CXdb Reference: Concepts and Messages

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:
Concepts and Messages

Order No. DSW-478

First Edition
November 1992

CONVEX Press
Richardson, Texas
United States of America

CONVEX
CXdb Reference:
Concepts and Messages

Order No. DSW-478

Printed in the United States of America

Revision Information for

CONVEX CXdb Reference: Concepts and Messages

Edition	Document No.	Description
First Edition	710-024130-000	Initial release, November 1992. This document describes concepts and messages used in CONVEX CXdb V2.0. This document is part of a two-volume set that also includes the *CONVEX CXdb Reference: Commands and Parameters* (Document No. 710-015430-003). 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

Purpose and audience

The *CONVEX CXdb Reference: Concepts and Messages* describes the major concepts associated with using CXdb. It also lists all the messages generated by CXdb.

This book is part of a two-volume set that also includes *CONVEX CXdb Reference: Commands and Parameters*. 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 book are also available through the CXdb online help system.

Organization

This book is organized as follows:

- **Chapter 1, "Concepts"**—Explains the major concepts involved in using the CXdb commands.
- **Chapter 2, "Messages"**—Lists and describes all CXdb messages.
- **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 source code is 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 book is not a complete explanation of CXdb. For more information, refer to:

- *CONVEX CXdb Reference: Commands and Parameters* (DSW-477)—The complement to this book, which presents detailed information about CXdb commands and associated command parameters.
- *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 help 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

Concepts

1

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

- **Description** — Text explaining the concept.
- **Examples** — One or more examples illustrating the concept, where applicable.
- **Related Commands** — A list of CXdb commands related to the concept. The commands are described in the complement to this volume, *CONVEX CXdb Reference: Commands and Parameters*.
- **Related Concepts** — A list of other concepts related to the concept being described. The related concepts are also described in this chapter.
- **Related Parameters** — A list of command parameters related to the concept. CXdb parameters are described in the complement to this volume, *CONVEX CXdb Reference: Commands and Parameters*.

For more details about the concepts described in this chapter, refer to the *CONVEX CXdb User's Guide*.

background execution

Description

CXdb process execution commands can be run in the background.

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

With background execution, your process executes while you continue to use CXdb. To run a command in the background, follow the command with the ampersand (&) on the command line. When a command is run in the background, the CXdb command prompt returns, allowing you to enter other CXdb commands. While a command is running in the background you cannot use any CXdb commands that require the process to be stopped. The command runs in the background until the command is completed or process execution is stopped. Only one command may run in the background at a time.

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

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

To move one of the above commands to the background after the command has begun execution you can type **CTRL-c** in the command window. This does not stop the process.

Once a process execution command is running in the background you can stop the process with the `stop` command. When the process is stopped (or terminates) the command in the background is completed. CXdb displays a message when the command finishes executing.

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

Examples

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

```
(CXdb) run &
Command [#3] backgrounded

Beginning execution of process [#0]
(CXdb)
```

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

```
(CXdb) step statement 1000 &
Command [#45] backgrounded

Stepping process [#0] by statement 1000 times
Command [#45] completed.
(CXdb)
```

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

Related Commands

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

Related Concepts

stepping
windows

breakpoints

Description

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

There are several different ways to set a breakpoint. The associated commands are described below:

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

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

Several commands allow you to interact with existing breakpoints. These commands are described below:

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

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

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

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

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

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

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

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

Examples

The following series of examples create and manipulate several different breakpoints in one process object.

```
(CXdb) break line 60

#0: break line, on [#0/*], Enabled, ignore 0/0
        [0x80001620] BESTMV in pickup.f line 60
```

The above command sets a breakpoint at line 60 in the current source file. The eventpoint number for this breakpoint is 0 and the address of the breakpoint is `80001620` in routine `BESTMV` at line 60 in the source file `pickup.f`.

```
(CXdb) break routine BESTMV

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

The above command sets a breakpoint at the first executable source unit in the routine containing the address of the routine `BESTMV`. Obviously the routine `BESTMV` contains its own address, so the breakpoint is set at the first executable source unit of the routine `BESTMV`. The first executable source unit of a routine is usually the first statement of a routine after local variables have been declared unless there are local initializations.

```
(CXdb) break instruction BESTMV

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

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

```
(CXdb) break source 135

#3: break source, on [#0/*], Enabled, ignore 0/0
        [0x800016f0] BESTMV in pickup.f line 65
```

The above command sets a breakpoint at the starting address of source unit 135. The source unit numbers for a given line can be displayed using the `info line` command.

```
(CXdb) info break
Event   Enabled     Ignore     proc/td      Address        Where
#0        y          0/0        0/*      [0x80001620] BESTMV in pickup.f line 60
#1        y          0/0        0/*      [0x800015f2] BESTMV in pickup.f line 59
#2        y          0/0        0/*      [0x800015f0] BESTMV in pickup.f line 55
#3        y          0/0        0/*      [0x800016f0] BESTMV in pickup.f line 65
```

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

```
(CXdb) run
Beginning execution of Process [#0]
Process [#0/0] stopped by Bkpt 2, at [0x800015f0] BESTMV in pickup.f line 55
```

The above example begins process execution without passing any arguments to the process. When execution reaches the address of `800015f0`, breakpoint 2 is reached because it is enabled. Because it does not have an ignore count, the breakpoint is triggered. Because the breakpoint did not have its own eventpoint handler, the default handler for breakpoints is executed, which prints the message telling you what caused process execution to stop.

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

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

```
(CXdb) break line 60 $Middle

#4: break line, on [#0/*], Enabled, ignore 1/2
        [0x80001620] BESTMV in pickup.f line 60

INFO: Eventpoint 0 also has a breakpoint at address 0x80001620.
```

The above command sets another breakpoint at line 60 of the current source file. The debugger variable `$Middle` was created and assigned to this eventpoint. CXdb informs you that two eventpoints now reside at the same address location.

```
(CXdb) break line 60 {echo "Breakpoint 5 reached" ; resume;}

#5: break line, on [#0/*], Disabled, ignore 0/0
        [0x80001620] BESTMV in pickup.f line 60
 {
      echo "Breakpoint 5 reached" ;
      resume;
 }

INFO: Eventpoint 4 also has a breakpoint at address 0x80001620.
```

The above command sets a third breakpoint at line 60. An eventpoint handler is specified for this eventpoint. The handler prints a message and then resumes execution of the process.

```
(CXdb) continue
Resuming execution of process [#0]
Breakpoint 5 reached
Resuming execution of process [#0]
Process [#0/0] stopped by Bkpt 3, at [0x800016f0] BESTMV in pickup.f line 65
```

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

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

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

```
(CXdb) run
Process [#0] is already running with pid 16139.
Terminate existing process and restart? y
Beginning execution of Process [#0]
Process [#0/0] stopped by Bkpt 0, at [0x80001620] BESTMV in pickup.f line 60
```

When process execution reaches address `80001620`, breakpoint 4 is reached because eventpoint 5 is disabled. Breakpoint 4 has an ignore count, so its counter is incremented by one. Because an eventpoint has still not been triggered, breakpoint 0 is reached. Because it is enabled and does not have an ignore count, breakpoint 0 is triggered.

```
(CXdb) info event *
#0: break line, on [#0/*], Enabled, ignore 0/0
        [0x80001620] BESTMV in pickup.f line 60
#3: break source, on [#0/*], Enabled, ignore 0/0
        [0x800016f0] BESTMV in pickup.f line 65
#4: break line, on [#0/*], Enabled, ignore 1/2
        [0x80001620] BESTMV in pickup.f line 60
#5: break line, on [#0/*], Disabled, ignore 0/0
        [0x80001620] BESTMV in pickup.f line 60
 {
       echo "Breakpoint 5 reached" ;
       resume;
 }
```

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

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

The above command enables breakpoint 5. This will cause breakpoint 5 to be triggered the next time address `80001620` is reached because it is the highest-numbered, enabled breakpoint.

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

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

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

Related Commands

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

Related Concepts

eventpoints	eventpoint handlers
tracepoints	watchpoints

Related Parameters

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

C language expressions

Description

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

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

1. Identifiers:

- Restrictions:
  - Identifiers are case sensitive.
  - If a file name appears in the scope path prefix to the identifier, and if the file name contains a special character such as dot (.) that is a lexical element in the alphabet of the language, then precede the special character with a backslash (\).
  - If a dollar sign ($) is part of the identifier name, precede the dollar sign with a backslash (\).
  - An identifier qualified by "`const`" is tracked, and a warning is issued if it is modified.
- Program variables can be either:
  - Unqualified
  - Qualified by a scope path prefix
- CXdb debugger variables:
  - Debugger variables must begin with the CXdb scope prefix `cxdb$` or `$.`
  - Debugger variable names are case sensitive.
  - An assignment to a debugger variable modifies it to have the value, type, and precision of the right-hand side of the assignment.
- Hardware registers:
  - Register names must be qualified with the CXdb scope prefix `cxdb$` or `$.`
  - Register names are not case sensitive, except for the scalar, vector, and communication registers.
  - An assignment to a register modifies its value only; its type and precision remain the same.

2. Constants:

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

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

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

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

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

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

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

  - The following trigraph sequences are accepted:

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

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

3. Expressions:

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

  - Function calls
  - Structure and union members:

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

    `++` postfix increment
    `--` postfix decrement

- Unary operators include:
  - Prefix operators:

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

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

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

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

- Multiplicative operators include:

  `*` multiply
  `/` divide
  `%` mod left operand by right

- Additive operators include:

  `+` add
  `-` subtract

- Shift operators include:
  - `<<` shift left
  - `>>` shift right
- Relational operators include:
  - `<` test for less than
  - `>` test for greater than
  - `<=` test for less than or equal to
  - `>=` test for greater than or equal to
- Equality operators include:
  - `==` test for equal to
  - `!=` test for not equal to
- Bitwise operators include:
  - `&` AND
  - `^` exclusive OR
  - `|` inclusive OR
- Logical operators include:
  - `&&` test for AND
  - `||` test for OR
- The conditional operator (`?:`):
  - If the test condition is true, evaluate the operand to the left of the colon (`:`).
  - Otherwise, evaluate the operand to the right of the colon.
- Assignment operators include:
  - `=` simple assignment
  - `*=` multiply and assign
  - `/=` divide and assign
  - `%=` mod and assign
  - `+=` add and assign
  - `-=` subtract and assign
  - `&=` AND and assign
  - `^=` exclusive OR and assign
  - `|=` inclusive OR and assign
  - `<<=` shift left and assign
  - `>>=` shift right and assign
- The comma operator (`,`):
  - Evaluate the left operand, then the right.
  - Return the result of the right operand.

Examples

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

```
(CXdb) break routine 0x800013d4
#1: break routine, on [#0/0], Enabled, ignore 0/0
        [0x800013d4] numbers'suba in numbers.c line 19
```

The above command sets a breakpoint at the beginning of the routine that contains the hexadecimal address `800013d4`. The expression `0x800013d4` is C language notation for a hexadecimal address.

```
(CXdb) print ++k
(int) 5
```

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

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

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

```
(CXdb) print/x &y
(int*) 0x8000c268
```

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

```
(CXdb) print matrix[0][3][0..4]
float[1][3..3][5]
[0][3][0..4] :      28.5585     17.6754     6.7924     -4.0906     -14.9737
```

The above command prints an array slice, or subset. The slice consists of elements `[0][3][0]` through `[0][3][4]` of `matrix`.

```
(CXdb) find memory forward ff &matrix:k+50
Data found at 0x8000d438
```

The above command searches for the hexadecimal byte pattern `ff` in a region of process memory. The memory region is specified by the language expression `&matrix:k+50`. The first part of the expression uses the operator & to return the starting address of `matrix`. The second part of the expression is `k+50`, and it evaluates to the number of bytes of memory to search.

Related Commands

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

Related Concepts

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

Related Parameters

array-slice
debugger-variable
language-expression
string

cmderr

Description

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

To direct the CXdb messages to different destinations, you create a list of viewports for cmderr. The commands for maintaining this list are:

- `add cmderr` — Add new viewports to the viewport list for cmderr.
- `remove cmderr` — Remove viewports from the viewport list for cmderr.
- `set cmderr` — Delete the current viewport list for cmderr and replace it with the specified list.

For viewports that are files, you might want to control whether or not the new data will overwrite an existing file. The following commands give you this control:

- `clear noclobber` — Allow existing files to be overwritten.
- `set noclobber` — Respond with an error message if the specified viewport file already exists.

