CXdb Reference: Commands and Parameters

First Edition

CONVEX COMPUTER CORPORATION

606

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

CONVEX CXdb Reference: Commands and Parameters

Order No. DSW-477

First Edition
November 1992

CONVEX Press
Richardson, Texas
United States of America

CONVEX CXdb Reference: Commands and Parameters

Order No. DSW-477

Copyright © 1992 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 WARRANTY 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 OR 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 UNIX System Laboratories, Inc.

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

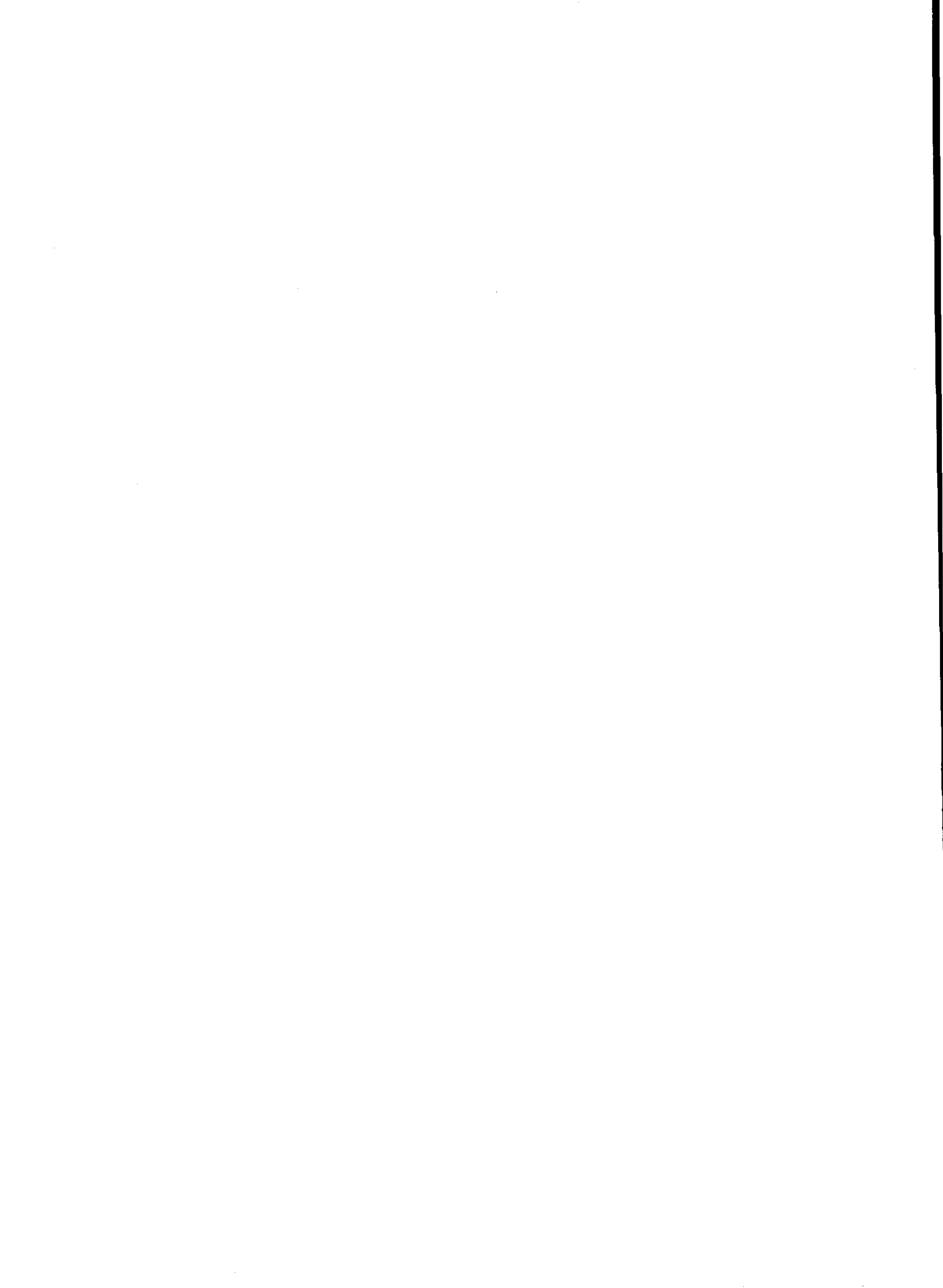
Maryland Windows is copyrighted (c) 1983 University of Maryland Computer Science Department.

Printed in the United States of America

Revision Information for

CONVEX CXdb Reference: Commands and Parameters

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



Contents

How to use this book xi
Purpose and audience xi
Organization xi
Notational conventions xii
Command syntax xii
General conventions xii
Notes and cautions xiii
Associated documents xiii
Ordering documentation xiv
Technical assistance xiv
The `contact` utility 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 autocreate` 47
`clear default environment` 49
`clear default fixed sched` 51
`clear default handler` 53

clear default remotewd 55
clear echo 57
clear environment 59
clear fixed sched 61
clear handler 63
clear logging 65
clear noclobber 67
clear seq 69
clear sqs 71
clear step 73
clear typehandler 75
continue 77
copy 79
core 81
csd 83
cxdb 87
debug core 95
debug exec 97
debug proc 101
detach 105
dirpath 107
disable event 109
disable eventtype 111
disassemble 113
display disassembly 117
display examine 119
display file. 121
display routine 123
display source. 125
display stack 127
echo 129
edit 131
enable event. 133
enable eventtype. 135
evaluate 139
event exec 141
event join 143
event modify. 147
event reached instruction 153
event reached line 157
event reached routine 161
event reached source 165
event relation. 169
event signal. 173
event spawn 177
examine 181
executable 185
fill 187

find memory backward 191
find memory forward 195
find window backward 199
find window forward 201
finish 203
frame 207
gdb 209
get 213
goto address 217
goto line 219
goto source 221
help 223
if 225
info alias 229
info args 231
info bind 233
info break 235
info cregisters 237
info cxdb 239
info default environment 243
info dirpath 245
info dynamicobject 247
info environment 249
info errno 251
info event 253
info eventtype 257
info expression 261
info formatting 265
info frame 267
info frame at 271
info history 273
info line 275
info locals 279
info macro 281
info objectmap 283
info path 285
info process 287
info psw 291
info registers 293
info scope 295
info signal 297
info sourceunit 301
info stack 303
info symbols 305
info threads 307
info trace 311
info type 313
info vregisters 317

info watch 319
kill process 321
list 323
load object 327
macro 329
next 335
next instruction 339
next over 341
print 345
put 351
pwd 355
quit 357
recall 359
remove alias 361
remove cmderr 363
remove cmdlog 365
remove cmdout 367
remove default environment 369
remove default path 371
remove dirpath 373
remove environment 375
remove event 377
remove eventtype 379
remove macro 381
remove path 383
remove variable 385
rerun 387
resume 389
return 391
run 393
set autocreate 397
set cmderr 399
set cmdlog 401
set cmdout 403
set default environment 405
set default fixed sched 407
set default format 409
set default fpmode 411
set default handler 413
set default memory 415
set default path 417
set default pshell 419
set default remotewd 421
set default step 423
set directory 425
set echo 427
set environment 429
set evalopts fpmode 431

set evalopts iprecision 433
set evalopts rprecision 435
set fixed sched 437
set format 439
set fpmode 441
set handler 443
set ignore 445
set logging 447
set memory 449
set noclobber 451
set path 453
set printopts maxarray 455
set printopts nopadding 457
set printopts padding 459
set printopts precision 461
set pshell 463
set remotewd 465
set seq 467
set shell 469
set signal 471
set sqs 473
set step 475
set threads 477
set typehandler 479
shell 481
signal process 483
signal thread 485
source 487
step 489
step instruction. 493
step over 495
stop 499
trace instruction 501
trace line 505
trace routine 509
trace source 513
watch 517

2 Parameters 523

<array-slice> 525
<debugger-variable> 529
<directory-specifier> 531
<environment-variable> 533
<event-handler> 535
<event-specifier> 537
<eventtype-specifier> 539
<file-name> 543
<frame-specifier> 545
<function-name> 547
<granularity> 549
<key-name> 553
<language-expression> 555
<line-specifier> 557
<process-list> 559
<redirection-operator> 563
<regular-expression> 567
<signal-specifier> 571
<source-unit> 573
<string> 575
<synthesized-variable> 577
<thread-list> 581
<viewport> 583

Master index 585

How to use this book

Purpose and audience

The *CONVEX CXdb Reference: Commands and Parameters* describes each of the CXdb commands. It also describes major parameters used with the commands.

This book is part of a two-volume set that also includes *CONVEX CXdb Reference: Concepts and Messages*. Together, these two books form a complete reference manual for CXdb.

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 *CONVEX CXdb User's Guide* can also help you to understand the information in this manual.

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

Organization

This manual is organized as follows:

- **Chapter 1, "Commands"**—Contains descriptions, syntax rules and examples for CXdb commands.
- **Chapter 2, "Parameters"**—Describes the major parameters used in CXdb commands.
- **Master index**—Indexes both volumes of the *CONVEX CXdb Reference.*

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
  - Indicate default aliases for CXdb commands
- `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 online man pages appear in the form exec(2), where the name of the man page is followed by its section number enclosed in parentheses.
- The shell prompt is shown as a percent sign (%).
- Unless otherwise indicated, source code examples are in FORTRAN. Where there are differences between how CXdb handles C and FORTRAN, examples in C are also shown.
- FORTRAN examples are shown in uppercase letters. You can, however, use lowercase.

Notes and cautions

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

Caution

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

Associated documents

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

- *CONVEX CXdb Reference: Concepts and Messages* (DSW-478)—The complement to this book, which presents conceptual information about how to use the CXdb commands. It also contains expanded explanations of all CXdb messages.
- *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 practical debugging examples.
- *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.
- All other locations, contact the nearest CONVEX office.

The `contact` utility

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

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

Using `contact` requires:

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

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

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 V2.0: Gary Brooks, Suzanne McBride, Steve Simmons, and Larry Streepy.
- The team who created tests to help affirm the accuracy of CXdb and this book: Marianne Becker and Lloyd Tharel.
- The people who provided vision and management for the product and documentation: Presley Smith, Mark Chiarelli, Marilyn Lutz, and Gary Brooks.
- Keith Knox, for his help in verifying the accuracy of the information in this book.
- Mary Clare Bernier, for her editorial comments that helped make this book more consistent and readable.
- The field test sites who provided valuable feedback that improved CXdb and this book.

—Raymond Cetrone and Kenneth S. Harward

Commands 1

This chapter contains reference pages for 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. The related commands are also described in this chapter.
- **Related Concepts** — A list of concepts related to the command. Concepts are described in the companion to this volume, *CONVEX CXdb Reference: Concepts and Messages.*
- **Related Parameters** — A list of parameters related to the command. CXdb parameters are described more fully in Chapter 2 of this book.

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

Full command name ——► break line

Shortest abbreviation ——► bre l

Default alias or aliases (if any exist) ——► *bl*

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

add cmderr

ad cmde

Add 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`
`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

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.

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

Examples

The following examples add directories to the default search path.

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

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

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

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

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 path`
`remove path`
`set path`
`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 default environment
clear environment            info default environment
info environment             remove default environment
remove environment           set default environment
set environment
```

Related Concepts

default environment
environment
process object

Related Parameters

environment-variable
process-list
string

add path

ad p

p+

Add directories to the search path.

Syntax

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

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

Description

The `add path` command appends the specified directories to the search path of the process object.

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

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

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

Examples

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

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

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

```
(CXdb) add path /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`
`info cxdb`
`info process`
`remove path`
`set path`
`cxdb`
`info path`
`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
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 the image of a running process.

Syntax

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

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

Description

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

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

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

To debug a process running on a remote host, precede the process ID with the remote host name and a colon.

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

Examples

The following command attaches CXdb to a running process.

```
(CXdb)  attach 12345
Attaching Process [#0] to pid 22632
Process [#0/0] stopped by attach at [0x800013de] FIB in program.f line 10
```

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

Related Commands

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

Related Concepts

process object

remote debugging

Related Parameters

process-list

backtrace

ba

bt

Display the frames of the call stack.

Syntax

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

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

Description

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

Examples

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 at`
`info frame`
`info stack`

Related Parameters

process-list
thread-list

bind

bin

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 semicolon (`;`).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

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

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

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

Examples

The following examples set breakpoints at specific instruction addresses.

```
(CXdb) break instruction 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 semicolon (;).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

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

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

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

Examples

The following examples set breakpoints at specific source lines.

```
(CXdb) break line 18

#0: break line, on [#0/*], Enabled, ignore 0/0
        [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 join
event modify
event reached instruction
event reached line
event reached routine
event reached source
event relation
event signal
event spawn
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 semicolon (;).
<*debugger-variable*>	The debugger variable assigned to this eventpoint.

Description

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

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

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

Examples

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

```
(CXdb) break routine 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 join                    event modify
event reached instruction     event reached line
event reached routine         event reached source
event relation                event signal
event spawn                   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 semicolon (`;`).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

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

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

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

Examples

The following examples set breakpoints at specific source units.

```
(CXdb) break source 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 autocreate

cl a

Disable the dynamic creation of source windows.

Syntax

```
clear autocreate
```

Description

The `clear autocreate` command disables the dynamic creation of source windows. When this setting is disabled, CXdb cannot automatically create a new source window. Initially the autocreate setting is enabled. If autocreate is disabled, it can be enabled again using the `set autocreate` command. The `clear autocreate` command is equivalent to using the `-ns` option on the `cxdb` command line.

In the CXwindows interface, you can toggle the autocreate setting using the autocreate option of the CommandWindow menu in the command window.

If used in an initialization file, or on the command line as `-ns`, CXdb does not create a source window at start up when invoked with the name of an executable file.

You can create source windows using the `display source` command. You can also display source code in the command window using the `list` command.

Examples

The following example illustrates how to disable the autocreate option.

```
(CXdb) clear autocreate
```

The above command disables the autocreate option.

Related Commands

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

Related Concepts

initialization files
windows

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 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`
`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 default remotewd

cl de re

Clear the default remote working directory.

Syntax

```
clear default remotewd
```

Description

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

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

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

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

Examples

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

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

The above command clears the default remote working directory.

clear default remotewd

Related Commands

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

Related Concepts

console working directory	process object
process working directory	remote debugging

clear echo

cl ec

Disable echoing of input from initialization files and command files.

Syntax

```
clear echo
```

Description

The `clear echo` command disables echoing.

