

```
K + J
K
J
```

An expression can be as small as a single variable or as large as a complete statement. Notice that different appearances of the same expression count as different source units. For example, the variable `K` appears twice in line 7 above, so it counts as two different expression source units for that line. As with the other types of source units, the source language determines what constitutes a valid expression.

Consider the same section of code written in C:

```
1       prime(n)
2       int n;
3       {
4         int j,k;
5
6         for (j=2; j<=n; j++)
7         {
8           if (a[j]==1)
9           {
10            k = j*2;
11            while ((k<=n) && (k<=1000))
12            {
13              a[k] = 0;
14              k = k+j;
15            }
16          }
17        }
18      }
```

In the above example, there is one routine source unit. It consists of lines 3 through 18, inclusive.

There are two loop source units in the above example: a `for` loop and a `while` loop. They consist of the following:

- `for` loop — Lines 6 through 17, inclusive.
- `while` loop — Lines 11 through 15, inclusive.

There are four block source units in the above example. The first is a routine block, the second is a `for` block, the third is an `if` block, and the fourth is a `while` block. These blocks consist of the following statements:

- routine block — Lines 3 through 18, inclusive.
- `for` block — Lines 7 through 17, inclusive.
- `if` block — Lines 9 through 16, inclusive.
- `while` block — Lines 12 and 15.

The statement source units in the above example are:

- Line 6 — `for (j=2; j<=n; j++)`
- Line 8 — `if (a[j]==1)`
- Line 10 — `k = j*2`
- Line 11 — `while ((k<=n) && (k<=1000))`
- Line 13 — `a[k] = 0`
- Line 14 — `k = k+j`

The expression source units in the above example are:

- Line 6:
```
j<=n
j++
j=2
j (first occurrence)
2
j (second occurrence)
n
j (third occurrence)
```
- Line 8:
```
a[j]==1
a[j]
a
j
1
```
- Line 10:
```
k = j*2
j*2
k
2
j
```
- Line 11:
```
(k<=n) && (k<=1000)
(k<=1000)
k<=1000
(k<=n)
k<=n
1000
k (first occurrence)
n
k (second occurrence)
```

- Line 13:
  ```
  a[k] = 0
  a[k]
  a
  k
  0
  ```
- Line 14:
  ```
  k = k+j
  k+j
  k (first occurrence)
  k (second occurrence)
  j
  ```

Related Commands

```
break source            clear step
event reached source    info line
info sourceunit         next
next over               set default step
set step                step
step over               trace source
```

Related Concepts

breakpoints, eventpoints, stepping, tracepoints, windows

Related Parameters

granularity, source-unit

stepping

Description

Stepping is the incremental execution of a program. CXdb provides a sophisticated set of stepping commands that allow you to control not only the number of steps to execute but also the size of each step.

In the broadest sense, stepping can be done in one of two ways: by machine instructions or by source units. The commands for stepping by machine instruction are:

- `next instruction` — Execute the specified number of machine instructions. Do not count instructions within a called routine as part of the specified number.
- `step instruction` — Execute the specified number of machine instructions. Count instructions within a called routine as part of the specified number.

The commands for stepping by source units allow you to specify the source unit granularity, or step size, as well as the number of steps. These commands are:

- `next` — Continue executing the process until it reaches the next source unit of the specified granularity. Do not count the source units within a called routine.
- `step` — Continue executing the process until it reaches the next source unit of the specified granularity. Count the source units within a called routine.
- `finish` — Finish executing the innermost active source unit of the specified granularity. Stop execution at the next source unit of default granularity.
- `next over` — Complete execution of the current source unit of the specified granularity. Stop execution at the next source unit of default granularity. Do not count the source units within a called routine.
- `step over` — Complete execution of the current source unit of the specified granularity. Stop execution at the next source unit of default granularity. Count the source units within a called routine.

The source unit granularities are:

```
routine
loop
block
statement
expression
```

If you do not specify the granularity for a particular stepping command, CXdb uses the default granularity. Initially the default granularity is statement, but you can modify this with the following commands:

- `set default step` — Set the default granularity (or step size) for all new process objects.
- `set step` — Set the default granularity (or step size) for an existing process object.

As a general rule, you will probably want to use a larger stepping granularity when the point of execution is far away from a problem area of the program and a finer granularity as the point of execution approaches the problem area.

The current source unit starts at the address indicated by the current value of the program counter (PC). Several source units of different granularities might all start at the same location. Therefore, all of these source units can be current at the same time. That is why it is important for you to specify the granularity of the particular current source unit you want.

The innermost active source unit is the one of specified granularity whose address range (or extent) includes the current value of the PC. If you start at the location indicated by the current PC and look backward through the code, the innermost active source unit is the first one of the specified granularity that you encounter. In other words, it is the innermost source unit of specified granularity that encloses or contains the location indicated by the current PC.

Some examples might help to clarify the difference between the current source unit and the innermost active one. Consider the following pseudocode:

```
start Loop 1
    Statement 5
    start Loop 2
        Statement 10
        Statement 11
    end Loop 2
    Statement 15
end Loop 1
```

If the PC is pointing to Statement 11 in the above pseudocode, then the innermost active loop is Loop 2, and the innermost active block is the block that includes Statement 10 and Statement 11. However, the current source unit is Statement 11 because the PC is pointing to it. Even thought Loop 2 is active, it is not current because the PC does not point directly at the beginning of Loop 2.

With the PC pointing at Statement 11, Loop 2 and its included block are considered active because they contain the location indicated by the current value of the PC. Loop 1 is also active, but it is not the *innermost* active loop in this case.

If the PC is pointing to Statement 5 in the above pseudocode, then Loop 1 is the innermost active loop and Statement 5 is the current source unit. If the PC is pointing directly at the start of Loop 2, then Loop 2 is both the current loop source unit and the innermost active loop source unit.

One final and most important point: stepping is a dynamic activity. It proceeds according to the order of execution of the object code, not according to the order of the source code. When CXdb is searching for the particular source unit you want to step to, it checks each source unit individually before executing it. Therefore, if conditional constructs in the source code cause execution to skip over the source unit you want, or if optimization has eliminated the desired source unit, then stepping cannot take you to that source unit.

Examples

The stepping examples shown below relate to the following FORTRAN source code:

```
1        PROGRAM EXAMPLE
2        PRINT *, "The example program has started."
3        DO I = 1, 10
4          PRINT 99, "I = ", I
5          CALL SUBA(I)
6        ENDDO
7        PRINT *, "The example program is done."
8     99 FORMAT (A,I2)
9        END
10
11       SUBROUTINE SUBA(N)
12       INTEGER N
13       PRINT 98, "Subroutine SUBA has started. The value of N is ", N
14       DO K = 1, N
15         PRINT 98, "K = ", K
16         IF (K .LE. 5) THEN
17           DO L = 1, N
18             PRINT 98, "L = ", L
19           ENDDO
20           PRINT 98, "The loop for L is done, with L = ", L
21           IF (L .LE. 5) THEN
22             DO M = 1, N
23               PRINT 98, "M = ", M
24             ENDDO
25             PRINT 98, "The loop for M is done, with M = ", M
26           ENDIF
27         ENDIF
28       ENDDO
29       PRINT 98, "Subroutine SUBA is done. The value of K is ", K
30       RETURN
31    98 FORMAT (A,I2)
32       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 2.

Enter the following command:

```
(CXdb) step
Stepping process [#0/*] by 1 statement
Process [#0/0] stopped stepping at [0x80001368] EXAMPLE in example.f line 3
```

Because statement is the default granularity, the above command steps the current process by one statement. The PC now points to the beginning of line 3, which is the beginning of a `DO` loop.

```
(CXdb) step 2
Stepping process [#0/*] by 2 statements
Process [#0/0] stopped stepping at [0x800013a2] EXAMPLE in example.f line 5
```

The above command steps the current process by two statements. The PC points to the beginning of line 5, which is a call to a subroutine.

```
(CXdb) next
Nexting process [#0/*] by 1 statement
Process [#0/0] stopped stepping at [0x80001372] EXAMPLE in example.f line 4
```

The above command again steps the process by one statement. However, before the command executed, the PC was at the beginning of line 5, which is a subroutine call. The `next` command ignores all source units in a called subroutine. Therefore, the process continues to execute until it reaches the next statement after returning from the subroutine. When the process stops, the PC is pointing to the beginning of line 4.

In the above example, note that the process does not stop again at the `DO` statement on line 3 because the `DO` loop is already active. This is just another iteration of the loop, so there is no new statement to be executed at line 3. Therefore, the process does not stop in this case until it reaches line 4.

At this point, you enter the following command:

```
(CXdb) step loop 2
Stepping process [#0/*] by 2 loops
Process [#0/0] stopped stepping at [0x800014c2] SUBA in example.f line 17
```

The above command steps the process by two loops. Because the step command does look at source units in called routines, this command continues execution of the process until it reaches the second `DO` loop in subroutine `SUBA`. When the process stops, the PC is pointing to the beginning of line 17.

```
(CXdb) step over loop
Stepping process [#0/*] by 1 loop
Process [#0/0] stopped stepping at [0x80001540] SUBA in example.f line 20
```

The above command completes execution of the current loop that begins on line 17. Since the default granularity is statement, execution stops at the next statement after the loop. The PC now points to the beginning of line 20.

```
(CXdb) finish loop
Finishing innermost loop in Process [#0/*]
Process [#0/0] stopped stepping at [0x8000166c] SUBA in example.f line 29
```

The above command completes execution of the innermost active loop. When the PC is pointing to line 20, the innermost active loop is the `DO` loop that begins on line 14. The above command completes execution of this entire loop and stops the process at the first default source unit (statement) after the loop. Therefore, when the process stops, the PC is points to the beginning of line 29.

Related Commands

`finish`
`info cxdb`
`info line`
`info process`
`info sourceunit`
`next`
`next instruction`
`next over`
`set default step`
`set step`
`step`
`step instruction`
`step over`

Related Concepts

process object

source units

Related Parameters

granularity

synthesized variables

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.

The `info expression` command also lists the reason for the use of each synthesized variable. The reasons are abbreviated to acronyms, which are defined as follows:

- `ALTE` — Alternate entry point of a loop that executes conditionally.
- `BITB` — Bit bucket storage.
- `BOOT` — Storage of a value removed from a scalar register.
- `BSS` — Starting address of BSS memory region.
- `CMIN` — Index used to find the minimum element of an array by pattern matching.
- `CREG` — Communication register storage.
- `CTMP` — Call temporary, used for temporary storage of arguments that are passed by value to a subroutine.
- `DATA` — Starting address of DATA memory region.
- `DEAD` — Loop induction variable whose current value is impossible to calculate. No synthesized variable is reported in this case.

- `DEXP` — Expression that has been distributed over several optimized loops.
- `FLNK` — Forward link to a constant value that is propagated to the distributends of a vectorized loop.
- `INDV` — Loop induction variable that has undergone strength reduction.
- `ISTR` — Inner strip counter of a strip-mined loop.
- `MLXS` — Subscript of a multi-dimensional array that has been transposed into a one-dimensional array during vectorization.
- `OSTR` — Outer strip counter of a strip-mined loop.
- `PBKE` — Loop-invariant expression.
- `PBKU` — Loop-invariant constant.
- `REXP` — Reduced subexpression that is hoisted out of a loop.
- `SEXP` — Subscript expansion that folds the subscripts of a multi-dimensional array into one subscript.
- `SINK` — Sink variable that replaces the original induction variable when the loop iteration count is too small to warrant strip mining.
- `SPLL` — Pointer to the spill area for scalar registers.
- `STML` — Strip mine length.
- `TBSS` — Starting address of TBSS memory region.
- `TDATA` — Starting address of TDATA memory region.
- `TEXT` — Starting address of TEXT memory region.
- `TPTR` — Temporary pointer, used for temporary storage of arguments that are passed by reference to a subroutine.
- `TRIP` — Trip count used to replace a more complex loop iteration quantity.
- `UREX` — Expression from an unrolled loop.
- `URIV` — Induction variable of an unrolled loop.
- `UTRP` — Trip counter of an unrolled loop.
- `VBOT` — Storage of a value removed from a vector register.
- `VMSK` — Vector mask storage.
- `VSPL` — Pointer to the spill area for vector registers.
- `ZMSK` — Zero mask storage.

You can use a synthesized variable in any *<language-expression>*, in the same way you would use a program variable. However, in most cases, you will 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 J
object type: Fortran identifier
   location: <none>
       size: 4 bytes
       type: INTEGER*4
      value: 4
       used to create 1 synthesized variable(s):
         1. <INDV>    ?i7 = ?i1+((4*N)*(J-1))
```

The above command displays information about the program variable `J`. The response indicates that the current value of `J` is 4. This value is not stored (location = `<none>`) because the synthesized variable `?i7` replaces the use of `J`. The reason for the replacement is `INDV`, which means the induction variable has undergone strength reduction. The equation used to generate the synthesized variable is `?i7=?i1+((4*N)*(J-1))`.

In the source code, `J` is a loop induction variable that is used as an index to reference specific elements of an array. In the object file, `?i7` serves as a pointer to the array elements. The compiler replaces `J` with `?i7` because it is more efficient to increment the pointer than it is to increment `J` 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 `J` by solving for it in the equation shown for `?i7`.

```
(CXdb) info expression \?i7
object type: Fortran identifier
   location: register a2
       size: 4 bytes
       type: INTEGER*4
      value: -2147176320
       Reason: Loop induction variable
       created from 1 equation(s):
         1. <INDV>  ?i1+((4*N)*(J-1))
       2 liveness ranges:
             Start        End          Location
          1. 0x8000172e:0x80001750 - register a2
          2. 0x80001750:0x80001754 - register a2
```

The above command displays information about the synthesized variable `?i7`. The response shows the equation that the compiler uses to generate `?i7`. It also shows the liveness ranges and corresponding storage locations for the variable. The reason for generating `?i7` is that it replaces a loop induction variable.

```
(CXdb) print/x \?i7
INTEGER*4) 0x8004b080
```

The above command prints the current value of the synthesized variable `?i7` in hexadecimal format. Because `?i7` is a pointer to an array in this case, the current value of `?i7` is the starting address of the next array element to be accessed.

Related Commands

`evaluate` `info expression` `print`

Related Concepts

debugger variables language expressions

Related Parameters

language-expression synthesized-variable

tracepoints

Description

A tracepoint is a pre-defined eventpoint, or trap, that you place in your executable code. When process execution reaches the location of an enabled tracepoint, the set of actions associated with the tracepoint, called the eventpoint handler, are taken. Tracepoints allow you to trace the execution of your process past key locations in your program.

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

All tracepoints have a default handler that prints a message telling you that the tracepoint has been reached and then resumes process execution. You may specify a different set of actions to take for a particular tracepoint, or change the setting of the default handler itself.

Tracepoints, like all eventpoints, can be enabled or disabled. When process execution reaches the address of an enabled tracepoint, the tracepoint is said to be reached. A disabled tracepoint is treated as if it does not exist, and therefore can never be reached unless it is enabled again. The disabling of tracepoints allows you to prevent tracepoints being reached without having to completely remove them from the process object.

Once a tracepoint is reached, one of two actions can occur. If the tracepoint has an ignore count, the counter is incremented by one and process execution continues. If the tracepoint does not have an ignore count, then the eventpoint is said to be triggered. When a tracepoint is triggered, the commands in its eventpoint handler are executed. If the tracepoint does not have its own eventpoint handler, the default eventpoint handler for tracepoints is used.

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

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

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

There are several different ways to set a tracepoint. Tracepoint commands are described below:

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

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

Several commands allow you to interact with existing tracepoints. These commands are described below:

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

Examples

The following series of examples create and manipulate several different tracepoints in one process object.

```
(CXdb) trace line 60

#0: trace line, on [#0/*], Enabled, ignore 0/0
        [0x80001620] BESTMV in pickup.f line 60
```

The above command sets a tracepoint at line 60 in the current source file. The eventpoint number for this tracepoint is 0, and the address of the tracepoint is `80001620` in routine `BESTMV` at line 60 in the source file `pickup.f`.

```
(CXdb) trace routine BESTMV

#1: trace routine, on [#0/*], Enabled, ignore 0/0
        [0x800015f2] BESTMV in pickup.f line 59
```

The above command sets a tracepoint at the first executable source unit in the routine containing the address of the routine `BESTMV`. The routine `BESTMV` contains its own address, so the tracepoint is set at the first executable source unit of the routine `BESTMV`. The first executable source unit of a routine is usually the first statement of a routine after local variables have been declared unless there are local initializations.

```
(CXdb) trace instruction BESTMV

#2: trace routine, on [#0/*], Enabled, ignore 0/0
        [0x800015f0] BESTMV in pickup.f line 55
```

The above command sets a tracepoint at the first address of `BESTMV`. The first address of a routine begins the preamble of the routine. The preamble of a routine manages the stack for that routine. This address is different than that of the previous example, because tracepoint 1 is at the first source unit of `BESTMV`.

```
(CXdb) trace source 135

#3: trace source, on [#0/*], Enabled, ignore 0/0
        [0x800016f0] BESTMV in pickup.f line 65
```

The above command sets a tracepoint at the starting address of source unit 135. The source unit numbers of a given line may be found using the `info line` command.

```
(CXdb) info trace
Event   Enabled     Ignore     proc/td      Address         Where
#0        y          0/0         0/*      [0x80001620] BESTMV in pickup.f line 60
#1        y          0/0         0/*      [0x800015f2] BESTMV in pickup.f line 59
#2        y          0/0         0/*      [0x800015f0] BESTMV in pickup.f line 55
#3        y          0/0         0/*      [0x800016f0] BESTMV in pickup.f line 65
```

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

```
(CXdb) run
Beginning execution of Process [#0]
Process [#0/0] hit Tracepoint 2 at [0x800015f0] BESTMV in pickup.f line 55
Process [#0/0] hit Tracepoint 1 at [0x800015f2] BESTMV in pickup.f line 59
Process [#0/0] hit Tracepoint 0 at [0x80001620] BESTMV in pickup.f line 60
Process [#0/0] hit Tracepoint 3 at [0x800016f0] BESTMV in pickup.f line 65

Process [#0] exited normally.
```

The above example begins process execution without passing any arguments to the process. When execution reaches the address of `800015f0`, tracepoint 2 is reached because it is enabled. Because it does not have an ignore count, the tracepoint is triggered. Because the tracepoint did not have its own eventpoint handler, the default handler for tracepoints is executed, which prints a message and then resumes execution. Each of the tracepoints is triggered in turn. For this example, assume that the program runs to completion.

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

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

```
(CXdb) trace line 60 $Middle

#4: trace line, on [#0/*], Enabled, ignore 1/2
        [0x80001620] BESTMV in pickup.f line 60

INFO: Eventpoint 0 also has a tracepoint at address 0x80001620.
```

The above command sets another tracepoint at line 60 of the current source file. The debugger variable `$Middle` is created and assigned to this eventpoint. CXdb informs you that two eventpoints now reside at the same address location.

```
(CXdb) trace line 60 {echo "Tracepoint 5 reached" ;}

#5: trace line, on [#0/*], Disabled, ignore 0/0
        [0x80001620] BESTMV in pickup.f line 60
 {
       echo "Tracepoint 5 reached" ;
 }

INFO: Eventpoint 4 also has a tracepoint at address 0x80001620.
```

The above command sets a third tracepoint at line 60. An eventpoint handler is given to this eventpoint. The handler prints a message but does *not* resume execution of the process.

```
(CXdb) run
Beginning execution of process [#0]
Tracepoint 5 reached
```

The above example restarts execution of the process. Tracepoint 5 is triggered because it is the highest-numbered, enabled eventpoint at that location (the other two tracepoints at that location are 0 and 4), and it does not have an ignore count. The eventpoint handler for tracepoint 5 is used instead of the default handler for tracepoints. The handler for tracepoint 5 prints a message but does not resume process execution.