The default viewport for cmderr is Window #1 (the command window), and the default for noclobber is clear (off).

Examples

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

```
(CXdb) set noclobber
```

The above command prevents overwriting of existing files.

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

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

Related Commands

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

Related Concepts

cmdlog
cmdout
logging
viewports
windows

Related Parameters

redirection-operator
viewport

cmdlog

Description

cmdlog refers to any entries, or input, made in the CXdb command window. It includes input read from a command file or initialization file as well as entries you make directly in the command window.

Entries that you make directly are always echoed in the command window. Input read from command files or initialization files is not echoed in the command window unless the echo option is turned on (either by default or by means of the `set echo` command).

In addition to echoing input in the command window, you can store it in any number of files. You do this by specifying a viewport list that contains the names of the destination files for cmdlog. The commands to modify the viewport list for cmdlog are:

- `add cmdlog` — Add new viewports to the viewport list for cmdlog.
- `remove cmdlog` — Remove viewports from the viewport list for cmdlog.
- `set cmdlog` — Delete the current viewport list for cmdlog and replace it with the specified list.

Once you have specified a viewport list for cmdlog, you can enable or disable logging to all the viewports in that list by using the following commands:

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

If you are using files to log input data, you can use the following commands to specify whether or not the new data will overwrite an existing file:

- `clear noclobber` — Allow existing files to be overwritten.
- `set noclobber` — Respond with an error message if the specified viewport file already exists.

Examples

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

```
(CXdb) set noclobber
```

The above command prevents overwriting of existing files.

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

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

```
(CXdb) set logging
```

The above command enables logging to all viewports of cmdlog. In this case, any further input to the command window is stored in the file `my_input.log`.

Related Commands

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

Related Concepts

cmderr
cmdout
logging
viewports
windows

Related Parameters

viewport

cmdout

Description

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

By default, the CXdb output appears in the command window. However, you can also send this output to other destinations, called viewports. A viewport may be either a file or the CXdb command window.

To direct the CXdb output to different destinations, you create a list of viewports for cmdout. The commands for maintaining this list are:

- `add cmdout` — Add new viewports to the viewport list for cmdout.
- `remove cmdout` — Remove viewports from the viewport list for cmdout.
- `set cmdout` — Delete the current viewport list for cmdout and replace it with the specified list.

For viewports that are files, you might want to control whether or not the new data will overwrite an existing file. The following commands give you this control:

- `clear noclobber` — Allow existing files to be overwritten.
- `set noclobber` — Respond with an error message if the specified viewport file already exists.

The default viewport for cmdout is Window #1 (the command window), and the default for noclobber is clear (off).

Examples

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

```
(CXdb) set noclobber
```

The above command prevents overwriting of existing files.

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

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

Related Commands

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

Related Concepts

cmderr
cmdlog
logging
viewports
windows

Related Parameters

redirection-operator
viewport

command files

Description

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

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

There are four primary uses for command files:

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

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

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

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

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

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

Examples

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

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

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

Also assume that echoing is enabled (on).

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

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

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

Related Commands

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

Related Concepts

cmdlog
console working directory
initialization files
logging
windows

Related Parameters

file-name

Compiler-Debugger Interface
CDI

Description

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

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

NOTE: If you move any of the CDI data files from their original location, you must use the `dirpath` command to give CXdb the new path to these files.

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

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

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

- Version 8.0 or later of the CONVEX FORTRAN compiler and version 5.0 or later of the CONVEX C compiler:

  `.ns` — Name space; provides information about external program symbols and scope blocks.

  `.fe` — Front end; provides language-specific data and lexical scope information about all program identifiers and source units. The source units reflect the syntax of the source code.

- Prior to version 8.0 of the CONVEX FORTRAN compiler or version 5.0 of the CONVEX C compiler:

  `.ns` — Name space; provides information about external program symbols and scope blocks.

  `.tsi` — Type and scope information; provides language-specific data and lexical scope information about all program identifiers.

  `.sut` — Source unit table; provides information about individual source units as well as the interrelationships between source units. The source units reflect the syntax of the source code.

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

- Version 8.0 or later of the CONVEX FORTRAN compiler and version 5.0 or later of the CONVEX C compiler:

  `.be` — Back end; specifies storage locations and liveness ranges for program variables, synthesized variables, and source units.

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

- Prior to version 8.0 of the CONVEX FORTRAN compiler or version 5.0 of the CONVEX C compiler:

  `.lrt` — Location range table; specifies the storage locations of each variable for various ranges of the program counter (PC).

  `.srt` — Source range table; specifies which source units are active over various ranges of the program counter (PC).

  `.vt` — Variable table; specifies the attributes of all variables, including synthesized variables. It is analogous to the symbol table used by the compiler.

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

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

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

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

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

Caution

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

Related Commands

`dirpath`
`info dirpath`
`remove dirpath`

Related Concepts

search path

console working directory

Description

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

The commands that use the console working directory are listed below:

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

The commands that affect the console working directory are described below:

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

The console working directory is initially set to the directory from which you invoked CXdb. The process working directory is initially set to the console working directory.

Examples

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

```
(CXdb) source cxdb/commfile.1
```

The above command executes the CXdb commands found in the `/mnt/jones/cxdb/commfile.1` file. Because the path name to the file is relative, the console working directory is used as the base path name.

```
(CXdb) cd cxdb
```

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

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

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

Related Commands

`add cmderr`
`add cmdlog`
`add cmdout`
`add default path`
`add path`
`cd`
`core`
`debug core`
`debug exec`
`executable`
`info cxdb`
`pwd`
`remove cmderr`
`remove cmdlog`
`remove cmdout`
`remove default path`
`remove path`
`set cmderr`
`set cmdlog`
`set cmdout`
`set default path`
`set directory`
`set path`
`source`

Related Concepts

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

Related Parameters

directory-specifier
file-name

csd debugger

Description

The CONVEX `csd` debugger is a symbolic debugger. The most frequently used `csd` commands are available in CXdb through aliases.

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

- From the UNIX command line — By entering the command `cxdb -csd`.
- From the CXdb command window — By entering the command `csd`.

If you always want to use the `csd` aliases in either of the above modes, you can include the `source /usr/lib/cxdb/aliases/csd_aliases` command in your `.cxdbinit` file. This incorporates the predefined `csd` aliases automatically each time you invoke CXdb.

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

The `csd` commands supported by CXdb aliases are:

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

csd command	CXdb equivalent

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

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

`fpmode ieee`	`set format ieee`
`fpmode native`	`set fpmode native`
`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`

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

Related Commands

csd
cxdb
gdb
info alias
source

Related Concepts

gdb debugger

debugger variables

Description

Debugger variables are variables that you can define in CXdb.

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

In some cases, it is necessary to distinguish a debugger variable from another type of symbol with the same name. To distinguish the symbols, you can precede the debugger variable with a dollar sign (`$`).

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

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

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

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

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

```
evaluate
print
```

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

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

In addition to debugger variables that you define, there are a number of special, predefined debugger variables that enable you to reference the process registers. These debugger variables are dynamic. When your process is running, they are updated to contain the current register values. The names of these special variables, and the registers they represent, are as follows:

- `$a0` to `$a7` (or `$A0` to `$A7`) — The 32-bit address registers A0 to A7.
- `$ap` (or `$AP`) — The argument pointer (AP or A6).
- `$c0` to `$c63` — The 32-bit communication registers from ring 4, regardless of which set is allocated.
- `$C0` to `$C63` — The 64-bit communication registers from ring 4, regardless of which set is allocated.
- `$cl0` to `$cl63` (or `$CL0` to `$CL63`) — The lock bits for the communication registers (one bit per register).
- `$cir` (or `$CIR`)— The 3-bit communication index register (CIR).
- `$fp` (or `$FP`) — The frame pointer (FP or A7).

- `$pc` (or `$PC`) — The program counter (PC).
- `$psw` (or `$PSW`) — The processor status word (PSW).
- `$s0` to `$s7` — The 32-bit scalar registers S0 to S7.
- `$S0` to `$S7` — The 64-bit scalar registers S0 to S7.
- `$sp` (or `$SP`) — The stack pointer (SP, or A0).
- `$v0` to `$v7` — The 32-bit vector registers V0 to V7. Each vector register may contain up to 128 elements (numbered 0 to 127), and each element is 32 bits long. To access a particular element, use array notation. For example, to access the 123rd element of vector register V4, use the notation `$v4(123)` in FORTRAN or `$v[122]` in C.
- `$V0` to `$V7` — The 64-bit vector registers V0 to V7. Each vector register may contain up to 128 elements (numbered 0 to 127), and each element is 64 bits long. To access a particular element, use array notation. For example, to access the 123rd element of vector register V4, use the notation `$V4(123)` in FORTRAN or `$V[122]` in C.
- `$vl` (or `$VL`) — The vector length register (VL).
- `$vml` (or `$VML`) — The lower half of the vector merge register (VM).
- `$vmu` (or `$VMU`) — The upper half of the vector merge register (VM).
- `$vs` (or `$VS`) — The vector stride register (VS).

Finally, there are two other predefined debugger variables that are updated dynamically:

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

Examples

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

```
(CXdb) break routine SUB_D $D
Breakpoint 2, [0x80001882] SUB_D in myprog.f line 93
```

The above command sets a breakpoint at the routine called `SUB_D`. The object number for this breakpoint is 2, and this object number is stored in the debugger variable `$D`.

Once the debugger variable has been created, it can be referenced in a subsequent command, as follows:

```
(CXdb) set ignore 3 $D
Event 2 will be ignored 3 times
```

The above command sets an ignore count for eventpoint 2, which is represented in this command by the debugger variable `$D`.

```
(CXdb) evaluate $A=x+y
```

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

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

The above command stores the value `1234` in the scalar register S0. The contents of the other predefined debugger variables, except `$self`, can be modified in a similar way.

Related Commands

break instruction	break line
break routine	break source
debug core	debug exec
debug proc	evaluate
event exec	event modify
event reached instruction	event reached line
event reached routine	event reached source
event relation	event signal
macro	print
trace instruction	trace line
trace routine	trace source
watch	

Related Concepts

command files
eventpoint handlers
initialization files

Related Parameters

debugger-variable

default environment

Description

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

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

Modifications to the default environment do not affect existing processes that were passed the default environment. You can modify the default environment with the following commands:

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

Examples

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

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

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

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

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

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

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

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

If a new process is created, and its process object does not have its own environment, it is passed these three environment variables of the default environment.

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

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

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

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

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

Related Commands

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

Related Concepts

environment

process object

Related Parameters

environment-variable

string

default search path

Description

The default search path is used as the basis for the search path for each new process object.

The search path of a process object is used to find program source files as well as CXdb compiler-generated data files.

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

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

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

Examples

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

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

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

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

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

The above example adds two directories to the default search path. The next process object that is created will receive the new default search path which now includes the `/mnt/jones/libraries` and `/mnt/jones/math/libraries` directories.

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

The above command removes the `/mnt/jones/math/libraries` directory from the default search path. The other directories in the default search path remain.

```
(CXdb) info path
Default search list:
        .
        /mnt/jones/libraries

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

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

```
(CXdb) set default path /mnt/jones/project2 , /mnt/jones/project2/source
(CXdb) info path
Default search list:
        /mnt/jones/project2
        /mnt/jones/project2/source


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

The above command removes all the existing directories from the default search path and sets the default search path to the two listed directories. Now each new process object will include the `/mnt/jones/project2` and `/mnt/jones/project2/source` directories, as well as the process working directory as its search path. The `info path` command displays the new default search path.

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

Related Commands

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

Related Concepts

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

Related Parameters

directory-specifier

environment

Description

The environment is the set of environment variables of a process object. A process object does not initially have an environment. An environment is created for a process object the first time you change its environment. The environment then becomes a copy of the default environment modified by the effects of the environment command issued.

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

You can modify the environment of a process object with the following commands. Only the `info environment` command does not create an environment for a process object.

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

Examples

The following examples illustrate how to use the environment commands.

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

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

```
(CXdb) clear environment
```

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

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

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

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

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

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

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

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

Related Commands

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

Related Concepts

default environment — process object

Related Parameters

environment-variable — string

eventpoint handlers

Description

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

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

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

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

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

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

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

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

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

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

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

Examples

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

```
(CXdb) break line 56 {print x;}
```

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

```
(CXdb) break line 56 {print x; resume;}
```

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

```
(CXdb) break line 56 {if (x==49) {echo "x is 49";} else {resume;} }
```

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

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

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

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

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

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

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

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

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

Related Commands

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

Related Concepts

breakpoints
debugger variables
eventpoints
tracepoints
watchpoints

Related Parameters

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

eventpoints

Description

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

You can trap the following types of events:

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

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

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

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

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

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

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

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

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

There are many different ways to set a eventpoint. The different commands are divided into their eventpoint types.