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

Related Concepts

default environment
environment
process object

Related Parameters

process-list

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

Related Concepts

cmdlog
viewports
logging

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 Manual (C Series).*

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 sequential store enable SQS bit.

Syntax

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

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

Description

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

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

For more information about the PSW and the SQS bit, refer to the *CONVEX Architecture Reference Manual (C Series).*

Examples

The following example illustrates how to clear the SQS bit.

```
(CXdb) clear sqs
```

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

Related Commands

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

Related Parameters

process-list thread-list

clear step

cl st

Reset the stepping granularity to the default setting.

Syntax

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

```
(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.
\;	The language expression terminator.
<*destination-address*>	The starting address of the memory region that receives the copied data. This can be any <*language-expression*> that evaluates to a valid address.

Description

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

Caution

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

Examples

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

```
(CXdb) copy array_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

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

Syntax

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

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

Description

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

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

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

A remote core file can be specified by preceding the core file with the name of the remote host, a colon, and the path to the core file. Relative path names on a remote host use the remote working directory as a base. For more information, refer to the concepts page on remote debugging in the *CXdb Reference: Concepts and Messages*.

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

Examples

The following example retrieves an image from a core file.

```
(CXdb) core corefile
Core file from: a.out
thread 0 received signal 6 (IOT trap)

Process [#0/0] stopped execution at [0x80014b86]
      ___ap$kill+0xe
```

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

Related Commands

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

Related Concepts

process object
remote debugging

Related Parameters

process-list
file-name

csd

csd

Enable compatibility with csd debugger commands.

Syntax

csd

Description

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

The csd commands supported by CXdb aliases are:

csd command	CXdb equivalent
&	Use print loc(*x*) or print &*x*.
?	find window backward
alias	Use alias command.
assign	evaluate
call	print
catch	Use set signal command.
cregs	info cregisters
delete	remove event
down	Use frame command.
dump	backtrace
edit	edit
file	display file
format	No equivalent.
format decimal	set format byte dec; set format half dec; set format word dec; set format long dec; set format quad dec
format hex	set format byte hex; set format half hex; set format word hex; set format long hex; set format quad hex

csd command	CXdb equivalent
fpmode ieee	set format ieee
fpmode native	set fpmode native
fpmode auto	set fpmode dual
func	info scope; Use backtrace or display routine commands for more information.
help	help
ignore	Use set signal command.
list	Use list command.
mode	No equivalent.
mode chained	clear seq
mode sequential	set seq
next all	next
nexti	next instruction
print	Use print command.
quit	quit
rerun	Use rerun command.
return	Use return command.
run	Use run command.
regs	info registers
set num_elements =	set printopts maxarray
set precision =	set printopts precision 10
set deref_aaregs	Change format in register window.
set dump_lfmt	Use info commands.
set dumpvregs	Use info vregisters commands.
status	info event *
step	step
step all	step
stepi	step instruction
stop	No equivalent.
stop at	break line
stop in	break routine
stop if	event relation
stop threads	event spawn; event join
stopi at	break instruction
thread	info process
treads false	remove eventtype spawn, join
threads true	event spawn; event join
trace threads	event spawn {backtrace 1; echo 'thread spawned'; resume;}; event join {backtrace 1; echo 'thread joined'; resume;};
unalias	remove alias
up	Use up alias.
use	set path

csd command	CXdb equivalent
`vregs`	`info vregisters`
`whatis`	`info expression`
`when at`	`event reached line`
`when in`	`event reached routine`
`where`	`backtrace`
`whereis`	`info symbols`
`which`	`info symbols`

Examples

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

```
(CXdb) csd
```

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

Related Commands

`cxdb`
`gdb`
`info alias`
`source`

Related Concepts

csd debugger
gdb debugger

cxdb

cxdb

Invoke CXdb from the shell.

Syntax

cxdb [**-a** *<process-id>*] [**-csd**] [**-D** *<directory-specifier>*[**,** ...]]
[**-f** *<file-name>*] [**-F**] [**-l**] [**-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.
-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.
-csd	Invokes CXdb in line mode (as with the `-l` option) and automatically incorporates aliases for `csd` debugger commands.

-cw *<geometry>*
: Specifies a geometry for the command window with the Maryland Windows interface.

-D *<directory-specifier>*
: Specifies 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.

-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.

-F
: Enables fixed scheduling. This option performs the `set default fixed sched` 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.

-fw *<geometry>*
: Specifies a geometry for the display file window with the Maryland Windows interface.

-hw *<geometry>*
: Specifies a geometry for the help window with the Maryland Windows interface.

-l	Invokes CXdb in line mode. Line mode allows you to enter CXdb commands interactively through ConvexOS command-line editing, without invoking any of the CXdb windows. In line mode, echoing of command files and initialization files is disabled by default.
-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.
-pw *<geometry>*	Specifies a geometry for the process interface window with the Maryland Windows interface.
-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).
-x *<command-list>*	Specifies 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.
<x-toolkit-options>	Specifies an X toolkit option. For more information about toolkit options, refer to your X Windows documentation.

Description

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

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

- Attach CXdb to a process
- Run CXdb in line mode
- Execute `csd` debugger commands within CXdb
- Add directories to the default search path
- 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 window geometry for 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 -csd
CXdb version 2.0, Copyright (C) 1992, Convex Computer Corp.
(CXdb) clear echo
(CXdb) csd
(CXdb)
```

The above command invokes CXdb in line mode and automatically incorporates a set of predefined aliases for `csd` debugger commands. CXdb first executes the `clear echo` command to disable echoing of initialization files and command files. (To enable echoing again, use the `set echo` command.) Next CXdb executes the `csd` command to incorporate the predefined `csd` aliases. You can then enter many of the `csd` commands directly and they will be translated into equivalent CXdb commands. Refer to Appendix C of the *CONVEX CXdb User's Guide* for a list of the `csd` commands supported by 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 -l
CXdb version 2.0, Copyright (C) 1992, Convex Computer Corp.
(CXdb) clear echo
(CXdb)
```

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

```
% cxdb -nx
```

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

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

The above command invokes CXdb. After the command window appears, the two alias commands are executed. The string of commands is delimited by double quotes. Note that the alias definitions 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 `a.out` 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
csd debugger
default search path
initialization files
Maryland Windows
process object
windows
Xdefaults

Related Parameters

directory-specifier
file-name

debug core

deb c

dbgc

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

Syntax

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

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

Description

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

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

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

A core image can be debugged using absolute addresses. It can be debugged symbolically if the executable file that created the core file is specified using either the `executable` flag on the `debug core` command or the `executable` command.

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

A remote core file can be specified by preceding the file name with the name of the remote host, a colon, and the path to the core file. Relative path names use the remote working directory as a base. For more information, refer to the concepts page on remote debugging in the *CXdb Reference: Concepts and Messages*.

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

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

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

Examples

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

```
(CXdb) debug core corefile
Core file from: a.out
thread 0 received signal 6 (IOT trap)

Process [#0/0] stopped execution at 0x80033bfe

Process [#0] created
```

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

Related Commands

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

Related Concepts

debugger variables
remote debugging
process object

Related Parameters

debugger-variable
file-name

debug exec

deb e

dbg

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

Syntax

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

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

Description

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

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

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

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

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

A remote executable can be specified by preceding the executable name with the remote machine name, a colon, and the path to the executable file. Relative path names use the remote working directory as a base. For more information, refer to the concepts page on remote debugging in the *CXdb Reference: Concepts and Messages.*

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

CXdb uses the search path to find the CDI data files and source files specified in the executable file. The CDI data files exist only if the executable file was compiled with the `-cxdb` option of the CONVEX FORTRAN or C compilers. The .CXdb directory can be located in any directory on the search path.

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

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

Examples

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

```
(CXdb) debug exec a.out

Default source file: ./prog.f
Default source language: Fortran

Process [#0] created
```

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

```
(CXdb) debug exec ben:/usr/smith/a.out

Default source file: ./prog.f
Default source language: Fortran

Process [#0] created
```

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

The `run` command would now create a new process on the remote host named `ben` from this executable image.

Related Commands

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

Related Concepts

debugger variables
remote debugging
process object

Related Parameters

debugger-variable
file-name

debug proc

deb p

dbgp

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

Syntax

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

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

Description

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

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

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

A process image can be debugged using absolute addresses. It can be debugged symbolically if the CDI information for the process is specified using either the `executable` flag of the `debug proc` command or the `executable` command.

A remote process can be specified by preceding the process ID with the remote host name and a colon.

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

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

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

Examples

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

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

This command creates a process object. CXdb attaches to the process and stops it. The process image becomes the image of the process object. The process object does not yet have any CDI information.

You can specify the basis for CDI information by using the `executable` command.

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

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

Related Commands

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

Related Concepts

debugger variables	process object
remote debugging	

Related Parameters

debugger-variable

detach

det

Detach from a process.

Syntax

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

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

Description

The `detach` command detaches CXdb from a process.

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

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

Examples

The following example detaches CXdb from a process.

```
(CXdb) detach
```

The above command detaches CXdb from the process of the current process object. The process is left running outside 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

dirpath

dir

Specify an alternate directory path for the CDI data files.

Syntax

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

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

Description

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

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

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

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

CXdb searches for the CDI data files in the following order:

- The original CDI directory (hard-coded into the executable file)
- Alternate CDI directories specified with the `dirpath` command
- Directories specified in the search path

An alternate CDI directory specified with the `dirpath` command remains in effect during the entire debugging session, unless you explicitly delete it with the `remove dirpath` command. If you want to specify the same CDI directory paths in future debugging sessions, you should put the appropriate `dirpath` commands in a CXdb command file or initialization file.

Examples

The following example specifies an alternate directory path for the CDI data files.

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

The above command tells CXdb to search the directory /usr/smith/.CXdb for the CDI data files that were originally generated in the directory /usr/jones/.CXdb. Note that this command also applies to other paths that begin with /usr/jones and /usr/smith. For example, the above command also tells CXdb to search the directory /usr/smith/devl/proj2/.CXdb for the CDI data files that were originally generated in the directory /usr/jones/devl/proj2/.CXdb.

Related Commands

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

Related Concepts

command files
initialization files
Compiler-Debugger Interface
search path

disable event

disab event

dis

Disable eventpoints.

Syntax

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

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

Description

The `disable event` command disables all specified eventpoints.

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

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

Related Concepts

breakpoints
eventpoints
eventpoint handlers
tracepoints
watchpoints

Related Parameters

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.

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 disassembly

disp d

Create a disassembly window in CXwindows.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads used in evaluating the address to disassemble. The default is all threads in the current process.
<address-expression>	The address to begin disassembling from. 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.
`\;`	The language expression terminator.
<thread-number>	The number of the thread to associate with the new window. The default is no threads.
[`,` ...]	An optional list of threads to associate with the new disassembly window. Multiple thread numbers are separated by commas.

Description

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

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

You can change the threads associated with a disassembly window by using the `set threads` command, or by using the threads option under the DisassemblyWindow menu.

This command has no effect when using the Maryland Windows interface.

Examples

The following examples open new disassembly windows for the current process.

```
(CXdb) display disassembly
```

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

```
(CXdb) display disassembly fib_calculation
```

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

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

The above command opens a new disassembly window that displays the disassembled code. The debugger variable `$pc` is used to specify the starting address for disassembling. The language expression terminator (`\;`) separates the starting address from the thread list. The new window is associated with threads 0 and 1 of the current process.

Related Commands

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

Related Parameters

language-expression
thread-list
process-list

display examine

disp e

Create an examine window in CXwindows.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads used in evaluating the address to examine. The default is all threads in the current process.
<address-expression>	The address to begin displaying memory from. This can be any *<language-expression>* that evaluates to an address in the syntax of the current source language.
\;	The language expression terminator.
<thread-number>	The number of the thread to associate with the new window. The default is the current thread.

Description

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

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

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

This command has no effect when using the Maryland Windows interface.

Examples

The following examples open new examine windows for the current process.

Using FORTRAN syntax:

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

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

Using C syntax:

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

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

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

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

Related Commands

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

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
edit
display routine
pwd
```

Related Parameters

file-name

display routine

disp r

Display the source code for a routine.

Syntax

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

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

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.

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

You can change the threads associated with a source window by using the `set threads` command. In the CXwindows interface, you can set the threads for a source window using the Thread Selection dialog box. This dialog box can be opened using the threads option under the SourceWindow menu.

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 disassembly
display examine	display file
display source	display stack
examine	set threads

Related Parameters

language-expression	process-list
thread-list	

display source

disp so

Create a new source window to display a source file.

Syntax

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

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

Description

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

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

You can change the threads associated with a source window by using the `set threads` command. In the CXwindows interface, you can set the threads for a source window using the Thread Selection dialog box. This dialog box can be opened using the threads option under the SourceWindow menu.

Examples

The following examples open new source windows for the current process.

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

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

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

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

Related Commands

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

Related Parameters

file-name
process-list

display stack

disp st

Create a stack window in CXwindows.

Syntax

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

Parameter	Meaning
<*process-list*>	A list of processes affected by this command. The default is the current process.
<*thread-list*>	A list of threads used in evaluating the frame-specifier expression. The default is all threads in the current process.
<*frame-specifier*>	A relative or absolute frame number.
\;	The language expression terminator.
<*thread-number*>	The number of the thread to associate with the new window. The default is the current thread.

Description

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

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

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

You can change the thread associated with a stack window by using the `set threads` command, or by using the threads option under the StackWindow menu.

This command has no effect when using the Maryland Windows interface.

Examples

The following examples open new stack windows for the current process.

```
(CXdb) display stack
```

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

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

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

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

The above command opens a new stack window that displays the current stack for the current thread. The current frame for the current thread is set to one greater (numerically) than the previous frame. Thus, if frame 0 was the current frame, the current frame is now frame 1. The `\;` separates the frame specifier from the thread list. The new window is associated with thread 0 of the current process.

Related Commands

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

Related Concepts

scope

Related Parameters

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

Related Concepts

breakpoints
eventpoint handlers
watchpoints
eventpoints
tracepoints

Related Parameters

eventtype-specifier

evaluate

eva

Evaluate a language expression.

Syntax

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

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

Description

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

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

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

- `set evalopts fpmode` — 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.
<event-handler>	A sequence of CXdb commands enclosed in curly-braces ({ }). Each command must be terminated with a semicolon (;).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

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

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

Examples

The following examples create `exec` eventpoints.