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

The above example does two things. First, tracepoint 5 is disabled. Second, tracepoint 4 is given an ignore count of 2.

```
(CXdb) run
Process [#0] is already running with pid 16139.
Terminate existing process and restart? y
Beginning execution of Process [#0]
Process [#0/0] hit Tracepoint 0 at [0x80001620] BESTMV in pickup.f line 60
```

In the above example, the `run` command causes CXdb to display a warning message indicating that the current process still exists. If you answer yes, CXdb kills the current process and begins execution of a new process. When process execution reaches address `80001620,` tracepoint 4 is reached because eventpoint 5 is disabled. Tracepoint 4 has an ignore count, so its counter is incremented by one. Because an eventpoint has still not been triggered, tracepoint 0 is reached. Because it is enabled and does not have an ignore count, it is triggered.

```
(CXdb) info event *
#0: trace line, on [#0/*], Enabled, ignore 0/0
        [0x80001620] BESTMV in pickup.f line 60
#3: trace source, on [#0/*], Enabled, ignore 0/0
        [0x800016f0] BESTMV in pickup.f line 65
#4: trace line, on [#0/*], Enabled, ignore 1/2
        [0x80001620] BESTMV in pickup.f line 60
#5: trace line, on [#0/*], Disabled, ignore 0/0
        [0x80001620] BESTMV in pickup.f line 60
 {
      echo "Tracepoint 5 reached" ;
 }
```

The above command displays the status of all eventpoints. Using the info event command, you can display the eventpoint handler of an eventpoint. From the output of this command you can also see that tracepoint 5 is disabled and that tracepoint 4 has been ignored once.

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

The above command enables tracepoint 5. This causes tracepoint 5 to be triggered the next time address `80001620` is reached because it is the highest-numbered, enabled tracepoint.

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

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

For more information about the use of eventpoint handlers see the reference page on eventpoint handlers.

Related Commands

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

Related Concepts

breakpoints
eventpoints
eventpoint handlers
watchpoints

Related Parameters

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

viewports

Description

Viewports are destinations for CXdb input, output, and error messages. A viewport may be either the CXdb command window or a file.

There are three types of viewports, each of which is capable of receiving a different type of information. The types are:

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

CXdb maintains separate viewport lists for cmderr, cmdlog, and cmdout. Each viewport on a list receives a copy of the designated information for its given type. For example, all the viewports for cmdout receive a copy of the output simultaneously.

The commands for modifying the viewport lists are:

- `add cmderr` — Add new viewports to the current list of cmderr viewports.
- `add cmdlog` — Add new viewports to the current list of cmdlog viewports.
- `add cmdout` — Add new viewports to the current list of cmdout viewports.
- `remove cmderr` — Remove viewports from the current list of cmderr viewports.
- `remove cmdlog` — Remove viewports from the current list of cmdlog viewports.
- `remove cmdout` — Remove viewports from the current list of cmdout viewports.
- `set cmderr` — Remove the current cmderr viewports and replace them with a new list of viewports.
- `set cmdlog` — Remove the current cmdlog viewports and replace them with a new list of viewports.
- `set cmdout` — Remove the current cmdout viewports and replace them with a new list of viewports.

The default viewport for cmderr and cmdout is the CXdb command window (Window #1). There is no default viewport for cmdlog because your entries in the command window are automatically echoed there.

For logging, most of the viewports you specify will be files. If the specified file does not exist, CXdb creates it. If the specified file already exists, then you can use the following commands to control whether or not CXdb writes (either overwrites or appends) to the existing file:

- `clear noclobber` — Allow writing (either overwriting or appending) to existing files.
- `set noclobber` — Respond with an error message if the specified file already exists.

The default for noclobber is clear (off).

For cmderr and cmdout, you can override the viewport list and redirect the response of an individual command by using redirection operators with that command. Each redirection operator allows you to specify a viewport list that applies only to the command with which it appears.

For cmdlog, you can enable and disable the viewports with the commands `set logging` and `clear logging`, respectively. The default is logging disabled (clear).

The command `info cxdb` displays the current settings of cmderr, cmdlog, cmdout, log, and noclobber.

Examples

The following examples illustrate how to modify and use viewport lists.

```
(CXdb) add cmdout cxdb.info
New cmdout: Window #1, cxdb.info
```

The above command adds a viewport to the list for cmdout. The response indicates that the new viewports for cmdout are Window #1 (the command window) and the file `cxdb.info`.

```
(CXdb) add cmderr cxdb.info, cxdb.err
New cmderr: Window #1, cxdb.info, cxdb.err
```

The above command adds two new viewports to the list for cmderr. The response indicates that the new viewports for cmderr are Window #1 (the command window), the file `cxdb.info`, and the file `cxdb.err`.

```
(CXdb) remove cmderr cxdb.info
New cmderr: Window #1, cxdb.err
```

The above command removes a viewport from the list for cmderr. The response indicates that the new viewports for cmderr are Window #1 (the command window) and the file `cxdb.err`.

```
(CXdb) set cmdout output_data
```

The above command deletes the current viewport list for cmdout and replaces it with a new list that contains the file name `output_data`. This command establishes the file `output_data` as the only viewport for cmdout. Notice that the command window is no longer one of the cmdout viewports, so responses to CXdb commands will not appear in the command window.

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

The above command adds Window #1 (the command window) to the viewport list for cmdout. The response indicates that the new viewports for cmdout are the file `output_data` and Window #1.

With each individual CXdb command, you can override the viewport lists by using redirection operators on the command line. For example, to redirect the output of the `info process` command, enter the following:

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

The above command line redirects the output of the `info process` command to the file called `process_status` instead of to the viewports for cmdout. However, the viewports for cmderr and cmdlog are not affected by the redirection in this case.

To keep a log of the commands you use during the debugging session, enter the following:

```
(CXdb) add cmdlog debug_script
New cmdlog: debug_script
(CXdb) set logging
```

The above response indicates that the new viewport for cmdlog is the file `debug_script`. The `set logging` command enables logging to this file. This type of log file is a good way to keep track of your actions during a debugging session. If you ever need to retrace your steps, you can use the `source` command to execute this log file as if it were a command file.

Notice that the above response does not list Window #1 (the command window) as one of the viewports for cmdlog. This is because everything you type in the command window is automatically echoed there. If you add Window #1 to cmdlog, then everything you type will appear twice in the command window.

Related Commands

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

Related Concepts

cmderr
cmdlog
cmdout
viewports
windows

Related Parameters

redirection-operator
viewport

watchpoints

Description

A watchpoint is a pre-defined eventpoint, or trap, that you set to watch a region of process memory. When the value stored in the watched address region changes, process execution stops, and the set of actions associated with the watchpoint, called the eventpoint handler, are taken. Watchpoints enable you to watch for a specific address, or address range, to change value.

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

All watchpoints have a default handler that displays a message telling you that the watchpoint's region has been modified. You may specify a different set of actions to take for a particular watchpoint, or change the setting of the default handler itself.

Watchpoints, like all eventpoints, can be enabled or disabled. When there is a change in the address region watched by an enabled watchpoint, the watchpoint is said to be reached. A disabled watchpoint is treated as if it does not exist, and therefore can never be reached unless it is enabled again. The disabling of watchpoints allows you to prevent them being reached without having to completely remove them from the process object.

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

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

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

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

The `watch` command creates all watchpoints. A process image must exist before you can create a watchpoint.

Several commands allow you to interact with existing watchpoints. These commands are described below:

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

Examples

The following series of examples create and manipulate several different watchpoints in one process object. The following FORTRAN program is used throughout the examples:

```
PROGRAM TEST
INTEGER A,B,C

A = 0
B = 0
C = 0
DO I=1,1000
    A=A+1
    DO J=1,1000
        B=B+1
        DO K= 1,1000
            C=C+1
        ENDDO
    ENDDO
ENDDO
END
```

For the following examples, assume that process execution has stopped at the beginning of the program.

```
(CXdb) watch loc(C)

#0: watch 0x80029010..0x80029013, on [#0/0], Enabled, ignore 0/0
```

The above command creates a watchpoint that monitors any changes to the FORTRAN variable `C`. The response from CXdb shows the current settings for the created watchpoint. The output is the same as if an `info event` command had been issued for this eventpoint.

Because the variable is a four-byte integer, the address range is from `80029010` to `80029013`. The eventpoint number is 3, it is currently enabled, and it does not have an ignore count.

```
(CXdb) info expression B

object type: Fortran identifier
   location: 0x8002900c
       size: 4 bytes
       type: INTEGER*4
      value: 0

(CXdb) watch '8002900c'x .. '8002900f'x

#1: watch 0x8002900c..0x8002900f, on [#0/0], Enabled, ignore 0/0
```

The above example shows another way to set a watchpoint. First the address of the variable `B` is obtained using the `info expression` command. Second, a watchpoint is created to watch the address range starting with the hexadecimal address of `8002900c` and ending with `8002900f`. The syntax for specifying a hexadecimal number is FORTRAN specific.

```
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] memory region 0x8002900c..0x8002900f modified
Process [#0/0] stopped by Watchpoint 1, at [0x80001392] TEST in test.f line 11
```

The above example resumes execution of the current process. The process is stopped when the address region watched by watchpoint 1 changes. The default handler for watchpoints, which displays the above messages, is executed. Process execution does not resume.

```
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] memory region 0x80029010..0x80029013 modified
Process [#0/0] stopped by Watchpoint 0, at [0x8000139c] TEST in test.f line 12
(CXdb) print C
(INTEGER*4) 1
(CXdb) set ignore 5 0
Event 0 will be ignored 5 times
```

The above example does several things. First, process execution is continued. The process is stopped by the watchpoint 0 because the contents of the variable `C` have changed. Second, the current value of `C` is printed. Finally, watchpoint 0 is given an ignore count of 5. The next 5 times the watchpoint is reached, it will not be triggered. When it is reached a sixth time, it is triggered.

```
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] memory region 0x80029010..0x80029013 modified
Process [#0/0] stopped by Watchpoint 0, at [0x8000139c] TEST in test.f line 12
(CXdb) print C
(INTEGER*4) 6
(CXdb) disable event 0
Event 0 disabled
```

The above example resumes process execution. Watchpoint 0 eventually stops the process. The print command shows that the watchpoint was ignored 5 times. Finally, watchpoint 0 is disabled. The watchpoint can no longer be reached.

```
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] memory region 0x8002900c..0x8002900f modified
Process [#0/0] stopped by Watchpoint 1, at [0x80001392] TEST in test.f line 11
```

The above command resumes process execution. Because watchpoint 0 is disabled, the process is stopped by watchpoint 1 when the value of the variable `B` changes.

```
(CXdb) info watch
Event     Enabled Ignore  proc/td         Region
#0           y     0/0    0/0     0x80029010          0x80029013
#1           n     0/0    0/0     0x8002900c          0x8002900f
```

The above command displays information about all existing watchpoints. The eventpoint number, enabled status, ignore count, process and thread number, and region of memory being watched is displayed for each watchpoint.

```
(CXdb) enable event 0
Event 0 enabled
(CXdb) remove event 1
Event 1 removed
```

The above example performs two actions. First, watchpoint 0 is enabled again, so it can be reached. Second, watchpoint 1 is completely removed from the process. It can not be brought back.

```
(CXdb) watch '80029010'x :4 \; {print C; disable event $self; resume;}

#2: watch 0x80029010..0x80029013, on [#0/0], Enabled, ignore 0/0
    {
       print C;
       disable event $self;
       resume;
    }
```

The above example creates a new watchpoint that watches the variable `C`. A count of the number of bytes to watch, including the starting address of `80029010`, is given. In addition, an eventpoint handler is defined for this watchpoint. When the watchpoint is triggered, the value of `C` is printed, and then the watchpoint disables itself by using the predefined debugger variable `$self`. Finally, process execution is resumed. This allows the other watchpoint at this location, watchpoint 0, to be triggered the next time through the loop.

```
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] memory region 0x80029010..0x80029013 modified
Process [#0/0] stopped by Watchpoint 2, at [0x8000139c] TEST in test.f line 12
1001
Event 2 disabled
Resuming execution of Process [#0/*]
Process [#0/0] memory region 0x80029010..0x80029013 modified
Process [#0/0] stopped by Watchpoint 0, at [0x8000139c] TEST in test.f line 12
```

The above command continues process execution. Watchpoint 2 is triggered because it is the highest-numbered, enabled eventpoint to be reached. The commands of its eventpoint handler execute, causing the value of `C` to be printed, the watchpoint to be disabled, and process execution to resume.

The next time through the loop, watchpoint 0 is triggered because it is now the highest-numbered, enabled eventpoint to be reached. Watchpoint 0 stops process execution.

Related Commands

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

Related Concepts

breakpoints
debugger variables
eventpoints
eventpoint handlers
process object
tracepoints

windows

Description

Many different windows are associated with CXdb. Each window provides a different view of the program being debugged.

You can use either the CXwindows interface or the Maryland Windows interface with CXdb. Although the look of the windows is different between the two interfaces, their functionality is very similar.

Some of the windows available in the CXwindows interface are not available in the Maryland Windows interface, due to the inherent screen size limitations.

The windows that are available in both interfaces are:

- Command window
- Source window
- Process interface window
- Help window
- Display file window

Under the CXwindows interface, the windows are maintained by your window manager (for complete information on the manipulation of windows in CXwindows, refer to the window manager documentation). If you are using the Motif window manager it must be version 2.1. Each window has a menu bar at its top that provides access to the functions for that window. In each case, the first menu item is the name of the window.

You can set the resource values of various window settings for CXdb windows running under the CXwindows interface. You can add these settings to your `.Xdefaults` file. For a description of CXdb Xdefaults settings, refer to the reference page "Xdefaults."

Besides the above mentioned windows, the following windows are available only in the CXwindows interface:

- Disassembly window
- Examine window
- Stack window

Under the Maryland Windows interface, the screen can be divided into multiple windows, much like an X windows display. Interaction with the windows is performed using special keystrokes. You can raise, lower, move, and resize each window, move between windows, scroll and select text from a window, and edit the text on the command line.

You can set the geometry specifications for the windows of the Maryland Windows interface when invoking CXdb at the shell prompt. For more infomation on setting geometry specifications for windows, refer to the reference page for the `cxdb` command.

For a full description of the default key bindings used in the Maryland Windows interface, refer to the reference page for Maryland Windows or the reference page on the `bind` command.

The CXdb windows are described below.

- **command window** — The primary window for entering commands in CXdb. The command window is the default viewport for command output and command error messages.
- **source window** — Displays the source code of the current source file. The source window highlights the innermost source units associated with the program counter when process execution stops. Symbols are used to mark the location of eventpoints in the source code. A source window is brought up when an executable file is specified, when the `display routine` command is used, when you open a source window from a menu option, or when a new thread is created.
- **process interface window** — The interaction window between you and your process. In CXwindows, the process interface window is a standard xterm window. In Maryland Windows, it is another subdivision of the screen.
- **`help window`** — Displays the reference page for any topic in this manual. The window is brought up using the `help` command.
- **`display file window`** — Displays the contents of a file. This window can display any ASCII text file for browsing only. The window is brought up using the `display file` command.

- **disassembly window** (CXwindows only) — Displays a region of disassembly instructions for a thread of the process. From the disassembly window you can view the scalar, vector, and communication register sets, as well as the settings of the process status word (PSW) register, through popup windows. Symbols are used to mark the location of eventpoints in the disassembly code displayed in the disassembly window.
- **examine window** (CXwindows only) — Displays a region of process memory for a thread of the process. You can select the memory unit (byte, half, word, long) and the memory format (such as octal, hexadecimal, decimal, or unsigned) used to display the region of memory.
- **stack window** (CXwindows only) — Displays the current contents of the stack. The stack window displays the frames on the stack and can also display the arguments to each frame by clicking on the frame's description within the window.

Related Commands

```
bind                       cxdb
disassemble                display file
display routine            examine
find memory backward       find memory forward
find window backward       find window forward
help                       info bind
```

Related Concepts

eventpoints
logging
Maryland Windows
source units
stepping
viewports
Xdefaults

Xdefaults

Description

There are various Xdefault resource settings for the CXwindow interface of CXdb. The default settings are stored in the `/usr/lib/X11/app-defaults/CXdb` file. You can set these resources in your `.Xdefaults` file. These resources can be used to change the geometry, text font, and window size (in rows and columns) of the following windows:

- Command window
- Source window
- Disassembly window
- Stack window
- Examine window
- Display file window

The disassembly, stack, and examine windows can be set to automatically update when process execution stops, or to remain static. This setting is controlled with the `AutoUpdate` resource.

You can change the CXdb command prompt and secondary prompt of the command window. The secondary prompt is used when a CXdb command requires additional input.

The CXdb Xdefault settings are made in the `.Xdefaults` file in your home directory, just as any other Xdefault resource setting. The special CXdb resource names are listed below:

```
Cxdb.commandPrompt
Cxdb.secondaryPrompt
Cxdb.Command Window.geometry
Cxdb.Command Window.blinkrate
Cxdb.commandRows
Cxdb.commandColumns
Cxdb.Source Window.geometry
Cxdb.sourceRows
Cxdb.sourceColumns
Cxdb.Disassembly Window.geometry
Cxdb.disassemblyRows
Cxdb.disassemblyColumns
Cxdb.disassemblyAutoUpdate (default is OFF)
```

```
Cxdb.Stack Window.geometry
Cxdb.stackRows
Cxdb.stackColumns
Cxdb.stackAutoUpdate (default is ON)
Cxdb.Examine Window.geometry
Cxdb.examineRows
Cxdb.examineColumns
Cxdb.examineAutoUpdate (default is OFF)
Cxdb.Display File Window.geometry
Cxdb.displayFileRows
Cxdb.displayFileColumns
Cxdb.processWindowgeometry
Cxdb.commandMenu (default is ON)
Cxdb.commandButtons (default is OFF)
Cxdb.commandMenuImmediate (default is OFF)
```

Examples

The following example sets several CXdb Xdefaults.

Assume that the following lines have been added to the `.Xdefaults` file of your home directory.

```
Cxdb.Command Window.geometry: 700x900+0+0
Cxdb.Command Window.blinkRate: 0
Cxdb.examineRows: 60
Cxdb*font: fixed
Cxdb.disassemblyAutoUpdate: ON
```

Adding the above lines to your `~/.Xdefaults` file sets the command window geometry, turns off CXdb prompt blinking, sets the examine window width to 60 rows, specifies a "fixed" character font for all CXdb windows, and enables automatic screen updating in the disassembly window.

Related Concepts

Maryland Windows   windows

Parameters

3

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 are divided into 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 of this manual.
- **Related Concepts** — A list of major concepts related to the parameter. Related concepts are described in Chapter 2 of this manual.
- **Related Parameters** — A list of other parameters related to the parameter being described. The related parameters are also described in this chapter.

<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] :      43.5585     32.6754     21.7924     10.9094      0.0263
[0][1][0..4] :      40.5585     29.6754     18.7924      7.9094     -2.9737
[0][2][0..4] :      35.5585     24.6754     13.7924      2.9094     -7.9737
[0][3][0..4] :      28.5585     17.6754      6.7924     -4.0906    -14.9737
[0][4][0..4] :      19.5585      8.6754     -2.2076    -13.0906    -23.9737
           ***
[1][0][0..4] :      49.0000     38.1170     27.2339     16.3509      5.4679
[1][1][0..4] :      46.0000     35.1170     24.2339     13.3509      2.4679
[1][2][0..4] :      41.0000     30.1170     19.2339      8.3509     -2.5321
[1][3][0..4] :      34.0000     23.1170     12.2339      1.3509     -9.5321
[1][4][0..4] :      25.0000     14.1170      3.2339     -7.6491    -18.5321
           ***
[2][0][0..4] :      54.4415     43.5585     32.6754     21.7924     10.9094
[2][1][0..4] :      51.4415     40.5585     29.6754     18.7924      7.9094
[2][2][0..4] :      46.4415     35.5585     24.6754     13.7924      2.9094
[2][3][0..4] :      39.4415     28.5585     17.6754      6.7924     -4.0906
[2][4][0..4] :      30.4415     19.5585      8.6754     -2.2076    -13.0906
           ***
[3][0][0..4] :      59.8830     49.0000     38.1170     27.2339     16.3509
[3][1][0..4] :      56.8830     46.0000     35.1170     24.2339     13.3509
[3][2][0..4] :      51.8830     41.0000     30.1170     19.2339      8.3509
[3][3][0..4] :      44.8830     34.0000     23.1170     12.2339      1.3509
[3][4][0..4] :      35.8830     25.0000     14.1170      3.2339     -7.6491
           ***
[4][0][0..4] :      65.3246     54.4415     43.5585     32.6754     21.7924
[4][1][0..4] :      62.3246     51.4415     40.5585     29.6754     18.7924
[4][2][0..4] :      57.3246     46.4415     35.5585     24.6754     13.7924
[4][3][0..4] :      50.3246     39.4415     28.5585     17.6754      0.0000
[4][4][0..4] :       0.0000      0.0000      0.0000      0.0000      0.0000
           ***
```

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] :      43.5585    32.6754    21.7924    10.9094     0.0263
```

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] :      43.5585    40.5585    35.5585    28.5585    19.5585
```

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] :      43.5585    49.0000    54.4415    59.8830    65.3246
```

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] :      43.5585    32.6754    21.7924    10.9094     0.0263
[0][1][0..4] :      40.5585    29.6754    18.7924     7.9094    -2.9737
[0][2][0..4] :      35.5585    24.6754    13.7924     2.9094    -7.9737
[0][3][0..4] :      28.5585    17.6754     6.7924    -4.0906   -14.9737
[0][4][0..4] :      19.5585     8.6754    -2.2076   -13.0906   -23.9737
```

The above command prints all rows and columns in the first level of the array.