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

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

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

The following command sets a *signal* eventpoint.

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

The following command sets a *modify* eventpoint.

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

The following command sets a *relational* eventpoint.

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

The following command sets an *exec* eventpoint.

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

The following command sets a *join* eventpoint.

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

The following command sets a *spawn* eventpoint.

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

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

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

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

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

Examples

For examples of how to set, manipulate, and remove eventpoints, refer to the individual reference pages for the above commands in the complement to this volume, *CONVEX CXdb Reference: Commands and Parameters*.

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

Related Commands

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

Related Concepts

eventpoints
eventpoint handlers
language expressions
tracepoints
watchpoints

Related Parameters

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

FORTRAN language expressions

Description

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

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

1. Identifiers:

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

2. Constants:

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

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

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

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

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

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

3. Arithmetic expressions:

- Additive operators:

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

- Multiplicative operators (binary):

  *
  /

- Exponential operators (binary):

  **

4. Relational expressions:

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

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

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

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

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

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

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

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

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

11. Cray pointers:
    - Refer to "Cray compatibility" in the *CONVEX FORTRAN Guide* for specifications on pointers.
12. Intrinsic functions:
    - Restrictions:
      - Intrinsic functions cannot be passed as arguments in calls.
      - White space is significant.
      - Data types are relaxed. For example, `IIDNNT` accepts `REAL*4` and `REAL*16` as well as `REAL*8`.
      - Intrinsic function names are not case sensitive.
    - Intrinsic functions for converting to integer:

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

    - Intrinsic functions for converting to real:

      ```
      REAL
      DBLE
      QEXT
      ```

    - Intrinsic functions for converting to complex:

      ```
      CMPLX
      DCMPLX
      ```

    - Intrinsic functions for character conversions:

      ```
      CHAR
      ICHAR
      ```

    - Intrinsic functions for string comparisons:

      ```
      LLT
      LLE
      LGT
      LGE
      ```

- Intrinsic functions for conversion to nearest integer:

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

- Intrinsic functions for address acquisition:

```
LOC
%LOC
```

- Miscellaneous intrinsic functions:

```
ABS
MOD
```

Examples

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

```
(CXdb) break routine '8000139a'x
#1: break routine, on [#0/0], Enabled, ignore 0/0
        [0x8000139a] SUBA in numbers.f line 16
```

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

```
(CXdb) print K=K+1
(INTEGER*4) 5
```

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

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

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

```
(CXdb) print/x loc(Y)
(INTEGER*4) 0x8004a00c
```

The above command evaluates the expression `loc(Y)` and prints the result in hexadecimal (`/x`) format. The FORTRAN function `loc()` returns the address of the program variable `Y`. Thus, `8004a00c` is the hexadecimal address of `Y`. (The `0x` in front of the address indicates that it is a hexadecimal number.)

```
(CXdb) print MATRIX(1,4,1..5)
REAL*4(1:1, 4:4, 1:5)
(1,4,1..5) :     28.5585    17.6754    6.7924    -4.0906    -14.9737
```

The above command prints an array slice, or subset. The slice consists of elements `(1,4,1)` through `(1,4,5)` of `MATRIX`.

```
(CXdb) find memory forward ff loc(ARRAY):J+50
Data found at 0x800ac438
```

The above command searches for the hexadecimal byte pattern `ff` in a region of process memory. The memory region is specified by the language expression `loc(ARRAY):J+50`. The first part of the expression uses the FORTRAN function `loc()` to return the starting address of `ARRAY`. The second part of the expression is `J+50`, and it evaluates to the number of bytes of memory to search.

Related Commands

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

Related Concepts

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

Related Parameters

array-slice
language-expression
debugger-variable
string

gdb debugger

Description

The CONVEX gdb debugger is a symbolic debugger. The most frequently used gdb commands are available in CXdb through aliases. To incorporate these aliases into CXdb, enter the `gdb` command in the CXdb command window.

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

With the predefined aliases incorporated, you can type in a gdb command while using CXdb. If the command has an alias, the alias is substituted, and the equivalent CXdb command is executed. If the 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`

gdb command	CXdb equivalent
`info comm-registers`	`info cregisters`
`info directories`	`info process`
`info display`	`info event`
`info files`	`info process`
`info functions`	`info symbols`
`info methods`	No equivalent.
`info sources`	`info objectmap`
`info types`	`info type`
`info variables`	`info symbols`
`jump`	Use `goto line` or `goto address` command.
`list`	Use `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.

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

Related Commands

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

Related Concepts

csd debugger

initialization files

Description

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

The primary uses for initialization files are:

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

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

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

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

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

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

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

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

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

Examples

The following example illustrates several uses of initialization files.

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

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

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

```
debug exec a.out
break routine 1$_MAIN__
run without
disassemble
alias s print
set default step loop
```

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

```
(CXdb)
Default source file: ./pickup6.f
Default source language: Fortran

Process [#0] created
Breakpoint 0, [0x80001334] PICKUP in pickup6.f line 1
Beginning execution of Process [#0]
Process [#0/0] stopped by Bkpt 0, at [0x80001334] PICKUP in pickup6.f line 1
Disassemble Process [#0/0] from 0x80001334 to 0x80001460

(CXdb)
```

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

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

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

Related Commands

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

Related Concepts

cmdlog
command files
console working directory
logging
windows

Related Parameters

file-name

language expressions

Description

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

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

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

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

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

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

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

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

- As a data value
- As an address

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

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

Examples

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

```
(CXdb) break routine SUBA \; $BreakA
#1: break routine, on [#0/0], Enabled, ignore 0/0
        [0x8000139a] SUBA in numbers.f line 16
```

The above command sets a breakpoint at the beginning of the routine called `SUBA`. In this case, CXdb interprets the language expression `SUBA` to be the starting address of the routine `SUBA`. The debugger variable `$BreakA` stores the number of this breakpoint, which is 1. The language expression terminator (\;) is required to separate the language expression `SUBA` from the debugger variable `$BreakA`.

```
(CXdb) break routine '8000139a'x
#1: break routine, on [#0/0], Enabled, ignore 0/0
        [0x8000139a] SUBA in numbers.f line 16
```

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

```
(CXdb) break routine 0x800013d4
#1: break routine, on [#0/0], Enabled, ignore 0/0
        [0x800013d4] numbers'suba in numbers.c line 19
```

The above command sets a breakpoint at the beginning of the routine that contains the hexadecimal address `800013d4`. The expression `0x800013d4` is C language notation for a hexadecimal address.

```
(CXdb) print Y+2
(REAL*4) 4.5000
```

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

```
(CXdb) print Y=Y+2
(REAL*4) 6.5000
```

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

```
(CXdb) print/x loc(Y)
(INTEGER*4) 0x8004a00c
```

The above command evaluates the expression `loc(Y)` and prints the result in hexadecimal (`/x`) format. The FORTRAN function `loc()` returns the address of the program variable `Y`. Thus, `8004a00c` is the hexadecimal address of `Y`. (The `0x` in front of the address indicates that it is a hexadecimal number.)

```
(CXdb) print/x &y
(int*) 0x8000c268
```

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

```
(CXdb) print "This is a test"
(CHARACTER*14) "This is a test"
```

The above command prints a literal string of characters.

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

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

```
(CXdb) print MATRIX(1,4,1..5)
REAL*4(1:1, 4:4, 1:5)
(1,4,1..5) :     28.5585     17.6754     6.7924     -4.0906     -14.9737
```

The above command prints an array slice, or subset. The slice consists of elements `(1,4,1)` through `(1,4,5)` of `MATRIX`. The subscripts in this example are in FORTRAN notation.

```
(CXdb) print matrix[0][3][0..4]
float[1][3..3][5]
[0][3][0..4] :     28.5585     17.6754     6.7924     -4.0906     -14.9737
```

The above command prints an array slice, or subset. The slice consists of elements `[0][3][0]` through `[0][3][4]` of `matrix`. The subscripts in this example are in C notation.

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

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

```
(CXdb) find memory forward ff loc(ARRAY):J+50
Data found at 0x800ac438
```

The above command searches for the hexadecimal byte pattern `ff` in a region of process memory. The memory region is specified by the language expression `loc(ARRAY):J+50`. The first part of the expression uses the FORTRAN function `loc()` to return the starting address of `ARRAY`. (The equivalent function for returning the address of a variable in C syntax is `&`.) The second part of the expression is `J+50`, and it evaluates to the number of bytes of memory to search.

Related Commands

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

Related Concepts

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

Related Parameters

array-slice	debugger-variable
language-expression	string

logging

Description

Logging is the process of recording the activity that takes place during a debugging session. Three types of information can be logged, and each type has a particular name. The types are:

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

The method for logging the information is to direct it to a viewport. A viewport is either a file or the CXdb command window. A file is a permanent log because it can be saved, but the command window is obviously a temporary log for display purposes only.

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

- `add cmderr` — Add new viewports to the list of cmderr viewports.
- `add cmdlog` — Add new viewports to the list of cmdlog viewports.
- `add cmdout` — Add new viewports to the list of cmdout viewports.
- `remove cmderr` — Remove viewports from the list of cmderr viewports.
- `remove cmdlog` — Remove viewports from the list of cmdlog viewports.
- `remove cmdout` — Remove viewports from the list of cmdout viewports.
- `set cmderr` — Delete the current list of cmderr viewports and replace it with a new list.
- `set cmdlog` — Delete the current list of cmdlog viewports and replace it with a new list.
- `set cmdout` — Delete the current list of cmdout viewports and replace it with a new list.

The default viewport for cmderr and cmdout is the CXdb command window (Window #1). There is no default viewport for cmdlog because your entries in the command window are automatically echoed there.

For logging, most of the viewports you specify will be files. If the specified file does not exist, CXdb creates it. If the specified file already exists, then you can use the following commands to control whether or not CXdb writes (either overwrites or appends) to the existing file:

- `clear noclobber` — Allow overwriting of existing files and creation of new files for appending.
- `set noclobber` — Respond with an error message if attempting to overwrite an existing file or attempting to append to a file that does not exist.

The default for noclobber is clear (off).

The viewport lists for cmderr and cmdout apply to all commands executed by CXdb, regardless of whether you enter the commands directly yourself or read them in from a command file. However, for each individual command, you can override the viewport lists for cmdout and cmderr by using redirection operators with the command. The redirection operators enable you to specify a special viewport list that applies only to the one command with which it appears.

For cmdlog, the following commands enable you to control whether or not input from command files and initialization files is echoed to the cmdlog viewports:

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

The default for echo is set (on).

Also for cmdlog, you can enable or disable logging of all input with the following commands:

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

The default is logging disabled (clear).

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

Examples

The following examples illustrate one way to modify the viewport lists for cmderr, cmdlog, and cmdout. They also illustrate how to use redirection operators to override the viewport lists.

```
(CXdb) set noclobber
```

The above command prevents writing to existing files.

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

The above command adds the file `error_log` to the viewport list for cmderr. Note that the list already contains Window #1 (the command window) because it is the default viewport for cmderr. All CXdb messages are now stored in `error_log` as well as being displayed in the command window.

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

The above command adds the file `output_log` to the viewport list for cmdout. This viewport list already contains Window #1 (the command window) as the default. All output from CXdb commands is now stored in `output_log` as well as being displayed in the command window.

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

The above command adds the file `input_log` to the viewport list for cmdlog. This viewport list does not contain Window #1 as the default because everything you enter in the command window is automatically echoed there. Adding Window #1 to the viewport list for cmdlog actually causes each input line to appear twice in the command window.

At this point, nothing is sent to the file `input_log` because logging has not been enabled for cmdlog. To enable it, enter the following:

```
(CXdb) set logging
```

Now, any commands you enter in the command window are stored in `input_log` as well as being displayed in the command window. This is a convenient way to create a command file. After exiting from CXdb, you can edit `input_log` so that it contains only the commands you want to use again. The next time you invoke CXdb, you can execute this file with the `source` command.