```
(CXdb) event exec

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

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

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

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

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

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

#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 semicolon (`;`).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

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

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

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

For more information about threads, refer to the *CONVEX CXdb User's Guide.*

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
eventpoints
tracepoints
debugger variables
eventpoint handlers
watchpoints

Related Parameters

debugger-variable
process-list
event-handler

event modify

eve m

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

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process object.
<thread-list>	A list of threads affected by this 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 semicolon (;).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

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

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

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

Modify eventpoints are triggered when the 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 `80001234`. The notation `0x80001234` is C-specific and indicates the address is in hexadecimal notation. When the value stored in this range changes, the eventpoint is triggered.

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

#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 semicolon (;).
<*debugger-variable*>	The debugger variable assigned to this eventpoint.

Description

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

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

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

Examples

The following examples set eventpoints at specific instruction addresses.

```
(CXdb) event reached instruction 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 semicolon (`;`).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

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

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

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

Examples

The following examples set eventpoints at specific source lines.

```
(CXdb) event reached line 18

#0: reached line, on [#0/*], Enabled, ignore 0/0
        [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 semicolon (`;`).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

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

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

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

Examples

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

```
(CXdb) event reached routine 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.

```
(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 semicolon (`;`).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

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

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

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

Examples

The following examples set eventpoints at specific source units.

```
(CXdb) event reached source 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 semicolon (;).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

The `event relation` command sets an eventpoint watching for the value of the specified language expression to become true.

The expression must evaluate to TRUE or FALSE, as defined by the current language. The expression cannot evaluate to TRUE when the eventpoint is created.

After the execution of each statement source unit, CXdb tests to see if any enabled relation eventpoint has become true. If it has, that eventpoint is triggered.

NOTE: Due to the extra checking involved with relation eventpoints, process execution is substantially slowed.

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

Relation eventpoints are not associated with a particular address. They are triggered when their condition evaluates to `TRUE`. Eventpoints of this type are known as asynchronous eventpoints. If multiple asynchronous eventpoints are reached at the same time only the first (lowest-numbered) eventpoint is triggered. Thus, only the handler of this eventpoint is executed.

Examples

The following examples set eventpoints for relations.

```
(CXdb) event relation 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.
<signal-specifier>	The signal to be caught.
<event-handler>	A sequence of CXdb commands enclosed within curly-braces (`{ }`). Each command must be terminated with a semicolon (`;`).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

The `event signal` command sets an eventpoint to catch the specified signal.

Any signals listed in the man pages for `sigvec(2)` can be caught. When an eventpoint is set to catch a signal, the eventpoint's handler takes precedence over the `stop`, `pass`, and `print` actions specified with the `set signal` command. The actions set with the `set signal` command are not removed, so if the eventpoint is ever disabled or removed, those actions will once again determine how CXdb handles a caught signal.

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

When specifying a signal you can use its full name, its name without the `SIG` prefix, or its signal number. Signal names are not case sensitive.

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 semicolon (`;`).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

The `event spawn` command creates an eventpoint to watch for the process to spawn a thread.

When one or more threads spawn, the eventpoint is triggered. When the eventpoint is triggered, process execution stops, and then the commands of the eventpoint's handler are executed. If the eventpoint does not have its own handler, then the default handler for eventpoints, which displays a message, is executed.

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

For more information about threads, refer to the *CONVEX CXdb User's Guide.*

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

Related Parameters

language-expression
process-list
thread-list

executable

exe

Specify an executable file for a CDI base and the executable image.

Syntax

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

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

Description

The `executable` command specifies the executable file to use as the basis for the process object's CDI information. The executable file is also used to create an executable image, from which new processes are created.

Any existing CDI information or executable image is removed from the process object, and replaced by the new information and executable image. The process object must already have been created using one of the three `debug` commands.

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

CXdb uses the search path to find the CDI data files and source files specified in the executable file. The CDI data files only exist if the executable file was compiled with the `-cxdb` option of the CONVEX FORTRAN or C compilers. The .CXdb directory can be located in any directory on the search path.

A remote executable may be specified by preceding the executable name with the name of the remote host, a colon, and the path to the executable file. Relative path names use the remote working directory as a base. For more information, refer to the concepts page on remote debugging in the *CXdb Reference: Concepts and Messages.*

Examples

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

```
(CXdb) executable a.out

Default source file: ./program.f
Default source language: Fortran
```

The above command establishes the executable file named a.out located in the console working directory as the basis for the CDI information of the process object. The directory where the file was found is added to the search path of the process object.

An executable image is also created from this executable file. If neither a process image nor a core image exists in the process object, the executable image is now the image being debugged.

```
(CXdb) executable ben:/usr/smith/a.out

Default source file: ./prog.f
Default source language: Fortran
```

The above command uses the executable file located on the remote host named `ben` as the basis for the CDI information in the process object. An executable image is also created from this executable file.

The `run` command would now create a new process on the remote host named `ben` from this executable image.

Related Commands

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

Related Concepts

process object
remote debugging

Related Parameters

file-name
process-list

fill

fil

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

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.
<memory-unit>	The type of memory unit to be filled. The possible types are: b – byte (8 bits) h – halfword (16 bits) w – word (32 bits) l – longword (64 bits) q – quadword (128 bits) If you do not specify a memory unit, CXdb uses the default memory unit for the specified memory region.
<starting-address>	The starting address of the memory region to be filled. This can be any *<language-expression>* that evaluates to a valid address.
<ending-address>	The ending address of the memory region to be filled. This can be any *<language-expression>* that evaluates to a valid address.
<unit-count>	The number of memory units to be filled. This can be any *<language-expression>* that evaluates to a positive integer. The default count is all units in the memory region specified by the starting address.

<language-expression>	Any valid expression in the source language. The resulting value of the expression is filled, or written, into the memory units of the specified memory region.

Description

The `fill` command fills the specified memory region with the value of the specified language expression.

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

Related Concepts

C language expressions
language expressions
FORTRAN language expressions

Related Parameters

language-expression
thread-list
process-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/w 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`	`examine`
`find memory backward`	`print`

Related Concepts

process object

Related Parameters

language-expression	process-list
thread-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.

```
(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 quotation marks are needed because the string contains white space. 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 quotation marks are needed because the string contains white space. 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
process-list
thread-list

gdb

gdb

Enable compatibility with gdb debugger commands.

Syntax

gdb

Description

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

The gdb commands supported by CXdb aliases are:

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

gdb command	CXdb equivalent
`info files`	`info process`
`info functions`	`info symbols`
`info methods`	No equivalent.
`info sources`	`info objectmap`
`info types`	`info type`
`info variables`	`info symbols`
`jump`	Use `goto line` or `goto address` command.
`list`	Use `display file` command.
`output`	No equivalent.
`printf`	No equivalent.
`printsyms`	`info symbols >`
`ptype`	`info type`
`reverse-search`	`find window backward`
`search`	No equivalent.
`set args`	Specify arguments with `run` or `rerun` command.
`set array-max`	`set printopts maxarray`
`set base`	Use `set format` and `examine`, or `print` with format options.
`set compile off`	No equivalent.
`set compile on`	No equivalent.
`set compiled-breakpoints`	No equivalent.
`set debug-flag`	No equivalent.
`set editing off`	No equivalent.
`set editing on`	No equivalent.
`set history expansion off`	No equivalent.
`set history expansion on`	No equivalent.
`set history file`	`add cmdlog`
`set history size`	No equivalent.
`set history write off`	`clear logging`
`set history write on`	`set logging`
`set parallel fixed`	`set fixed sched`
`set parallel off`	No equivalent.
`set parallel on`	`clear fixed sched`
`set pipeline off`	`set seq`
`set pipeline on`	`clear seq`
`set prettyprint`	No equivalent.
`set unionprint`	No equivalent.
`set prompt`	Done in .Xdefaults file.
`set screensize`	No equivalent.
`set verbose off`	No equivalent.
`set verbose on`	No equivalent.
`symbol-file`	Done automatically as needed.
`tbreak`	Use an eventpoint handler.

gdb command	CXdb equivalent
`term-status`	No equivalent.
`thread`	`info threads`
`tty`	Process interface window created automatically.
`undisplay`	No equivalent.
`unset environment`	`remove environment`
`until`	`finish loop`
`up`	`frame +1`

Examples

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

```
(CXdb) gdb
```

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

Related Commands

`csd`
`info alias`
`cxdb`
`source`

Related Concepts

csd debugger
gdb debugger

get

ge

Restore the contents of memory regions from a file.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes used in evaluating this command. The default is the current process.
<thread-list>	A list of threads used in evaluating this command. The default is all threads of the current process.
<file-name>	The name of the file from which to retrieve the memory contents. Relative path names use the console working directory as a base.
<starting-address>	The first address to load. This can be any *<language-expression>* that evaluates to an address in the syntax of the current source language. If the starting address is not followed by an ending address or byte count, CXdb determines the size of the memory region based on the type of structure at the starting address.
<ending-address>	The last address to load. This can be any *<language-expression>* that evaluates to an address in the syntax of the current source language. It must be preceded by two dots (`..`).
<byte-count>	The number of bytes to load. This can be any *<language-expression>* that evaluates to a positive integer in the syntax of the current source language. It must be preceded by a colon (`:`).
[`\;` ...]	An optional list of memory regions. Multiple memory regions are separated by the language-expression terminator (`\;`).

Description

The `get` command reads the specified binary file and loads each memory region in the file into the corresponding memory regions on the command line. If no memory regions are specified, CXdb attempts to load the variables back into the memory regions from which they originally came. Therefore, you should be at an equivalent scope before issuing the `get` command. The `get` command can only read binary files created with the `put` command.

If more memory regions are specified with the `get` command than exist in the file, an error occurs. If less memory regions are specified, only the necessary number of memory regions are loaded.

The `get` command can be used to restore FORTRAN common blocks and C structures. FORTRAN common block names can be given as address expression by delimiting the name with slashes (/).

The `get` command can be used to restore array slices saved with the `put` command; however, the contents must be restored into a contiguous memory region.

Caution

CXdb does not check for architectural data type dependencies when restoring data. If the `get` command is used to restore a floating point variable using IEEE format into a floating point variable using native format, the resulting value will be incorrect.

Examples

The examples below relate to the following FORTRAN source code.

```
SUBROUTINE FIB_CALCULATION
   INTEGER VALUE, ANSWER
   INTEGER I, FIB(50)
   COMMON /BLK/ VALUE, ANSWER

   FIB(1) = 1
   FIB(2) = 1
   IF (VALUE .GT. 2) THEN
      DO I=3, VALUE
         FIB(I) = FIB(I-1) + FIB(I-2)
      ENDDO
   ENDIF

   CALL PRINT_RESULT(VALUE, FIB)
   RETURN
   END
```

The examples below use the `get` command with the above FORTRAN source code.

```
(CXdb) get file1
```

The above command reads the contents of the file named file1 and places the contents into the memory region from which they came.

```
(CXdb) get file2 /BLK/:8
```

The above command reads file2 and places the contents of the file into the common block named `BLK` (which is 8 bytes long). If the size of the memory region in the file is larger than the size of the common block, CXdb only reads enough of the file to fill the common block.

```
(CXdb) get file3 loc(FIB(1))..loc(FIB(5))
```

The above example places the contents of file3 into the specified memory region. The memory region begins with the address of the start of the array `FIB` and extends through the first address of the seventeenth element of the array (for a total of 4 elements). The FORTRAN `loc()` function provides the addresses of the array elements.

```
(CXdb) get file4 FIB:16
```

The above example places the contents of file4 into the memory region starting at the beginning address of the array `FIB` and extending for 16 bytes (4 elements of the array).

```
(CXdb) get file5 loc(ANSWER) \; loc(VALUE)
```

The above command reads the values of the variables `ANSWER` and `VALUE` from the file5 file. The language expression terminator (`\;`) separates the two language expressions. The FORTRAN `loc()` function provides the address of the variables.

Because only a starting address is specified for each variable, the size of the memory region is automatically set to the size of the variable.

Related Commands	`examine`	`put`
Related Concepts	C language expressions FORTRAN language expressions	console working directory language expressions
Related Parameters	array-slice process-list	file-name thread-list

goto address

g a

Branch to the specified address.

Syntax

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

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

```
(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

g l

Branch to the specified source line.

Syntax

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

Parameter	Meaning
<*process-list*>	A list of processes affected by this command. The default is the current process.
<*thread-list*>	A list of threads affected by this command. The default is all threads of the current process.
<*line-specifier*>	The line specifier for the line of source code to branch to. The line specifier can include a source file name as well as the line number. (Line numbers are assigned 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 line`
`info line`
`info sourceunit`
`goto address`
`info frame`
`info registers`
`info stack`

Related Concepts

scope
source units

Related Parameters

process-list
thread-list
source-unit

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.
- **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 in the previous example. The relational expression uses C syntax rather than FORTRAN.

The following example uses the `if` command as it might appear in a command file. The entire command file is shown.

```
debug exec a.out
break line 10
run
if (A .lt. 1500) {\
   echo 'Number not large enough';\
   echo 'Increase X factor';}\
else {\
   echo 'Number large enough';\
   source cmdfile.2;}
echo 'command file finished'
```

The above command file demonstrates one possible use of the `if` command. The `if` command checks if `A` is less than 1500. If it is, the messages echo and the command file finishes.

If `A` is not less than 1500, the message echoes and the `cmdfile.2` command file is sourced. Using `if` commands, you can control the flow of command files.

Related Commands

`evaluate`
`set default handler`
`set typehandler`
`resume`
`set handler`
`source`

Related Concepts

command files
eventpoints
initialization files
debugger variables
eventpoint handlers

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

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.

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 are 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

process-list	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 (C Series).*

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 dirpath

i di

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

Syntax

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

Parameter	Meaning
<*regular-expression*>	A search pattern specified as a regular expression. A process object must exist before you can specify a regular expression with this command.

Description

The `info dirpath` command can be used in two different ways:

- Without a regular expression — To list the alternate CDI directory paths that have been created with the `dirpath` command.
- With a regular expression — To list the names of object files for the current process that match the regular expression.

A process object must exist before you can specify a regular expression with the `info dirpath` command.

Examples

The following examples illustrate both ways of using the `info dirpath` command.

```
(CXdb) info dirpath
    From                        To
 1. /usr/jones              -> /usr/smith
 2. /usr/jones              -> /mnt/special/proj5
```

The above command lists all CDI directory paths that have been created with the `dirpath` command during the current debugging session. In this example, the original path to the CDI data files is /usr/jones. The two alternate paths to the same CDI data files are /usr/smith and /mnt/special/proj5.

```
(CXdb) info dirpath v
    Object File              Directory
 1. value.o                  /usr/jones
 2. varDesc.o                /usr/jones
 3. varDescList.o            /usr/jones
 4. vector.o                 /usr/smith
```

The above command lists the names of all the object files for the current process that begin with the letter `v`. It also lists the directory where each object file was originally created during compilation.