```
(CXdb) print matrix[i-2][j+1][4]
(float)     -13.0906
```

The above command prints the single element referenced by the subscripts. In this example, the subscripts are expressions that evaluate to positive integers.

Related Commands

`evaluate`	`examine`
`print`	`set printopts maxarray`

<array-slice>

Related Concepts

C language expressions
language expressions
FORTRAN 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.

Examples

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

```
(CXdb) break routine SUB_D $D
Breakpoint 2, [0x80001882] SUB_D in myprog.f line 93
```

The above command sets a breakpoint at the routine called `SUB_D`. The object number for this breakpoint is 2, and this object number is stored in the debugger variable `$D`.

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
eventpoint handlers
debugger variables
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 in the CXdb default environment. It is expanded to its value before the command is executed. This can also be a debugger-variable.
- 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
Search 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
Search path:
        /mnt/jones/projects
        /mnt/jones/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.

```
(CXdb) add path .. , .
Search path:
        /mnt/jones/projects
        /mnt/jones/libraries
        ..
        .
```

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 which are always relative to the current console working directory.

```
(CXdb) add path ~smith/projects, $MYDIR/libraries
Search path:
        /mnt/jones/projects
        /mnt/jones/libraries
        ..
        .
        /mnt/smith/projects
        /mnt/jones/project/source/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 held in the environment variable MYDIR.

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, it is used by itself. When the value of an environment variable is needed, the variable 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 it 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`.

<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 semi-colon (`;`).

Description

An *<event-handler>* is a collection of CXdb commands to perform when the corresponding event is triggered.

The commands are enclosed in curly-braces ({ }). Each statement inside of an event-handler must end with a semi-colon (`;`).

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 semi-colon. Each command in the body of the `if` command must terminate with a semi-colon. 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 line 56 {print x;}
```

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

```
(CXdb) break line 56 {print x; resume;}
```

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

```
(CXdb) break line 56 {if (x==49) echo "x is 49"; else {print x; resume;}}
```

When the breakpoint in the above command is triggered, execution stops and a test is made on the value of the variable `x`. If the condition evaluates to `TRUE`, then the `print` command is executed and the eventpoint handler is done. If the condition evaluates to `FALSE`, the `print` command is skipped, 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 the asterisk (*) is used.

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
          [0x80001344] PICKUP in pickup3.f line 8

#2: break line, on [#0/*], Enabled, ignore 0/0
          [0x80001440] PICKUP in pickup3.f line 27
```

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
          [0x800014a0] PICKUP in pickup3.f line 32
```

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
          [0x80001344] PICKUP in pickup3.f line 8

#1: trace line, on [#0/*], Enabled, ignore 0/0
          [0x800014a0] PICKUP in pickup3.f line 32

#2: break line, on [#0/*], Enabled, ignore 0/0
          [0x80001440] PICKUP in pickup3.f line 27
```

The above command displays information about all eventpoints in the current process.

Related Commands

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

Related Concepts

breakpoints
eventpoint handlers
watchpoints
eventpoints
tracepoints

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	All event exec eventpoints.
join	All event join eventpoints.
modify	All event modify eventpoints.
reached	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
join
modify
reached
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 Tracepoint:

#2: trace line, on [#0/*], Enabled, ignore 0/0
          [0x80001394] PICKUP in pickup5.f line 14

Status of eventpoints of type Breakpoint:

#1: break routine, on [#0/*], Enabled, ignore 0/0
          [0x80001344] PICKUP in pickup5.f line 7

#0: break line, on [#0/*], Enabled, ignore 0/0
          [0x8000135a] PICKUP in pickup5.f line 9
```

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 source, on [#0/*], Enabled, ignore 0/0
            [0x80001424] PICKUP in pickup5.f line 23

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
            [0x80001394] PICKUP in pickup5.f line 14

Status of eventpoints of type Breakpoint:

#1: break routine, on [#0/*], Enabled, ignore 0/0
            [0x80001344] PICKUP in pickup5.f line 7

#0: break line, on [#0/*], Enabled, ignore 0/0
            [0x8000135a] PICKUP in pickup5.f line 9
```

The above command displays information about all the eventpoints of all the 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 `source` command.

```
(CXdb) source ~smith/commands1
```

The above command sources the `commands1` file found under the home directory of the user `smith`.

```
(CXdb) source $MYDIR/proj2setup
```

The above command sources the `proj2setup` command file found under the directory in the environment variable `MYDIR`.

Related Commands

```
add cmderr          add cmdlog
add cmdout          core
debug core          debug exec
display file        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 the current frame is frame 4.

```
(CXdb) info frame 1
Process [#0/0]
Frame : 1; [0x800013fe] PICKUP in pickup6.f line 19
Frame address : 0xffffcb80
Saved registers : pc=0x800013fe psw=0x7109680 fp=0xffffcba0 ap=0x80001c20
Floating point mode : NATIVE; Language : FORTRAN
Number of arguments : 0
```

The above command displays frame number 1 of the stack. The integer 1 is used here as an absolute frame number.

```
(CXdb) info frame -1
Frame : 3; [0x8000245c] _main+0x174
Frame address : 0xffffcba0
Saved registers : pc=0x8000245c psw=0x7909600 fp=0xffffcbe4 ap=0xffffcbb4
Floating point mode : NATIVE; Language : C
Number of arguments : 3
```

In the above command, the frame specifier is `-1`. The current frame is 4, so the command displays frame 3 (which is 4–1).

```
(CXdb) info frame +1
Process [#0/0]
Frame : 5; [0x800010b8] ___ap$envret+0x26
Frame address : 0xffffcbe4
Floating point mode : NATIVE; Language : C
Number of arguments : 0
```

In the above command, the frame specifier is `+1`. The current frame is 4, so the command displays frame 5 (which is 4+1).

Related Commands

`backtrace`	`frame`
`info frame`	`info locals`
`info process`	`info stack`

Related Concepts

scope

<function-name>

The name of a Maryland Windows function.

Syntax

<function-name>

Description

A *<function-name>* is the name of a special function in the Maryland Windows interface. The names of the functions are listed below in five categories:

- Window movement functions:

```
lower-window
move-window
next-window
previous-window
raise-window
resize-window
```

- Cursor movement and scrolling functions:

```
beginning-of-buffer
beginning-of-line
backward-char
backward-word
down-line
down-screen
end-of-buffer
end-of-line
forward-char
forward-word
left-character
right-character
up-line
up-screen
```

- History functions:

```
down-history
up-history
```

- Editing functions:

```
backward-delete-word
copy-region-as-kill
delete-backward-char
delete-char
exchange-point-and-mark
kill-line
kill-region
kill-word
set-mark-command
transpose-chars
yank
```

- Miscellaneous functions:

```
capitalize-word
digit
digit-argument
downcase-word
keyboard-quit
newline
prefix-meta
redraw-display
self-insert
undefined-key
universal-argument
upcase-word
```

Examples

The following example illustrates the use of a function name with the `bind` command.

```
(CXdb) bind transpose-chars c-t
```

The above command associates the keystroke sequence CTRL-t (represented as `c-t`) with the function `transpose-chars`. To enter the key name, you literally type `c-t`. However, to use the `transpose-chars` function in the command window of Maryland Windows, you hold down the CTRL key and then press t.

Related Commands

`bind` `info bind`

Related Concepts

Maryland Windows windows

<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.

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

<key-name>

A keystroke sequence.

Syntax

<key-name>[...]

Parameter	Meaning
<key-name>	The name of a key.
[...]	Additional keys in the sequence.

Description

A *<key-name>* is a particular keystroke sequence. The key name literally shows which keys to press.

Certain keys have special representations when used with the `bind` and `info bind` commands in the Maryland Windows interface. These keys and their representations are:

- CONTROL key
  ```
  C-
  c-
  ^
  ```
- ESCAPE key
  ```
  ESC
  ^[
  M-
  m-
  ```
- RETURN key
  ```
  C-M
  c-m
  ^M
  ^m
  RET
  RETURN
  ret
  return
  ```
- SPACE key
  ```
  SPC
  ```

- **LINEFEED** key
    LFD
    ^J
    c-j
    C-J
- **DELETE** key
    DEL
    TAB
    ^I
    c-i
    C-I

Examples

The following examples illustrate how to specify keystroke sequences with the `info bind` command.

```
(CXdb) info bind c-d
 ^D              delete-char
```

The above command displays information about the keystroke sequence **CTRL-d**. This sequence performs the `delete-char` function in the command window of Maryland Windows. To delete a character, you hold down the **CTRL** key and press **d**. However, in the `info bind` command, this same keystroke sequence is represented by `c-d`.

```
(CXdb) info bind ^d
 ^D              delete-char
```

The above command also displays information about **CTRL-d**.

```
(CXdb) info bind return
 Ret             newline
```

The above command displays information about the **RETURN** key.

Related Commands

`bind` `info bind`

Related Concepts

Maryland Windows windows

Related Parameters

function-name

<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.

<language-expression>

Examples

The following examples illustrate the use of language expressions as addresses and numerical values.

```
(CXdb) break routine SUBA
#0: break routine, on [#0/*], Enabled, ignore 0/0
        [0x800013aa] SUBA in arrays.f line 11
```

The above command sets a breakpoint at the routine called `SUBA.` In this case, CXdb evaluates the expression `SUBA` to determine the address of the routine.

```
(CXdb) print A+B
(REAL*4)      10.4415
```

The above command evaluates the expression `A+B` and prints the resulting value, which is a real number.

Related Commands

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

Related Concepts

debugger variables
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 74
Breakpoint 0, [0x80001788] SUBR2 in myfile.f line 74
```

The above command sets a breakpoint at the machine instruction corresponding to line 74 of the current source file.

```
(CXdb) break line sample.c:74
Breakpoint 1, [0x800017c4] sample in sample.c line 74
```

The above command sets a breakpoint at the machine instruction corresponding to line 74 of the file `sample.c`.

<line-specifier>

Related Commands

`break line`
`goto line`
`trace line`
`event reached line`
`info line`

Related Concepts

source units

Related Parameters

file-name

<process-list>

A list of processes.

Syntax

`:{p | 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 that means all processes.

Description

A *<process-list>* is a list of processes that are affected by a command.

NOTE: V1.0 of CXdb maintains only one current process object at any given time. Because of this, you may omit the process list in V1.0.

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, then the process list is required to specify which processes you want to affect. If you have multiple processes but you do not specify a process list, then 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 file
display routine                 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
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
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 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

<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.

`>>&`	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 of the other command parameters except the background directive (`&`). If a redirection operator follows a language expression, terminate the language expression with a backslash-semicolon (`\;`) escape sequence.

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.

The default is noclobber clear (disabled).

Examples

The following examples illustrate the use of redirection operators. For all these examples, assume that noclobber is enabled (on).

```
(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. (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 X+Y\; > 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 escape sequence (`\;`) is needed to delimit the end of the language expression `X+Y`.

```
(CXdb) print X+Y\; > 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 X+Y\; >>! 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 X+Y\; >>! 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 on).

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	windows

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"
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"
[         "step over"
```

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 its name, with or without the SIG prefix, or by its number. Signal names are *not* case sensitive.

The following is a list of signals followed by their signal numbers in parentheses:

```
SIGHUP(1)
SIGINT(2)
SIGQUIT(3)
SIGILL(4)
SIGTRAP(5)
SIGIOT(6)
SIGEMT(7)
SIGFPE(8)
SIGKILL(9)
SIGBUS(10)
SIGSEGV(11)
SIGSYS(12)
SIGPIPE(13)
SIGALRM(14)
SIGTERM(15)
SIGURG(16)
SIGTSTP(17)
SIGSTOP(18)
SIGCUNT(19)
SIGCHLD(20)
SIGTTIN(21)
SIGTTOU(22)
SIGIO(23)
SIGXCPU(24)
SIGFSZ(25)
SIGVTALRM(26)
SIGPROF(27)
SIGWINCH(28)
```

```
SIGLOST(29)
SIGUSR1(30)
SIGUSR2(31)
```

Signal 0 is used to indicate 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.

```
(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                  event signal
set signal                   signal process
signal thread
```

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 different commands.

```
(CXdb) break source 125
Breakpoint 6, [0x800017c2] MAIN in myfile.f line 43
```

The above command sets a breakpoint at source unit 125 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 newprog.c:78
```

The above command sets the program counter (PC) to the starting address of source unit 78 in the file `newprog.c`.

<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

<string>

<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 J
object type: Fortran identifier
   location: <none>
       size: 4 bytes
       type: INTEGER*4
      value: 4
       used to create 1 synthesized variable(s):
         1. <INDV>     ?i7 = ?i1+((4*N)*(J-1))
```

The above command displays information about the program variable `J`. The response indicates that the current value of `J` is 4. This value is not stored (location = `<none>`) because the synthesized variable `?i7` replaces `J`. The reason for the replacement is `INDV`, which means the induction variable has undergone strength reduction. The equation used to generate the synthesized variable is `?i7=?i1+((4*N)*(J-1))`.

In the source code, `J` is a loop induction variable that is used as an index to reference specific elements of an array. In the object file, `?i7` serves as a pointer to the array elements. The compiler replaces `J` with `?i7` because it is more efficient to increment the pointer than it is to increment `J` 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 `J` by solving for it in the equation shown for `?i7`.

```
(CXdb) info expression \?i7
object type: Fortran identifier
   location: register a2
       size: 4 bytes
       type: INTEGER*4
      value: -2147176320
       Reason: Loop induction variable
       created from 1 equation(s):
         1. <INDV>  ?i1+((4*N)*(J-1))
       2 liveness ranges:
               Start      End         Location
          1. 0x8000172e:0x80001750 - register a2
          2. 0x80001750:0x80001754 - register a2
```

The above command displays information about the synthesized variable `?i7`. The response shows the equation that the compiler uses to generate `?i7`. It also shows the liveness ranges and corresponding storage locations for the variable. The reason for generating `?i7` is that it replaces a loop induction variable.

```
(CXdb) print/x \?i7
INTEGER*4) 0x8004b080
```

The above command prints the current value of the synthesized variable `?i7` in hexadecimal format. Because `?i7` is a pointer to an array in this case, the current value of `?i7` is the starting address of the next array element to be accessed.

Related Commands

`evaluate` `info expression`
`print`

Related Concepts

debugger variables language expressions
synthesized variables

Related Parameters **language-expression**

<thread-list>

A list of process threads.

Syntax

`:{t | 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 that means 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, then 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) :t1,2 continue
```

The above command continues execution of threads 1 and 2 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                    clear step
continue                     copy
disassemble                  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 sourceunit
info stack                   info threads
info trace                   info type
info vregisters              info watch
next                         next instruction
next over                    print
return                       set format
set memory                   set seq
set sqs                      set step
signal thread                step
step instruction             step over
trace instruction            trace line
trace routine                trace source
watch
```

Related Parameters

process-list

<viewport>

A file name or window identifier.

Syntax

{1 | *<file-name>* | $*<debugger-variable>*}

Parameter	Meaning
1	The window number of the CXdb command window.
<file-name>	The name of a file to be used as a viewport.
$*<debugger-variable>*	A debugger variable that contains the number of the CXdb command window. The delimiter ($) is required in this case.

Description

A *<viewport>* is either a file or the CXdb command window. Viewports can receive copies of CXdb input, output, and error messages.

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 generated in response to commands.
- `cmdlog` — Commands entered in the command window.
- `cmdout` — Output generated in response to 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/local/Proj7/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/local/Proj7.`