To display all current settings, enter the following:

```
(CXdb) info cxdb

Current CXdb state:

     Environment:
             pid: 2130
             cwd: /doc/Smith/ug
   Command Modes: Echo On, Logging On, Noclobber On
          cmdout: Window #1, output_log
          cmderr: Window #1, error_log
          cmdlog: input_log
        EvalOpts: fpmode = dual, iprecision = 4, rprecision = 4
           Shell: tcsh

Process Defaults:
Fixed Scheduling: Off
       Step size: statement
   Process shell: csh
          fpmode: dual
     Memory size: (none)
  Memory Formats: byte=(none), halfword=(none), word=(none)
                  longword=(none), quadword=(none)
     Search path:
         .

       Processes:
```

The above response indicates that echo, logging, and noclobber are all enabled (on). It also list the current viewports for cmdout, cmderr, and cmdlog.

You can override the cmdout and cmderr viewport lists for individual commands by using redirection operators. For example, you could enter the following:

```
(CXdb) info cxdb >temp_out >&temp_err
```

Output from the above command is redirected to the file `temp_out`, and any CXdb messages associated with this command are redirected to `temp_err`. No other viewports receive the response from this command, but the command line itself is still logged in `input_log`. (Because the noclobber option is enabled, an error results if `temp_out` or `temp_err` exists.)

Related Commands

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

Related Concepts

cmderr
cmdlog
cmdout
viewports
windows

Related Parameters

redirection-operator
viewport

Maryland Windows

Description

The Maryland Windows interface provides a windowing environment for CXdb when run on a non-graphics terminal. The windows in the Maryland Windows interface are similar to their counterparts in the CXwindows interface. However, keystrokes are used to manipulate the Maryland Windows and to edit the text inside those windows.

Functions are activated by specific key bindings. In the key bindings below, the **META** key (sometimes labeled **ESC**) is represented as `M-` and the **CTRL** key by the caret (^).

The function names are grouped together in the list below. Before each function name is a default key binding that performs that function.

- Window movement functions:
  - `M-o` `lower-window`
  - `M-m` `move-window`
  - `M-n` `next-window`
  - `M-p` `previous-window`
  - `M-r` `raise-window`
  - `M-z` `resize-window`
  - `M-k` `close-window` (except command window)
- Cursor movement and scrolling functions:
  - `M-<` `beginning-of-buffer`
  - `M->` `end-of-buffer`
  - `^A` `beginning-of-line`
  - `^E` `end-of-line`
  - `^B` `backward-char` (except command window)
  - `^F` `forward-char` (except command window)
  - `M-b` `backward-word`
  - `M-f` `forward-word`
  - `^N` `down-line` (except command window)
  - `^P` `up-line` (except command window)
  - `^V` `down-screen`
  - `M-v` `up-screen`

To move the cursor around the command window, one character at a time, you must use the arrow keys.

In the command window you can perform the functions described below:

- Editing functions:

`M-h`	`backward-delete-word`
`M-^D,M-d,M-delete`	`kill-word`
`backspace,^h`	`delete-backward-char`
`delete,^D`	`delete-char`
`^K`	`kill-line`
`^W`	`kill-region`
`M-w,M-W`	`copy-region-as-kill`
`^@`	`set-mark-command`
`^X`	`exchange-point-and-mark`
`^T`	`transpose-characters`
`^Y`	`yank`

- History functions:

`^N`	`down-history`
`^P`	`up-history`

- Miscellaneous functions:

`M-c`	`capitalize-word`
`M-0 to M-9`	`digit-argument`
`^U`	`universal-argument`
`M-l`	`downcase-word`
`M-u`	`upcase-word`
`^L`	`redraw-display`

For a more complete description on the performance of these functions refer to the *CONVEX CXdb User's Guide* or the reference page for the `bind` command.

Related Commands

`bind` `info bind`

Related Concepts

windows

Related Parameters

function-name key-name

process object

Description

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

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

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

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

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

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

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

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

Examples

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

```
(CXdb) debug exec a.out

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

Process [#0] created
```

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

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

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

Process [#0/0] stopped execution at [0x80033bbe] ___ap$kill+0xe
```

The above example retrieves the core image from the file named `corefile`. The image being debugged is now the core image. The CDI information of the process object remains unchanged.

The source window updates to reflect the new image in the process object.

```
(CXdb) kill process
Discard core image on process [#0]? y
Core image discarded from Process [#0]
```

The above command removes the core image from the process object. The executable image is once again the image being debugged.

```
(CXdb) attach 20860
Attaching Process [#0] to pid 20860
Process [#0/0] stopped by attach at [0x80001554] FIB in program.f line 24
```

The above command attaches CXdb to the process whose process ID (`pid`) is `20860`. CXdb attaches to the process and then stops it. The process is now under CXdb's control. The process image for this process becomes the image being debugged. Again, the CDI information is not changed.

The source window updates to reflect the new image in the process object.

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

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

(CXdb) stop
Stopping Process [#0]
Process [#0/0] stopped at [0x800013d6] FIB in program.f line 10
```

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

The `stop` command stops execution of the process.

```
(CXdb) executable b.out

Default source file: ./program2.c
Default source language: C
```

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

The source window is deleted and a new source window opens for the new CDI information.

However, the image being debugged is still that created by the `run` command in the previous example. Typically, you would not want to continue debugging a process using another executable file's CDI information.

```
(CXdb) break routine fib_calculation

#0: break routine, on [#0/*], Enabled, ignore 0/0
        [0x8000146c] program2'fib_calculation in program2.c line 37
(CXdb) run
Process [#0] is already running with pid 8583.
Terminate existing process and restart? y
Starting process [#0]: b.out
Process [#0/0] stopped by Bkpt 0, at [0x8000146c] program2'fib_calculation
```

The `break routine` command creates a breakpoint at the beginning of the `fib_calculation` routine. Note that the breakpoint is placed according to the `b.out` executable file, not the `a.out` process currently running.

The `run` command replaces the existing process image with the process image from a new process. The new process is created from the executable image from the executable file `b.out`.

The source window updates to reflect the new process image.

```
(CXdb) info process
status of process [#0]:

      executable: b.out
       arguments: (none)
fixed scheduling: off
          pshell: csh
    image status: created pid 9211, state = stopped
     working dir: /usr/smith/programs
    default step: statement
default language: C
         threads: 1
  current thread: 0

 thread 0 status: stopped at [0x8000146c] program2'fib_calculation in program2.c

source file search path:
        .
```

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

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

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

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

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

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

Related Commands

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

Related Concepts

breakpoints
environment
eventpoints
eventpoint handlers
remote debugging
search path
tracepoints
watchpoints

Related Parameters

process-list
thread-list

process working directory

Description

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

The following commands work with the process working directory:

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

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

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

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

Examples

The following commands use the process working directory.

```
(CXdb) set directory /mnt/jones/projects
```

This command sets the process working directory to be the `/mnt/jones/projects` directory.

```
(CXdb) info process

status of process [#0]:

      executable: a.out
       arguments: (none)
fixed scheduling: off
          pshell: csh
    image status: no image
     working dir: (default to current CXdb directory)
    default step: statement
default language: Fortran

source file search path:
        .
```

The above command displays information about the current process object. The setting of the process working directory is shown to be the current console working directory.

Related Commands

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

Related Concepts

console working directory
default search path
search path

Related Parameters

directory-specifier

remote debugging

Description

CXdb enables you to debug an executable file, core file, or remote process residing on a remote host (a CONVEX machine other than the one on which CXdb is currently running). You can debug remotely just as you would debug locally. Once you specify the remote executable file, core file, or process, debugging proceeds as normal.

To debug a process or file on another host, the remote host must have CXdb (V1.2 or later) installed, and you must have an account on that host. If it is not installed, CXdb will not be able to create a server on the remote host, and thus not be able to allow remote debugging. For information on installing CXdb, contact your system administrator.

To debug an executable file on a remote host, specify the remote host when you specify the executable file. The remote host name is followed by a colon, followed by the path to the executable file on the remote host. The remote host name must be either an absolute Internet address or a name found in the `/etc/hosts` file. You can specify a remote executable using the `debug exec` or `executable` commands, as well as on the command line when invoking CXdb.

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

- Remote working directory—A remote directory can be specified for the process object, using the `set remotewd` command.
- Default remote working directory—A default remote directory can be specified to be used if the process object's remote directory is not set. The default remote working directory is specified using the `set default remotewd` command.
- The console working directory—The current working directory on the local host is used if a remote working directory has not been specified using one of the above methods.
- Your home directory on the remote host—If the console working directory does not exist on the remote host, your home directory on that host is used instead.

NOTE: The CDI data files and source files for the executable file are assumed to reside on the local host. CXdb searches for these files using the search path. For more information, refer to the concepts page on the search path.

To debug a core file on a remote host, specify the remote host when you specify the core file. The remote host name is followed by a colon, followed by the path to the core file on the remote host. The remote host name must be either an absolute Internet address or a name found in the `/etc/hosts` file. You can specify a remote core file using the `debug core` or `core` commands, as well as on the command line when invoking CXdb.

There are two ways of debugging a process on a remote host:

- Attaching to an existing process on the remote host—Using the `attach` or `debug proc` commands, you may specify a remote host name, followed by a colon, before specifying the process ID of the process to attach to.
- Creating a new process on the remote host—If the last executable file specified was on a remote host, then the `run` command will create a new process on the remote host. The process is created in the process object's process working directory (specified using the `set directory` command). If the process working directory has not been specified, the remote working directory is used instead.

All other aspects of debugging are the same between local debugging and remote debugging. The `info process` command displays the remote working directory and whether or not the executable file resides on a remote host.

Examples

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

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

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

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

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

Process [#0] created
```

This above example creates a process object. The executable file `a.out` provides the basis for the CDI information in the process object. The executable file is located on the remote host named `ben`. Because an absolute path was not given, CXdb searches for the file in the default remote working directory (`/usr/smith/programs`). CXdb uses the `a.out` file to find the appropriate CDI data files and source files on the *local* host.

The image being debugged at this point is the executable image created from the `a.out` file.

```
(CXdb) attach ben:1234
process on ben
Attaching Process [#0] to pid 1234
Process [#0/0] stopped by attach at [0x80001554] FIB in program.f line 24
```

The above command attaches CXdb to the process whose process ID (`pid`) is `1234` on the remote host named `ben`. The process should have been created from the remote executable file previously specified. CXdb attaches to the process and then stops it. This remote process can now be debugged as if it were a local process.

The image for this process is now being debugged. The CDI information is not changed. The source window updates to reflect the new image in the process object.

```
(CXdb) run

Process [#0] is already running with pid 1234.
Terminate existing process and restart? y

Starting process [#0]: a.out
```

The `run` command creates a new process from the executable image (from the `debug exec` command above). The old process image that was attached to in the previous example is replaced by the process image from the new process. The new process is running on the host `ben` because the executable file providing the executable image is located on `ben`.

```
(CXdb) info process
status of process [#0]:

      executable: a.out
       arguments: (none)
fixed scheduling: off
          pshell: csh
    image status: created pid 23001, state = stopped

     remote host: ben
     working dir: /usr/smith/programs
    default step: statement
default language: Fortran
         threads: 1
  current thread: 0

 thread 0 status: stopped at [0x800014fe] FIB in program.f line 21

source file search path:
        .
```

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

Related Commands

`add path`	`attach`
`clear default remotewd`	`core`
`debug core`	`debug exec`
`debug proc`	`detach`
`info path`	`info process`
`kill process`	`remove path`
`rerun`	`run`
`set default remotewd`	`set path`
`set remotewd`	

Related Concepts

background execution	process object
process working directory	search path

scope

Description

The scope of a program identifier determines where that identifier is visible. There are two concepts involving scope in CXdb:

- **Current scope** — The current scope is used to find identifiers that do not have a complete scope path. The current scope is determined by the current program counter (PC). Usually the PC is in the current frame, but by using the `frame` command you can change the current frame, thus also changing the current scope.
- **Scope path** — A scope path is a complete path to the declaration of a program identifier. Scope paths enable you to reference program identifiers that are not currently visible, such as static variables, shadowed variables, and global variables. Scope paths also enable you to reference variables in other namespaces, such as loader symbols or debugger variables.

A scope path can be used to access several different namespaces. Namespaces are the storage allocations given to the different types of language identifiers CXdb can recognize. The possible namespaces are:

- Debugger variables (Scope path is `cxdb$` or `$`.)
- Loader symbols (Scope path is `l$`, or `asm$`.)
- FORTRAN (Scope path is explained below.)
- C (Scope path is explained below.)

In FORTRAN, scope paths enable you to reference common block identifiers from any point in the program. This allows you to view the data of a common block from different perspectives.

In C, scope paths enable you to reference static identifiers and identifiers that are shadowed. An identifier is shadowed when another identifier with the same name is declared in an inner block of the same scope.

Because the scope rules are different for FORTRAN and C, so are their scope paths.

In FORTRAN:

`f$`*<routine-name>* **`'`** *<identifier>*

- *<routine-name>* — The routine in which the identifier is declared. If the routine name is omitted, the current scope is used to find the identifier.

In C:

c$*<file-name>*'*<routine-name>*'*<block-list>*'*<identifier>*

- *<file-name>* — The name of the file in which the identifier is declared. If this is omitted, the current source file is used.
- *<routine-name>* — The name of the routine in which the identifier is declared. If this is omitted, the current routine is used or, if a *file-name* has been specified, the global declarations of that file.
- *<block-list>* — A list of block numbers specifying the block in which the identifier is declared. A block is anything inside of braces. Unless you have compiled your program using the `-pcc` option, block numbers are not assigned to blocks in which variables are not declared. There can be one or more blocks in a block list, corresponding to the nesting of the blocks.

The first block, inside the routine, is block 1. Subsequent blocks have incremental block numbers. Each new level of blocks begins over at 1. The following C pseudocode uses identifiers named block to help demonstrate block numbers.