Related Commands

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

Related Concepts

Compiler-Debugger Interface
process object
search path

info dynamicobject

in dy
i dy

Display memory segments of dynamically loaded objects.

Syntax

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

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

Description

The `info dynamicobject` command displays a table showing the memory segments of all object files that have been dynamically loaded into memory. Object files that have been dynamically loaded can be specified to CXdb during a debugging session by using the `load object` command.

NOTE: CONVEX does not provide, 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
eventpoint handlers
watchpoints
eventpoints
tracepoints

Related Parameters

event-specifier

info eventtype

in eventt
et?

Display all eventpoints of the specified type.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<eventtype-specifier>	An eventtype whose eventpoints you want to display.
[, ...]	An optional list of additional eventpoint types. Multiple eventpoint types are separated by commas.

Description

The `info eventtype` command displays information about the eventpoints of the specified eventpoint types.

The following is a list of eventpoint types:

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

Eventpoints are separated by type. If the default handler has been changed for the eventpoint type, the new default handler is displayed.

For each eventpoint, the number, type, enabled setting, ignore count, address and symbolic location (if applicable) are displayed. If a handler has been specified for the eventpoint, then the commands of the handler are displayed.

Examples

The following examples display the eventpoints of different eventpoint types.

```
(CXdb) info eventtype break

#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 process`
`set evalopts fpmode`
`set evalopts rprecision`
`info cxdb`
`print`
`set evalopts iprecision`

Related Concepts

language expressions
synthesized variables

Related Parameters

language-expression
synthesized-variable
process-list
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:
              padding: Off
    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 `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

info frame

in fr

i fr

Display a stack frame.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.
<frame-specifier>	A relative or absolute frame number.

Description

The `info frame` command displays information about the specified frame of the process stack.

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

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

info frame at

in fr a
i fr a

Display the stack frame at the specified address.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.
<language-expression>	An expression that evaluates to a frame address in the source language.

Description

The `info frame at` command displays information about the stack frame stored at the specified address.

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

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.

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

Related Concepts

breakpoints
eventpoints
source units
stepping
tracepoints

Related Parameters

granularity
source-unit
line-specifier

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 o

Display the object map.

Syntax

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

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

Description

The `info objectmap` command displays summary information about the object map for the specified process. The object map information is available only for files compiled with the `-cxdb` option. The information 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 path

in pa

p+

Display the directories in the search path.

Syntax

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

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

Description

The `info path` command displays the directories in the search path of the process object. CXdb uses the search path to find the source code for the current executable.

The search path can be modified using the `add path` and `remove path` commands. The `set path` command can be used to replace the entire search path with a new set of directories.

Examples

The following example displays the search path of the current process object.

```
(CXdb) info path
Default search list:
        .


Process [#0] search list for: a.out
        .
```

The above command displays the search path. In this example, the search path consists of the current working directory (.).

Related Commands

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

Related Concepts

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

Related Parameters

process-list

info process

in pr

p?

Display the status of the process.

Syntax

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

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

Description

The `info process` command displays information about the current process object. The information displayed includes the status of the process, the image, and the search path. If CXdb is running under ConvexRTS/rtk, additional rtk information is displayed.

Examples

The following example displays information about the 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

     remote host: pixel
     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.
- `remote host` — If debugging is being done remotely, the name of the remote host is displayed.
- `working dir` — The process working directory. This is the directory from which your process is run.
- `default step` — The granularity used if a granularity is not specified in a stepping command. The possible granularities are routine, block, loop, statement, and expression. The initial default is statement.
- `default language` — The default language used by this program. CXdb determines the default language from the language of the main routine of your program.
- `threads` — The threads in your process. If multiple threads are active, their numbers are displayed.
- `current thread` — The current thread of your process 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.

The following example shows the `info process` command being used on a ConvexRTS/rtk system. For ConvexRTS/rtk systems, `info application` is an alias for the `info process` command.

```
(CXdb) info process

status of process [#0]:

      executable: /usr/smith/work/environ
       arguments: arg1 arg2
fixed scheduling: off
          pshell: csh
    image status: created pid 0, state = stopped

     remote host: shelrtk
     working dir: (none)
    default step: statement
default language: C
         threads: 2 [active: 0,1]
  current thread: 0

 thread 0 status: stopped at [0x800052a2] environ'main in environ.c line 13
 thread 1 status: stopped at [0x8000fb52] _msg_recv+0xa

Values of application attributes:

default application name: environ
       default max tasks: 16
        actual max tasks: 16
     default file system: nfs
      actual file system: nfs

source file search path:
        .
        /usr/smith/work
```

The above command displays information about the current process object. In addition to the normal information displayed, the application attributes for the application being run are listed.

Related Commands

```
add path              attach
clear fixed sched     core
debug core            debug exec
debug proc            detach
executable            info cxdb
info threads          remove path
rerun                 run
set fixed sched       set pshell
set path              set remotewd
```

Related Concepts

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

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 (C Series).*

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 (C Series).*

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

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 concepts page on signals in the *CXdb Reference: Concepts and Messages.*

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 `11`, column `10` 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 or tasks 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 or tasks.

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` — The status of each active thread (in this case thread 0). A thread is said to be active if it is alive. If a process exists, at least one thread is alive. The information displayed for each thread is described below:
  - `stopped at` — The address at which the thread was stopped. In this case, the thread was stopped at address `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.

The following example shows the `info threads` command used under ConvexRTS/rtk.

```
(CXdb) info threads

Status of process [#0] threads:

  thread count: 2
active threads: 0
current thread: 0

      Thread 0: stopped at [0x800052a2] environ'main in environ.c line 13
                by breakpoint

Values of task attributes
              default priority: 128
               actual priority: 128
    default maximum stack size: 16384
     actual maximum stack size: 16384
default task private data size: 4096
 actual task private data size: 1320
```

The above command displays information about the task of the current process.

Related Commands

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

Related Concepts

eventpoints

Related Parameters

process-list
thread-list

info trace

in tr
t?

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 (C Series).*

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

Related Concepts

breakpoints
eventpoint handlers
watchpoints
eventpoints
tracepoints

Related Parameters

process-list
thread-list

kill process

k p
k

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

Syntax

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

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

Description

The `kill process` command terminates a running process. The `kill process` command also removes the core or process image being debugged from the process object.

After the image is removed, the executable image of the process object becomes the image being debugged. If the process object does not have an executable image, then nothing is being debugged. You can specify an executable image for the process object using the `executable` command.

When you use the `kill process` command, CXdb asks you to confirm that you want to kill the specified process. If you answer yes, the process is terminated, and the image of the process is removed from the process object. If you answer no, the process is not terminated.

Examples

The following example kills a process.

```
(CXdb) kill process
Kill process [#0]? y
Terminated execution of Process [#0]
```

The above command kills the current process. The image of the process is removed from the process object. A new process can be created using the `run` or `rerun` command, or attached using the `attach` command.

Related Commands

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

Related Concepts

process object

Related Parameters

process-list

list

l

List lines of source code to cmdout.

Syntax

list [[*<file-name>*:] *<starting-line>*] [*<number-of-lines>*]

Parameter	Meaning
<file-name>	The name of the source file. Relative path names use the console working directory as a base.
<starting-line>	The line in the source file to start the listing.
<number-of-lines>	The number of lines to list. The default is 10.

Description

The `list` command lists to cmdout the source code from a source file. Initially, cmdout is the command window.

If the starting line is not specified, CXdb displays the source code from 5 lines before the beginning of the active source unit. If there is not an active source unit, the list begins with the first line of the specified source file. If the source file is not specified, the source file containing the main routine is chosen.

You must specify a starting line number when specifying a source file. The source file must have been compiled as part of the process currently being debugged.

If the number of lines to list is not specified, 10 lines are listed.

Subsequent `list` commands without arguments continue listing the same source file, 10 lines at a time.

Examples

The following examples list lines of source code. For the purpose of these examples, assume that the program being debugged has not yet begun execution. The source file for the program is fib.f.

```
(CXdb) list
 1: c      A FORTRAN PROGRAM TO CALCULATE FIBONACCI VALUES
 2:
 3:        PROGRAM FIB
 4:        INTEGER I, VALUE, ANSWER, NUMARGS
 5:        CHARACTER ARG
 6:
 7:        NUMARGS = IARGC()
 8:        IF (NUMARGS .GT. 0) THEN
 9:           PRINT *, "NUMBER  FIBONACCI"
10:           PRINT *, "------  ---------"
```

The above example displays 10 lines of source code, beginning with the first line of the source file that contains the main routine, because this is the first use of the `list` command.

```
(CXdb) list
11:           DO I = 1,NUMARGS
12:              VALUE = GETVALUE(I)
13:              IF (VALUE .NE. 0) THEN
14:                 CALL FIB_CALCULATION(VALUE, ANSWER)
15:              ENDIF
16:           ENDDO
17:        ELSE
18:           PRINT *, "No numbers were specified on the command line.  Specify the"
19:           PRINT *, "numbers you want fibonacci values for with blanks between"
20:           PRINT *, "each number.  Numbers must be between 0 and 40."
```

The above command displays 10 lines of source code from the same source file, but starting with the line after the last line listed. Line 10 was the last line listed in the first example, so this listing begins with line 11.

```
(CXdb) list 1
 1: c      A FORTRAN PROGRAM TO CALCULATE FIBONACCI VALUES
 2:
 3:        PROGRAM FIB
 4:        INTEGER I, VALUE, ANSWER, NUMARGS
 5:        CHARACTER ARG
 6:
 7:        NUMARGS = IARGC()
 8:        IF (NUMARGS .GT. 0) THEN
 9:           PRINT *, "NUMBER  FIBONACCI"
10:           PRINT *, "------  ---------"
```

The above command lists 10 lines of source code from the source file containing the `main` routine, starting with line 1.

```
(CXdb) list 7 5
 7:        NUMARGS = IARGC()
 8:        IF (NUMARGS .GT. 0) THEN
 9:           PRINT *, "NUMBER  FIBONACCI"
10:           PRINT *, "------  ---------"
11:           DO I = 1,NUMARGS
```

The above command lists 5 lines from the source file containing the `main` routine, starting with line 7.

```
(CXdb) list fib.f:25 5
25:
26:        FUNCTION GETVALUE(ARGNUM)
27:        INTEGER I, ARGNUM, TEMP
28:        CHARACTER*2 ARG
29:
```

The above command lists 5 lines from the source file fib.f, starting at line 25.

For the next example, assume that program execution has stopped at line 56 of the fib.f source file.

```
(CXdb) list
51:
52:       FIB(1) = 1
53:       FIB(2) = 1
54:       IF (VALUE .GT. 2) THEN
55:          DO I=3, VALUE
56:             FIB(I) = FIB(I-1) + FIB(I-2)
57:          ENDDO
58:       ENDIF
59:
60:       CALL PRINT_RESULT(VALUE, FIB)
```

In the above example, 10 lines were listed surrounding the current point of execution. Line 56 contained the beginning of the active source unit, so the listing began from five lines above, at line 51.

Related Commands

`display routine` `display source`

Related Concepts

console working directory source units

Related Parameters

file-name

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

macro

m

Define a macro.

Syntax

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

Parameter	Meaning
<name>	The name of the macro. The name may consist of alphanumeric characters only, and it is case sensitive.
<parameter>	A positional parameter that is passed to the macro. If a comma is to be passed as part of the parameter, a backslash (\) must precede the comma.
<default>	The default value for the specified parameter. A colon (:) must separate the parameter name from its default value.
[, ...]	Additional parameters and their corresponding default values. Multiple parameters in the list must be separated by commas. Spaces between the entries are optional.
<string>	The text that forms the body of the macro. If the text contains spaces, then it must be enclosed in quotation marks. Wherever CXdb encounters the macro name, it substitutes the body of the macro along with the specified parameters.

Description

The `macro` command defines a macro in the CXdb command language.

CXdb treats a macro as a text substitution. A macro can substitute for part of a command, a complete command, or multiple commands. Macros can accept parameters, and the parameters can have default values. Macros may be nested within other macros, and they can execute iteratively.

To invoke a macro, use an at-sign (@) in front of its name. Whenever the macro is invoked, CXdb expands it by substituting the text of the macro body in place of the macro name.

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

Examples

The following examples illustrate how to define and invoke macros.

```
(CXdb) macro sl(N:1) "step loop N; info locals"
```

The above command defines a macro called `sl`. This macro contains two CXdb commands: `step loop` and `info locals`. The parameter `N` represents the number of loops to step, and the default value for `N` is `1`.

To invoke the above macro, enter the following:

```
(CXdb) @sl
Stepping process [#0/*] by 1 loop
Process [#0/0] stopped 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

n

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

n i

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.

```
(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

n o

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

Related Concepts

process object
stepping
source units

Related Parameters

granularity
thread-list
process-list

print

pr

p

Evaluate a language expression and print the result.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.
n	A line control that suppresses the newline character at the end of the printed data.
<format>	The format for displaying the memory units. If a format is not specified, the default format for the specified process is used. The format specifications are:

- B — binary
- C — FORTRAN complex
- L — FORTRAN logical
- c — ASCII character
- d — signed decimal
- e[*<width>*.*<precision>*] — scientific notation
- f[*<width>*.*<precision>*] — floating point notation
- i — instruction
- o — octal
- s — string
- u — unsigned decimal
- x — hexadecimal

<fpmode>	The mode for floating point calculations, which can be one of the following: • `D` — Dual floating point mode • `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`
`info formatting`
`set default format`
`set default memory`
`set format`
`set memory`
`set printopts maxarray`
`set printopts precision`

Related Concepts

C language expressions
debugger variables
FORTRAN language expressions
language expressions
scope

Related Parameters

array-slice
language-expression
redirection-operator
debugger-variable
process-list
thread-list

put

pu

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

Syntax

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

Parameter	Meaning
<process-list>	A list of processes used in evaluating this command. The default is the current process.
<thread-list>	A list of threads used in evaluating this command. The default is all threads of the current process.
<file-name>	The name of the destination file for the memory contents. Relative path names use the console working directory as a base.
<starting-address>	The first address to save. This can be any *<language-expression>* that evaluates to an address in the syntax of the current source language. If the starting address is not followed by an ending address or byte count, CXdb determines the size of the memory region based on the type of structure at the starting address.
<ending-address>	The last address to save. This can be any *<language-expression>* that evaluates to an address in the syntax of the current source language. It must be preceded by two dots (`..`).
<byte-count>	The number of bytes to save. This can be any *<language-expression>* that evaluates to a positive integer in the syntax of the current source language. It must be preceded by a colon (`:`).
[`\;` ...]	An optional list of memory regions. Multiple memory regions are separated by the language-expression terminator (`\;`).

Description

The put command creates the specified file and stores the size and contents of the specified memory regions in the file. The contents can then be loaded back into the same memory regions at a later time using the get command.

The put command can be used to save and restore FORTRAN common blocks and C structures. FORTRAN common block names can be given as address expressions by delimiting each name with slashes (/). You can also save an array slice, but an array slice may only be restored to a contiguous memory region.

If the file exists, it is overwritten unless the NOCLOBBER option is enabled.

Examples

The examples below relate to the following FORTRAN source code.