```
(CXdb) set cmdout 1,save_cxdbout
New cmdout: Window #1, save_cxdbout
```

The above command establishes a new viewport list for `cmdout.` This list contains Window #1 (the command window) and the 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
viewports	windows

Related Parameters

file-name	redirection-operator

CXdb messages

4

This chapter contains reference pages that explain the CXdb messages. The messages are listed in order by number. Each explanation is divided into the following sections:

- Message — The exact text of the message. Variable parameters are enclosed in angle brackets (<>) and are shown in italic type. For example, <*char*> is a variable.
- Type — The message type, which can be one of the following:
  - `INFO` — A condition that is not normal or expected, but it is not severe enough to cause an error.
  - `ERROR` — A condition that prevents completion of the CXdb command.
  - `FATAL` — A condition that prevents further execution of the process being debugged.
- Explanation — A more detailed explanation of the message, including possible actions to correct the situation.

Msg. No.

1 Message: Illegal character in alias identifier: *<char>*

Type: ERROR

Explanation: The alias identifier contains an illegal character that would prohibit parsing. The illegal character is indicated in the text of the error message. Please select a different name for your alias.

2 Message: Alias *<alias name>* exists already, please try a new identifier

Type: ERROR

Explanation: This alias currently exists. Therefore, you should provide a new name in the alias definition.

3 Message: An alias must have a unique identifier

Type: ERROR

Explanation: An alias must be given a name in order to be created.

4 Message: Alias *<alias name>* does not exist

Type: ERROR

Explanation: An alias with this name does not exist.

5 Message: Corrupt executable file for *<filename>*

Type: ERROR

Explanation: The executable file is corrupt. That is, one or more of the internal data structures have been corrupted.

6 Message: Corrupt location range table for *<filename>*

Type: ERROR

Explanation: A location range table could not be opened. Therefore, check to see if it exists. If it does, then possibly some of the internal data representations have been corrupted.

7 Message: Corrupt section table for *<filename>*

Type: ERROR

Explanation: The section table for this program is corrupt.

8 Message: Corrupt source file map for *<filename>*

Type: ERROR

Explanation: The source file map for this program is corrupt.

9 Message: Corrupt source table for <*filename*>
Type: ERROR
Explanation: The source table for this program is corrupt.

10 Message: Corrupt source range table for <*filename*>
Type: ERROR
Explanation: A source range table could not be opened. Therefore, check to see if it exists. If it does, then possibly some of the internal data representations have been corrupted.

11 Message: Corrupt source unit table for <*filename*>
Type: ERROR
Explanation: A source unit table could not be opened. Therefore, check to see if it exists. If it does, then possibly some of the internal data representations have been corrupted.

12 Message: Corrupt TSI table for <*filename*>
Type: ERROR
Explanation: A TSI table could not be opened. Therefore, check to see if it exists. If it does, then possibly some of the internal data representations have been corrupted.

13 Message: Corrupt variable table for <*filename*>
Type: ERROR
Explanation: A variable table could not be opened. Therefore, check to see if it exists. If it does, then possibly some of the internal data representations have been corrupted.

14 Message: Syntax Error - line:<*line number*> col:<*column number*>; <*expecting or missing*>: <*specific error token*>
Type: ERROR
Explanation: A syntax error was encountered in the parsing of your command. You should review what you typed, and make sure that it is a valid command. If you are uncertain about the syntax for a given command, you can use the on-line help system to get more information on specific commands.

15 Message: Syntax Error - line:<*line number*> col:<*column number*>; <*expecting or missing*>: <*specific error token*>
Type: ERROR
Explanation: The Fortran language expression you entered contained syntax errors. Please refer to the Fortran language guide for details on Fortran sytax.

16 Message: Display format '<*format specifier*>' may not be associated with memory size '<*size specifier*>'.

Type: ERROR

Explanation: You have tried to associate a display format with an incompatible memory size. Due to size and machine representations for specific types, not all memory sizes can be displayed in all formats.

17 Message: Invalid event-id,<*event-id*>

Type: ERROR

Explanation: The event number is either negative or not a currently defined event.

18 Message: Invalid thread-id,<*thread-id*>

Type: ERROR

Explanation: The thread number is either negative or larger than the number of threads allowed by the architecture.

19 Message: No such thread-id,<*thread-id*>, for process [#<*process number*>].

Type: ERROR

Explanation: The thread number is either negative or larger than the number of threads currently active for the indicated process.

20 Message: Unable to select thread-id,<*thread-id*>, for process <*process-id*>.

Type: ERROR

Explanation: CXdb was unable to select the indicated thread as the current thread within the process shown. This will probably lead to more errors.

21 Message: Process number <*process number*> is invalid.

Type: ERROR

Explanation: The process number you specified is either negative or not a currently defined process. Use the 'info cxdb' or 'info process' command to determine which process numbers are valid.

22 Message: Invalid directory pathname,<*file specifier*>

Type: ERROR

Explanation: The file specifier is not a directory.

23 Message: IOCTL error on file descriptor <*descriptor number*>

Type: ERROR

Explanation: An error ocurred while performing an ioctl system call on the indicated file descriptor. This may lead to additional errors. Other error messages should indicate the reason the ioctl was being performed if the failure of the ioctl was considered harmful to that operation.

Msg. No.

24
Message: IO read / write error on file descriptor <*descriptor number*>
Type: ERROR
Explanation: An error occurred while trying to read or write on the specified file descriptor. This may lead to additional errors. Other error messages should indicate the reason the ioctl system call was being performed if the failure of the ioctl was considered harmful to that operation.

25
Message: Macro <*macro-name*> already exists; please try a new identifier
Type: ERROR
Explanation: This macro currently exists. Therefore, you should provide a new name in the macro definition.

26
Message: A macro must have a unique identifier
Type: ERROR
Explanation: A macro must be given a name in order to be created.

27
Message: Macro <*macro-name*> does not exist
Type: ERROR
Explanation: A macro with this name does not exist.

28
Message: Macro parameters beyond position <*count*> are ignored.
Type: INFO
Explanation: CXdb imposes a limit on the number of parameters allowed in a macro invocation. The remainder are ignored.

29
Message: Macro invocations nested more than <*count*> levels.
Type: ERROR
Explanation: CXdb imposes a limit on the level of nesting of macro invocations. This limit has been exceeded.

30
Message: Dynamic memory exhausted.
Type: FATAL
Explanation: No more virtual memory is available. CXdb has used up all the dynamic memory available to it. This can come from many causes. This is generally a fatal error.

31
Message: Process [#<*process-number*>] is not stopped.
Type: ERROR
Explanation: To execute the command, the process must be stopped.

32 Message: Process [#<*process-number*>] has no running image.
Type: ERROR
Explanation: The command you entered operates on the image of an executable. The process you specified does not have an active image. Images may be created by executing the 'debug proc', 'run', 'attach', or 'core' commands.

33 Message: Parameter <*name*> already exists
Type: ERROR
Explanation: A parameter with that name already exists.

34 Message: Macro parameter name is missing.
Type: ERROR
Explanation: The name of a parameter in a macro definition has been omitted. Every formal parameter in a macro definition must have a name.

35 Message: An error occurred during a read or write on descriptor <*descriptor number*>
Type: ERROR
Explanation: An error occurred during a read, write, or seek system call. This may lead to additional errors. Other error messages should indicate the reason the ioctl was being performed if the failure of the ioctl was considered harmful to that operation.

36 Message: Syntax error in the following command:
<*command line*>
Type: ERROR
Explanation: You have typed something incorrectly. The CXdb parser cannot continue. Please read the command summary to resolve the problem.

37 Message: Error accessing system dependent information
Type: ERROR
Explanation: Error encountered during the getsysinfo system call. This is a fatal error.

38 Message: Unable to open <*filename*>'s executable file
Type: ERROR
Explanation: The executable file cannot be opened. No debugging information can be inferred.

39 Message: Unable to find CDI location range table for <*filename*> in search path

Type: ERROR

Explanation: A location range table was not found. This file is generated by the compiler to inform CXdb about symbolic information. Therefore, check to see if it exists. If it does, then make sure that it is within one of the directories in the search path. Reference the ADD PATH command for adding another directory to your search path.

40 Message: Unable to CDI open source file map for <*filename*>

Type: ERROR

Explanation: The source file map for this map cannot be opened. This file is generated by the compiler to inform CXdb about symbolic information.

41 Message: Unable to find source file for <*filename*> in search path

Type: ERROR

Explanation: A source file was not found. Therefore, check to see if it exists. If it does, then make sure that it lies within the search path. Reference the ADD PATH command for adding another directory to your search path.

42 Message: Unable to find CDI source range table for <*filename*> in search path

Type: ERROR

Explanation: A source range table was not found. This file is generated by the compiler to inform CXdb about symbolic information. Therefore, check to see if it exists. If it does, then make sure that it is within one of directories in the search path. Reference the ADD PATH command for adding another directory to your search path.

43 Message: Unable to find CDI source unit table for <*filename*> in search path

Type: ERROR

Explanation: A source unit table could not be found. This file is generated by the compiler to inform CXdb about symbolic information. Therefore, check to see if it exists. If it does, then make sure that it is within one of directories in the search path. Reference the ADD PATH command for adding another directory to your search path.

44 Message: Unable to find CDI type scope information for <*filename*> in search path

Type: ERROR

Explanation: A type-scope table could not be found. This file is generated by the compiler to inform CXdb about symbolic information. Therefore, check to see if it exists. If it does, then make sure that it is within one of directories in the search path. Reference the ADD PATH command for adding another directory to your search path.

Msg. No.

45
Message: Unable to find CDI variable table for <*filename*> in search path
Type: `ERROR`
Explanation: A variable table could not be found. This file is generated by the compiler to inform CXdb about symbolic information. Therefore, check to see if it exists. If it does, then make sure that it is within one of directories in the search path. Reference the ADD PATH command for adding another directory to your search path.

46
Message: Unmatched parentheses
Type: `ERROR`
Explanation: There are unmatched parentheses in the command. Please retype the command correctly so that all of the parentheses are balanced.

47
Message: Internal Error: <*reason for internal error*>
Type: `FATAL`
Explanation: An internal error in the compiler has occurred. Please submit a problem report.

48
Message: Informational: <*reason for internal error*>
Type: `INFO`
Explanation: An internal error in the compiler has occured. Please submit problem report.

49
Message: Unable to locate your home directory.
Type: `INFO`
Explanation: None of the environment variables $DOTDIR, $LOGDIR, or $HOME are defined. Unable to locate your home directory to process any CXdb initialization file.

50
Message: Pathname for <*use of pathname*> is too long.
Type: `ERROR`
Explanation: The pathname constructed is longer than the maximum allowed by the system. Contact your system administrator.

51
Message: Couldn't <*operation*> file <*filename*>.
System error text: <*errno description*>
Type: `ERROR`
Explanation: An operation on a file failed. The error message contains the text of the error code returned by the failing operation.

52 Message: The initialization file <*init filename*> is not owned by either you or root. Skipped.

Type: `INFO`

Explanation: The initialization file specified is neither owned by you nor root. This is a potential security problem, so the file is being skipped.

53 Message: Executable file <*filename*> not found.

Type: `ERROR`

Explanation: The file you specified as an executable image could not be accessed. Either you do not have permissions to access the directories leading to the file, or the file doesn't exist.

54 Message: File <*filename*> is not marked executable.

Type: `ERROR`

Explanation: The file you specified exists but does not have the execute permission bit set for you. It is either not executable, or you do not have the permissions to execute it.

55 Message: Multiple process/thread specifier pairs not allowed.
All but first ignored.

Type: `ERROR`

Explanation: You have specified more than one process/thread specifier pair on a command. Only one pair is allowed. All specifier pairs except the first have been ignored. A common mistake is to enter ':T1 :P0' to specify thread 1 of process 0. The ordering of the process/thread specifiers is important. The process must always come first. The correct specification would be ':P0 :T1'. Starting a process/thread pair with a thread specifier means to process the specified thread in ALL processes. In version 1.0 this is always a single process.

56 Message: You may not specify threads on this command. Specifiers ignored.

Type: `INFO`

Explanation: You have specified specific threads on a command that only operates at the process level (for example, the 'kill process' command only operates on processes, not individual threads). The thread specifiers have been ignored.

57 Message: You may not specify processes or threads on this command.
Specifiers ignored.

Type: `INFO`

Explanation: You have specified specific processes or threads on a command which does not operate on processes (for example, 'info cxdb' and all forms of the 'debug' command do not operate on specific processes). The specifiers were ignored.

Msg. No.

58 Message: No process object exists.
Type: ERROR
Explanation: You have tried to execute a command that operates on a process object, and there are no processes defined. Use a form of the 'debug' command to create a process object.

59 Message: Process object <*process number*> vanished!
Type: ERROR
Explanation: A process which was previously verified to be on the global process list has disappeared.

60 Message: Event object <*event number*> vanished!
Type: ERROR
Explanation: An event which was previously verified to be on the global event list has disappeared.

61 Message: No such offset type.
Type: ERROR
Explanation: An offset type has been used which does not exist.

62 Message: Frame specified is out of range.
Type: ERROR
Explanation: You have specified a frame, either by absolute or relative number, which does not exist on the stack. You can use the 'backtrace' command to see the frames currently on the stack.

63 Message: Environment variable <*variable name*> not found.
Type: INFO
Explanation: The environment variable you specified does not exist within the environment being operated upon. Many environments are maintained: CXdb's default environment table and each process' environment.

64 Message: Process ID <*process-id*> is invalid.
Type: ERROR
Explanation: The process ID you specified is invalid. It must be positive and less than 30000.

65 Message: Only one process object may be attached to a running process.
Type: ERROR
Explanation: You tried to specify multiple process objects on the 'attach' command. Only a single CXdb process object can be attached to a running process at a time.

Msg. No.

66 Message: Unable to attach to process *<process-id>*.

Type: `ERROR`

Explanation: CXdb was unable to attach to a child process. During the normal startup process for a target process CXdb will attach to several processes. If any of these fail, the startup will fail. Also, if you specified a process to attach to with the 'debug proc' or 'attach' commands and you don't have the correct permissions to attach to it, the attach may fail.

67 Message: The system call 'sigaction' failed! Reason: *<failure reason>*.

Type: `ERROR`

Explanation: CXdb tried to perform a 'sigaction' system call to alter the way it handles signals. This failed for the indicated reason. This is nearly always a fatal error.

68 Message: This command may not be run asynchronously. Ignoring asynch operator.

Type: `INFO`

Explanation: You specified that the command should run asynchronously with the '&' operator. However, the command you entered can not be executed asynchronously. Only commands that control the execution of a process are allowed to run asynchronously. This includes operations such as run, attach, continue, step, next, stop, and kill.

69 Message: Process [#*<process number>*] already has a running image.

Type: `ERROR`

Explanation: You tried to start the execution of or attach to a process when the process object already has a running image. If you want to restart the process, you should first terminate the current running image with the 'kill process' or 'detach' command and then issue the 'run' command again.

70 Message: Process [#*<process number>*] has no executable specified for it.

Type: `ERROR`

Explanation: You tried to start the execution of a process and no executable file has been specified. You can associate an executable file with an existing process object with the 'executable' command.

71 Message: Couldn't open a pty connection for process [#*<process number>*].

Type: `ERROR`

Explanation: In trying to create the target process a pty channel is created to communicate with it. This pty connection failed. Previous error messages should indicate the reason for failure. Typically it is because the system is out of pty's.

72 Message: Unable to block receipt of <*signal name*> signal.
Type: `FATAL`
Explanation: CXdb tried to block the receipt of a signal, and the system call failed.

73 Message: Unable to unblock receipt of <*signal name*> signals.
Type: `FATAL`
Explanation: CXdb tried to unblock the receipt of a signal, and the system call failed.

74 Message: Unable to fork to create process [#<*process number*>].
Type: `ERROR`
Explanation: CXdb tried to fork a new process to create the image to be debugged, and the fork system call failed.

75 Message: Unable to set fixed scheduling on process [#<*process number*>].
Type: `INFO`
Explanation: CXdb tried to set fixed scheduling on the target process, but the setpattr system call failed.

76 Message: Unable to clear PIXINHERIT process [#<*process number*>].
Type: `INFO`
Explanation: CXdb uses a process mode called PIXINHERIT to obtain the processes as they start up. This mode must be cleared once the target process is started. CXdb was unable to clear this mode.

77 Message: Unable to set SEQ and SQS modes for process [#<*process number*>].
Type: `ERROR`
Explanation: CXdb failed to set the Sequential Store (SQS) and Sequential Execution (SEQ) bits in the target process' PSW. Processes can not be debugged if these modes are not set.

78 Message: Unable to determine the parent process for process <*process-id*>.
Type: `ERROR`
Explanation: CXdb was unable to determine the parent process for a process that it had attached to during the target startup handling. The process was discarded.

79 Message: Parent process <*process-id*> for process <*process-id*> is unknown.
Type: `ERROR`
Explanation: CXdb determined the parent process for a process that it had attached to during the target startup handling, but it was not any of the processes that CXdb is currently controlling. The new process was discarded.

80
Message: Parent process <*process-id*> for process <*process-id*> has unknown type.
Type: `ERROR`
Explanation: CXdb determined the parent process for a process that it had attached to during the target startup handling, but its type was 'unknown'. This error will cause the process startup phase to fail.

81
Message: Parent process <*parent process-id*> for process <*child process-id*> is the initial process.
Type: `ERROR`
Explanation: CXdb determined the parent process for a process that it had attached to during the target startup handling, but it was the initial process that CXdb forked during the startup phase. The initial process may have forked another process before performing the exec. This error will cause the process startup phase to fail.

82
Message: The xterm process <*xterm process-id*> forked process <*child process-id*>.
Type: `ERROR`
Explanation: During the startup phase for a target process under the X windows system, CXdb starts an xterm to invoke the process to be debugged. However, the xterm has forked a second process for some reason. This error will cause the process startup phase to fail.

83
Message: The shell process <*shell process-id*> forked process <*child process-id*>.
Type: `ERROR`
Explanation: During the startup phase for a target process CXdb starts a shell to invoke the process to be debugged. However, the shell has forked a second process for some reason. This error will cause the process startup phase to fail.

84
Message: CXdb has inherited target's child <*process-id*>. It was released.
Type: `INFO`
Explanation: For some reason, CXdb has inherited one of the child processes belonging to the target process. It was detached so that it can run normally.

85
Message: Process <*parent process-id*> is unattached.
Type: `FATAL`
Explanation: CXdb determined that a process it received from wait() was not already pattached. This condition should never arise and is a fatal error.

86
Message: Unexpected SIGTRAP code <*SIGTRAP subcode*> in process <*process-id*>.
Type: `INFO`
Explanation: The process indicated received a SIGTRAP signal, but the subcode indicated is unkown. The SIGTRAP was ignored.

87 Message: Unable to clear the Trace Trap PSW bit in process <*process-id*>.
Type: INFO
Explanation: CXdb was unable to clear the Trace Trap bit in one of the processes it is managing. This is only a warning but will probably lead to other errors later. If this error occurs during target startup, the startup is terminated.

88 Message: Unknown process <*process-id*> exec'ed.
Type: ERROR
Explanation: One of the processes that CXdb is controlling has an unknown state, and it performed an exec. The process has been detached. If this error occurred during target startup, the startup is terminated.

89 Message: Process [#<*process number*>] exec'ed.
Type: INFO
Explanation: The target process you are debugging has performed an exec. This will probably make all the debugging information that CXdb has about the process obsolete. You should issue the 'executable' command to inform CXdb about the executable the process is now executing. CXdb has removed all your PC based eventpoints from this process because they now contain invalid locations and data.

90 Message: Command [#<*command number*>] is pending on process [#<*process number*>].
Type: ERROR
Explanation: You tried to execute a command on the indicated process, but there was already another command pending (running asynchronously) on that process. You can not execute another command on that process until the previous command completes.

91 Message: The shell controlling process [#<*process number*>] exited.
Type: INFO
Explanation: The shell process that CXdb started to control the execution of the target process exited unexpectedly. This will cause the target to be terminated.

92 Message: The xterm controlling process [#<*process number*>] exited.
Type: INFO
Explanation: The xterm process that CXdb started to control the execution of the target process exited unexpectedly. This will cause the target to be terminated.

Msg. No.

93 Message: Thread [#<*process number*>/<*thread-id*>] is already running.

Type: `ERROR`

Explanation: You entered a command which would continue the execution of one or more threads within a process. At least one of these threads is already running. Your command can not be performed until all of the threads it affects are stopped.

94 Message: Floating point mode not recognized.

Type: `ERROR`

Explanation: You have referenced a floating point mode which is either unknown or not allowed with this command.

95 Message: No debugging information available.

Type: `ERROR`

Explanation: You have tried to execute a command that requires the full debugging information for a program. There is currently no debugging information available. You can specify an executable to retrieve debugging information using the 'executable' command.

96 Message: Source file <*file name*> not found.

Type: `ERROR`

Explanation: The source file you specified could not be found within the debugging information associated with this process. You may have misspelled it, or the directory search path for the process may not include the directory where that source file resides. You can check the search path with the 'info process' command.

97 Message: No source statements found.

Type: `ERROR`

Explanation: The location you specified does not contain any source statements. Possibly no source statements start at the line you specified or the region you indiciated with the cursor contains no source statements.

98 Message: No debugging information available for stepping thread <*thread-id*>. Continuing until routine exit.

Type: `INFO`