```
            main()
 |----------{
 |              static float block = 0;
 |       |------{
 |       |          static float block = 1;
 |       |    |-----{
 |       |    1         static float block = 1.1;
 |       1    |-----}
 |       |    |-----{
 |       |    2         static float block = 1.2;
 |       |    |-----}
main     |------}
 |               {    /* block ignored unless -pcc option used */
 |               }
 |       |------{
 |       2          static float block = 2;
 |       |------}
 |       |------{
 |       3          static float block = 3;
 |       |------}
 |               execution_stopped_here();
 |----------}
```

Assuming execution was stopped at the end of the routine, the following scope paths print the different identifiers named block:

```
print block - 0
print '1'block - 1
print '1'1'block - 1.1
print '1'2'block - 1.2
print '2'block - 2
print '3'block - 3
```

The current scope from the end of the routine is the main routine. Thus, no scope path is needed to reach the identifier with a value of zero.

By starting a scope path with a backquote, you are asking CXdb to find the identifier within the current routine. In the above commands, the current scope was the main routine. Thus, all of the scope paths that start with a backquote begin with an implied `c$prog'main`.

Examples

The following examples demonstrate the use of scope paths; first for FORTRAN and then for C.

Consider the following FORTRAN program.

```
      PROGRAM PROG
      INTEGER I,A,B
      COMMON /B1/ A,B

      DO I=1,10
         A=1111
         B=8888
         CALL SUB1(I)
      ENDDO
      END

      SUBROUTINE SUB1(I)
      INTEGER I
      INTEGER C,D
      COMMON /B1/ C,D

      C=.2222
      D=.9999
      END
```

Assume that program execution has been stopped at the call to the function `SUB1` after five iterations of the `DO` loop. The current scope is based from frame 0, the current point of execution.

```
(CXdb) info scope
Process [#0/0], frame 0 scope: f$PROG
(CXdb) print A
INTEGER*4 1111
(CXdb) print B
INTEGER*4 8888
(CXdb) print SUB1'I
Variable has not been assigned storage: line: 1 col: 7
(CXdb) print SUB1'C
INTEGER*4 1111
(CXdb) print SUB1'D
INTEGER*4 8888
```

The above example prints out the values of the identifiers. Because `C` and `D` are in the same `COMMON` block as `A` and `B`, they really represent the same identifier location. Thus, they can be referenced in the subroutine because they share the same storage as `A` and `B`. The parameter to the subroutine cannot be referenced because it is only assigned storage when the subroutine is executing.

For the next series of examples, assume that process execution is stopped just before the end of `SUB1`.

```
(CXdb) info scope
Process [#0/0], frame 0 scope: f$SUB1
(CXdb) print C
INTEGER*4 2222
(CXdb) print D
INTEGER*4 9999
(CXdb) print PROG'A
INTEGER*4 2222
(CXdb) print PROG'B
INTEGER84 9999
```

The above examples print out the values of the various identifiers. Again, because `A` and `B` represent the same memory storage as `C` and `D`, they can be referenced from the subroutine.

The next series of examples use the following C program, made up of two source files. The first source file is `prog.c`:

```
#include <stdio.h>
extern void sub1();
int x = 0;

main()
{
   int i ;
   int x = 1111;

   for (i=1; i<=20; i++)
   {
       int x = 2222 ;
       if (i < 10)
           if (i < 6)
           {
              int x = 3333 ;
           }
           else
           {
              int x = 4444 ;
              sub1() ;
           }
   }
   printf("Finished\n");
}
```

The second source file is `sub.c`:

```
extern int x;

void sub1()
{
   static int x = 5555 ;
}
```

CXdb scope paths allow you to access the different identifiers all named `a`. Assume that execution has stopped just before the call to the function `sub1`.

```
(CXdb) info scope
Process [#0/0], frame 0 scope: c$prog'prog'main'1'2
(CXdb) print x
int 4444
(CXdb) print prog'x
int 0
(CXdb) print prog'main'x
int 1111
(CXdb) print prog'main'1'x
int 2222
(CXdb) print prog'main'1'1'x
int 3333
```

The above examples reference the different identifiers named `x`. The current scope is found using the `info scope` command.

The current scope is shown to be the block in which `x` is set to 4444. The first command does not use a scope path, so `x` is found in the current scope.

Now assume that execution has continued to just before the `printf` statement at the end of the program.

```
(CXdb) info scope
Process [#0/0], frame 0 scope: c$prog'prog'main
(CXdb) print x
int 1111
(CXdb) print c$prog'main'1'2'x
int 4444
(CXdb) print sub'x
int 0
(CXdb) print sub'sub1'x
int 5555
```

The above example again print the values of the identifiers named `x`. The current scope is the main routine. The global identifier `x` is referenced in the file `sub.c`. The local identifier of the subroutine `sub1`, located in `sub.c`, can be referenced because it was declared as a static identifier.

Related Commands

`evaluate`	`frame`
`info scope`	`print`

Related Concepts

source units

search path

Description

The search path is a list of directories CXdb searches when it is looking for either source files or the compiler-generated CDI data files. Initially, the search path for each process object is set to the default search path. If an executable is specified for the process object, CXdb adds the directories stored in the executable to the search path. The executable has absolute paths to the directories where the source code is located.

NOTE: If you have compiled your source code using a version of the CONVEX FORTRAN compiler later than V7.0 or the CONVEX C compiler later than V4.3, you may not need to specify a search path. If the source code has not changed location since compilation time, CXdb can find the source and CDI files from the information in the executable. If the source code has changed location, or if different source files or compiler-generated data files are desired, you may need to update the search path.

The following commands manipulate the search path of a process object:

- `add path` — Adds directories to the search path.
- `info path` — Displays the directories in the search path.
- `remove path` — Remove directories from the search path.
- `set path` — Sets the search path to the specified directories.

Examples

The following examples use the search path commands.

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

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

The above example displays the default search path and the search path of the process object. Currently the search path consists of the current working directory, represented by a dot (.).

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

The above example adds two directories to the search path. The next time CXdb is searching for a source file it will use the new search path, which now includes the `/mnt/jones/libraries` and `/mnt/jones/math/libraries` directories.

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

The above command removes the `/mnt/jones/math/libraries` directory from the search path. The other directories in the search path remain.

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

Process [#0] search list for: a.out
        .
        /mnt/jones/libraries
```

The above command displays the updated search path, after the `add path` and `remove path` commands.

```
(CXdb) set path /mnt/jones/project2 , /mnt/jones/project2/source
(CXdb) info path
Default search list:
        .

Process [#0] search list for: a.out
        /mnt/jones/project2
        /mnt/jones/project2/source
```

The above command removes all the existing directories from the search path and sets it to the two listed directories. The `info path` command displays the new search path for the process object.

Related Commands

`add default path`	`add path`
`attach`	`cd`
`core`	`cxdb`
`debug core`	`debug exec`
`debug proc`	`executable`
`info cxdb`	`info path`
`remove default path`	`remove path`
`set default path`	`set path`

Related Concepts

command files	Compiler-Debugger Interface
console working directory	default search path
process object	process working directory
remote debugging	

Related Parameters

directory-specifier

signals

Description

Signals sent to an executing process can be controlled through several CXdb commands. Eventpoints can be set to watch for the receipt of a particular signal. The actions to be taken when CXdb catches a signal can be set for each different signal. The signal received by a process can be specified.

CXdb catches all signals sent to the process before the process ever receives them, unless the signal was sent by CXdb itself. When CXdb catches a signal, the signal number for that signal is placed in the debugger variable `$signal`.

If an eventpoint is triggered by a signal, the commands of the eventpoint handler for that eventpoint are executed. If an eventpoint is not triggered, the actions CXdb takes after a signal is caught are as follows.

- **Stop** —Stop process execution. If stop is not set, process execution is not stopped.
- **Print** — Print a message telling what signal was caught. If print is not set, no message is printed.
- **Pass** — Pass the value of the pre-defined debugger variable `$signal` to the process when process execution resumes. By modifying the value stored in `$signal`, you can change what signal is passed to the process. If pass is not set, no signal is passed to the process.

The above actions are independent of one another, and can be set or unset for each signal. The signals are briefly described below. The number in parentheses is the signal number for that signal.

- `SIGHUP(1)` — Hangup
- `SIGINT(2)` — Interrupt
- `SIGQUIT(3)` — Quit
- `SIGILL(4)` — Illegal instruction
- `SIGTRAP(5)` — Trace/breakpoint trap
- `SIGIOT(6)` — IOT trap
- `SIGEMT(7)` — EMT trap
- `SIGFPE(8)` — Floating point exception
- `SIGKILL(9)` — Killed

- `SIGBUS(10)` — Bus error
- `SIGSEGV(11)` — Segmentation violation
- `SIGSYS(12)` — Bad system call
- `SIGPIPE(13)` — Broken pipe
- `SIGALRM(14)` — Alarm clock
- `SIGTERM(15)` — Terminated
- `SIGURG(16)` — Urgent I/O condition
- `SIGTSTP(17)` — Stopped (terminal)
- `SIGSTOP(18)` — Stopped
- `SIGCONT(19)` — Continued
- `SIGCHLD(20)` — Child exited
- `SIGTTIN(21)` — Stopped (tty input)
- `SIGTTOU(22)` — Stopped (tty output)
- `SIGIO(23)` — I/O possible
- `SIGXCPU(24)` — CPU time limit exceeded
- `SIGXFSZ(25)` — File size limit exceeded
- `SIGVTALRM(26)` — Virtual timer expired
- `SIGPROF(27)` — Profiling timer expired
- `SIGWINCH(28)` — Window size change
- `SIGLOST(29)` — Resource lost
- `SIGUSR1(30)` — User-defined signal 1
- `SIGUSR2(31)` — User-defined signal 2

Signal 0 is used to indicate that no signal should be sent to the process.

Signal names are not case sensitive.

The following commands work with signals:

- `info signal` — Displays the current actions for a signal or all signals.
- `event signal` — Sets an eventpoint for a signal.
- `set signal` — Sets the actions for a signal.
- `signal process` — Sends a signal to the specified process.
- `signal thread` — Sends a signal to the specified thread.

Examples

The following commands control the effect of the `SIGINT` signal on the process. For these examples, assume that the process is currently stopped.

```
(CXdb) info signal SIGINT

The current signal actions are:

Signal                        Signal
number   Stop   Pass   Print   name
------   ----   ----   -----   ----------------------
2         Yes     No    Yes    Interrupt
```

The above command displays the current settings for the actions to take when CXdb catches the `SIGINT` signal. Initially CXdb will stop the process, not pass the signal, and print a message when the signal is caught.

```
(CXdb) event signal SIGINT {eval $signal = 0; resume;}

#1: signal 2 on [#0], Enabled, ignore 0/0
  {
      eval $signal=0;
      resume;
  }
```

The above command sets an eventpoint watching for the `SIGINT` signal. The eventpoint handler sets the debugger variable `$signal` to zero and then resumes execution. Because the value of `$signal` is zero, when execution resumes the signal is not sent to the process. In effect, this handler causes the `SIGINT` signal to be ignored.

```
(CXdb) set signal INT nostop print nopass
```

The above command changes the default settings for the `SIGINT` signal. When CXdb catches the signal, a message is printed saying the signal has been caught. Because the stop action is not set, process execution continues. However, the signal is not passed to the process. This has the same effect as the previously set eventpoint; the `SIGINT` signal is ignored.

```
(CXdb) signal process 2
Resuming execution of Process [#0] with signal 2
```

The above command causes process execution to continue with the `SIGINT` signal (signal number 2) immediately being sent to the process. Because this signal is generated by CXdb, the signal is not caught by CXdb.