```
SUBROUTINE FIB_CALCULATION
   INTEGER VALUE, ANSWER
   INTEGER I, FIB(50)
   COMMON /BLK/ VALUE, ANSWER

   FIB(1) = 1
   FIB(2) = 1
   IF (VALUE .GT. 2) THEN
      DO I=3, VALUE
         FIB(I) = FIB(I-1) + FIB(I-2)
      ENDDO
   ENDIF

   CALL PRINT_RESULT(VALUE, FIB)
   RETURN
   END
```

The examples below use the put command with the above FORTRAN source code.

```
(CXdb) put file1 /BLK/:8
```

The above command places the contents of the common block BLK into the binary file file1. The memory region extends for 8 bytes, beginning with the starting address of the common block. In this case, the entire common block is saved because the common block consists of two integer variables, VALUE and ANSWER, each of which is 4 bytes.

```
(CXdb) put file2 FIB
```

The above command creates a binary file named file2 and places the contents of the array `FIB` into it. Because only a starting address was specified, CXdb assumes the size of the region to be the size of the entire array.

```
(CXdb) put file3 loc(FIB(1))..loc(FIB(5))
```

The above example places the contents of the memory region into a new binary file named `file3`. The memory region begins with the address of the start of the array and extends through to the starting address of the fifth element of the array (for a total of 4 elements of the array). The FORTRAN `loc()` function is used to determine the address of the array elements.

```
(CXdb) put file4 FIB:16
```

The above example stores the contents of the memory region into file4. The memory starts at the address of the array `FIB` and extends for 16 bytes (4 elements of the array, because each element is 4 bytes).

```
(CXdb) put file5 loc(ANSWER) \; loc(VALUE)
```

The above command places the values of the variables `ANSWER` and `VALUE` into the file5 file. The language-expression terminator (`\;`) is necessary to separate the two address expressions. The FORTRAN `loc()` function is used to determine the address of the variables.

Because only a starting address is specified for each variable, the size of the memory region is automatically set to the size of the variable.

The `put` command can also be used to save the contents of C language structures. If the name of a C language structure is given as the starting address of a memory region to save, CXdb determines the size of the structure and saves the appropriate memory region.

Related Commands

`clear noclobber`
`examine`
`get`
`set noclobber`

Related Concepts

C language expressions
console working directory
FORTRAN language expressions
language expressions

Related Parameters

array-slice
file-name
process-list
thread-list

pwd

pw

Display the name of the console working directory.

Syntax

pwd

Description

The `pwd` command displays the current setting of the console working directory.

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

Related Concepts

cmderr
cmdlog
cmdout
logging
viewports
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
```

To remove the file `cxdb_input` from the list, 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.

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

Examples

The following examples remove directories from the default search path.

```
(CXdb) remove default path /mnt/jones/project/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.

```
(CXdb) remove default path
      /mnt/jones/libraries,/mnt/jones/math/libraries
```

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

Related Commands

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

Related Concepts

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

Related Parameters

directory-specifier

remove dirpath

rem di

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

Syntax

remove dirpath [<*original-directory*>]

Parameter	Meaning
<*original-directory*>	The directory where the program object files were originally compiled. If you do not specify this parameter, all alternate CDI directory paths are removed. The directory path name must begin with root (/). Do not include the .CXdb subdirectory as part of the path name.

Description

The `remove dirpath` command removes one or more path names from the list of alternate CDI (Compiler-Debugger Interface) directory paths. You can remove only alternate CDI directory paths that were added with the `dirpath` command.

NOTE: If you do not specify a directory path to remove, CXdb removes all CDI directory paths created with the `dirpath` command during the current debugging session.

Examples

The following examples remove CDI directory paths.

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

The above command removes all alternate directory paths substituted for /usr/jones. These alternate directory paths were previously specified with the `dirpath` command.

```
(CXdb) remove dirpath
```

The above command removes all alternate directory paths created with the `dirpath` command during the current debugging session.

Related Commands

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

Related Concepts

Compiler-Debugger Interface
search path

remove environment

rem en

env-

Delete environment variables from the process environment.

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process object.
<environment-variable>	An environment variable to remove.
[, ...]	A list of additional environment variables to remove. Multiple environment variables are separated by commas.

Description

The `remove environment` command removes the specified environment variables from the environment of the process object.

If the process object does not yet have its own environment, the `remove environment` command creates an environment for the process object. The new environment consists of the default environment minus the environment variables specified in the command.

Each new process will receive the modified environment. Existing processes are not affected.

Examples

The following examples remove environment variables from the environment of the current process object. These examples assume that the process object does not yet have an environment.

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

The above command creates an environment for the process object and then removes the environment variable `EDITOR` from the newly created environment. The `remove environment` command indicates to CXdb that you want to modify the environment for this process object. CXdb creates an environment for this process object consisting of the default environment with the `EDITOR` variable removed.

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.
<eventtype-specifier>	A list of eventpoint types whose eventpoints are to be removed. The asterisk (`*`) is used to specify all eventpoint types.
`[, ...]`	An optional list of additional eventpoint types. Multiple eventpoint types are separated by commas.

Description

The `remove eventtype` command removes all eventpoints of the specified eventpoint type.

The following is a list of eventpoint types:

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

Removed eventpoints cannot be restored and can no longer be referenced in CXdb commands. The eventpoint numbers assigned to removed eventpoints are never reused during a CXdb session.

If you want to prevent the eventpoints of an eventpoint type from being reached but do not want to completely remove them from their process objects, you can disable them with the `disable eventtype` command or give them each an ignore count with the `set ignore` command.

Examples

The following examples illustrate how to remove the eventpoints of eventpoint types.

```
(CXdb) remove eventtype trace
Event 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`
`enable event`
`info event`
`remove event`
`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

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

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

Examples

The following examples remove directories from the search path.

```
(CXdb) remove path /mnt/jones/project/libraries
```

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

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

The preceding example removes the listed directories from the search path of the current process. When CXdb searches for a source file, it no longer searches these two directories.

Related Commands

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

Related Concepts

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

Related Parameters

directory-specifier
process-list

remove variable

rem v

Delete a debugger variable.

Syntax

remove variable *<debugger-variable>*

Parameter	Meaning
<debugger-variable>	The debugger variable to be removed.

Description

The `remove variable` command removes the specified debugger variable.

Once a debugger variable has been removed, it may not be referenced in a CXdb command. You can, however, create a new debugger variable with the same name.

Examples

The following example illustrates how to remove debugger variables.

```
(CXdb) remove variable 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

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

Syntax

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

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

Description

The `rerun` command creates a process from the executable image of the process object and then begins 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

resume

res

Continue execution of the process from within an eventpoint handler.

Syntax

```
resume
```

Description

The `resume` command resumes process execution from within an eventpoint handler. 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

ret

Return to the calling routine.

Syntax

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

Parameter	Meaning
<*process-list*>	A list of processes affected by this command. The default is the current process.
<*thread-list*>	A list of threads affected by this command. The default is all threads of the specified process.
<*language-expression*>	Any expression that evaluates to a valid return value.

Description

The `return` command returns the specified value to the calling routine. This forced return pops the top frame from the process stack. The program counter (PC) is set to the next instruction after the call that has been returned.

When you issue the `return` command, CXdb prompts you to specify whether or not you want the function to return immediately. If you respond with a yes (`y`), the process returns the specified value to the calling routine and resets the PC. If you respond with a no (`n`), then CXdb aborts the `return` command. If you press the **RETURN** key without responding yes or no, the default is yes (`y`).

Examples

The following examples illustrate how to force a return to a calling routine.

```
(CXdb) return 5
Make function return 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

Create and execute a new process.

Syntax

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

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

Description

The `run` command creates a new process from the executable image of the process object and then begins execution of that process.

If the executable image was created from an executable file on the local host, the process is run from the process working directory on the local host. The process working directory can be set with the `set directory` command. CXdb expands the arguments passed to the process through the `run` command. The expansion is done according to the rules of the shell specified as the process shell. This shell can be specified using the `set pshell` command.

If the executable image was created from an executable file on a remote host, the process is run on the remote host in the process working directory. If the process working directory has not been set, the remote working directory is used. If the remote working directory for the process object has not been set (using the `set remotewd` command), the default remote working directory is used. If the default remote working directory has not been set (using the `set default remotewd` command), the console working directory on the local host is used as the remote working directory.

The argument list contains the arguments to be passed to the process after being expanded by CXdb according to the rules of 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. Redirection operators for the shell must be preceded by a backslash (\) to prevent interpretation by CXdb.

If a process already exists, CXdb asks if you want to terminate the process and restart. If you answer yes, CXdb kills the current process and creates a new process with any specified arguments. If you answer no, CXdb leaves the current process as is, and the CXdb prompt returns.

After the process is created, execution begins. Process execution continues until the process terminates or is stopped.

Examples

The following examples begin execution of a process.

```
(CXdb) run \< data
Beginning execution of Process [#0]
```

The above command creates a new process from the executable file of the current process object. The arguments passed to the shell are `< data`, which cause standard input to come from the data file. The `<` is preceded by a backslash to prevent interpretation by CXdb.

Because the process is run from the process working directory, the process working directory is used as the base directory for relative path names. `< data` becomes the argument list for the current process object.

```
(CXdb) run &
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 argument list is omitted, no arguments are passed to the process.

The `&` on the command line runs the command in the background. This causes the CXdb command prompt to return, allowing you to enter other CXdb commands that do not require the process to be stopped.

```
(CXdb) run arg1 arg2 \< data > cmdout.log
Beginning execution of Process [#0]
```

The above command creates a new process. The process shell is passed `arg1 arg2 < data`. This string is interpreted by the process shell, and then the arguments `arg1` and `arg2` are passed to the process while standard input is redirected from the data file.

The arguments `> cmdout.log` redirect 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 preceded by a backslash.

Related Commands

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

Related Concepts

background execution
process object
process working directory
remote debugging
windows

Related Parameters

process-list

set autocreate

se a

Enable dynamic creation of source windows.

Syntax

```
set autocreate
```

Description

The `set autocreate` command enables dynamic creation of source windows. When this setting is enabled, CXdb can automatically create a new source window. Initially the autocreate option is enabled. It can be disabled using the `clear autocreate` command, or by using the `-ns` option when invoking CXdb with the `cxdb` command.

In the CXwindows interface, you can toggle the autocreate setting using the autocreate option of the CommandWindow menu in the command window.

The new source window is initially associated with all threads of the current process. You can set the threads associated with a source window by using the `set threads` command. In CXwindows you can set the threads for a window using the Thread Selection dialog box. You can open this dialog box by selecting the threads option under the SourceWindow menu in the source window.

If the `set autocreate` command is used in an initialization file, CXdb creates a source window when it is invoked with the name of an executable file.

Examples

The following example illustrates how to set the autocreate option.

```
(CXdb) set autocreate
```

The above command enables the autocreate option. If a thread is spawned, CXdb creates a new source window.

Related Commands

`clear autocreate`
`cxdb`
`display source`
`list`
`set threads`

Related Concepts

initialization files
windows

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

Related Concepts

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

Related Concepts

cmderr
cmdout
viewports
cmdlog
logging
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 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.

set default handler

Related Commands

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

Related Concepts

breakpoints
eventpoints
eventpoint handlers
tracepoints
watchpoints

Related Parameters

event-handler

set default memory

se de m

Set the default unit size for displaying memory.

Syntax

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

Parameter	Meaning
<memory-unit>	The type of memory unit displayed. The possible types are: `byte` `halfword` `word` `longword` `quadword`

Description

The `set default memory` command sets the CXdb default for the type of memory unit used to display the contents of memory. This memory unit becomes the default for any new process objects that have not explicitly had their default memory unit set with the `set memory` command. The memory unit affects the appearance of output from the `examine` command.

The types of memory units are:

- `byte` — 8 bits
- `halfword` — 16 bits
- `word` — 32 bits
- `longword` — 64 bits
- `quadword` — 128 bits

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.

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

Examples

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

```
(CXdb) set default 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.

```
(CXdb) set default path /mnt/jones/libraries , 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 .
```

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 path`
`remove default path`
`remove path`
`set path`

Related Concepts

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

Related Parameters

directory-specifier

set default pshell

se de ps

Set the default process shell.

Syntax

`set default pshell {sh | csh}`

Description

The `set default pshell` command sets the CXdb default process shell to either `sh` or `csh`. Initially, the default process shell is the `csh` if the shell in which CXdb is running is a `csh` or `tcsh`. Otherwise, the default process shell is initially `sh`.

The CXdb default process shell is used to set the process shell for all new process objects. Existing process objects are not affected. The process shell of a process object is the type of shell in which a new process begins execution. It is also the type of shell used to interpret the arguments passed with the `run` command.

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 remotewd

se de re

Set the default remote working directory.

Syntax

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

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

Description

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

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

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

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

You can clear the default remote working directory using the `clear default remotewd` command.

Examples

The following examples show how to set the default remote working directory.

```
(CXdb) set default remotewd /mnt/jones/project
```

The above command sets the default remote working directory to the /mnt/jones/project directory. The next process that is created on a remote machine will be run from this directory unless a directory is explicitly set using the `set remotewd` command.

```
(CXdb) set default remotewd $REMOTEPROJ
```

The above command sets the default remote working directory to the value of the `$REMOTEPROJ` environment variable. The environment variable is expanded before the default remote working directory is set.

Related Commands

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

Related Concepts

console working directory
process working directory
process object
remote debugging

Related Parameters

directory-specifier

set default step

se de s

Set the default stepping granularity.

Syntax

`set default step` *<granularity>*

Parameter	Meaning
<granularity>	The desired step size, which can be one of the following: `routine` `loop` `block` `statement` `expression`

Description

The `set default step` command sets the CXdb default granularity, or step size, for the stepping commands. This default granularity is used by new processes that have not explicitly had their default granularity set with the `set step` command. It is also used by existing processes that have had their default granularity reset with the `clear step` command.