Explanation: There is no debugging information available for the current location of the thread indicated. Stepping by source units is not possible. The execution of the thread will be continued until it returns from the current routine.

99 Message: Unable to clear bits <*bit names*> in PSW.
Type: ERROR
Explanation: The indicated bits could not be cleared in the current PSW or on the stack. This will probably lead to more cascading errors. There is no real way to recover from this.

100 Message: Unable to reset memory occupied by breakpoint at <*address*>.
Type: ERROR
Explanation: The original value of memory currently occupied by a breakpoint could not be restored. This will probably lead to incorrect operation of you program. You will probably not be able to continue the execution of any thread past this address.

101 Message: Unable to replace breakpoint at <*address*>.
Type: ERROR
Explanation: The breakpoint at the indicated address could not be reinstated. This will probably lead to incorrect operation of your program. It is probable that the current stepping operation will fail and further execution may also fail.

102 Message: Source unit <*source unit index*> does not exist within file <*filename*>.
Type: ERROR
Explanation: The source unit index you specified does not exist within the indicated source file. You can use the 'info source' command to get the source unit indices for all source units on a given source file line.

103 Message: <*operation*> only works with the X Window System interface.
Type: ERROR
Explanation: This operation's interface requires CXdb's X Window System interface to function.

104 Message: Eventpoint <*event number*> also has a breakpoint at address <*address*>.
Type: INFO
Explanation: You have created an eventpoint that placed a breakpoint at the same location another eventpoint placed a breakpoint. The last eventpoint created will take precedence over the previous one(s). You may wish to disable or remove the previous event by using the 'disable event' or 'remove event' commands.

105 Message: CXdb variable <*variable name*> not found.
Type: ERROR
Explanation: The CXdb variable you specified does not exist. CXdb variables are created with optional command parameters in some commands, or when used as the storage location of a value in an assignment expression.

Msg. No.

106 Message: CXdb variable <*variable name*> cannot be deleted.
Type: ERROR
Explanation: The CXdb variable you specified is predefined by CXdb and cannot be deleted. Only variables that you create can be deleted.

107 Message: CXdb variable <*variable name*> cannot have its value set.
Type: ERROR
Explanation: The CXdb variable you specified is predefined by CXdb to be read-only.

108 Message: Invalid command in eventpoint handler. Handler aborted.
Type: ERROR
Explanation: The handler for the eventpoint indicated contained a command which is not allowed within eventpoint handlers. No form of process execution may be started from within an eventpoint handler. This includes all forms of the step, next, continue, finish, return, signal, stop, kill, run, attach, and detach commands. You may resume the execution of the process with the 'resume' command. Execution of the eventpoint handler was terminated.

109 Message: Command not valid interactively. It may only be used in eventpoint handlers.
Type: ERROR
Explanation: The command you entered is not allowed interactively. It may only be used within an eventpoint handler. Commands of this type, 'resume' for example, have no meaning outside of an eventpoint handler and cannot be entered interactively.

110 Message: Ambiguous file <*filename*>, use a qualified pathname.
Type: ERROR
Explanation: The given source file name is ambiguous. Please provide the pathname for the specified source file. That is the file name with one of the directory names in source file search list prepended to it.

111 Message: Source file <*name*> is out of date: last modified <*modified date*>, compiled <*compiled date*>.
Type: INFO
Explanation: The source file is out of date with respect to the version used for compilation. This is only an informational message; the modified version of the source file will be used.

112 Message: Cannot get the value for <*symbol*>
Type: ERROR
Explanation: CXdb could not determine the value for the given symbol. Thus, evaluation cannot continue.

113 Message: File <*name*> is not a core file.
Type: ERROR
Explanation: The file you specifed was not a core file. Please be certain that you typed the name correctly. The operation was aborted.

114 Message: Address <*hex address*> is not within a defined region of the process image.
Type: ERROR
Explanation: The address you specified was not within any of the section maps contained in the current process image. Either you entered an invalid address or the process image is not from the executable you are debugging.

115 Message: A process image already exists! Use 'kill process' or 'detach' first.
Type: ERROR
Explanation: You tried to associate a new image with a process object that already has an image. You must remove the existing image with either the 'kill process' or 'detach' command and then try again.

116 Message: Vector register <*name*> doesn't have enough space.
Type: INFO
Explanation: The vector register you specified does not have enough space left for the operation you requested. Verify that the number of bytes you are trying to read/write exist in the register being accessed from the location you specified.

117 Message: Your SHELL setting, '<*shell path*>', is unsupported. Defaulting to '/bin/sh'.
Type: INFO
Explanation: Your SHELL environment variable specifies a unknown shell. CXdb only supports the use of sh, csh, ksh, and tcsh. Your SHELL setting will not be honored, and '/bin/sh' will be used instead.

118 Message: Illegal regular expression, <*value*>
Type: ERROR
Explanation: This is an illegal regular expression. Please reference the ed(1) man page to determine the proper syntax of the regular expression. If certain special characters are part of the identifier, you may need to escape those characters with "".

119 Message: System call failed: *<syscall name>* - *<failure reason>*
Type: ERROR
Explanation: The indicated system call failed. The reason for the failure is also shown. The ramifications of this failure are entirely dependent upon what you were doing when the error occurred.

120 Message: Error reading scalar registers for [#*<process number>*/*<thread-id>*].
Type: ERROR
Explanation: An error has been encountered while trying to read the scalar registers. This will almost certainly cascade into other errors. Depending on when this error was encountered, CXdb may not be able to determine the state of your process correctly.

121 Message: Error reading vector registers for [#*<process number>*/*<thread-id>*].
Type: ERROR
Explanation: An error has been encountered while trying to read the vector registers. This is not a fatal error, but CXdb will not be able to display the contents of these registers, nor will you be allowed to modify them.

122 Message: Error reading communication registers for [#*<process number>*/*<thread-id>*].
Type: ERROR
Explanation: An error has been encountered while trying to read the communication registers. This is not a fatal error, but CXdb will not be able to display the contents of these registers, nor will you be allowed to modify them.

123 Message: Invalid command! You may not resume the execution of a core file.
Type: ERROR
Explanation: You attempted to perform some type of process excution (such as 'step' or 'next') while examining a core file. Core files can not be executed in any fashion. You may start a new process image with the 'run' command, but you can not continue the execution of a core file.

124 Message: Unable to get trap addresses for thread *<thread id>*.
Type: ERROR
Explanation: During process creation CXdb was unable to read the user mode trap addresses from the process. This probably indicates a more serious problem, and the process object will be unusable.

125 Message: Read of address <*address*> to get <*datum*> failed.
Type: ERROR
Explanation: The read operation on the process image failed. The address being read and the data wanted were listed in the error message. The operation that required the read will fail. Further debugging may not be possible depending on the cause of the failure.

126 Message: Write of address <*address*> to put <*datum*> failed.
Type: ERROR
Explanation: The write operation on the process image failed. The address being written to and the data being written were listed in the error message. The operation that required the write will fail. Further debugging may not be possible depending on the cause of the failure.

127 Message: Unable to set bits <*bit names*> in PSW.
Type: ERROR
Explanation: The indicated bits could not be set in the current PSW or on the stack. This will probably lead to more cascading errors. There is no real way to recover from this.

128 Message: There is currently no default source file, please specify one.
Type: ERROR
Explanation: You entered a command that requires a file name specification. You did not enter a file name, and there is currently no default file. The default file is set each time you access a file from the source window. Until the source window is created, there is no default source file.

129 Message: Error in constructing a frame [#<*process number*>/<*thread-id*>].
Type: ERROR
Explanation: An error was encountered in trying to construct a stack frame for the thread of the process. It was impossible either to read the current registers, or to read from memory.

130 Message: Evaluation of the language expression failed.
Type: ERROR
Explanation: The evaluation of the language expression you entered failed for the indicated reason(s). How to resolve the problem is dependent on the reason for failure.

131 Message: Unable to locate main routine! No default source file set.
Type: `INFO`
Explanation: When CXdb loads an executable, it tries to locate the main routine for the application and then uses that to set up the default source file name and source language. CXdb was unable to determine the location of the main routine. This usually means that either the main routine was not compiled with the -cxdb option, or it is written in a language that CXdb doesn't support. No default source file has been setup and the default language has been set to C.

132 Message: There is no location in memory for <*identifier*>.
Type: `ERROR`
Explanation: There is currently no location in memory for the referenced variable. Although the identifier exists, there is no corresponding location in memory. The identifier may be a compile time construct. If the identifier does represent a variable, the references to it were optimized away.

133 Message: Too many threads selected (<*thread count*>). Only 1 allowed.
Type: `ERROR`
Explanation: You have tried to select more than one thread for an operation that only allows a single thread. Please use the ':t' focus specifier to select a single thread and try the command again.

134 Message: Formal parameter <*identifier*> is never used in macro body.
Type: `INFO`
Explanation: Formal parameter was not used in the macro body. This is only an informational message, and the macro definition will persist.

135 Message: A Source Window could not be created.
Type: `INFO`
Explanation: A Source Window could not be created because there was either no executable known to the compiler-debugger interface, or because there is no stack frame in the current state of the process which contains compiler-debugger interface information, or because no source file for any stack frame containing compiler-debugger interface information could be found.

136 Message: An incomplete geometry specification was given.
Type: `INFO`
Explanation: An incomplete geometry specification was given. Either the height, X origin, or Y origin were missing. The format for a geometry specification is (width)x(height)+(x-origin)+(y-origin).

137 Message: Bad width field '<*width*>' was given in the geometry specification.
Type: INFO
Explanation: An invalid width was given in the geometry specification for a window. Either the width was the only part of the geometry given, or it was non-numeric. The format for a geometry specification is (width)x(height)+(x-origin)+(y-origin).

138 Message: Bad height field '<*height*>' was given in the geometry specification.
Type: INFO
Explanation: A invalid height was given in the geometry specification for a window. Either the height was the only part of the geometry given, or it was non-numeric. The format for a geometry specification is (width)x(height)+(x-origin)+(y-origin).

139 Message: Bad width and height separator '<*separator*>' was given in the geometry specification.
Type: INFO
Explanation: A invalid width and height separator was given in the geometry specification for a window. Either the width was the only part of the geometry given, or it was not an 'x' character. The format for a geometry specification is (width)x(height)+(x-origin)+(y-origin).

140 Message: Bad origin separator '<*separator*>' was given in the geometry specification.
Type: INFO
Explanation: A invalid origin separator was given in the geometry specification for a window. Either the width and height or width, height, and X origin were the only part of the geometry given, or the separator was not a '+' character. The format for a geometry specification is (width)x(height)+(x-origin)+(y-origin).

141 Message: Bad origin '<*origin*>' was given in the geometry specification.
Type: INFO
Explanation: A invalid origin value was given in the geometry specification for a window. Either an incomplete geometry specification was given, the origin value was non-numeric, or an origin value was given that extends beyond the edge of the physical CRT screen. The format for a geometry specification is (width)x(height)+(x-origin)+(y-origin).

142 Message: Too wide geometry width '<*width*>' was given in the geometry specification.
Type: INFO
Explanation: The value for the width in the geometry specification for the window is wider than the physical CRT screen.

Msg. No.

143 Message: Too high geometry height '*<height>*' was given in the geometry specification.

Type: `INFO`

Explanation: The value for the height in the geometry specification for the window is higher than the physical CRT screen.

144 Message: The geometry specification of the X origin plus the width ('*<X origin + width>*') would extend past the edge of the physical screen.

Type: `INFO`

Explanation: The values for the X origin and width in the geometry specification for the window would make the window extend past the edge of the physical CRT screen.

145 Message: The geometry specification of the Y origin plus the height ('*<Y origin + height>*') would extend past the bottom of the physical screen.

Type: `INFO`

Explanation: The values for the Y origin and height in the geometry specification for the window would make the window extend past the bottom of the physical CRT screen.

146 Message: Object file *<name>* compiled with early version of compiler.

Type: `INFO`

Explanation: An object file used to create the executable being debugged was created with a compiler which is older than the current version supported by this version of CXdb. Object files must be created with the following two compilers: Convex Fortran 6.1.1 or Convex C 4.0, or greater. Recompiling the indicated file will allow CXdb to provide complete debugging support.

147 Message: Line *<line number>* is out of bounds. Valid range is 1 to *<max line number>*.

Type: `ERROR`

Explanation: You have specified a line number that is not within the valid range for a source file. The valid range is included in the error message text. You should retry your command using a line number within the valid range.

148 Message: Source unit *<number>* is out of bounds. Valid range is 0 to *<maximum>*.

Type: `ERROR`

Explanation: You have specified a source unit number which is not within the valid range for a source file. The valid range is included in the error message text. You should retry your command using a source unit number within the valid range.

149 Message: Line <*line number*> has been optimized away.

Type: ERROR

Explanation: You have specified a line number that, due to optimizations performed by the compiler, has no object code associated with it. It is impossible to place eventpoints on this line.

150 Message: Source unit <*source unit number*> has been optimized away.

Type: ERROR

Explanation: You have specified a source unit number that, due to optimizations performed by the compiler, has no object code associated with it. It is impossible to place eventpoints on this source unit.

151 Message: A process object already exists. Try the '<*command name*>' command.

Type: ERROR

Explanation: You have used a command which would create another process object. Version 1.0 of CXdb only operates on a single process. Each of the commands that create process objects have equivalents that modify an existing process object. For 'debug exec' use 'executable', for 'debug core' use 'core', and for 'debug proc' use 'attach'.

152 Message: Address <*address*> is an invalid PC setting for process [#<*process*>/<*thread*>].

Type: ERROR

Explanation: You have specified an invalid address with the 'goto' command. It is either outside the address regions of the program, or read permissions are not set for the page containing the address. The PC for this process and thread will not be modified.

153 Message: <*source unit or line*> <*file*>:<*index*> has multiple entry points.

Type: ERROR

Explanation: You have specified a source unit or line number with the 'goto' command which has multiple entry points. Because this is an ambiguous situation, the PC of the thread has not been modified. You can use the 'info sourceunit' or 'info line' command to see the entry point information and then try the 'goto address' command with a specific address from that list.

154 Message: Multiple threads selected, evaluating expression in thread <*thread-id*>

Type: INFO

Explanation: Expression evaluations must be made in the context of a process image. If that you selected more than 1 thread in the command focus, then one of the threads will be selected to be used as the context for performing the expression evaluation. This may be incorrect if the threads selected are in different contexts (or scopes). If this is the case, you should retry the command selecting only a single thread using the ':t' focus specifier.

155 Message: Expression has invalid type for use as an address.

Type: ERROR

Explanation: The expression you entered as part of a command that required an address evaluated to a type which could not be converted to an address. When an address is required, CXdb will only convert from integer types. All other types are invalid. Please try the command again using a different expression or, if you are in C, try casting the result to an integer type.

156 Message: Conversion of expression result to <*type name*> failed.

Type: ERROR

Explanation: The expression you entered could not be converted to the type indicated. Please try the command again using a different expression or, if you are in C, try casting the result to a type that can be converted to the desired type.

157 Message: Invalid signal number <*signal number*>.

Type: ERROR

Explanation: The signal number you entered is invalid. Signal numbers must be in the range 0 - 31. Please enter the command again with a valid signal number.

158 Message: Unable to push <*byte count*> bytes onto the process stack.

Type: ERROR

Explanation: CXdb was unable to push the indicated number of bytes onto the process' stack. This operation is done in preparation for calling a function within the target process (as in evaluating an expression which includes a function call). Because the data could not be placed on the stack, the call will not be performed. Further errors may result.

159 Message: CXdb is assuming the screen contains <*screen rows*> rows.

Type: INFO

Explanation: The shell from which cxdb was invoked did not provide a valid number of rows for the terminal screen. CXdb is assuming a default height as indicated.

160 Message: CXdb is assuming the screen contains <*screen cols*> columns.

Type: INFO

Explanation: The shell from which cxdb was invoked did not provide a valid number of columns for the terminal screen. CXdb is assuming a default width as indicated.

Msg. No.		
161	Message:	File *<name>* doesn't exist. Can't append with NOCLOBBER set.
	Type:	ERROR
	Explanation:	Noclobber is currently set, and you specified an append operation in a viewport specification on a file that doesn't exist. Because noclobber is set, CXdb will not create this file. You can clear noclobber mode and try again, or you can use the override redirection operator '>>!' to force the file creation.
162	Message:	File *<name>* exists. Can't overwrite with NOCLOBBER set.
	Type:	ERROR
	Explanation:	Noclobber is currently set, and you specified an overwrite operation in a viewport specification on a file that already exists. Because noclobber is set, CXdb will not truncate this file. You can clear noclobber mode and try again, or you can use the override redirection operator '>!' to force the file truncation.
163	Message:	Window ID *<id>* is invalid for use as a viewport target.
	Type:	ERROR
	Explanation:	Noclobber is currently set, and you specified an overwrite operation in a viewport specification on a file that already exists. Because noclobber is set CXdb will not truncate this file. You can clear noclobber mode and try again or you can use the override redirection operator '>!' to force the file truncation.
164	Message:	Eventpoint *<eventpoint id>* expression evaluation failed.
	Type:	ERROR
	Explanation:	The evaluation of the language expression you entered failed while testing the evenpoint's conditional for the indicated reason(s). How to resolve the problem is dependent on the reason for failure.
165	Message:	Relational expression already TRUE. Eventpoint not created.
	Type:	ERROR
	Explanation:	The language expression you entered currently evaluates to TRUE. Because a relation eventpoint halts execution when a relation becomes true, the eventpoint was not created. Please make sure that you entered the expression correctly.
166	Message:	File '*<name>*' not found within current search path.
	Type:	ERROR
	Explanation:	The file you specified could not be located anywhere within the directories currently in your search path. Check to be sure that you typed the file name correctly and that you have the necessary directories in your search path.

Msg. No.

167

Message: Reading compiler data files...

Type: `INFO`

Explanation: When reading the compiler data files, an unusually long response time may occur. This message informs the user that everything is fine.

168

Message: Unable to read previous stack frame.

Type: `ERROR`

Explanation: CXdb was attempting to read a stack frame and the attempt failed. This error will almost certainly lead to additional errors. In the case of creating relation eventpoints, the eventpoint will not be created.

169

Message: Relation eventpoint <*eventpoint #*> now out of scope. Disabled.

Type: `INFO`

Explanation: Relation type events are tied to a specific frame on the stack. When this frame is popped off the stack during normal program execution, any relation eventpoints that were associated with that frame are automatically disabled.

170

Message: File <*name*> is <*desired kind*>.

Type: `ERROR`

Explanation: An operation on a file failed. The error message contains the reason the file operation filed.

171

Message: Unknown scope path, <*scope path*>

Type: `ERROR`

Explanation: An unknown scope path was used.

172

Message: Evaluating expression in each thread context

Type: `INFO`

Explanation: Expression evaluations must be made in the context of a specific thread within the process. The evalutation of the expression you entered will be evaluated within the context of EACH thread processed. It is possible that errors may result from evaluation within one thread and not in others. This can happen if the threads are currently within different scopes in which all the same identifiers are not visible.

173

Message: Invalid offset <*number*>

Type: `ERROR`

Explanation: The offset you entered is invalid. It must be a non-zero positive integer. Additionally, it is an error if the offset specified would place the ending address outside the memory allocated to the process.

174 Message: Invalid address range *<start>*..*<end>*

Type: ERROR

Explanation: The address range you specified is invalid. Both addresses specified must reside within the address range occupied by the process, and the starting address of the range must be less than the ending address.

175 Message: Data region lies on stack. Eventpoint will be disabled when frame is popped.

Type: INFO

Explanation: The address range you specified lies within the bounds of the stack. When the stack frame containing this region is popped off the stack, the eventpoint will be automatically disabled.

176 Message: Debugger variable '*<name>*' is not *<object>* specifier

Type: ERROR

Explanation: Debugger variables have a type associated with them. You used a debugger variable in a context that required it to have a specific type and it didn't have that type. For example, when an eventpoint is created, a debugger variable may be associated with that eventpoint. Later, that variable may be used to reference that eventpoint in other commands. However, it couldn't be used in commands that operate on processes because it references an eventpoint.

177 Message: Stack popped region being monitored by eventpoint *<eventpoint #>*. Disabled.

Type: INFO

Explanation: The data region being monitored by the eventpoint indicated resided within the stack. The stack has now been popped to a point that no longer contains this region. Any data stored there 'technically' no longer exists. To prevent inaccurate results, the evenpoint has been disabled.

178 Message: Expression not valid for use as an address.

Type: ERROR

Explanation: The language expression entered is not usable as an address specifier. You can use the ':' or '..' notation to construct explicit address ranges or you can use the address operators (%LOC in Fortran and '&' in C) to obtain the address of an identifier. When using the address operator form, the size of the region will be determined from the identifier referenced.

179 Message: Cycle detected in alias translation with alias *<alias name>*

Type: `INFO`