Related Commands

`info signal`
`set signal`
`signal thread`
`event signal`
`signal process`

Related Parameters

process-list
thread-list
signal-specifier

source units

Description

Source units are syntactic units of source code. There are five granularities of source units:

- Expression — Any valid combination of constants, operators, and operands.
- Statement — A combination of expressions that constitutes a complete instruction in the source language.
- Block — The statements that make up the body of a routine, a loop, or a conditional construct.
- Loop — A special type of statement that encloses a block.
- Routine — A subroutine or function.

Source units let you control the granularity used in analyzing your source code. You can specify source units for stepping as well as for setting breakpoints, eventpoints, and tracepoints. Source units are essential for debugging optimized code, because optimized code often does not execute in the same sequence as the original source statements.

The compiler assigns an identification number to each source unit when you compile your program with the `-cxdb` option. These source unit numbers are unique within a single source file. To list the identification numbers of all source units on a given line of source code, use the `info line` command.

Examples

The following two examples present the same section of source code written in both FORTRAN and C. This will help you compare source units in the two languages.

First consider the following section of FORTRAN source code:

```
1       SUBROUTINE PRIME(N)
2       INTEGER A(1000)
3
4       DO J = 2, N
5         IF (A(J) .EQ. 1)
6           K = J*2
7           DO WHILE ((K .LE. N).AND.(K .LE. 1000))
8             A(K) = 0
9             K = K + J
10          ENDDO
11        ENDIF
12      ENDDO
13      RETURN
14      END
```

In the above code, there is only one routine source unit, and `PRIME` is the name of the routine represented by this source unit. This source unit consists of everything from the `SUBROUTINE` statement on line 1 up to and including the `END` statement on line 14.

There are two loop source units in the above example: a `DO` loop and a `DO WHILE` loop. They consist of the following:

- `DO` loop — Lines 4 through 12, inclusive.
- `DO WHILE` loop — Lines 7 through 10, inclusive.

There are four block source units in the above example. The first is a routine block, the second is a `DO` block, the third is an `IF` block, and the fourth is a `DO WHILE` block. These blocks consist of the following statements:

- Routine block — Lines 2 through 14, inclusive.
- `DO` block — Lines 5 through 11, inclusive.
- `IF` block — Lines 6 through 10, inclusive.
- `DO WHILE` block — Lines 8 and 9.

The statement source units in the above example are:

- Line 4 — `DO J = 2, N`
- Line 4 — `J = 2`
- Line 5 — `IF (A(J) .EQ. 1) THEN`
- Line 6 — `K = J*2`
- Line 7 — `DO WHILE((K .LE. N).AND.(K .LE. 1000))`
- Line 8 — `A(K) = 0`
- Line 9 — `K = K + J`
- Line 13 — `RETURN`
- Line 14 — `END`

Notice that a statement must contain some executable code in order to be counted as a source unit. Thus, statements such as `SUBROUTINE`, `INTEGER`, `ENDIF`, and `ENDDO` are not considered to be source units. Neither are comment lines or blank lines.

Also notice that the `DO` and `DO WHILE` statements are both statement source units and loop source units at the same time.

The expression source units in the above example are:

- Line 4:
  ```
  N
  2
  ```
- Line 5:
  ```
  A(J) .EQ. 1
  A(J)
  J
  1
  ```
- Line 6:
  ```
  J*2
  J
  2
  ```

- Line 7:

```
(K .LE. N).and.(K .LE. 1000)
(K .LE. 1000)
K .LE. 1000
(K .LE. N)
K .LE. N
1000
K (first occurrence)
N
K (second occurrence)
```

- Line 8:

```
K
0
```

- Line 9:

```
K + J
K
J
```

An expression can be as small as a single variable or as large as a complete statement. Notice that different appearances of the same expression count as different source units. For example, the variable `K` appears twice in line 7 above, so it counts as two different expression source units for that line. As with the other types of source units, the source language determines what constitutes a valid expression.

Consider the same section of code written in C:

```
1       prime(n)
2       int n;
3       {
4         int j,k;
5
6         for (j=2; j<=n; j++)
7         {
8           if (a[j]==1)
9           {
10            k = j*2;
11            while ((k<=n) && (k<=1000))
12            {
13              a[k] = 0;
14              k = k+j;
15            }
16          }
17        }
18      }
```

In the above example, there is one routine source unit. It consists of lines 3 through 18, inclusive.

There are two loop source units in the above example: a `for` loop and a `while` loop. They consist of the following:

- `for` loop — Lines 6 through 17, inclusive.
- `while` loop — Lines 11 through 15, inclusive.

There are four block source units in the above example. The first is a routine block, the second is a `for` block, the third is an `if` block, and the fourth is a `while` block. These blocks consist of the following statements:

- Routine block — Lines 3 through 18, inclusive.
- `for` block — Lines 7 through 17, inclusive.
- `if` block — Lines 9 through 16, inclusive.
- `while` block — Lines 12 and 15.

The statement source units in the above example are:

- Line 6 — `for (j=2; j<=n; j++)`
- Line 8 — `if (a[j]==1)`
- Line 10 — `k = j*2`
- Line 11 — `while ((k<=n) && (k<=1000))`
- Line 13 — `a[k] = 0`
- Line 14 — `k = k+j`

The expression source units in the above example are:

- Line 6:
  `j<=n`
  `j++`
  `j=2`
  `j` (first occurrence)
  `2`
  `j` (second occurrence)
  `n`
  `j` (third occurrence)
- Line 8:
  `a[j]==1`
  `a[j]`
  `a`
  `j`
  `1`
- Line 10:
  `k = j*2`
  `j*2`
  `k`
  `2`
  `j`
- Line 11:
  `(k<=n) && (k<=1000)`
  `(k<=1000)`
  `k<=1000`
  `(k<=n)`
  `k<=n`
  `1000`
  `k` (first occurrence)
  `n`
  `k` (second occurrence)

- Line 13:
  ```
  a[k] = 0
  a[k]
  a
  k
  0
  ```
- Line 14:
  ```
  k = k+j
  k+j
  ```
  k (first occurrence)
  k (second occurrence)
  ```
  j
  ```

Related Commands

`break source`	`clear step`
`event reached source`	`info line`
`info sourceunit`	`next`
`next over`	`set default step`
`set step`	`step`
`step over`	`trace source`

Related Concepts

breakpoints	eventpoints
stepping	tracepoints
windows	

Related Parameters

granularity	source-unit

stepping

Description

Stepping is the incremental execution of a program. CXdb provides a sophisticated set of stepping commands that allow you to control not only the number of steps to execute but also the size of each step.

In the broadest sense, stepping can be done in one of two ways: by machine instructions or by source units. The commands for stepping by machine instruction are:

- `next instruction` — Execute the specified number of machine instructions. Do not count instructions within a called routine as part of the specified number.
- `step instruction` — Execute the specified number of machine instructions. Count instructions within a called routine as part of the specified number.

The commands for stepping by source units allow you to specify the source unit granularity, or step size, as well as the number of steps. These commands are:

- `next` — Continue executing the process until it reaches the next source unit of the specified granularity. Do not count the source units within a called routine.
- `step` — Continue executing the process until it reaches the next source unit of the specified granularity. Count the source units within a called routine.
- `finish` — Finish executing the innermost active source unit of the specified granularity. Stop execution at the next source unit of default granularity.
- `next over` — Complete execution of the current source unit of the specified granularity. Stop execution at the next source unit of default granularity. Do not count the source units within a called routine.
- `step over` — Complete execution of the current source unit of the specified granularity. Stop execution at the next source unit of default granularity. Count the source units within a called routine.

The source unit granularities are:

- Routine
- Loop
- Block
- Statement
- Expression

If you do not specify the granularity for a particular stepping command, CXdb uses the default granularity. Initially the default granularity is statement, but you can modify this with the following commands:

- `set default step` — Set the default granularity (or step size) for all new process objects.
- `set step` — Set the default granularity (or step size) for an existing process object.

As a general rule, you will probably want to use a larger stepping granularity when the point of execution is far away from a problem area of the program and a finer granularity as the point of execution approaches the problem area.

The current source unit starts at the address indicated by the current value of the program counter (PC). Several source units of different granularities might all start at the same location. Therefore, all of these source units can be current at the same time. That is why it is important for you to specify the granularity of the particular current source unit you want.

The innermost active source unit is the one of specified granularity whose address range (or extent) includes the current value of the PC. If you start at the location indicated by the current PC and look backward through the code, the innermost active source unit is the first one of the specified granularity that you encounter. In other words, it is the innermost source unit of specified granularity that encloses or contains the location indicated by the current PC.

Some examples might help to clarify the difference between the current source unit and the innermost active one. Consider the following pseudocode:

```
start Loop 1
    Statement 5
    start Loop 2
        Statement 10
        Statement 11
    end Loop 2
    Statement 15
end Loop 1
```

If the PC is pointing to Statement 11 in the above pseudocode, then the innermost active loop is Loop 2, and the innermost active block is the block that includes Statement 10 and Statement 11. However, the current source unit is Statement 11 because the PC is pointing to it. Even thought Loop 2 is active, it is not current because the PC does not point directly at the beginning of Loop 2.

With the PC pointing at Statement 11, Loop 2 and its included block are considered active because they contain the location indicated by the current value of the PC. Loop 1 is also active, but it is not the *innermost* active loop in this case.

If the PC is pointing to Statement 5 in the above pseudocode, then Loop 1 is the innermost active loop and Statement 5 is the current source unit. If the PC is pointing directly at the start of Loop 2, then Loop 2 is both the current loop source unit and the innermost active loop source unit.

One final and most important point: stepping is a dynamic activity. It proceeds according to the order of execution of the object code, not according to the order of the source code. When CXdb is searching for the particular source unit you want to step to, it checks each source unit individually before executing it. Therefore, if conditional constructs in the source code cause execution to skip over the source unit you want, or if optimization has eliminated the desired source unit, then stepping cannot take you to that source unit.

Examples

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

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

Assume that the default stepping granularity is statement. Also assume that the process is stopped, and the program counter (PC) is pointing to the beginning of line 2.

Enter the following command:

```
(CXdb) step
Stepping process [#0/*] by 1 statement
Process [#0/0] stopped stepping at [0x80001368] EXAMPLE in example.f line 3
```

Because statement is the default granularity, the above command steps the current process by one statement. The PC now points to the beginning of line 3, which is the beginning of a `DO` loop.

```
(CXdb) step 2
Stepping process [#0/*] by 2 statements
Process [#0/0] stopped stepping at [0x800013a2] EXAMPLE in example.f line 5
```

The above command steps the current process by two statements. The PC points to the beginning of line 5, which is a call to a subroutine.

```
(CXdb) next
Nexting process [#0/*] by 1 statement
Process [#0/0] stopped stepping at [0x80001372] EXAMPLE in example.f line 4
```

The above command again steps the process by one statement. However, before the command executed, the PC was at the beginning of line 5, which is a subroutine call. The `next` command ignores all source units in a called subroutine. Therefore, the process continues to execute until it reaches the next statement after returning from the subroutine. When the process stops, the PC is pointing to the beginning of line 4.

In the above example, note that the process does not stop again at the `DO` statement on line 3 because the `DO` loop is already active. This is just another iteration of the loop, so there is no new statement to be executed at line 3. Therefore, the process does not stop in this case until it reaches line 4.

At this point, you enter the following command:

```
(CXdb) step loop 2
Stepping process [#0/*] by 2 loops
Process [#0/0] stopped stepping at [0x800014c2] SUBA in example.f line 17
```

The above command steps the process by two loops. Because the step command does look at source units in called routines, this command continues execution of the process until it reaches the second `DO` loop in subroutine `SUBA`. When the process stops, the PC is pointing to the beginning of line 17.

```
(CXdb) step over loop
Stepping process [#0/*] by 1 loop
Process [#0/0] stopped stepping at [0x80001540] SUBA in example.f line 20
```

The above command completes execution of the current loop that begins on line 17. Since the default granularity is statement, execution stops at the next statement after the loop. The PC now points to the beginning of line 20.

```
(CXdb) finish loop
Finishing innermost loop in Process [#0/*]
Process [#0/0] stopped stepping at [0x8000166c] SUBA in example.f line 29
```

The above command completes execution of the innermost active loop. When the PC is pointing to line 20, the innermost active loop is the `DO` loop that begins on line 14. The above command completes execution of this entire loop and stops the process at the first default source unit (statement) after the loop. Therefore, when the process stops, the PC is points to the beginning of line 29.