Initially, the CXdb default granularity is statement. To display the current setting, use the `info cxdb` command.

Examples

The following example illustrates how to set the CXdb default step size (granularity).

```
(CXdb) set default step loop
```

The above command selects `loop` as the default granularity.

Related Commands

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

Related Concepts

source units

stepping

Related Parameters

granularity

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 the directory in which a new process (created by the `run` or `rerun` command) is run. If the process is created on a remote host, the process is run in the process working directory on the remote host.

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

Related Commands

`cd`
`pwd`
`info process`

Related Concepts

console working directory
process object
remote debugging
default search path
process working directory
search path

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
command files
logging

set environment

se en

env=

Clear and redefine the environment variables for the process environment.

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process object.
<environment-variable>	An environment variable to add after the environment has been cleared.
<string>	The value to be given to the environment variable.
[, ...]	An optional list of additional environment variable assignments. Multiple assignments must be separated by a comma (,).

Description

The `set environment` command clears the environment of the process object and then adds the specified environment variables to the environment.

If the process object does not yet have its own environment, the `set environment` command creates an environment for the process object consisting of the environment variables specified in the command.

Each new process receives the modified environment. An existing process will not be affected.

Examples

The following examples set the environment for the current process object. For these examples, assume that the process object does not yet have its own environment.

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

The above command sets the environment of the current process object to be the environment variable `EDITOR`. The `set environment` command indicates to CXdb that you want to modify the environment, so CXdb creates an environment for the current process object. The environment created is cleared, then the environment variable `EDITOR` is added to the empty environment.

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

```
(CXdb) set environment LESS = -MQ , LIBRARIES = "/usr/lib /mnt/jones/lib"
```

The above command clears the current environment and then adds the two environment variables `LESS` and `LIBRARIES`. In this case, the quotes are needed for the variable `LIBRARIES` because the string contains a white space character (a blank).

Related Commands

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

Related Concepts

default environment
process object
environment

Related Parameters

environment-variable
string
process-list

set evalopts fpmode

se ev f

Set the floating point mode for evaluating expressions.

Syntax

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

Parameter	Meaning
ieee	IEEE floating point mode.
native	Native floating point mode.
dual	Dual mode. With this setting, CXdb evaluates language expressions by using the same floating point mode (IEEE or native) as the current process.

Description

The `set evalopts fpmode` command sets the floating point mode used by CXdb to evaluate language expressions. The available modes are:

- `ieee` — CXdb uses IEEE floating point mode to evaluate language expressions, regardless of the mode used by the current process.
- `native` — CXdb uses native floating point mode to evaluate language expressions, regardless of the mode used by the current process.
- `dual` — CXdb uses the same floating point mode (either IEEE or native) as the current process.

Dual is the default floating point mode for evaluating language expressions.

To display the current settings of the evalopts, use the `info cxdb` command.

Examples

The following examples illustrate how to select the floating point mode for language expression evaluations done by CXdb.

```
(CXdb) set evalopts fpmode native
```

The above command sets the CXdb floating point mode to native.

```
(CXdb) set evalopts fpmode ieee
```

The above command sets the CXdb floating point mode to IEEE.

```
(CXdb) set evalopts fpmode dual
```

The above command sets the CXdb floating point mode to dual. This means that CXdb will evaluate language expressions in the mode used by the current process.

Related Commands

`evaluate`
`info cxdb`
`print`
`set default fpmode`
`set evalopts iprecision`
`set evalopts rprecision`
`set fpmode`

Related Concepts

C language expressions
FORTRAN language expressions
language expressions

Related Parameters

language-expression

set evalopts iprecision

se ev i

Set the size of integer constants for CXdb.

Syntax

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

Parameter	Meaning
4	4-byte integers.
8	8-byte integers.

Description

The `set evalopts iprecision` command sets the size for integer constants used by CXdb in evaluating language expressions. The available sizes are:

- 4-byte integers
- 8-byte integers

The default is 4-byte integers.

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

Examples

The following examples illustrate how to set the size for integer constants used by CXdb in evaluating language expressions.

```
(CXdb) set evalopts iprecision 8
```

The above command selects an integer size of `8` bytes.

```
(CXdb) set evalopts iprecision 4
```

The above command selects an integer size of `4` bytes.

Related Commands

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

Related Concepts

C language expressions
FORTRAN language expressions
language expressions

Related Parameters

language-expression

set evalopts rprecision

se ev r

Set the size of real numbers for CXdb.

Syntax

set evalopts rprecision {4 | 8}

Parameter	Meaning
4	Single precision, or 4-byte real numbers.
8	Double precision, or 8-byte real numbers.

Description

The `set evalopts rprecision` command sets the level of precision for real number (floating point) constants used by CXdb in evaluating language expressions. The available precision levels are:

- Single precision (4-byte real)
- Double precision (8-byte real)

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
```

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`

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

Related Concepts

breakpoints
eventpoint handlers
watchpoints
eventpoints
tracepoints

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

Related Concepts

cmdlog
viewports
logging

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

se n

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.

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

Examples

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

```
(CXdb) set 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.

```
(CXdb) set path /mnt/jones/libraries , 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 .
```

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

Related Commands

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

Related Concepts

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

Related Parameters

directory-specifier
process-list

set printopts maxarray

se 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.

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 remotewd

se r

Set the remote working directory.

Syntax

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

Parameter	Meaning
<*process-list*>	A list of process objects affected by this command. The default is the current process object.
<*directory-specifier*>	The directory to become the remote working directory.

Description

The `set remotewd` command sets the remote working directory.

The remote working directory acts as the console working directory for the remote host when doing remote debugging. The remote working directory is used as:

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

If the remote working directory is not set by the `set remotewd` command, CXdb uses the directory specified by the `set default remotewd` command. If neither of these is set, CXdb uses the console working directory as the remote working directory. For more information on the remote working directory, refer to the concepts page on remote debugging in the *CXdb Reference: Concepts and Messages*.

Examples

The following examples show how to set the remote working directory.

```
(CXdb) set remotewd /mnt/jones/project
```

The above command sets the remote working directory to the /mnt/jones/project directory. The next process that is created on a remote host will be run from this directory.

```
(CXdb) set remotewd $REMOTEPROJ
```

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

Related Commands

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

Related Concepts

console working directory
process working directory
process object
remote debugging

Related Parameters

directory-specifier
process-list

set seq

se se

Set the sequential mode (SEQ) bit.

Syntax

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

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

Description

The `set seq` command sets the sequential mode (SEQ) bit of the processor status word (PSW).

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

For more information about the PSW and the SEQ bit, refer to the *CONVEX Architecture Reference Manual (C Series).*

Examples

The following example illustrates how to set the SEQ bit.

```
(CXdb) set seq
```

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

Related Commands

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

Related Parameters

process-list thread-list

set shell

se sh

Set the type of shell invoked from within CXdb.

Syntax

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

se si

Set the actions for the specified signal.

Syntax

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

Parameter	Meaning
<process-list>	A list of process objects affected by this command. The default is the current process object.
<signal-specifier>	The signal whose actions you want to set.

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.

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

Related Concepts

signals

Related Parameters

process-list
signal-specifier

set sqs

se sq

Set the sequential store enable (SQS) bit.

Syntax

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

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

Description

The `set sqs` command sets the sequential store enable (SQS) bit of the processor status word (PSW).

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

For more information about the PSW and the SQS bit, refer to the *CONVEX Architecture Reference Manual (C Series).*

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 threads

se th

Associate windows with particular threads.

Syntax

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

Parameter	Meaning
<window>	The object number of a CXdb window. The window must be able to be made thread-specific.
[, ...]	A list of additional windows. Multiple windows are separated by commas.
<thread-number>	The number of the thread to associate the with the window.
[, ...]	A list of additional threads. Multiple threads in the list must be separated by commas.

Description

The `set threads` command sets the threads associated with another window. The `set threads` command can be used to set threads for the following windows:

- Source window
- Disassembly window (CXwindows only)
- Examine window (CXwindows only)
- Stack window (CXwindows only)
- Process status word window (CXwindows only)
- Scalar registers window (CXwindows only)
- Vector registers window (CXwindows only)
- Communication registers window (CXwindows only)

In the CXwindows interface, you can also set the threads for a window by using the Thread Selection dialog box. You can open this dialog box for a specific window by selecting the threads option from the first menu of each window.

Examples

The following examples illustrate how to set threads for various windows.

```
(CXdb) set threads 2 0
```

The above command associates window 2 with thread 0 of the current process.

```
(CXdb) set threads 3,4 0,1
```

The above command associates windows 3 and 4 with threads 0 and 1.

Related Commands

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

Related Concepts

windows

set typehandler

se t

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

Syntax

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

Parameter	Meaning

<eventtype-specifier>	An eventtype to associate with the specified eventpoint handler. Possible eventtypes are:
	`break`
	`trace`
	`watch`
	`exec`
	`join`
	`modify`
	`reached`
	`relation`
	`signal`
	`spawn`
	`*` (all)

[, ...]	An optional list of additional eventtypes. Multiple eventtypes are separated by commas.
<event-handler>	The eventpoint handler to assign to the specified eventtypes.

Description

The `set typehandler` command defines the default handler for the specified eventtypes.

If an eventpoint does not have its own handler, then it uses the default handler for its eventtype. If the eventtype does not have its own handler, then the default handler for general eventpoints is used.

When a default handler for a given type is changed, the new setting can be displayed using the `info eventtype` command. The handler can be removed using the `clear typehandler` command.

Examples

The following examples illustrate how to define a default handler for various eventtypes.

```
(CXdb) set typehandler trace {echo "Reached tracepoint: "; print $self;
      resume; }
```

The above command sets the default handler for tracepoints. The default handler echoes a message, displays the current eventpoint number stored in the debugger variable `$self,` and resumes process execution. All tracepoints that do not have a specified handler will use the new default tracepoint handler when they are next triggered.

```
(CXdb) set typehandler break, reached {echo "Reached eventpoint: "; print
      $self; }
```

The above command sets the default handler for breakpoints and event reached eventpoints. The handler echoes a message and displays the current eventpoint number stored in the debugger variable `$self`. Process execution does not resume.

```
(CXdb) set typehandler * {echo "Using default handler"; print $self;}
```

The above command sets the default handler for all eventtypes. The new handler echoes a message and displays the current value of the `$self` debugger variable. Process execution does not resume. This command sets all eventpoints, no matter what the type, to perform the same function, unless the eventpoint has its own eventpoint handler.

Related Commands

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

```
(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
info signal
signal process
event signal
set signal

Related Concepts

background execution
signals

Related Parameters

process-list
thread-list
signal-specifier

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`
`clear echo`
`info cxdb`
`set echo`
`set logging`

Related Concepts

command files
cmderr
cmdlog
cmdout
initialization files
logging
viewports

Related Parameters

file-name

step

ste

s

Step to the next source unit.

Syntax

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

Parameter	Meaning
<process-list>	A list of processes affected by this command. The default is the current process.
<thread-list>	A list of threads affected by this command. The default is all threads of the specified process.
<granularity>	The type of source unit, or step size. Available granularities are: `routine` `block` `loop` `statement` `expression` If you do not specify a granularity, CXdb uses the default granularity of the specified process.
<count>	The number of times to repeat this command. The default is 1.
&	Runs the command in the background.

Description

The `step` command continues execution of your process until it reaches the next source unit of the specified granularity. If the current routine does not contain another source unit of the specified granularity, then the process continues executing until it reaches the end of the current routine.

Examples

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

```
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

ste i

si, stepi

Step to the next instruction.

Syntax

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

Parameter	Meaning
<*process-list*>	A list of processes affected by this command. The default is the current process.
<*thread-list*>	A list of threads affected by this command. The default is all threads of the specified process.
<*count*>	The number of times to repeat this command. The default is 1.
&	Runs the command in the background.

Description

The `step instruction` command steps the process by the specified number of machine instructions.

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

```
(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
next
next over
step over
finish
next instruction
step

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 statement
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

t i

ti

Set a tracepoint at an instruction.

Syntax

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

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

Description

The `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 must be sure that the address is actually the starting address for the instruction. If the tracepoint is placed at an address in the middle of an instruction, the tracepoint will be interpreted as a portion of the instruction, which can cause unpredictable results.

```
(CXdb) trace instruction 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
eventpoints
tracepoints
debugger variables
eventpoint handlers
watchpoints

Related Parameters

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

trace line

t l

tl

Set a tracepoint at a source line.

Syntax

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

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

Description

The `trace line` command sets a tracepoint before the first statement on the specified line.

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

When the tracepoint is triggered, process execution stops and the commands of the tracepoint's handler are executed. If the tracepoint does not have its own handler, the default handler for tracepoints, which displays a message and resumes process execution, is executed.

Examples

The following examples set tracepoints at specific source lines.

```
(CXdb) trace line 18

#0: trace line, on [#0/*], Enabled, ignore 0/0
          [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 semicolon (`;`).
<debugger-variable>	The debugger variable assigned to this eventpoint.

Description

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

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

When the tracepoint is triggered, process execution stops and the commands of the tracepoint's handler are executed. If the tracepoint does not have its own handler, the default handler for tracepoints, which displays a message and then resumes process execution, is executed.

Examples

The following examples set tracepoints at the first executable source 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 semicolon (`;`).
<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 28
```

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 semicolon (`;`).
<*debugger-variable*>	The debugger variable assigned to this eventpoint.

Description

The `watch` command sets a watchpoint to watch for a change to occur at the specified address range. A process image must exist for a watchpoint to be created.

After the execution of each 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 `80001234`. The notation `0x80001234` is C-specific and indicates the address is in hexadecimal notation. When the value stored in this range changes, the watchpoint is triggered.

```
(CXdb) watch '80001234'x..'80001237'x

#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

Parameters

2

This chapter contains reference pages that explain the parameters used with CXdb commands. There is a separate reference page for each parameter. The reference pages 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 book.
- **Related Concepts** — A list of major concepts related to the parameter. Related concepts are in *CONVEX CXdb Reference: Concepts and Messages*.
- **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
```

Related Concepts C language expressions | FORTRAN language expressions | language expressions

Related Parameters language-expression

<debugger-variable>

A variable used in CXdb commands.

Syntax

[cxdb$ | $]*<variable-name>*

Parameter	Meaning
cxdb$	One of the symbols to delimit the debugger variable name and distinguish it from the name of a process variable.
$	One of the symbols to delimit the debugger variable name and distinguish it from the name of a process variable.
<variable-name>	An alphanumeric string that is the name of the debugger variable.