Explanation: A cycle has been detected in alias translation. The command line is still processed, but alias translation ceases. Only the alias itself may expand with itself as part of the name. The given name is one of the aliases within the cycle. Note, by definition, there will be several other aliases within the cycle.

180 Message: Syntax Error - line:*<line number>* col:*<column number>*; *<expecting or missing>*: *<specific error token>*

Type: `ERROR`

Explanation: The C language expression you entered contained syntax errors. Please refer to a C language guide for details on C syntax.

181 Message: Identifier not visible from the current lexical scope, line: *<line number>* col: *<column number>* '*<identifier>*'

Type: `ERROR`

Explanation: The specified identifier is not visible from the current lexical scope. There are two principal causes for this message. This message can be caused either by a misspelled identifier or the use (or omission) of a scope prefix. If a scope prefix is used, the message indicates the identifier is not visible in the specified scope. If the scope prefix is omitted, the message indicates the identifier is not visible from the present lexical scope established from the current PC (or selected frame).

182 Message: Floating point overflow, line: *<line number>* col: *<column number>*

Type: `ERROR`

Explanation: This message occurs when you apply an operator to a floating point operand(s) which yields a result too large to be represented by the resulting type.

183 Message: Mixed mode operation, line: *<line number>* col: *<column number>*. Left operand of '*<operator>*' is mixed mode.

Type: `INFO`

Explanation: The left operand of the indicated operator has an internal floating point representation that is not compatible with the floating point mode established by the 'set evalopts fpmode' command. Try changing the fpmode of the evaluator to match the fpmode of the left operand. Refer to the 'set evalopts fpmode' command for more information.

184 Message: Mixed mode operation, line: <*line number*> col: <*column number*> Right operand of '<*operator*>' is mixed mode

Type: `INFO`

Explanation: The right operand of the indicated operator has an internal floating point representation, which is not compatible with the floating point mode established by the 'set evalopts fpmode' command. Try changing the fpmode of the evaluator to match the fpmode of the right operand. Refer to the 'set evalopts fpmode' command for more information.

185 Message: Divide by zero, line: <*line number*> col: <*column number*>

Type: `ERROR`

Explanation: The operation's divisor has a value of zero, which causes the operation to produce a mathematically undefined result. Verify the origin of the divisor and make the necessary changes to insure the divisor is non-zero, then retry the operation.

186 Message: Subscript truncated to integer, line: <*line number*> col: <*column number*>

Type: `INFO`

Explanation: The precision of the integral data type used in a subscript expression is greater than the precision supported by the hardware architecture. The subscript expression has been converted (truncated) to the precision supported by the hardware architecture.

187 Message: Subscript not integer type, line: <*line number*> col: <*column number*>

Type: `ERROR`

Explanation: Subscript expressions must result in an integral data type. Try converting this expression using the INT() Fortran intrinsic or by using the C cast operator (int).

188 Message: Too many subscripts specified, line: <*line number*> col: <*column number*>

Type: `ERROR`

Explanation: The number of subscripts specified in the array expression outnumber the number of subscripts defined by the referenced array. Refer to the 'info expression' command to obtain the array definition of the array expression.

189 Message: Too few subscripts specified, line: <*line number*> col: <*column number*>

Type: `ERROR`

Explanation: The number of subscripts specified in the array expression are fewer than the number of subscripts defined by the referenced array. Refer to the 'info expression' command to obtain the array definition of the array expression.

Msg. No.

190 Message: Left Hand Side of '.' is not a structure/union type, line: *<line number>* col: *<column number>*

Type: `ERROR`

Explanation: The selection operator (.) is being applied to an expression that does not yield a structure or union type. Refer to the 'info expression' command to obtain the type yielded by the expression.

191 Message: Subscript required, line: *<line number>* col: *<column number>*

Type: `ERROR`

Explanation: The array reference requires a subscript. Refer to the 'info expression' command to obtain the number of subscripts required by the type yielded by the array reference.

192 Message: Illegal indirection, line: *<line number>* col: *<column number>*

Type: `ERROR`

Explanation: An attempt has been made to reference an object through a nonpointer expression. Refer to the 'info expression' command to obtain the type yielded by the expression.

193 Message: Illegal address reference, line: *<line number>* col: *<column number>*

Type: `ERROR`

Explanation: An attempt has been made to take the address of an operand which is not a function designator, an lvalue which designates an object which is a bit field or has been declared with register storage class.

194 Message: Cannot dereference a pointer to 'void', line: *<line number>* col: *<column number>*

Type: `ERROR`

Explanation: The address expression refers to a non-existent object (void). Dereferencing the address expression is prohibited. Try casting the address expression to a type which refers to a legal object type.

195 Message: Illegal pointer/integer combination, line: *<line number>* col: *<column number>*

Type: `ERROR`

Explanation: An attempt has been made to take the address of either (1) an operand that is not a function designator, or (2) an lvalue that designates an object which is a bit field, or (3) an object that has been declared with register storage class.

196 Message: Variable's storage is not available, line: *<line number>* col: *<column number>, <identifier>.*

Type: ERROR

Explanation: The referenced variable's storage is not available. There are many reasons that may cause this. The variable may neither be read nor written by the program due to program logic. If it is a stack local variable, it's frame may not be active. If it's frame is active, the variable may be logically dead because it is no longer referenced past this point in the routine. In fact, it may have been replaced by a temporary variable introduced by the compiler in the case of induction variables.

197 Message: CXdb variable undefined, line: *<line number>* col: *<column number>*

Type: ERROR

Explanation: The CXdb variable has not been previously defined.

198 Message: Loader symbol not found, line: *<line number>* col: *<column number>*

Type: ERROR

Explanation: The specified loader symbol does not appear in the nlist for the executable assigned to the specified process. Check the spelling of the symbol. Note that global Fortran symbols have underscores prepended and appended to the symbol root. C symbols have underscores prepended to the symbol root. Assembler symbols appear exactly as defined in the assembly source.

199 Message: Variable has not yet become active, line: *<line number>* col: *<column number>*

Type: ERROR

Explanation: The variable specified in the expression is visible from the current scope. However, at present there are no active locations available for use in the evaluation. This can occur if the process has not been activated with the 'run' command. Start the process and try evaluating the expression again when the process stops as a result of an eventpoint such as a breakpoint.

200 Message: Incomplete type, line: *<line number>* col: *<column number>*

Type: ERROR

Explanation: An attempt has been made to evaluate an expression in which the definition of the object designated by the expression is not visible. An example of this situation frequently occurs when a pointer to a derived type is declared in the source file, yet the source never references the object except to assign (or access) a value to the pointer. In this case, try prefixing the derived type's tag identifier with a scope path in which the type's definition is visible in a cast expression.

Msg. No.

201 Message: Illegal sizeof expression, sizeof(void) not permitted, line: *<line number>* col: *<column number>*

Type: `ERROR`

Explanation: It is illegal to use the sizeof operator on an object of type void. The void type is an incomplete type that cannot be completed, hence no size can be computed.

202 Message: Illegal sizeof expression, sizeof(function) not permitted, line: *<line number>* col: *<column number>*

Type: `ERROR`

Explanation: It is illegal to use the sizeof operator on an object that designates a function.

203 Message: Illegal sizeof expression, sizeof(bitfield) not permitted, line: *<line number>* col: *<column number>*

Type: `ERROR`

Explanation: It is illegal to use the sizeof operator on an object that designates a bit field.

204 Message: Illegal sizeof expression, 'type' is incomplete, line: *<line number>* col: *<column number>*

Type: `ERROR`

Explanation: An attempt has been made to compute the size of an object whose definition is not visible from the present scope. This situation frequently occurs when a pointer to a derived type is declared in the source file, yet the source never references the object except to assign (or access) a value to the pointer. In this case, try prefixing the derived type's tag identifier with a scope path in which the type's definition is visible in a cast expression and re-apply the sizeof operator.

205 Message: Illegal type combination: line, *<line number>* col: *<column number>*

Type: `ERROR`

Explanation: The type name specification includes conflicting type specifiers. An example of this might be a type name specification that includes both "float" and "struct foo" type specifiers.

206 Message: Storage class specifier ignored: line, *<line number>* col: *<column number>*

Type: `INFO`

Explanation: The storage class specifier identified is not applicable in the expression and is therefore ignored. The evaluation is not affected by its specification.

207 Message: Illegal type combination, both 'signed' and 'unsigned' specified, line: *<line number>* col: *<column number>*

Type: `ERROR`

Explanation: The type name specification includes conflicting type modifiers. The type modifiers "signed" and "unsigned" semantically conflict.

208 Message: Illegal type combination, both 'short' and 'long' specified, line: *<line number>* col: *<column number>*

Type: `ERROR`

Explanation: The type name specification includes conflicting type modifiers. The type modifiers "short" and "long" semantically conflict.

209 Message: Illegal type combination, both 'short' and 'char' specified, line: *<line number>* col: *<column number>*

Type: `ERROR`

Explanation: The type name specification includes conflicting type modifiers and specifiers. The type modifier "short" cannot be applied to the type "char".

210 Message: Illegal type combination, both 'long' and 'char' specified, line: *<line number>* col: *<column number>*

Type: `ERROR`

Explanation: The type name specification includes conflicting type modifiers and specifiers. The type modifier "long" cannot be applied to the type "char".

211 Message: Illegal type combination, 'volatile' specified more than once, line: *<line number>* col: *<column number>*

Type: `ERROR`

Explanation: The type qualifier "volatile" has been specified more than once.

212 Message: Illegal type combination, 'const' specified more than once, line: *<line number>* col: *<column number>*

Type: `ERROR`

Explanation: The type qualifier "const" has been specified more than once.

213 Message: 'volatile' qualifier ignored, line: *<line number>* col: *<column number>*

Type: `INFO`

Explanation: The "volatile" type qualifier has no effect on the evaluation of the expression, therefore it has been ignored.

214 Message: 'const' qualifier ignored: line, *<line number>* col: *<column number>*

Type: `INFO`

Explanation: The "const" type qualifier has no effect on the evaluation of the expression, therefore it has been ignored.

Msg. No.

215 Message: Type redeclared 'volatile', line: *<line number>* col: *<column number>*

Type: `ERROR`

Explanation: The type name has already been declared "volatile" and is now being redeclared as such. A type may be qualified "volatile" only once.

216 Message: Type redeclared 'const', line: *<line number>* col: *<column number>*

Type: `ERROR`

Explanation: The type name has already been declared "const" and is now being redeclared as such. A type may be qualified "const" only once.

217 Message: Illegal type in cast expression, line: *<line number>* col: *<column number>*

Type: `ERROR`

Explanation: The type name specified in the cast expression is not a scalar or void type. Examples of scalar types are char, int, float, and double, where each may be further modified by short/long or signed/unsigned and qualified by const and/or volatile. A pointer to one of the above simple types or to a derived type may be specified. All derived types (except pointer) are disallowed.

218 Message: Illegal type in cast expression, cannot cast to 'function' type, line: *<line number>* col: *<column number>*

Type: `ERROR`

Explanation: It is illegal to cast an expression to "function" type.

219 Message: Operand of 'cast' has incompatible type, line: *<line number>* col: *<column number>*

Type: `ERROR`

Explanation: The operand in the specified cast expression is not a scalar. Examples of scalar types are char, int, float, and double, where each may be further modified by short/long or signed/unsigned and qualified by const and/or volatile. A pointer to one of the above simple types or to a derived type may be specified. All derived types (except pointer) are disallowed.

220 Message: Invalid expression, constant expression required, line: *<line number>* col: *<column number>*

Type: `ERROR`

Explanation: This is a syntax error. The syntax for the expression requires a constant expression; that is, an expression that consists of literals only.

Msg. No.

221 Message: Illegal subscript value, must be greater than zero, line: *<line number>* col: *<column number>*

Type: ERROR

Explanation: The type name definition requires that the constant expression evaluate to an integral value greater than zero.

222 Message: 'void' is not a legal array element type, line: *<line number>* col: *<column number>*

Type: ERROR

Explanation: An array type may not be derived from the void type. The void type is an incomplete type that cannot be completed.

223 Message: 'function' is not a legal array element type, line: *<line number>* col: *<column number>*

Type: ERROR

Explanation: An array type may not be derived from a function type.

224 Message: Functions cannot return array types, line: *<line number>* col: *<column number>*

Type: ERROR

Explanation: The specified type name defines a function that returns an array type as its result. This is an illegal type constructor.

225 Message: Functions cannot return function types, line: *<line number>* col: *<column number>*

Type: ERROR

Explanation: The specified type name defines a function that returns a function type as its result. This is an illegal type constructor.

226 Message: Illegal array dimension, dimension is null, line: *<line number>* col: *<column number>*

Type: ERROR

Explanation: The type name definition requires that the constant expression describing the bounds of the array evaluate to an integral value greater than zero. This expression evaluates to a zero sized dimension.

227 Message: Bad type name, specified tag does not identify a struct type, line: *<line number>* col: *<column number>*

Type: ERROR

Explanation: The type tag specified in the structure type name designates a type that is not a structure type.

Msg. No.

228

Message: Bad type name, specified tag does not identify a union type, line: *<line number>* col: *<column number>*

Type: `ERROR`

Explanation: The type tag specified in the union type name designates a type that is not a union type.

229

Message: Bad type name, specified tag does not identify an enumeration type, line: *<line number>* col: *<column number>*

Type: `ERROR`

Explanation: The type tag specified in the enumeration type name designates a type that is not a enumeration type.

230

Message: Incomplete type, struct/union type has not been defined, line: *<line number>* col: *<column number>*

Type: `ERROR`

Explanation: A type definition corresponding to the specified type tag cannot be found from the present scope. Try prefixing the derived type's tag identifier with a scope path in which the type's definition is visible.

231

Message: Parameter's type is expected to be an array type, line: *<line number>* col: *<column number>*

Type: `INFO`

Explanation: Parameter type mismatch. The called function expects an array type as its actual parameter. This warning does not inhibit the evaluation of the function call.

232

Message: Parameter's type is expected to be a function type, line: *<line number>* col: *<column number>*

Type: `INFO`

Explanation: Parameter type mismatch. The called function expects a function type as its actual parameter. This warning does not inhibit the evaluation of the function call.

233

Message: Actual and formal are different sized objects, line: *<line number>* col: *<column number>*

Type: `INFO`

Explanation: This message results when the called subroutine or function declares a dummy argument of one type and/or precision, and the caller passes an actual argument with a different type and/or precision.

234 Message: Too many actual parameters specified, line: <*line number*> col: <*column number*>

Type: INFO

Explanation: The function or subroutine is being called with more actual parameters than the formal definition has declared. This is an informational message only, and its purpose is to provide information which might expose a potential programming error.

235 Message: Too few actual parameters specified, line: <*line number*> col: <*column number*>

Type: INFO

Explanation: The function or subroutine is being called with fewer actual parameters than the formal definition has declared. This is an informational message only, and its purpose is to provide information which might expose a potential programming error.

236 Message: Operand truncated to integer, line: <*line number*> col: <*column number*>

Type: INFO

Explanation: The integral result of the subexpression has been truncated to an integer with a precision consisting of four bytes. This occurs as a result of constraints imposed by the hardware architecture. The resulting value of this expression will be used to obtain an address in the process' virtual memory space.

237 Message: Incompatible formal and actual parameter types, line: <*line number*> col: <*column number*>

Type: ERROR

Explanation: The type of the actual parameter is incompatible with the declaration of the formal parameter. Try referring to the function declarator (prefixing the declarator with an explicit scope path leading to its definition, such as c$foo (extern) or c$file'foo (static)) in an "info expression" command to obtain its definition.

238 Message: 'const' qualifier ignored, line: <*line number*> col: <*column number*>

Type: INFO

Explanation: The side effect produced by evaluating the expression has been permitted to occur in an effort to permit a more flexible debugging environment. This sole purpose of this message is to report the actions of the evaluator.

239 Message: Scope block not found, line: <*line number*> col: <*column number*> '<*block*>'
Type: ERROR
Explanation: The specified block is not visible from the current lexical scope. There are two main causes for this message. The first cause may be the result of a typing error in which the specified block is misspelled. The second cause arrises as a result of the use of a scope prefix. When a scope prefix is used, the message indicates the block is not visible in the specified scope chain.

240 Message: Syntax Error - line: <*line number*> col: <*column number*> Must be constant
Type: ERROR
Explanation: The real and imaginary components of a COMPLEX constant must be constant expressions.

241 Message: Syntax Error - line: <*line number*> col: <*column number*> Precision of constant too high
Type: ERROR
Explanation: The precision of the real and imaginary components of a COMPLEX constant must be either single or double precision. Specifying quad (REAL*16) precision causes this error.

242 Message: Syntax Error - line: <*line number*> col: <*column number*> REAL or INTEGER constant required
Type: ERROR
Explanation: The real and imaginary components of a COMPLEX constant must be of type INTEGER or type REAL.

243 Message: Merged scope blocks, disambiguate identifier specification, line: <*line number*> col: <*column number*> <*identifier*>
Type: ERROR
Explanation: This message occurs as a result of compiler optimizations. The compiler has merged two or more scope blocks into a single scope block. As a result of optimization, the identifier which would normally shadow an identifier with the same name in an outer scope block, now becomes visible. To disambiguate between these two (or more) identifiers, a scope prefix must be used. CXdb preserves the lexical scope, thereby permitting a scope prefix to address the desired identifier.

244 Message: No address returned after successful lookup of loader symbol: <*symbol*>
Type: ERROR
Explanation: No address returned after successful lookup of loader symbol. This is an internal error. This should be reported to the CONVEX Technical Assistance Center.

Msg. No.

245 Message: Illegal scope prefix, line: *<line number>* col: *<column number>* Scope paths are not used with the 'loader$' prefix

Type: ERROR

Explanation: This message results when a scope path has been specified in conjunction with the loader or assembler language specifier (eg. l$, loader$, or asm$). Loader symbols do not have "lexical" scope. Remove the the portion of the scope path delimited by the language specifier and the target symbol to correct the specification.

246 Message: Floating point modes conflict: CXdb [*<fpmode>*] Process [*<fpmode>*]

Type: INFO

Explanation: This message reports the condition when the evaluator's floating point mode and the process' floating point mode are incompatible. This is an informational message only; evaluation proceeds as directed.

247 Message: Floating point hardware not available, evaluation aborted

Type: ERROR

Explanation: This is a fatal evaluation error. The floating point mode assigned to the evaluator (refer to the "set evalopts fpmode" command) requires IEEE floating point hardware which is not present.

248 Message: Unacceptable type conversion, line: *<line number>* col: *<column number>* Cannot make '*<target type>*' from '*<source type>*'

Type: ERROR

Explanation: This is an internal evaluation error. This should be reported to the CONVEX Technical Assistance Center.

249 Message: Mixed mode conversion, line: *<line number>* col: *<column number>* '*<source type>*' to '*<target type>*'

Type: INFO

Explanation: This is an informational evaluation error. This message occurs as a result of the formation of an expression that contains floating point operand(s) defined in one floating point mode but used in a different mode. Evaluation proceeds as directed.

250 Message: Incompatible type, line: *<line number>* col: *<column number>* Operand of '*<unary operator>*' has incompatible type

Type: ERROR

Explanation: The operand of the unary operator has a type that is incompatible with unary operation.

251 <u>Message:</u> Constant too large, Specification of '*<constant>*' too large to be represented

<u>Type:</u> `ERROR`

<u>Explanation:</u> This message occurs if an integer constant specification is too large to be represented by the integral data types available in the language.

252 <u>Message:</u> Invalid use, line: *<line number>* col: *<column number>* '*<identifier>*' is not an array/function type

<u>Type:</u> `ERROR`

<u>Explanation:</u> This message occurs as a result of an illegal use of the identifier in a routine call or array expression. The identifier is not a routine or array designator.

253 <u>Message:</u> Subscript out of range, line: *<line number>* col: *<column number>* Value: *<subscript>* Range: *<lower bound>*..*<upper bound>*

<u>Type:</u> `INFO`

<u>Explanation:</u> The value of the subscript is not within the bounds of the dimension defined for this array.

254 <u>Message:</u> Invalid member, line: *<line number>* col: *<column number>* Left operand of '.' does not have a member '*<field name>*'

<u>Type:</u> `ERROR`

<u>Explanation:</u> The expression on the left hand side of the selector operator designates a structure or union object that does not have a member identified by "field name".

255 <u>Message:</u> Incomplete type, line: *<line number>* col: *<column number>* Size of type is unknown

<u>Type:</u> `ERROR`

<u>Explanation:</u> The referenced object has an incomplete type. No size information is available to permit evaluation to continue. This can occur as a result of a forward declaration in which the definition is not visible from the specified scope.

256 <u>Message:</u> Bad argument type, line: *<line number>* col: *<column number>* Argument to intrinsic function has incorrect type

<u>Type:</u> `ERROR`

<u>Explanation:</u> Argument to the intrinsic function has a type that is inappropriate for use by the intrinsic function. Refer to the CONVEX Fortran Language Reference manual for a description of the intrinsic function.

Msg. No.		
257	Message:	Too many arguments, line: <*line number*> col: <*column number*> Too many arguments to intrinsic function
	Type:	ERROR
	Explanation:	Too many arguments have been passed to the intrinsic function. Refer to the CONVEX Fortran Language Reference manual for a description of the intrinsic function.
258	Message:	Invalid CHARACTER length, line: <*line number*> col: <*column number*> Arguments to ICHAR() must be of length 1
	Type:	ERROR
	Explanation:	The CHARCTER argument to the ICHAR() intrinsic function has a precision greater than one. Refer to the CONVEX Fortran Language Reference manual for a description of the intrinsic function.
259	Message:	Too few arguments, line: <*line number*> col: <*column number*> Too few argments to intrinsic function
	Type:	ERROR