Related Commands

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

Related Concepts

process object
source units

Related Parameters

granularity

synthesized variables

Description

A synthesized variable is a variable generated by the CONVEX FORTRAN or CONVEX C compiler at optimization level `-O1` or higher. Synthesized variables enhance the performance of a program in two major ways:

- By replacing a program variable with a more efficient construct. For example, a synthesized variable can be used as a pointer to a particular array element. This pointer can replace a loop induction variable that acts as an index to an array element.
- By providing runtime support for the program. For example, synthesized variables can be used to maintain register spill areas in memory.

To generate a synthesized variable, the compiler performs transformations based on mathematical equations. CXdb can solve these equations to determine the current value of the synthesized variable as well as the current value of the program variable that is replaced by the synthesized variable. The `info expression` command displays the equations used to derive the synthesized variables.

The `info expression` command also lists the reason for the use of each synthesized variable. The reasons are abbreviated to acronyms, which are defined as follows:

- `ALTE` — Alternate entry point of a loop that executes conditionally.
- `BITB` — Bit bucket storage.
- `BOOT` — Storage of a value removed from a scalar register.
- `BSS` — Starting address of BSS memory region.
- `CMIN` — Index used to find the minimum element of an array by pattern matching.
- `CREG` — Communication register storage.
- `CTMP` — Call temporary, used for temporary storage of arguments that are passed by value to a subroutine.
- `DATA` — Starting address of DATA memory region.
- `DEAD` — Loop induction variable whose current value is impossible to calculate. No synthesized variable is reported in this case.

- DEXP — Expression that has been distributed over several optimized loops.
- FLNK — Forward link to a constant value that is propagated to the distributends of a vectorized loop.
- INDV — Loop induction variable that has undergone strength reduction.
- ISTR — Inner strip counter of a strip-mined loop.
- MLXS — Subscript of a multidimensional array that has been transposed into a one-dimensional array during vectorization.
- OSTR — Outer strip counter of a strip-mined loop.
- PBKE — Loop-invariant expression.
- PBKU — Loop-invariant constant.
- REXP — Reduced subexpression that is hoisted out of a loop.
- SEXP — Subscript expansion that folds the subscripts of a multi-dimensional array into one subscript.
- SINK — Sink variable that replaces the original induction variable when the loop iteration count is too small to warrant strip mining.
- SPLL — Pointer to the spill area for scalar registers.
- STML — Strip mine length.
- TBSS — Starting address of TBSS memory region.
- TDATA — Starting address of TDATA memory region.
- TEXT — Starting address of TEXT memory region.
- TPTR — Temporary pointer, used for temporary storage of arguments that are passed by reference to a subroutine.
- TRIP — Trip count used to replace a more complex loop iteration quantity.
- UREX — Expression from an unrolled loop.
- URIV — Induction variable of an unrolled loop.
- UTRP — Trip counter of an unrolled loop.
- VBOT — Storage of a value removed from a vector register.
- VMSK — Vector mask storage.
- VSPL — Pointer to the spill area for vector registers.
- ZMSK — Zero mask storage.

You can use a synthesized variable in any *<language-expression>*, in the same way you would use a program variable. However, in most cases, you will only need to display the current value of the synthesized variable by using the `print` command.

Examples

The following examples illustrate how to display and reference synthesized variables.

```
(CXdb) info expression J
object type: Fortran identifier
   location: <none>
       size: 4 bytes
       type: INTEGER*4
      value: 4
       used to create 1 synthesized variable(s):
         1. <INDV>    ?i7 = ?i1+((4*N)*(J-1))
```

The above command displays information about the program variable `J`. The response indicates that the current value of `J` is 4. This value is not stored (location = <none>) because the synthesized variable `?i7` replaces the use of `J`. The reason for the replacement is `INDV`, which means the induction variable has undergone strength reduction. The equation used to generate the synthesized variable is `?i7=?i1+((4*N)*(J-1))`.

In the source code, `J` is a loop induction variable that is used as an index to reference specific elements of an array. In the object file, `?i7` serves as a pointer to the array elements. The compiler replaces `J` with `?i7` because it is more efficient to increment the pointer than it is to increment `J` and recalculate the address of the desired array element on each iteration of the loop.

For purposes of the `info expression` command, CXdb calculates the current value of `J` by solving for it in the equation shown for `?i7`.

```
(CXdb) info expression \?i7
object type: Fortran identifier
   location: register a2
       size: 4 bytes
       type: INTEGER*4
      value: -2147176320
       Reason: Loop induction variable
       created from 1 equation(s):
         1. <INDV>  ?i1+((4*N)*(J-1))
       2 liveness ranges:
              Start       End          Location
          1. 0x8000172e:0x80001750 - register a2
          2. 0x80001750:0x80001754 - register a2
```

The above command displays information about the synthesized variable `?i7`. The response shows the equation that the compiler uses to generate `?i7`. It also shows the liveness ranges and corresponding storage locations for the variable. The reason for generating `?i7` is that it replaces a loop induction variable.

```
(CXdb) print/x \?i7
INTEGER*4) 0x8004b080
```

The above command prints the current value of the synthesized variable `?i7` in hexadecimal format. Because `?i7` is a pointer to an array in this case, the current value of `?i7` is the starting address of the next array element to be accessed.

Related Commands

`evaluate` `info expression` `print`

Related Concepts

debugger variables language expressions

Related Parameters

language-expression synthesized-variable

tracepoints

Description

A tracepoint is a pre-defined eventpoint, or trap, that you place in your executable code. When process execution reaches the location of an enabled tracepoint, the set of actions associated with the tracepoint, called the eventpoint handler, are taken. Tracepoints allow you to trace the execution of your process past key locations in your program.

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

All tracepoints have a default handler that prints a message telling you that the tracepoint has been reached and then resumes process execution. You may specify a different set of actions to take for a particular tracepoint, or change the setting of the default handler itself.

Tracepoints, like all eventpoints, can be enabled or disabled. When process execution reaches the address of an enabled tracepoint, the tracepoint is said to be reached. A disabled tracepoint is treated as if it does not exist, and therefore can never be reached unless it is enabled again. The disabling of tracepoints allows you to prevent tracepoints being reached without having to completely remove them from the process object.

Once a tracepoint is reached, one of two actions can occur. If the tracepoint has an ignore count, the counter is incremented by one and process execution continues. If the tracepoint does not have an ignore count, then the eventpoint is said to be triggered. When a tracepoint is triggered, the commands in its eventpoint handler are executed. If the tracepoint does not have its own eventpoint handler, the default eventpoint handler for tracepoints is used.

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

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

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

There are several different ways to set a tracepoint. Tracepoint commands are described below:

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

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

Several commands allow you to interact with existing tracepoints. These commands are described below:

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

Examples

The following series of examples create and manipulate several different tracepoints in one process object.

```
(CXdb) trace line 60

#0: trace line, on [#0/*], Enabled, ignore 0/0
        [0x80001620] BESTMV in pickup.f line 60
```

The above command sets a tracepoint at line 60 in the current source file. The eventpoint number for this tracepoint is 0, and the address of the tracepoint is `80001620` in routine `BESTMV` at line 60 in the source file `pickup.f`.

```
(CXdb) trace routine BESTMV

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

The above command sets a tracepoint at the first executable source unit in the routine containing the address of the routine `BESTMV`. The routine `BESTMV` contains its own address, so the tracepoint is set at the first executable source unit of the routine `BESTMV`. The first executable source unit of a routine is usually the first statement of a routine after local variables have been declared unless there are local initializations.

```
(CXdb) trace instruction BESTMV

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

The above command sets a tracepoint at the first address of `BESTMV`. The first address of a routine begins the preamble of the routine. The preamble of a routine manages the stack for that routine. This address is different than that of the previous example, because tracepoint 1 is at the first source unit of `BESTMV`.

```
(CXdb) trace source 135

#3: trace source, on [#0/*], Enabled, ignore 0/0
        [0x800016f0] BESTMV in pickup.f line 65
```

The above command sets a tracepoint at the starting address of source unit 135. The source unit numbers of a given line may be found using the `info line` command.

```
(CXdb) info trace
Event   Enabled     Ignore     proc/td      Address       Where
#0        y          0/0        0/*      [0x80001620] BESTMV in pickup.f line 60
#1        y          0/0        0/*      [0x800015f2] BESTMV in pickup.f line 59
#2        y          0/0        0/*      [0x800015f0] BESTMV in pickup.f line 55
#3        y          0/0        0/*      [0x800016f0] BESTMV in pickup.f line 65
```

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

```
(CXdb) run
Beginning execution of Process [#0]
Process [#0/0] hit Tracepoint 2 at [0x800015f0] BESTMV in pickup.f line 55
Process [#0/0] hit Tracepoint 1 at [0x800015f2] BESTMV in pickup.f line 59
Process [#0/0] hit Tracepoint 0 at [0x80001620] BESTMV in pickup.f line 60
Process [#0/0] hit Tracepoint 3 at [0x800016f0] BESTMV in pickup.f line 65

Process [#0] exited normally.
```

The above example begins process execution without passing any arguments to the process. When execution reaches the address of `800015f0`, tracepoint 2 is reached because it is enabled. Because it does not have an ignore count, the tracepoint is triggered. Because the tracepoint did not have its own eventpoint handler, the default handler for tracepoints is executed, which prints a message and then resumes execution. Each of the tracepoints is triggered in turn. For this example, assume that the program runs to completion.

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

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

```
(CXdb) trace line 60 $Middle

#4: trace line, on [#0/*], Enabled, ignore 1/2
         [0x80001620] BESTMV in pickup.f line 60

INFO: Eventpoint 0 also has a tracepoint at address 0x80001620.
```

The above command sets another tracepoint at line 60 of the current source file. The debugger variable `$Middle` is created and assigned to this eventpoint. CXdb informs you that two eventpoints now reside at the same address location.

```
(CXdb) trace line 60 {echo "Tracepoint 5 reached" ;}

#5: trace line, on [#0/*], Disabled, ignore 0/0
         [0x80001620] BESTMV in pickup.f line 60
 {
       echo "Tracepoint 5 reached" ;
 }

INFO: Eventpoint 4 also has a tracepoint at address 0x80001620.
```

The above command sets a third tracepoint at line 60. An eventpoint handler is given to this eventpoint. The handler prints a message but does *not* resume execution of the process.

```
(CXdb) run
Beginning execution of process [#0]
Tracepoint 5 reached
```

The above example restarts execution of the process. Tracepoint 5 is triggered because it is the highest-numbered, enabled eventpoint at that location (the other two tracepoints at that location are 0 and 4), and it does not have an ignore count. The eventpoint handler for tracepoint 5 is used instead of the default handler for tracepoints. The handler for tracepoint 5 prints a message but does not resume process execution.

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

The above example does two things. First, tracepoint 5 is disabled. Second, tracepoint 4 is given an ignore count of 2.

```
(CXdb) run
Process [#0] is already running with pid 16139.
Terminate existing process and restart? y
Beginning execution of Process [#0]
Process [#0/0] hit Tracepoint 0 at [0x80001620] BESTMV in pickup.f line 60
```

In the above example, the `run` command causes CXdb to display a warning message indicating that the current process still exists. If you answer yes, CXdb kills the current process and begins execution of a new process. When process execution reaches address `80001620`, tracepoint 4 is reached because eventpoint 5 is disabled. Tracepoint 4 has an ignore count, so its counter is incremented by one. Because an eventpoint has still not been triggered, tracepoint 0 is reached. Because it is enabled and does not have an ignore count, it is triggered.

```
(CXdb) info event *
#0: trace line, on [#0/*], Enabled, ignore 0/0
        [0x80001620] BESTMV in pickup.f line 60
#3: trace source, on [#0/*], Enabled, ignore 0/0
        [0x800016f0] BESTMV in pickup.f line 65
#4: trace line, on [#0/*], Enabled, ignore 1/2
        [0x80001620] BESTMV in pickup.f line 60
#5: trace line, on [#0/*], Disabled, ignore 0/0
        [0x80001620] BESTMV in pickup.f line 60
 {
      echo "Tracepoint 5 reached" ;
 }
```

The above command displays the status of all eventpoints. Using the `info event` command, you can display the eventpoint handler of an eventpoint. From the output of this command you can also see that tracepoint 5 is disabled and that tracepoint 4 has been ignored once. (Tracepoints 1 and 2 were removed in a previous example.)