Description

A *<debugger-variable>* is a variable that you can define in CXdb. A debugger variable can store the following types of data:

- A CXdb object number
- The result of a language expression
- A signal number
- The contents of a register

The data type of a debugger variable is the same as the data type of the value assigned to it. Once a value has been assigned to a debugger variable, you can use the variable anywhere a value of that type would be valid. The data type of a debugger variable is not fixed. If you assign a new value to an existing debugger variable, that variable takes on the data type of the new value.

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
debugger variables
eventpoint handlers
initialization files

Related Parameters

event-handler

<directory-specifier>

A directory path name.

Syntax

<directory-specifier>

Description

A *<directory-specifier>* is a relative or absolute directory path name.

In addition to the names of directories and the slash (/), a path name can begin with the following special items:

- `$`*variable* — An environment variable 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 .

Parameters

<environment-variable>

Related Commands

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

Related Concepts

default environment

environment

Related Parameters

string

<event-handler>

A handler for an eventpoint.

Syntax

{ *<command>* ; [...] }

Parameter	Meaning
<command>	A CXdb command. The `resume` command must be used as the last command in a handler when process execution is to be continued.
[...]	An optional list of additional CXdb commands. Each command must be terminated with a semicolon (`;`).

Description

An *<event-handler>* is a collection of CXdb commands 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 semicolon (`;`).

The one exception to the above rule is in the use of the `if` command. The `if` command itself does not need to be terminated with a semicolon. Each command in the body of the `if` command must terminate with a semicolon. Multiple commands in the body can be enclosed in curly-braces.

Examples

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

```
(CXdb) break 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 `echo` command is executed and the eventpoint handler is done. If the condition evaluates to `FALSE`, the `print` command is executed, and process execution resumes.

Related Commands

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

Related Concepts

breakpoints
C language expressions
eventpoints
eventpoint handlers
FORTRAN language expressions
language expressions
tracepoints
watchpoints

<event-specifier>

An eventpoint identifier.

Syntax

{*<eventpoint-number>* | *<debugger-variable>* | * }

Parameter	Meaning
<eventpoint-number>	The eventpoint number assigned by CXdb to the eventpoint when it is created.
<debugger-variable>	A debugger variable that you assigned to the eventpoint.
*	All eventpoints in the current process.

Description

An *<event-specifier>* identifies an eventpoint to be affected by a CXdb command. The eventpoint may be specified by its eventpoint number or by a debugger variable if a debugger variable has been assigned to the eventpoint. To specify all eventpoints, use the asterisk (*).

In a list, multiple event specifiers are separated by commas.

Examples

The following examples use an event specifier with the `info event` command.

```
(CXdb) info event 0,2

#0: break routine, on [#0/*], Enabled, ignore 0/0
         [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          disable eventtype
info event             info eventtype
remove event           remove eventtype
set handler            set typehandler
set ignore
```

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.

Parameters

<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
```

Related Concepts	breakpoints	eventpoints
	eventpoint handlers	tracepoints
	watchpoints	
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 (or `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 (or `4 + 1`).

Related Commands

backtrace
frame
info locals
info stack
display stack
info frame
info process

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 (`{ }`).

Parameters

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.

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`
`event reached line`
`goto line`
`info line`
`trace line`

Related Concepts

source units

Related Parameters

file-name

<process-list>

A list of processes.

Syntax

`:p` {*<process-number>* [, ...] | *}

Parameter	Meaning
<process-number>	The number of a CXdb process object. The number can be expressed as an integer or as a debugger variable that contains the value of the process object number.
[, ...]	Additional process numbers in the list. Commas must separate the entries in the list. Spaces between the entries are optional.
*	The wildcard symbol that means all processes.

Description

A *<process-list>* is a list of processes that are affected by a command.

NOTE: The current version of CXdb maintains only one process object (process object 0) at any given time. Because of this, you may omit the process list from commands in this release of CXdb.

If you are debugging only one process at a time, you can omit the process list from the command. If you are debugging multiple processes, 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.

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 preceding 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 indicates that no signal should be sent to the process.

Examples

The following examples use signals with the `signal process` command.

```
(CXdb) signal process SIGINT
```

The above command sends the `SIGINT` signal to the current process.

```
(CXdb) signal process int
```