	Explanation:	Too few arguments have been passed to the intrinsic function. Refer to the CONVEX Fortran Language Reference manual for a description of the intrinsic function.
260	Message:	Bad argument type, line: <*line number*> col: <*column number*> Cannot take the address of a CXdb variable
	Type:	ERROR
	Explanation:	An attempt to obtain the address of a CXdb debugger variable has been discovered. CXdb variables do not exist in the virtual memory space of the debugged process.
261	Message:	Undefined result, line: <*line number*> col: <*column number*> Mathematical result is undefined.
	Type:	ERROR
	Explanation:	The formation of the expression produces a result that is mathematically undefined. Verify the values of the components in the expression to ensure the integrity of the result.
262	Message:	Incompatible type, line: <*line number*> col: <*column number*> Operands of '<*operator*>' point to incompatible types
	Type:	ERROR
	Explanation:	The operands of the specified operator refer to incompatible types.
263	Message:	Unsupported feature: <*message*>
	Type:	ERROR
	Explanation:	This functionality has been omitted from this release of CXdb.

264 Message: CXdb type information unavailable, line: *<line number>* col: *<column number>*.

Type: `INFO`

Explanation: The evaluator is unable to obtain the necessary type information to provide assistance in further debugging. Recompile the file in the present scope (if possible) using the -cxdb flag to permit the evaluator to gain access to the missing type information.

265 Message: Invalid scope specification: line: *<line number>* col: *<column number>* Descending scope path specification requires a descendent block

Type: `ERROR`

Explanation: To look "forward" in scope, one must choose the desired scope block (descendent) from which the target identifier is visible.

266 Message: Incompatible type: line: *<line number>* col: *<column number>* Left operand of '(:)' is not a CHARACTER type

Type: `ERROR`

Explanation: The substring operator requires its left operand to have CHARACTER type. Refer to the CONVEX Fortran Language Reference manual for a description of the operation.

267 Message: Incompatible type, line: *<line number>* col: *<column number>* Operands of '*<operator>*' have incompatible types

Type: `ERROR`

Explanation: The operands of the reported operator have incompatible types. Refer to either the CONVEX Fortran Language Reference manual or the CONVEX C Language Reference manual for a description of the operation.

268 Message: Lvalue required, line: *<line number>* col: *<column number>*

Type: `ERROR`

Explanation: The operation requires that the left operand of the assignment operator designate a modifiable object.

269 Message: No such scalar register < *register number>*.

Type: `ERROR`

Explanation: An attempt has been made to reference a nonexistent scalar register.

270 Message: No such vector register < *register number>*.

Type: `ERROR`

Explanation: An attempt has been made to reference a nonexistent vector register.

Msg. No.

271 Message: No such address register < *register number*>.
Type: ERROR
Explanation: An attempt has been made to reference a nonexistent address register.

272 Message: Can't reference CXdb variable < *variable name*>.
Type: ERROR
Explanation: An error has been encountered in the referencing of a CXdb variable.

273 Message: Error in accessing vector registers.
Type: ERROR
Explanation: An address was encountered that is outside the range of CXdb's cache of vector register variables.

274 Message: Invalid repeat count: <*count*>
Type: ERROR
Explanation: The repeat count specified in the command was invalid. Repeat counts for all step, next, and continue commands must be greater than zero.

275 Message: Address <*hex address*> is not within the text region of the process: [<*text start*>..<*text end*>]
Type: ERROR
Explanation: The address you specified is not within the defined text region of the process image. The 'info objectmap' command can be used to determine the object file regions that make up the text segment.

276 Message: Find pattern too long (<*length*>), max is <*max length*>
Type: ERROR
Explanation: The pattern you specified to the find command was too long. The maximum length is indicated in the message. Please try the command again with a shorter pattern.

277 Message: Invalid find pattern length (<*length*>), must be an even number.
Type: ERROR
Explanation: The pattern you specified to the find memory command was an odd number of characters. The 'find memory' command only works on whole bytes, so you must specify an even number of hex characters. Please try the command again with a pattern specifying an even number of characters.

Msg. No.

278 Message: Invalid find pattern character (<*char*>), must be a hex digit

Type: `ERROR`

Explanation: The pattern you specified to the 'find memory' command contains an invalid character. The pattern may only be composed of an even number of hexadecimal digits (representing a byte pattern in memory). The character that was invalid was included in the error message. Please try the command again with a pattern specifying only hex digits.

279 Message: Corrupt name space table for <*filename*>

Type: `INFO`

Explanation: A name space table could not be opened. Therefore, check to see if it exists. If it does, then possibly some of the internal data representations have been corrupted.

280 Message: Unable to find CDI name space table for <*filename*> in search path

Type: `INFO`

Explanation: A name space table could not be found. This file is generated by the compiler to inform CXdb about symbolic information. Therefore, check to see if it exists. If it does, then make sure that it is within one of directories in the search path. Reference the ADD PATH command for adding another directory to your search path.

281 Message: <*This operation*> only works with the CXdb Maryland Windows interface.

Type: `ERROR`

Explanation: This operation's interface requires CXdb's Maryland Window interface to function. If you are using the 'bind' or 'info bind' commands, they are not available in batch mode and they are set via the X resources interface for the X Window System.

282 Message: Ambiguous specification: line: <*line number*> col: <*column number*> '<*keyword prefix*>' is not unique among:<*keyword list*>

Type: `ERROR`

Explanation: The abbreviated command keyword prefix matches more than one possible keyword. Examine the list of possible keywords to determine a specification that uniquely specifies the keyword desired.

283 Message: Process memory read failure, address: <*address*> Cause: <*errno description*>

Type: `ERROR`

Explanation: An attempt to read from the virtural memory space of the debugged process failed. The exact cause is derived from the failure code returned by errno which is used to obtain the text associated with error. Refer to the ERRNO.H(3) man page for further details describing the failure code.

284 Message: Process memory write failure, address: <*address*> Cause: <*errno description*>

Type: ERROR

Explanation: An attempt to write to the virtual memory space of the debugged process failed. The exact cause is derived from the failure code returned by errno which is used to obtain the text associated with error. Refer to the ERRNO.H(3) man page for further details describing the failure code.

285 Message: Examine does not support <*format*>

Type: ERROR

Explanation: An attempt was made to specify a memory size and format type combination that is not supported.

286 Message: Conflicting floating point mode and format type.

Type: ERROR

Explanation: An attempt was made to specify both a floating point mode and a nonfloating point format.

287 Message: Bad floating point precision specification.

Type: ERROR

Explanation: An attempt was made to specify an unsupported or incorrect floating point width and precision specifier.

288 Message: Incomplete type, line: <*line number*> col: <*column number*> Pointer expression references an incomplete object type.

Type: ERROR

Explanation: Pointer expression references an object of type 'void' or an incomplete type. Refer to the 'info expression' command to obtain the type referred to through the pointer expression.

289 Message: Symbolic location read failure, line: <*line number*> col: <*column number*> Cause: <*errno description*>

Type: ERROR

Explanation: An attempt to read from the symbolic location of the program data specified at line/column in the debugged process failed. The exact cause is derived from the failure code returned by errno which is used to obtain the text associated with error. Refer to the ERRNO.H(3) man page for further details describing the failure code.

Msg. No.

290 Message: Symbolic location write failure, line: *<line number>* col: *<column number>* Cause: *<errno description>*

Type: `ERROR`

Explanation: An attempt to write to the symbolic location of the program data specified at line/column in the debugged process failed. The exact cause is derived from the failure code returned by errno which is used to obtain the text associated with error. Refer to the ERRNO.H(3) man page for further details describing the failure code.

291 Message: *<feature>* only available on C2 architectures.

Type: `ERROR`

Explanation: The feature indicated in the error message text is only available on the C2 Series architecture. Either the core file you are debugging is not from a process that was running on a C2, or the currrently running process is not on a C2.

292 Message: Ambiguous specification: line: *<line number>* col: *<column number>* '*<field name prefix>*' is not unique among:*<field list>*

Type: `ERROR`

Explanation: The abbreviated structure or union field prefix matches more than one possible field name. Examine the list of possible field names to determine a specification that uniquely specifies the field desired.

293 Message: Permanent memory allocations can't be done without 'sbrk' linked in the target

Type: `ERROR`

Explanation: You tried to perform an operation that required permanently allocating memory within the target process' address space. For example, this can arise when performing assignments of character strings to 'char *' types in expressions. However, the target image does not contain the 'sbrk' library function. Memory allocations can not occur without this routine. You might consider relinking your application to include this library function, or not performing the operation that requires memory allocation.

294 Message: Allocation of *<byte count>* bytes in the target process failed.

Type: `ERROR`

Explanation: You tried to perform an operation that required permanently allocating memory within the target process' address space. For example, this can arise when performing assignments of character strings to 'char *' types in expressions. However, the operation to allocate this memory failed. This could be due to several reasons: swap space is exhausted, the program has exceeded its data size limit (as set by setrlimit(2), or the maximum possible size of a data segment (compiled into the system) was exceeded.

295 Message: Process [#<*process number*>] is already stopped.
Type: INFO
Explanation: You tried to stop the execution of a process when it was already stopped. Use the 'info process' command to determine the complete status of the process.

296 Message: No process image, line: <*line number*> col: <*column number*> A process image must be available to evaluate string literals.
Type: ERROR
Explanation: All string literals are allocated in the debugged processes memory space. The evaluator cannot evaluate the literals in the expression without a process image. Refer to either the 'run', 'rerun', or 'attach' command for details on obtaining a process image.

297 Message: Memory Allocation failure, line: <*line number*> col: <*column number*> Cannot allocate memory in the process' memory space.
Type: ERROR
Explanation: All string literals are allocated in the debugged process' memory space. An attempt to allocate memory in the process' virtual memory space failed.

298 Message: Window [#<*window id*>] not found.
Type: ERROR
Explanation: The window ID you specified does not correspond to any currently active window. Please check the window ID and try the command again.

299 Message: Window [#<*window id*>] is not a source window
Type: ERROR
Explanation: You specified a window that is not a source window in a command which only operates on source windows. Please check the window ID and try the command again.

300 Message: No command matching '<*text*>' found
Type: ERROR
Explanation: The text you specified in the 'recall' command did not match any of the commands currently in the command history. Please check the text you used and try the command again.

301 Message: Operation not available in batch mode.
Type: ERROR
Explanation: You requested an operation that is not supported in batch mode. Consider using one of the interactive modes if you must use the feature.

Msg. No.

No.	Field	Text
302	Message:	Process has multiple threads. Terminated.
	Type:	ERROR
	Explanation:	The process you were debugging spawned additional threads. This release of CXdb does not support multi-threaded applications. To prevent incorrect debugger behavior, the process was terminated.
303	Message:	On-line help directory '<*directory name*>' not found.
	Type:	ERROR
	Explanation:	The on-line help directory specified does not exist. Either the on-line help database was not installed properly or it was deleted.
304	Message:	There is no previous command in the command history.
	Type:	ERROR
	Explanation:	The command history is empty, so there is no previous command to recall.
305	Message:	The environment provided to CXdb is corrupt. The first invalid entry is '<*string*>'.
	Type:	ERROR
	Explanation:	When CXdb is started, it is provided with a set of environment variables by the shell that starts it. This environment becomes the default environment in CXdb. Each entry in the environment is of the form name=string. The environment provided to CXdb when it was started contains at least one entry not of the proper form.
306	Message:	Viewport '<*string*>' is not on the viewport list.
	Type:	ERROR
	Explanation:	An attempt was made to remove a file viewport which did not exist on the viewport list specified. Check the current value of the viewport list with the 'info cxdb' command.
307	Message:	Viewport window '<*id*>' is not on the viewport list.
	Type:	ERROR
	Explanation:	An attempt was made to remove a window viewport which did not exist on the viewport list specified. Check the current value of the viewport list with the 'info cxdb' command.
308	Message:	Viewport stream '<*string*>' is not on the viewport list.
	Type:	ERROR
	Explanation:	An attempt was made to remove a stream viewport which did not exist on the viewport list specified. Check the current value of the viewport list with the 'info cxdb' command.

309 Message: A null viewport file name was passed internally.
Type: ERROR
Explanation: An internal error was discovered when a NULL viewport file name was passed. The command was aborted.

310 Message: The invalid viewport window ID '<*id*>' was passed internally.
Type: ERROR
Explanation: An internal error was discovered when an invalid viewport window ID was passed. The command was aborted.

311 Message: A null viewport stream was passed internally.
Type: ERROR
Explanation: An internal error was discovered when a NULL viewport stream was passed. The command was aborted.

312 Message: Xterm process <*process-id*> exec'ed.
Type: ERROR
Explanation: The xterm process that CXdb used to start the target process has performed an exec system call. This makes it impossible for CXdb to continue properly controlling the target process, so it has been terminated.

313 Message: Shell process <*process-id*> exec'ed.
Type: ERROR
Explanation: The shell process that CXdb used to start the target process has performed an exec system call. This makes it impossible for CXdb to continue properly controlling the target process, so it has been terminated.

314 Message: An unknown FormatCode was specified.
Type: ERROR
Explanation: An internal error was discovered during the formatting of a piece of data. This should be reported to the CONVEX Technical Assistance Center.

315 Message: An unknown type descriptor was specified.
Type: ERROR
Explanation: An internal error was discovered during the formatting of a piece of data. This should be reported to the CONVEX Technical Assistance Center.

316 Message: REAL*16 cannot be represented in IEEE floating point mode.
Type: INFO
Explanation: REAL*16 data types are only supported using native floating point mode.

Msg. No.

317

Message: Operand incompatible with /C format.

Type: `INFO`

Explanation: An attempt was made to format data as a complex floating point number. This failed either because the size of the data made it inappropriate for use as a complex floating point number, or because the data itself was incompatible with being formatted as a complex floating point.

318

Message: Mixed mode conversion from '<*source type*>' to '<*target type*>'

Type: `INFO`

Explanation: This message occurs as a result of a specification of a floating point mode which is different from the default floating point mode. Evaluation proceeds as specified.

319

Message: Operand incompatible with /i format.

Type: `INFO`

Explanation: In order to format data as an instruction, the data must be at least two bytes in length.

320

Message: The pattern '<*pattern*>' was not found.

Type: `ERROR`

Explanation: The pattern given to the 'find window' command was not found in the specified source window.

321

Message: Syntax Error - line:<*line number*> col:<*column number*>; <*expecting or missing*>: <*lexical element*>

Type: `ERROR`

Explanation: A lexical error was encountered in the parsing of your command. If at all possible, this message will try to provide the missing lexical element. After reviewing your command entry and this error message, if you are unable to rectify your specification, please refer to the online help (type "help") for this command to obtain a presentation of its syntax.

322

Message: All references to synthesized variable < *id*> are optimized away

Type: `INFO`

Explanation: A front end synthesized variable has been optimized away and so the original value is used. Front end synthesized variables are created so that language semantics are strictly adhered to. In most cases, it is used to retain an old value of the variable so that the variable may be updated later. An example in FORTRAN is when a parameter's value is saved upon entry into the routine. The synthesized variable may be optimized away later when there are no assignments to the actual parameter.

323 Message: Scope Path required, line: *<line number>* col: *<column number>* Descending scope specification is missing a block specifier.

Type: ERROR

Explanation: Descending scope specifications (those that look FORWARD in the scope chain) require at least one intervening block specifier before specifying the target identifier.

324 Message: Invalid Scope Specification, line: *<line number>* col: *<column number>* Program scope has not yet been established.

Type: ERROR

Explanation: Descending scope specifications (those that look FORWARD in the scope chain) require the process to have an active scope. This message can occur if the process has not yet been started and therefore has undefined scope. Use the 'run' or 'rerun' command to start the process combined with an eventpoint to stop the running process to establish an active scope, or, provide a complete scope path by prefixing it with a block specifier which is global to the entire program and descending from it.

325 Message: Operand incompatible with format.

Type: INFO

Explanation: Data which require sixteen bytes of storage, such as REAL*16 and COMPLEX*16, cannot be formatted as an integral type.

326 Message: Data file *<file name>* is out of date: last compiled *<date>*, created *<data file date>*

Type: ERROR

Explanation: A data file can become out of date with respect to an executable, if the data file is regenerated (by recompiling the source file) after the executable has been linked. Relinking the executable should correct the problem. If not, make sure the search path is properly set.

327 Message: Incompatible type, line: *<line number>* col: *<column number>* Left operand of '()' does not identify a function.

Type: ERROR

Explanation: The operand of '()' does not have function type. This message may appear as a result of the operand not being visible from the current scope, a typing error, or an illegal use of an operand which does not have type "function returning ...".

328 Message: Data cannot be formatted as floating point.
Type: `INFO`
Explanation: An attempt was made to format data as a floating point number. This failed either because the size of the data made it inappropriate for use as a floating point number, or because the data itself was incompatible with floating point format.

329 Message: Syntax Error - line: <*line number*> col: <*column number*> Illegal slice range specifier.
Type: `ERROR`
Explanation: A slice range has been recognized in a context to which it is not applicable. Slice ranges are only applicable in array expressions. A slice range is used in place of a subscript in an array expression to define the subset of the array's bounds.

330 Message: Specified thread <*thread-id*> ignored.
Type: `INFO`
Explanation: A thread that was specified on the command line has been ignored in the processing of this command. The specified thread is not currently active.

331 Message: Invalid upper bound, line: <*line number*> col: <*column number*>
Type: `ERROR`
Explanation: The upper bound of the specified slice range must either be equal to or greater than the specified lower bound.

332 Message: No process image; accesses to memory require a process image.
Type: `ERROR`
Explanation: The commanded operation cannot continue without a process image. Refer to either the 'run' or 'attach' commands for details on obtaining a process image.

333 Message: The Specified Region (<*region size*>) is too large.
Type: `ERROR`
Explanation: The region you specified in a watchpoint or modify eventpoint is too large. CXdb must make a copy of the region being monitored to determine when it changes. The acquisition of the memory to hold this copy failed. You must specify a smaller region to monitor.

334 Message: Invalid Slice Expression, line: <*line number*> col: <*column number*> Left operand of '[]' is not an array type.

Type: ERROR

Explanation: Slice expressions require the left operand of '[]' to have type "array of ...". Either the operand is not an array type, or its type has been defined with assumed bounds (the operand is possibly a pointer). Cast pointer types to array of ..., defining the bounds of the array.

335 Message: All threads selected in Process [#<*process number*>] are exiting.

Type: ERROR

Explanation: All the threads that you selected for execution within the process indicated are currently in the process of exiting (due to a join, wfork, or idle instruction). These threads can't be executed by themselves. At least one non-exiting thread must be executed along with these threads.

336 Message: Invalid type, line: <*line number*> col: <*column number*> Precision of integer type too small or too large.

Type: ERROR

Explanation: Integer constants (and loader symbols) used as function designators, require a precision of four bytes. The result of the subexpression to be used as the function designator is either greater than or is less than this precision. You can obtain the precision the evaluator has arrived at by using the 'info expression' command and passing it the function designator's expression, that is the left operand of the ()'s.

337 Message: Illegal pointer arithmetic, line: <*line number*> col: <*column number*>

Type: ERROR

Explanation: Function pointers are not permitted in arithmetic expressions. For more information, refer to the CONVEX C Language reference manual.

338 Message: Inappropriate value <*maxarray*> for printopts ignored.

Type: INFO

Explanation: An inappropriate value was specified for the printopts maxarray setting. Maxarray must be greater than zero and less than 2147483648.

339 Message: File < *name*> compiled with prerelease compiler, recompile

Type: INFO

Explanation: The source file was compiled with a prerelease version of the FORTRAN compiler. Please recompile the specfic module in order to have access to debugging information.

340 Message: Wildcard specifications are not allowed in this command.
Type: ERROR
Explanation: You specified a wildcard character (*) in a command that does not allow it. Please refer to the reference guide or online help system for the correct syntax of the command.

341 Message: Memory request failure, line: <*line number*> col: <*column number*>
Type: ERROR
Explanation: The evaluation of array slice expressions require heap allocation in the inferior process. This request for space has been denied for one of two reasons; either there is no additional heap space available, or the allocation request will exhaust the heap.

342 Message: Character format converted memory size to byte
Type: INFO
Explanation: The character format is available only on byte size. If any other memory size was chosen, it was ignored.

343 Message: Encountered <*a large number of*> <*args or locals*>. Processing <*a default number*>.
Type: INFO
Explanation: A large number of arguments or locals was encountered. This number is ignored in the processing of the command. However, the first few arguments or locals are accessed and processed.

344 Message: There are no text lines in < *this file*>.
Type: ERROR
Explanation: There are no lines of text in this file, although the source file is known to exist. Try updating the source search path so that the correct source file is found.

345 Message: There are no source units for < *this file*>.
Type: ERROR
Explanation: There are no source units in this file, although the source file is known to exist. Try updating the source search path so that the correct data files are found.

346 Message: Reading data file <*name*>, <*type*>.
Type: INFO
Explanation: Data files are read incrementally throughout the user's debugging session. Currently, a large data file is in the process of being read into memory. Therefore, an unusual delay will occur in this command.

Msg. No.

347 Message: Expression too complex, line: *<line number>* col: *<column number>* expr: *<expression>*

Type: ERROR

Explanation: The language expression is too complex to be parsed completely. This condition can occur if: the expression is arbitrarily large; the expression is nested too deeply; or the expression is the result of successively expanding a recursive macro. The entire preprocessed version of the expression being parsed is provided so that the effect of preprocessing can be observed.

348 Message: Command line too complex, line: *<line number>* col: *<column number>* input: *<command>*

Type: ERROR

Explanation: The CXdb command line is too complex to be parsed completely. This condition can occur if a CXdb command is nested too deeply or the command has become arbitrarily too complex as a result of successively expanding a recursive macro (or alias). The entire preprocessed version of the command line being parsed is provided so that the effect of preprocessing can be observed.

349 Message: Initializer expression type is not compatible with '*<memory type>*'.

Type: ERROR

Explanation: The data type obtained from the initializer expression is not compatible with the optional memory type specified on the command. CXdb cannot safely determine how to initialize the memory region specified when the data type of the initializer is incompatible with the specified optional memory type. If the expression syntax is C, try casting the initializer expression to match the precision of the memory type specified. If Fortran, try invoking one of CXdb's builtin intrinsics to convert the initializer to the precision of the specified memory type (eg. INT(), REAL(), QEXT(), CMPLX(), DCMPLX(), etc...). Use the 'info expression' command to determine the data type and precision of the initializer expression if needed.

350 Message: Precision of COMPLEX initializer is not compatible with '*<memory type>*'.

Type: ERROR

Explanation: The precision of the COMPLEX data type obtained from the initializer expression is not compatible with the optional memory type specified on the command. CXdb cannot safely determine how to initialize the memory region specified when the data type of the initializer is incompatible with the specified optional memory type. Try invoking one of CXdb's built-in Fortran intrinsics to convert the initializer to the precision of the specified memory type (eg. INT(), REAL(), QEXT(), CMPLX(), DCMPLX(), etc...). Use the 'info expression' command to determine the data type and precision of the initializer expression if needed.

351 Message: Precision of floating point initializer is not compatible with '<*memory type*>'.

Type: ERROR

Explanation: The precision of the floating point data type obtained from the initializer expression is not compatible with the optional memory type specified on the command. CXdb cannot safely determine how to initialize the memory region specified when the data type of the initializer is incompatible with the specified optional memory type. If the expression syntax is C, try casting the initializer expression to match the precision of the memory type specified. If Fortran, try invoking one of CXdb's built-in Fortran intrinsics to convert the initializer to the precision of the specified memory type (eg. INT(), REAL(), QEXT(), CMPLX(), DCMPLX(), etc...). Use the 'info expression' command to determine the data type and precision of the initializer expression if needed.

352 Message: Partial initialization, <*starting address*>..<*ending address*> (<*memory range*> bytes) too small for initializer (<*initializer*> bytes).

Type: INFO

Explanation: The specified memory region to be initialized does not contain enough space to permit an even multiple of stores. The last element cannot be written without exceeding the region specified. Therefore, these remaining bytes will not be initialized.

353 Message: Precision of initializer exceeds memory region, initializer: <*element size*> memory region: <*memory region*>.

Type: ERROR

Explanation: The precision of the initializer expression is greater than the capacity of the specified memory region. If the expression syntax is C, try casting the initializer expression to match the precision corresponding to the capacity of the specified memory region. If Fortran, try invoking one of CXdb's built-in Fortran intrinsics to convert the initializer to a precision corresponding to the capacity of the memory region (eg. INT(), REAL(), QEXT(), CMPLX(), DCMPLX(), etc...).

354 Message: Invalid evalopts precision setting, must be 4 or 8.

Type: ERROR

Explanation: The evalopts precision setting you entered is invalid. The values of 'iprecision' and 'rprecision' must be either 4 or 8. These values are used to determine the default size of integer and floating point constants in expression evaluation.

355 Message: Invalid ignore count: <*count*>

Type: ERROR

Explanation: The ignore count you specified is invalid. It must be greater than or equal to zero. If you used a debugger variable to specify the ignore count, rounding or truncation affects may have caused this result. The debugger variable value is implicitly converted to an integer before it is used.

356 Message: Too many statements, input: <*command*>

Type: ERROR

Explanation: The CXdb command line has too many statements. This condition is usually the result of an alias or macro expansion. If an alias or macro definition contains multiple statements, and the last statement is another macro or alias that is directly or indirectly recursive, the preprocessor and parser can loop indefinitely. This message is displayed if the parser believes this to be the case. The entire preprocessed version of the command line being parsed is displayed so that the effect of preprocessing can be observed.

357 Message: The '<*Command string*>' command expects a <*composition type*>, but received a <*composition type*>

Type: ERROR

Explanation: The command being completed with the composition expected a specific type, but received another. The composition display tells what kind of type is expected.

358 Message: Process [#<*process*>] has invalid state (<*routine*>): <*%d,state*>

Type: ERROR

Explanation: This is an internal error within CXdb. It should be reported to the Technical Assistance Center immediately. In general, CXdb has detected that the state of the process control information is inconsistent. This will almost assuredly lead to further errors.

359 Message: PIXRUN error on process [#<*process*>]: <*error text*>

Type: ERROR

Explanation: This is an internal error within CXdb. It should be reported to the Technical Assistance Center immediately. CXdb was attempting to restart the execution of the target process, and the system call failed. The reason for failure is included in the error message.

360 Message: Error trying to <*get/set*> process attributes on process [#<*process*>]: <*error text*>

Type: ERROR

Explanation: This is an internal error within CXdb. It should be reported to the Technical Assistance Center immediately. CXdb was attempting to get or set the process attributes of the process indicated, and the system call failed. The reason for failure is included in the error message.

361 Message: Error trying to <*clear/set*> the inherit mode on process [#<*process*>]: <*error text*>

Type: ERROR

Explanation: This is an internal error within CXdb. It should be reported to the Technical Assistance Center immediately. CXdb was attempting to clear or set the inherit mode of the process indicated, and the system call failed. The reason for failure is included in the error message.

362 Message: Error trying to detach process [#<*process*>]: <*error text*>

Type: ERROR

Explanation: This is an internal error within CXdb. It should be reported to the Technical Assistance Center immediately. CXdb was attempting to detach from the process indicated, and the system call failed. The reason for failure is included in the error message.

363 Message: Error trying to get signal subcode on thread [#<*thread-id*>]: <*error text*>

Type: ERROR

Explanation: This is an internal error within CXdb. It should be reported to the Technical Assistance Center immediately. CXdb was attempting to signal subcode of the thread indicated, and the system call failed. The reason for failure is included in the error message.

364 Message: Error trying to <*step/continue*> thread [#<*thread-id*>]: <*error text*>

Type: ERROR

Explanation: This is an internal error within CXdb. It should be reported to the Technical Assistance Center immediately. CXdb was attempting to step or continue the execution of the thread indicated, and the system call failed. The reason for failure is included in the error message.

365 Message: Error trying to select thread [#<*thread-id*>]: <*error text*>

Type: ERROR

Explanation: This is an internal error within CXdb. It should be reported to the Technical Assistance Center immediately. CXdb was attempting to select the thread indicated for read and write access, and the system call failed. The reason for failure is included in the error message.

366 Message: Error trying to get CWD for process [#<*process*>]: <*error text*>
Type: ERROR
Explanation: This is an internal error within CXdb. It should be reported to the Technical Assistance Center immediately. CXdb was attempting to get the current working directory for the process indicated, and the system call failed. The reason for failure is included in the error message.

367 Message: Error trying to get proc structure for process [#<*process*>]: <*error text*>
Type: ERROR
Explanation: This is an internal error within CXdb. It should be reported to the Technical Assistance Center immediately. CXdb was attempting to get the process control structure for the process indicated, and the system call failed. The reason for failure is included in the error message.

368 Message: Error trying to get thread count for process [#<*process*>]: <*error text*>
Type: ERROR
Explanation: This is an internal error within CXdb. It should be reported to the Technical Assistance Center immediately. CXdb was attempting to get the number of threads in the process indicated, and the system call failed. The reason for failure is included in the error message.

369 Message: Process [#<*process*> with pid <*pid*> is terminated.
Type: ERROR
Explanation: During the process startup phase CXdb tried to handle a process for the first time and it was found terminated. This will usually lead to a general startup failure, and the target process will probably not start. This may also leave the process control data in an inconsistent state. Use the 'kill process' command to try to resolve the problem.

370 Message: Process [#<*process*> with pid <*pid*> in unknown wait state.
Type: ERROR
Explanation: During the process startup phase CXdb tried to handle a process for the first time and it had un unknown state as indicated by the wait system call return value. This usually leads to a general startup failure, and the target process will probably not start. This can also leave the process control data in an inconsistent state. Use the 'kill process' command to try to resolve the problem.

371 Message: Error trying to get uarea for process [#<*process*>]: <*error text*>
Type: ERROR
Explanation: This is an internal error within CXdb. It should be reported to the Technical Assistance Center immediately. CXdb was attempting to get the user area structure for the process indicated, and the system call failed. The reason for failure is included in the error message.

372 Message: Error trying to pattach process with pid *<pid>*: *<error text>*
Type: `ERROR`
Explanation: This is an internal error within CXdb. It should be reported to the Technical Assistance Center immediately. CXdb was attempting to pattach to the process indicated, and the system call failed. The reason for failure is included in the error message.

373 Message: Event *<event-id>* is already disabled
Type: `INFO`
Explanation: The eventpoint indicated in the message is already disabled. Trying to disable it again has no real effect.

374 Message: Event *<event-id>* is already enabled
Type: `INFO`
Explanation: The eventpoint indicated in the message is already enabled. Trying to enable it again has no real effect.

375 Message: Descending slice (or substring) range, line: *<line number>* col: *<column number>* lower bound: *<lower bound>* upper bound: *<upper bound>*
Type: `ERROR`
Explanation: The lower bound must be less than or equal to the upper bound for the slice (or substring) expressions.

376 Message: Window functions can not be rebound to different keys.
Type: `ERROR`
Explanation: An attempt was made to bind a window function to a key. Only command line editing functions may be rebound.

377 Message: '*<function name>*' is an unknown command line editing function.
Type: `ERROR`
Explanation: An unknown command line editing function was specified as an argument to the 'bind' command. Please see the CXdb Reference page 'function-name' for a list of the available command line editing functions.

378 Message: The key name *<key name>* is unknown.
Type: `ERROR`
Explanation: An unknown key name was specified as an argument to the 'bind' command. Please see the CXdb Reference page 'key-name' for a description of the valid key names.

379 Message: The command string given to 'recall' was NULL.
Type: ERROR
Explanation: The string given to the 'recall' command was NULL. A non-NULL string is required for the 'recall' command.

380 Message: The number of history elements '<*number*>' to print is invalid.
Type: ERROR
Explanation: The number of history elements to print is invalid. A zero or negative value was given.

381 Message: No object file object was passed to a source window screen update function.
Type: ERROR
Explanation: An internal error occurred when a compiler-debugger interface object file object was not given to a source window screen update function.

382 Message: There is no active object file at the current point of execution.
Type: INFO
Explanation: There is no compiler-debugger interface information for any of the frames currently on the stack, so the source window could not be updated to reflect the current point of execution.

383 Message: There is no source file corresponding to the current CDI object file.
Type: ERROR
Explanation: An internal error occurred because there is no source file corresponding to the currently active compiler-debugger interface object file.

384 Message: No source file object was passed to a source window screen update function.
Type: ERROR
Explanation: An internal error occurred when a compiler-debugger interface source file object was not given to a source window screen update function.

385 Message: The string given to the 'find window' command was NULL.
Type: ERROR
Explanation: A NULL string was given to the 'find window' command. The 'find window' command requires a search string.

386 Message: No source unit exists in an internal source unit tree.
Type: ERROR
Explanation: An internal error occurred when a source unit tree node did not contain a source unit when one was expected. The attempted operation failed.

387 Message: No source unit exists at line *<line number>* column *<column number>* of the file '*<source file>*'.

Type: ERROR

Explanation: An attempt was made to select a source unit at a source position that has no source units associated with it. Try selecting the source unit again at a source position with executable code.

388 Message: No parent could be found for the currently selected source unit.

Type: ERROR

Explanation: No parent source unit could be found because there is no source unit tree at the position of the currently selected source unit.

389 Message: The currently selected source unit has no parent source unit.

Type: ERROR

Explanation: The currently selected source unit has no parent source unit. The most likely cause is that the currently selected source unit is a routine.

390 Message: No child could be found for the currently selected source unit.

Type: ERROR

Explanation: No child source unit could be found because there is no source unit tree at the position of the currently selected source unit.

391 Message: The currently selected source unit has no child source unit.

Type: ERROR

Explanation: The currently selected source unit has no child source units. The most likely cause is that the currently selected source unit is an innermost source unit.

392 Message: The currently selected source unit has no sibling source unit.

Type: ERROR

Explanation: The currently selected source unit has no further sibling source units; however, it may have other siblings. These other sibling source units can be found by finding the current source unit's parent and then selecting the parent's child and further sibling source units.

393 Message: There is no stack frame with CDI information at the current point of execution.

Type: ERROR

Explanation: No routines currently on the stack at or above the currently selected stack frame have compiler-debugger interface information. This means that the view in the source window may be out of date.

394 Message: Highlighting failed from line <*line*>, column <*column*>, to line <*line*>, column <*column*>.

Type: ERROR

Explanation: An internal error occurred when the range for highlighting was out of the bounds on the window being operated on.

395 Message: Unhighlighting failed from line <*line*>, column <*column*>, to line <*line*>, column <*column*>.

Type: ERROR

Explanation: An internal error occurred when the range for unhighlighting was out of the bounds on the window being operated on.

396 Message: The address expression '<*expression*>' does not match any routine with CDI information.

Type: ERROR

Explanation: The address expression given for displaying a new routine in the source window did not evaluate to be within the range of any routine known to the compiler-debugger interface.

397 Message: The line number '<*line*>' is out of the bounds of the source file.

Type: INFO

Explanation: The line number given in the filename/line number specification was out of the range of line numbers for the given source file. The line number '1' was assumed.

398 Message: The file name '<*file*>' is an unknown source file.

Type: ERROR

Explanation: The file name given in the filename/line number specification was a source file unknown to the compiler-debugger interface.

399 Message: Nested handler for eventpoint #<*event-id*> aborted.

Type: ERROR

Explanation: The eventpoint handler for the eventpoint indicated was invoked while another eventpointhandler was active. This is is possible when eventpoint handlers cause function calls to be evaluated. The execution of eventpoint handlers can not be nested so the indicated eventpoint handler was aborted.

400 Message: Process [#<*process-id*>/<*thread*>] stopped by <*signal*> can't be restarted.

Type: ERROR

Explanation: There are hardware generated signals (SIGBUS, SIGSEGV, and SIGILL) which can not be recovered from. Once one of these signals is generated further execution of the thread is not possible. There is no way to work around this restriction.

401 Message: Integer overflow on <*number*>.

Type: ERROR

Explanation: All numbers used in CXdb commands (for repeat counts, line numbers, ignore counts, etc.) must be less than 2147483647 (the maximum value of a 4-byte signed integer). The number you entered is larger than this. Please reenter the command with a valid number.

402 Message: Return value expression has an invalid type: <*type*>

Type: ERROR

Explanation: You have specified a return value that is incompatible with the current function. Please verify the return type of the function and try the command again. The 'info expression' command can give you more details on both the function name and the expression you are trying to use as a return value.

403 Message: The current function has void return type. Result ignored.

Type: INFO

Explanation: You have specified a return value for a function which has a void return type. The return value you specified is being ignored. The function will still be forced to return.

404 Message: No CDI information on current function. Assuming <*type*> return type.

Type: INFO

Explanation: There is no CDI information pertaining to the current function. The return type specified has been assumed for conversion of the return value you specified.

405 Message: Initial process failed to exec; cause: <*reason*>

Type: ERROR

Explanation: During the startup of the target process something went wrong. CXdb initially forked but was unable to exec the command to create the target process. The reason for the failure is indicated. This generally indicates something is wrong with the installation of CXdb.

406 Message: <*type-of*> process failed to exec. Startup terminated.

Type: ERROR

Explanation: During the startup of the target process something went wrong. One of the processes used to start the target process has exited for some reason. It normally comes from not being able to exec the image for some reason. Typical causes are the file is marked executable but is not in a format that the kernel will exec; the executable is not meant for the current architecture (such as running a C2 executable on a C1); or a system limit may have been exceeded. The startup process will be terminated.

407 Message: Unable to read line <*line number*> from file <*file*>

Type: ERROR

Explanation: While trying to read from the source file indicated an error was encountered. This may be due to the file having been modified since it was compiled (which may lead CXdb to try to access portions of the file that no longer exist).

408 Message: Unable to find CDI expression table information for <*filename*> in search path

Type: ERROR

Explanation: A synthesized variable table could not be found. This file is generated by the compiler to inform CXdb about symbolic information. Therefore, check to see if it exists. If it does, then make sure that it is within one of directories in the search path. Reference the ADD PATH command for adding another directory to your search path.

409 Message: Corrupt synthesized variable table for <*filename*>

Type: ERROR

Explanation: A synthesized table could not be opened. Therefore, check to see if it exists. If it does, then possibly some of the internal data representations have been corrupted.

410 Message: Inconsistent induction value for < *name*>, using <*value*>, from the set of {<*list of identifiers*>}

Type: INFO

Explanation: Several equations are used to determine the loop induction value. Sometimes, these equations may become inconsistent. Therefore, all calculated values are displayed and the lowest value is arbitrarily chosen to be used in expressions.

411 Message: Inconsistent induction value for <*identifier*> from the set of {<*list of identifiers*>}

Type: `ERROR`

Explanation: Several equations are used to determine the loop induction value. Sometimes, one of these equations may become inconsistent. Therefore, all calculated values are displayed; however, the operation cannot precede because of this inconsistency.

412 Message: Calculated induction value for following <*number of*> variable(s): <*list of identifiers*>

Type: `INFO`

Explanation: Calculated the induction value for the variable and stored the value in the above mentioned variables. That is, the induction variable was removed and replaced with the above mention synthesized variables.

413 Message: Unique object file <*name*> is not found.

Type: `ERROR`

Explanation: The unique name of the provide file was not found. This may be a result of two reasons. One reason is that no object file has that name. Another reason is that more than one object file has that identical name.

414 Message: The file <*name*> is invalid: <*reason*>

Type: `ERROR`

Explanation: The file specified as an object file was invalid for the indicated reason. This is typically because the file is not in a supported object file format.

415 Message: Breakpoint address <*address*> not in any defined text region

Type: `INFO`

Explanation: The address that you specified as a breakpoint address does not lie within any of the defined text regions of the program. This may be an error unless you are somehow intending to execute code not within the defined text areas as can happen with dynamically loaded object code.

416 Message: alias id '<*id*>' is a prefix for alias '<*other id*>'

Type: `ERROR`

Explanation: The alias name that you tried to define is a prefix of an alias that is already defined. It is illegal to define an alias which is a prefix of another alias. For example, it is illegal to have aliases 'foo' and 'foo bar'. Please choose a new alias name.

417 Message: alias id '<*id*>' would be prefixed by alias '<*other id*>'
Type: `ERROR`
Explanation: There is already an alias defined that is a prefix of the alias name you just tried to define. It is illegal to define an alias which is a prefix of another alias. For example, it is illegal to have aliases 'foo' and 'foo bar'. Please choose a new alias name.

418 Message: Invalid automatic dynamic load data: <*reason*>
Type: `ERROR`
Explanation: CXdb will automatically process dynamically loaded object files whenever a function by the special name of "_cxdb_dynamic_load" is called. The arguments to this function are expected to contain the data necessary to load the CDI information for the object file. Some error was encountered while trying to process this data. The reason for the error is shown in the message.

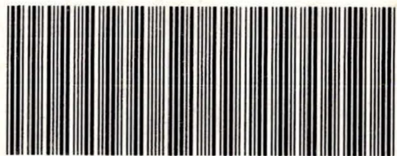
419 Message: Assignment could not be performed on <*identifier*>
Type: `ERROR`
Explanation: The assignment operation could not be performed on the above variable. This variable may either not have storage assign or be a constant typed variable.

420 Message: Subscript not arithmetic type, line: <*line number*> col: <*column number*>
Type: `ERROR`
Explanation: Subscript expressions must result in an arithmetic data type. Some examples of arithmetic types are: integer, logical, real and complex.

421 Message: Invalid operation! You may not write to a core file.
Type: `INFO`
Explanation: An operation has been attempted which would result in a write to a core file. Core files may not be written to, and the values of variables in a core file may not be modified.

422 Message: Incompatible pointers for formal and actual parameter types, line: <*line number*> col: <*column number*>
Type: `INFO`
Explanation: The type of the actual parameter is incompatible with the declaration of the formal parameter. However, these are both pointers and so execution continues.

Msg. No.



Order Number
DSW-472

Document Number
710-015430-002