The above command sends the `SIGINT` signal to the current process. The signal name can be abbreviated by dropping the `SIG` prefix. Signal names are not case sensitive.

```
(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`
`goto source`
`info sourceunit`
`event reached source`
`info line`
`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

Parameters

<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.

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`
`print`
`info expression`

Related Concepts

debugger variables
synthesized variables
language expressions

Related Parameters language-expression

<thread-list>

A list of process threads.

Syntax

`:t` {*<thread-number>* `[,` ...] | `*`}

Parameter	Meaning
<thread-number>	A thread number. The number must be expressed as an integer.
`[,` ...]	Additional thread numbers in the list. Commas must separate the entries in the list. Spaces between the entries are optional.
*	The wildcard symbol 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

Master index

This is a master index for both volumes of the CXdb reference manual. Pages are numbered sequentially across both volumes and are distributed as follows:

- Pages 1 to 596 — *CONVEX CXdb Reference: Commands and Parameters*
- Pages 597 to 850 — *CONVEX CXdb Reference: Concepts and Messages*

Symbols

& (background execution) 599
\; (*language-expression* terminator) 555

A

accessing memory
  commands
    copy 79
    examine 181
    fill 187
    find memory backward 191
    find memory forward 195
    info formatting 265
    print 345
    set format 439
    set memory 449
  concepts
    language expressions 673
    synthesized variables 731
  see also
    disassembly window
    examine window
    registers
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
aliases
  commands
    alias 17
    info alias 229
    remove alias 361
  concepts
    initialization files 669
  parameters
    regular-expression 567
    string 575
  see also
    csd debugger
    gdb debugger
    macros
altering execution order
  commands
    goto address 217
    goto line 219
    goto source 221
    return 391
  see also
    executing a process
arrays
  commands
    examine 181
    info expression 261
    print 345
    set printopts maxarray 455
  concepts
    language expressions 673
  parameters
    array-slice 525
  see also
    examine window
    memory
array-slice 525

Master index

`attach` 21
attaching to a process
  commands
    `attach` 21
    `continue` 77
    `debug proc` 101
    `detach` 105
    `executable` 185
    `info process` 287
    `kill process` 321
  see also
    executing a process
    loading object code

B

background execution 599
  commands
    `continue` 77
    `finish` 203
    `next` 335
    `next instruction` 339
    `next over` 341
    `rerun` 387
    `run` 393
    `signal process` 483
    `signal thread` 485
    `step` 489
    `step instruction` 493
    `step over` 495
  concepts
    background execution 599
`backtrace` 23
`bind` 25
blocks. *see* source units
`break instruction` 29
`break line` 33
`break routine` 37
`break source` 41
breakpoints 601
  commands
    `break instruction` 29
    `break line` 33
    `break routine` 37
    `break source` 41
    `clear handler` 63
    `disable event` 109
    `disable eventtype` 111
    `enable event` 133
    `enable eventtype` 135
    `info break` 235
    `info event` 253
    `remove event` 377
    `remove eventtype` 379
    `set handler` 443
    `set ignore` 445
  concepts
    breakpoints 601
    eventpoint handlers 647
    eventpoints 651
  parameters
    event-handler 535
    language-expression 555
    line-specifier 557

C

C language expressions 609
`cd` 45
CDI. *see* Compiler-Debugger Interface
`clear autocreate` 47
`clear default environment` 49
`clear default fixed sched` 51
`clear default handler` 53
`clear default remotewd` 55
`clear echo` 57
`clear environment` 59
`clear fixed sched` 61
`clear handler` 63
`clear logging` 65
`clear noclobber` 67
`clear seq` 69
`clear sqs` 71
`clear step` 73
`clear typehandler` 75
cmderr 617
cmdlog 619
cmdout 621
command files 623
  commands
    `clear echo` 57
    `if` 225
    `set echo` 427
    `source` 487
  concepts
    command files 623
    initialization files 669
command window
  commands
    `cxdb` 87
    `info cxdb` 239
    `info history` 273
    `quit` 357
    `recall` 359
  concepts
    viewports 743
    windows 755
    Xdefaults 759
  parameters
    redirection-operator 563
    viewport 583
Compiler-Debugger Interface 625

- compiling source code
  - concepts
    - Compiler-Debugger Interface 625
    - source units 717
  - *see also*
    - `debug exec`
- console working directory 629
  - commands
    - `cd` 45
    - `pwd` 355
  - concepts
    - console working directory 629
    - process working directory 693
  - parameters
    - *directory-specifier* 531
- `continue` 77
- `copy` 79
- `core` 81
- `csd` 83
- csd debugger 631
  - commands
    - `csd` 83
    - `cxdb` 87
  - concepts
    - csd debugger 631
  - *see also*
    - gdb debugger
- `cxdb` 87

D

- data files
  - commands
    - `dirpath` 107
    - `info dirpath` 245
    - `remove dirpath` 373
  - concepts
    - Compiler-Debugger Interface 625
    - search path 709
- data. *see*
  - language expressions
  - memory
- `debug core` 95
- `debug exec` 97
- `debug proc` 101
- debugger variables 635
- *debugger-variable* 529
- default environment 639
- default search path 641
- `detach` 105
- detaching from a process. *see* attaching to a process
- directories
  - commands
    - `cd` 45
    - `dirpath` 107
    - `pwd` 355
    - `set directory` 425
  - concepts
    - console working directory 629
    - process working directory 693
  - parameters
    - *directory-specifier* 531
  - *see also*
    - search path
- *directory-specifier* 531
- `dirpath` 107
- `disable event` 109
- `disable eventtype` 111
- `disassemble` 113
- disassembled code
  - commands
    - `disassemble` 113
    - `display disassembly` 117
  - *see also*
    - disassembly window
- disassembly window
  - commands
    - `disassemble` 113
    - `display disassembly` 117
  - concepts
    - windows 755
    - Xdefaults 759
  - *see also*
    - examine window
- `display disassembly` 117
- `display examine` 119
- `display file` 121
- display file window
  - commands
    - `display file` 121
  - concepts
    - windows 755
    - Xdefaults 759
- `display routine` 123
- `display source` 125
- `display stack` 127
- displaying information. *see*
  - `display` commands
  - `examine`
  - `info` commands
  - printing data
  - windows

E

`echo` 129
`edit` 131
`enable event` 133
`enable eventtype` 135
environment 645
  commands
    `add default environment` 9
    `add environment` 13
    `clear default environment` 49
    `clear environment` 59
    `info default environment` 243
    `info environment` 249
    `remove default environment` 369
    `remove environment` 375
    `set default environment` 405
    `set environment` 429
  concepts
    default environment 639
    environment 645
  parameters
    environment-variable 533
environment-variable 533
`evaluate` 139
evaluating expressions
  commands
    `evaluate` 139
    `print` 345
    `set evalopts fpmode` 431
    `set evalopts iprecision` 433
    `set evalopts rprecision` 435
  concepts
    language expressions 673
  parameters
    language-expression 555
`event exec` 141
`event join` 143
`event modify` 147
`event reached instruction` 153
`event reached line` 157
`event reached routine` 161
`event reached source` 165
`event relation` 169
`event signal` 173
`event spawn` 177
event-handler 535
eventpoint handlers 647
  commands
    `clear handler` 63
    `clear typehandler` 75
    `echo` 129
    `evaluate` 139
    `if` 225
    `info event` 253
    `resume` 389
    `set default handler` 413
    `set handler` 443
    `set typehandler` 479
  concepts
    eventpoint handlers 647
    eventpoints 651
  parameters
    event-handler 535
eventpoints 651
  commands
    `clear default handler` 53
    `clear handler` 63
    `clear typehandler` 75
    `disable event` 109
    `disable eventtype` 111
    `enable event` 133
    `enable eventtype` 135
    `event exec` 141
    `event join` 143
    `event modify` 147
    `event reached instruction` 153
    `event reached line` 157
    `event reached routine` 161
    `event reached source` 165
    `event relation` 169
    `event signal` 173
    `event spawn` 177
    `info event` 253
    `info eventtype` 257
    `remove event` 377
    `remove eventtype` 379
    `set default handler` 413
    `set handler` 443
    `set ignore` 445
    `set typehandler` 479
  concepts
    debugger variables 635
    eventpoint handlers 647
    eventpoints 651
  parameters
    event-handler 535
    event-specifier 537
    eventtype-specifier 539
    language-expression 555
  see also
    breakpoints
    tracepoints
    watchpoints
event-specifier 537
eventtype-specifier 539

`examine` 181
examine window
  commands
    `display examine` 119
    `examine` 181
  concepts
    windows 755
    Xdefaults 759
  see also
    disassembly window
`executable` 185
executing a process
  commands
    `continue` 77
    `executable` 185
    `kill process` 321
    `rerun` 387
    `resume` 389
    `run` 393
    `stop` 499
  concepts
    process object 687
    remote debugging 695
  see also
    altering execution order
    background execution
    stepping
expressions. *see*
  language expressions
  source units

F

file-name 543
`fill` 187
`find memory backward` 191
`find memory forward` 195
`find window backward` 199
`find window forward` 201
`finish` 203
FORTRAN language expressions 657
`frame` 207
frames. *see* stack frames
frame-specifier 545
function-name 547
functions. *see*
  language expressions
  routines

G

`gdb` 209
gdb debugger 665
  commands
    `cxdb` 87
    `gdb` 209
  concepts
    gdb debugger 665
  see also
    csd debugger
`get` 213
`goto address` 217
`goto line` 219
`goto source` 221
granularity 549
  see also source units

H

handlers. *see* eventpoint handlers
`help` 223
help window
  commands
    `help` 223
  concepts
    Maryland Windows 685
    windows 755

I

`if` 225
incremental execution. *see* stepping
`info alias` 229
`info args` 231
`info bind` 233
`info break` 235
`info cregisters` 237
`info cxdb` 239
`info default environment` 243
`info dirpath` 245
`info dynamicobject` 247
`info environment` 249
`info errno` 251
`info event` 253
`info eventtype` 257
`info expression` 261
`info formatting` 265
`info frame` 267
`info frame at` 271
`info history` 273
`info line` 275
`info locals` 279
`info macro` 281
`info objectmap` 283
`info path` 285
`info process` 287
`info psw` 291
`info registers` 293
`info scope` 295
`info signal` 297
`info sourceunit` 301
`info stack` 303

`info symbols` 305
`info threads` 307
`info trace` 311
`info type` 313
`info vregisters` 317
`info watch` 319
initialization files 669
  see also command files
invoking CXdb
  commands
    `cxdb` 87
    `info cxdb` 239
  concepts
    initialization files 669
    Xdefaults 759

K

key bindings. *see* Maryland Windows
key-name 553
`kill process` 321

L

language expressions 673
  commands
    `evaluate` 139
    `info expression` 261
    `print` 345
  concepts
    C language expressions 609
    FORTRAN language expressions 657
    language expressions 673
  parameters
    language-expression 555
language-expression 555
line-specifier 557
`list` 323
`load object` 327
loading object code
  commands
    `info dynamicobject` 247
    `info objectmap` 283
    `load object` 327
  concepts
    Compiler-Debugger Interface 625
    process object 687
  see also
    attaching to a process
    executing a process
logging 679
  commands
    `add cmderr` 3
    `add cmdlog` 5
    `add cmdout` 7
    `clear logging` 65
    `clear noclobber` 67
    `remove cmderr` 363
    `remove cmdlog` 365
    `remove cmdout` 367
    `set cmderr` 399
    `set cmdlog` 401
    `set cmdout` 403
    `set logging` 447
    `set noclobber` 451
  concepts
    cmderr 617
    cmdlog 619
    cmdout 621
    logging 679
    viewports 743
  parameters
    viewport 583
loops. *see* source units

M

`macro` 329
macros
  commands
    `info macro` 281
    `macro` 329
    `remove macro` 381
  concepts
    initialization files 669
  parameters
    regular-expression 567
    string 575
  see also
    aliases
Maryland Windows 685
  commands
    `bind` 25
    `info bind` 233
  concepts
    Maryland Windows 685
    windows 755
  parameters
    function-name 547
    key-name 553
memory. *see* accessing memory

N

`next` 335
`next instruction` 339
`next over` 341
nexting. *see* stepping

O

object code. *see* loading object code
optimized code
  commands
    disassemble 113
    examine 181
    info expression 261
    info line 275
    next instruction 339
    step instruction 493
  concepts
    source units 717
    synthesized variables 731
  parameters
    granularity 549
    synthesized-variable 577

P

path. *see*
  dirpath
  search path
print 345
printing data
  commands
    examine 181
    info expression 261
    print 345
    set printopts maxarray 455
    set printopts nopadding 457
    set printopts padding 459
    set printopts precision 461
  concepts
    language expressions 673
  see also
    arrays
    memory
process execution. *see* executing a process
process interface window
  commands
    kill process 321
    rerun 387
    run 393
    set pshell 463
    stop 499
  concepts
    windows 755
process object 687
  commands
    debug core 95
    debug exec 97
    executable 185
    info process 287
    kill process 321
  concepts
    Compiler-Debugger Interface 625
    process object 687
  parameters
    process-list 559
process settings
  commands
    add environment 13
    add path 15
    clear environment 59
    clear seq 69
    clear sqs 71
    clear step 73
    fixed sched 437
    info environment 249
    info path 285
    info process 287
    info signal 297
    remove environment 375
    remove path 383
    set directory 425
    set environment 429
    set format 439
    set fpmode 441
    set memory 449
    set pshell 463
    set seq 467
    set signal 471
    set sqs 473
    set step 475
  concepts
    environment 645
    process object 687
    process working directory 693
    search path 709
  parameters
    environment-variable 533
process working directory 693
  commands
    cd 45
    set directory 425
  concepts
    console working directory 629
    process object 687
    process working directory 693
    search path 709
  parameters
    directory-specifier 531
process-list 559
processor status word. *see* info psw
psw. *see* info psw
put 351
pwd 355

Master index

Q

`quit` 357

R

`recall` 359

recording a session. *see* logging

redirection-operator 563

register windows. *see* disassembly window

registers

  commands

    `evaluate` 139

    `info cregisters` 237

    `info psw` 291

    `info registers` 293

    `info vregisters` 317

    `print` 345

  see also

    debugger variables

    disassembly window

    synthesized variables

regular-expression 567

remote debugging 695

  commands

    `clear default remotewd` 55

    `set default remotewd` 421

    `set remotewd` 465

  concepts

    process object 687

    remote debugging 695

`remove alias` 361

`remove cmderr` 363

`remove cmdlog` 365

`remove cmdout` 367

`remove default environment` 369

`remove default path` 371

`remove dirpath` 373

`remove environment` 375

`remove event` 377

`remove eventtype` 379

`remove macro` 381

`remove path` 383

`remove variable` 385

`rerun` 387

resource settings. *see* Xdefaults

`resume` 389

`return` 391

routines

  commands

    `break routine` 37

    `display routine` 123

    `evaluate` 139

    `event reached routine` 161

    `info args` 231

    `info frame` 267

    `info frame at` 271

    `print` 345

    `trace routine` 509

  concepts

    language expressions 673

    source units 717

`run` 393

running a process. *see* executing a process

S

scope 701

  commands

    `frame` 207

    `info expression` 261

    `info scope` 295

    `print` 345

  concepts

    scope 701

    synthesized variables 731

  see also

    stack frames

search path 709

  commands

    `add default path` 11

    `add path` 15

    `dirpath` 107

    `info dirpath` 245

    `info path` 285

    `remove default path` 371

    `remove dirpath` 373

    `remove path` 383

    `set default path` 417

    `set path` 453

  concepts

    default search path 641

    search path 709

  see also

    `dirpath`

`set autocreate` 397

`set cmderr` 399

`set cmdlog` 401

`set cmdout` 403

`set default environment` 405

`set default fixed sched` 407

`set default format` 409

`set default fpmode` 411

`set default handler` 413

`set default memory` 415

`set default path` 417

`set default pshell` 419

`set default remotewd` 421

`set default step` 423

`set directory` 425

`set echo` 427

`set environment` 429

set evalopts fpmode 431
set evalopts iprecision 433
set evalopts rprecision 435
set fixed sched 437
set format 439
set fpmode 441
set handler 443
set ignore 445
set logging 447
set memory 449
set noclobber 451
set path 453
set printopts maxarray 455
set printopts nopadding 457
set printopts padding 459
set printopts precision 461
set pshell 463
set remotewd 465
set seq 467
set shell 469
set signal 471
set sqs 473
set step 475
set threads 477
set typehandler 479
shell 481
shell window
  commands
    set shell 469
    shell 481
  see also
    edit
    set pshell
signal process 483
signal thread 485
signals 713
  commands
    event signal 173
    info signal 297
    set signal 471
    signal process 483
    signal thread 485
  concepts
    debugger variables 635
    eventpoints 651
    signals 713
  parameters
    signal-specifier 571
signal-specifier 571
source 487
source code
  commands
    display file 121
    display routine 123
    display source 125
    list 323
  concepts
    Compiler-Debugger Interface 625
    source units 717
  see also
    compiling source code
    source window
source units 717
  commands
    clear step 73
    info line 275
    info sourceunit 301
    set default step 423
    set step 475
  concepts
    source units 717
  parameters
    granularity 549
    source-unit 573
source window
  commands
    clear autocreate 47
    debug exec 97
    debug proc 101
    display routine 123
    display source 125
    executable 185
    find window backward 199
    find window forward 201
    info line 275
    set autocreate 397
    set threads 477
  concepts
    source units 717
    windows 755
    Xdefaults 759
source-unit 573
stack frames
  commands
    backtrace 23
    frame 207
    info frame 267
    info frame at 271
    info stack 303
  concepts
    scope 701
  parameters
    frame-specifier 545
  see also
    altering execution order
stack window
  commands
    backtrace 23
    frame 207
    info frame 267
    info stack 303

concepts
windows 755
Xdefaults 759
statements. *see* source units
step 489
step instruction 493
step over 495
stepping 725
commands
clear step 73
finish 203
next 335
next instruction 339
next over 341
set default step 423
set step 475
step 489
step instruction 493
step over 495
concepts
background execution 599
source units 717
stepping 725
parameters
granularity 549
stop 499
string 575
synthesized variables 731
commands
info expression 261
print 345
concepts
language expression 673
synthesized variables 731
parameters
synthesized-variable 577
synthesized-variable 577

T

thread-list 581
threads
commands
clear default fixed sched 51
clear fixed sched 61
event exec 141
event join 143
event spawn 177
info threads 307
set default fixed sched 407
set fixed sched 437
set threads 477
signal thread 485
parameters
thread-list 581
trace instruction 501
trace line 505
trace routine 509
trace source 513
tracepoints 735
commands
clear handler 63
disable event 109
disable eventtype 111
enable event 133
enable eventtype 135
info event 253
info trace 311
remove event 377
remove eventtype 379
set handler 443
set ignore 445
trace instruction 501
trace line 505
trace routine 509
trace source 513
concepts
eventpoint handlers 647
eventpoints 651
tracepoints 735
parameters
event-handler 535
language-expression 555
line-specifier 557

V

variables. *see*
debugger variables
language expressions
memory
synthesized variables
viewport 583
viewports 743
commands
add cmderr 3
add cmdlog 5
add cmdout 7
clear logging 65
clear noclobber 67
remove cmderr 363
remove cmdlog 365
remove cmdout 367
set cmderr 399
set cmdlog 401
set cmdout 403
set logging 447
set noclobber 451
concepts
cmderr 617
cmdlog 619
cmdout 621

logging 679
viewports 743
parameters
redirection-operator 563
viewport 583

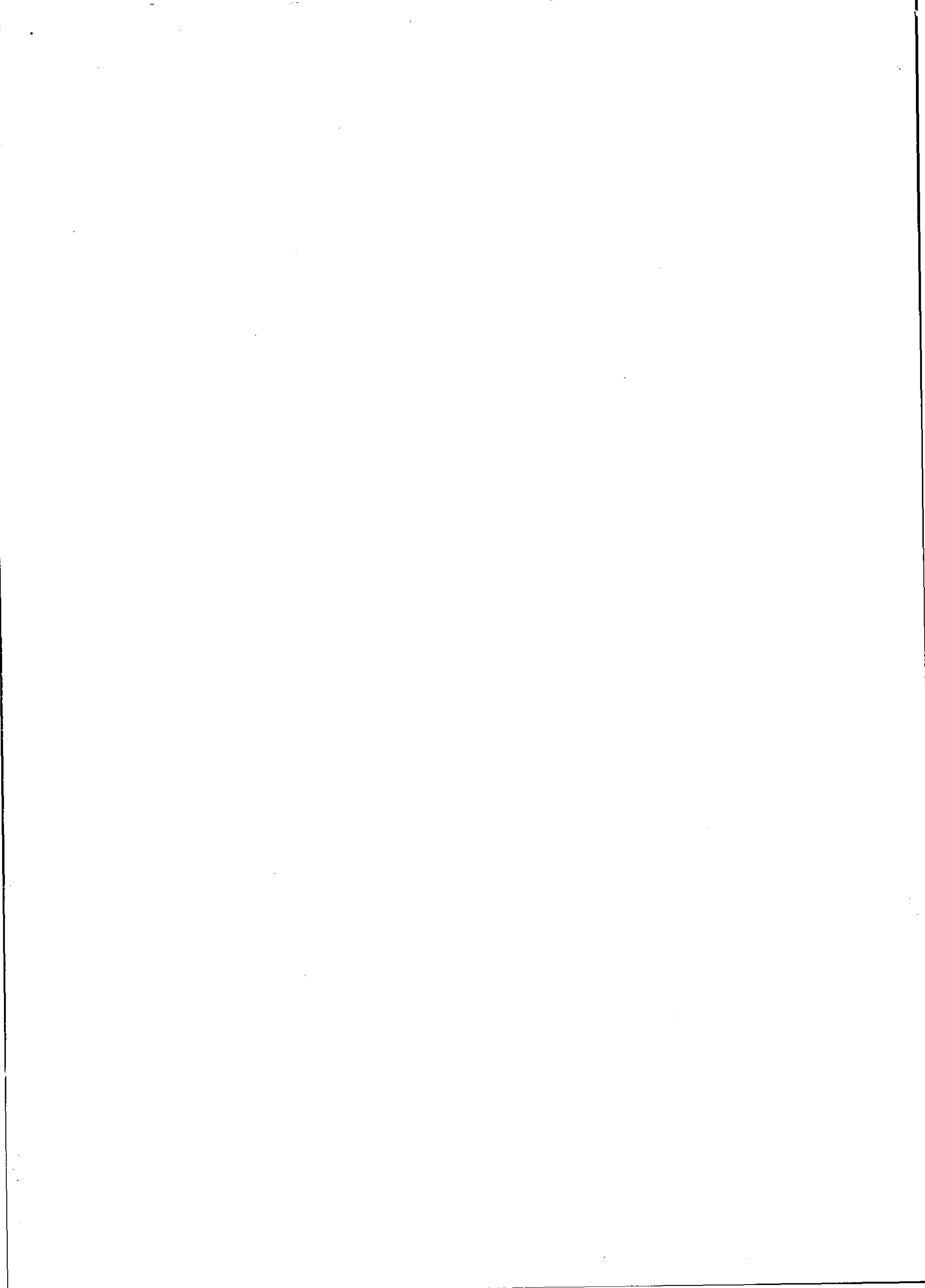
W

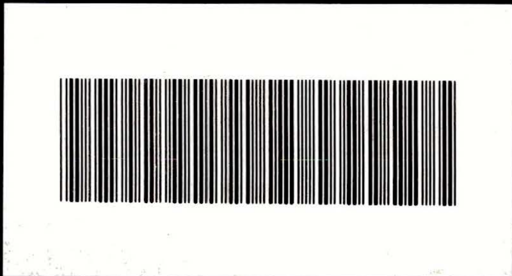
`watch` 517
watchpoints 747
commands
`clear handler` 63
`disable event` 109
`disable eventtype` 111
`enable event` 133
`enable eventtype` 135
`info event` 253
`info watch` 319
`remove event` 377
`remove eventtype` 379
`set handler` 443
`set ignore` 445
concepts
eventpoint handlers 647
eventpoints 651
parameters
event-handler 535
language-expression 555
windows 755
commands
`clear autocreate` 47
`display disassembly` 117
`display examine` 119
`display file` 121
`display routine` 123
`display source` 125
`display stack` 127
`find window backward` 199
`find window forward` 201
`set autocreate` 397
concepts
Maryland Windows 685
windows 755
working directory. *see*
console working directory
process working directory

X

X resources. *see* Xdefaults
Xdefaults 759

Master index





Order Number
DSW-477

Document Number
710-015430-003