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

The above command enables tracepoint 5. This causes tracepoint 5 to be triggered the next time address `80001620` is reached because it is the highest-numbered, enabled tracepoint.

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

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

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

Related Commands

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

Related Concepts

breakpoints
eventpoints
eventpoint handlers
watchpoints

Related Parameters

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

viewports

Description

Viewports are destinations for CXdb input, output, and error messages. A viewport may be either the CXdb command window or a file.

There are three types of viewports, each of which is capable of receiving a different type of information. The types are:

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

CXdb maintains separate viewport lists for cmderr, cmdlog, and cmdout. Each viewport on a list receives a copy of the designated information for its given type. For example, all the viewports for cmdout receive a copy of the output simultaneously.

The commands for modifying the viewport lists are:

- `add cmderr` — Add new viewports to the current list of cmderr viewports.
- `add cmdlog` — Add new viewports to the current list of cmdlog viewports.
- `add cmdout` — Add new viewports to the current list of cmdout viewports.
- `remove cmderr` — Remove viewports from the current list of cmderr viewports.
- `remove cmdlog` — Remove viewports from the current list of cmdlog viewports.
- `remove cmdout` — Remove viewports from the current list of cmdout viewports.
- `set cmderr` — Remove the current cmderr viewports and replace them with a new list of viewports.
- `set cmdlog` — Remove the current cmdlog viewports and replace them with a new list of viewports.
- `set cmdout` — Remove the current cmdout viewports and replace them with a new list of viewports.

The default viewport for cmderr and cmdout is the CXdb command window (Window #1). There is no default viewport for cmdlog because your entries in the command window are automatically echoed there.

For logging, most of the viewports you specify will be files. If the specified file does not exist, CXdb creates it. If the specified file already exists, then you can use the following commands to control whether or not CXdb writes (either overwrites or appends) to the existing file:

- `clear noclobber` — Allow writing (either overwriting or appending) to existing files.
- `set noclobber` — Respond with an error message if the specified file already exists.

The default for noclobber is clear (off).

For cmderr and cmdout, you can override the viewport list and redirect the response of an individual command by using redirection operators with that command. Each redirection operator allows you to specify a viewport list that applies only to the command with which it appears.

For cmdlog, you can enable and disable the viewports with the commands `set logging` and `clear logging`, respectively. The default is logging disabled (clear).

The command `info cxdb` displays the current settings of cmderr, cmdlog, cmdout, log, and noclobber.

Examples

The following examples illustrate how to modify and use viewport lists.

```
(CXdb) add cmdout cxdb.info
New cmdout: Window #1, cxdb.info
```

The above command adds a viewport to the list for cmdout. The response indicates that the new viewports for cmdout are Window #1 (the command window) and the file `cxdb.info`.

```
(CXdb) add cmderr cxdb.info, cxdb.err
New cmderr: Window #1, cxdb.info, cxdb.err
```

The above command adds two new viewports to the list for cmderr. The response indicates that the new viewports for cmderr are Window #1 (the command window), the file `cxdb.info`, and the file `cxdb.err`.

```
(CXdb) remove cmderr cxdb.info
New cmderr: Window #1, cxdb.err
```

The above command removes a viewport from the list for cmderr. The response indicates that the new viewports for cmderr are Window #1 (the command window) and the file `cxdb.err`.

```
(CXdb) set cmdout output_data
```

The above command deletes the current viewport list for cmdout and replaces it with a new list that contains the file name `output_data`. This command establishes the file `output_data` as the only viewport for cmdout. Notice that the command window is no longer one of the cmdout viewports, so responses to CXdb commands will not appear in the command window.

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

The above command adds Window #1 (the command window) to the viewport list for cmdout. The response indicates that the new viewports for cmdout are the file `output_data` and Window #1.

With each individual CXdb command, you can override the viewport lists by using redirection operators on the command line. For example, to redirect the output of the `info process` command, enter the following:

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

The above command line redirects the output of the `info process` command to the file called `process_status` instead of to the viewports for cmdout. However, the viewports for cmderr and cmdlog are not affected by the redirection in this case.

To keep a log of the commands you use during the debugging session, enter the following:

```
(CXdb) add cmdlog debug_script
New cmdlog: debug_script
(CXdb) set logging
```

The above response indicates that the new viewport for cmdlog is the file `debug_script`. The `set logging` command enables logging to this file. This type of log file is a good way to keep track of your actions during a debugging session. If you ever need to retrace your steps, you can use the `source` command to execute this log file as if it were a command file.

Notice that the above response does not list Window #1 (the command window) as one of the viewports for cmdlog. This is because everything you type in the command window is automatically echoed there. If you add Window #1 to cmdlog, then everything you type will appear twice in the command window.

Related Commands

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

Related Concepts

cmderr
cmdlog
cmdout
viewports
windows

Related Parameters

redirection-operator
viewport

watchpoints

Description

A watchpoint is a predefined eventpoint, or trap, that you set to watch a region of process memory. When the value stored in the watched address region changes, process execution stops, and the set of actions associated with the watchpoint, called the eventpoint handler, are taken. Watchpoints enable you to watch for a specific address, or address range, to change value.

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

All watchpoints have a default handler that displays a message telling you that the watchpoint's region has been modified. You may specify a different set of actions to take for a particular watchpoint, or change the setting of the default handler itself.

Watchpoints, like all eventpoints, can be enabled or disabled. When there is a change in the address region watched by an enabled watchpoint, the watchpoint is said to be reached. A disabled watchpoint is treated as if it does not exist, and therefore can never be reached unless it is enabled again. The disabling of watchpoints allows you to prevent them being reached without having to completely remove them from the process object.

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

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

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

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

The `watch` command creates all watchpoints. A process image must exist before you can create a watchpoint.

Several commands allow you to interact with existing watchpoints. These commands are described below:

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

Examples

The following series of examples create and manipulate several different watchpoints in one process object. The following FORTRAN program is used throughout the examples:

```
PROGRAM TEST
INTEGER A,B,C

A = 0
B = 0
C = 0
DO I=1,1000
    A=A+1
    DO J=1,1000
        B=B+1
        DO K= 1,1000
            C=C+1
        ENDDO
    ENDDO
ENDDO
END
```

For the following examples, assume that process execution has stopped at the beginning of the program.

```
(CXdb) watch loc(C)

#0: watch 0x80029010..0x80029013, on [#0/0], Enabled, ignore 0/0
```

The above command creates a watchpoint that monitors any changes to the FORTRAN variable `C`. The response from CXdb shows the current settings for the created watchpoint. The output is the same as if an `info event` command had been issued for this eventpoint.

Because the variable is a four-byte integer, the address range is from `80029010` to `80029013`. The eventpoint number is 0, it is currently enabled, and it does not have an ignore count.

```
(CXdb) info expression B

object type: Fortran identifier
   location: 0x8002900c
       size: 4 bytes
       type: INTEGER*4
      value: 0

(CXdb) watch '8002900c'x .. '8002900f'x

#1: watch 0x8002900c..0x8002900f, on [#0/0], Enabled, ignore 0/0
```

The above example shows another way to set a watchpoint. First the address of the variable `B` is obtained using the `info expression` command. Second, a watchpoint is created to watch the address range starting with the hexadecimal address of `8002900c` and ending with `8002900f`. The syntax for specifying a hexadecimal number is FORTRAN-specific.

```
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] memory region 0x8002900c..0x8002900f modified
Process [#0/0] stopped by Watchpoint 1, at [0x80001392] TEST in test.f line 11
```

The above example resumes execution of the current process. The process is stopped when the address region watched by watchpoint 1 changes. The default handler for watchpoints, which displays the above messages, is executed. Process execution does not resume.

```
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] memory region 0x80029010..0x80029013 modified
Process [#0/0] stopped by Watchpoint 0, at [0x8000139c] TEST in test.f line 12
(CXdb) print C
(INTEGER*4) 1
(CXdb) set ignore 5 0
Event 0 will be ignored 5 times
```

The above example does several things. First, process execution is continued. The process is stopped by the watchpoint 0 because the contents of the variable `C` have changed. Second, the current value of `C` is printed. Finally, watchpoint 0 is given an ignore count of 5. The next 5 times the watchpoint is reached, it will not be triggered. When it is reached a sixth time, it is triggered.

```
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] memory region 0x80029010..0x80029013 modified
Process [#0/0] stopped by Watchpoint 0, at [0x8000139c] TEST in test.f line 12
(CXdb) print C
(INTEGER*4) 6
(CXdb) disable event 0
Event 0 disabled
```

The above example resumes process execution. Watchpoint 0 eventually stops the process. The `print` command shows that the watchpoint was ignored 5 times. Finally, watchpoint 0 is disabled. The watchpoint can no longer be reached.

```
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] memory region 0x8002900c..0x8002900f modified
Process [#0/0] stopped by Watchpoint 1, at [0x80001392] TEST in test.f line 11
```

The above command resumes process execution. Because watchpoint 0 is disabled, the process is stopped by watchpoint 1 when the value of the variable `B` changes.

```
(CXdb) info watch
Event     Enabled Ignore  proc/td         Region
#0          y     0/0     0/0     0x80029010        0x80029013
#1          n     0/0     0/0     0x8002900c        0x8002900
      f
```

The above command displays information about all existing watchpoints. The eventpoint number, enabled status, ignore count, process and thread number, and region of memory being watched is displayed for each watchpoint.

```
(CXdb) enable event 0
Event 0 enabled
(CXdb) remove event 1
Event 1 removed
```

The above example performs two actions. First, watchpoint 0 is enabled again, so it can be reached. Second, watchpoint 1 is completely removed from the process. It can not be brought back.

```
(CXdb) watch '80029010'x :4 \; {print C; disable event $self; resume;}

#2: watch 0x80029010..0x80029013, on [#0/0], Enabled, ignore 0/0
    {
       print C;
       disable event $self;
       resume;
    }
```

The above example creates a new watchpoint that watches the variable `C`. A count of the number of bytes to watch, including the starting address of `80029010`, is given. In addition, an eventpoint handler is defined for this watchpoint. When the watchpoint is triggered, the value of `C` is printed, and then the watchpoint disables itself by using the predefined debugger variable `$self`. Finally, process execution is resumed. This allows the other watchpoint at this location, watchpoint 0, to be triggered the next time through the loop.

```
(CXdb) continue
Resuming execution of Process [#0/*]
Process [#0/0] memory region 0x80029010..0x80029013 modified
Process [#0/0] stopped by Watchpoint 2, at [0x8000139c] TEST in test.f line 12
1001
Event 2 disabled
Resuming execution of Process [#0/*]
Process [#0/0] memory region 0x80029010..0x80029013 modified
Process [#0/0] stopped by Watchpoint 0, at [0x8000139c] TEST in test.f line 12
```

The above command continues process execution. Watchpoint 2 is triggered because it is the highest-numbered, enabled eventpoint to be reached. The commands of its eventpoint handler execute, causing the value of `C` to be printed, the watchpoint to be disabled, and process execution to resume.

The next time through the loop, watchpoint 0 is triggered because it is now the highest-numbered, enabled eventpoint to be reached. Watchpoint 0 stops process execution.

Related Commands

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

Related Concepts

breakpoints
debugger variables
eventpoints
eventpoint handlers
process object
tracepoints

windows

Description

Many different windows are associated with CXdb. Each window provides a different view of the program being debugged.

You can use either the CXwindows interface or the Maryland Windows interface with CXdb. Although the look of the windows is different between the two interfaces, their functionality is very similar.

Some of the windows available in the CXwindows interface are not available in the Maryland Windows interface, due to the inherent screen size limitations.

The windows that are available in both interfaces are:

- Command window
- Source window
- Process interface window
- Help window
- Display file window

Under the CXwindows interface, the windows are maintained by your window manager (for complete information on the manipulation of windows in CXwindows, refer to the window manager documentation). If you are using the Motif window manager, it must be version 2.1. Each window has a menu bar at its top that provides access to the functions for that window. In each case, the first menu item is the name of the window.

You can set the resource values of various window settings for CXdb windows running under the CXwindows interface. You can add these settings to your `.Xdefaults` file. For a description of CXdb Xdefaults settings, refer to the concepts page "Xdefaults."

In addition to the above-mentioned windows, the following windows are available only in the CXwindows interface:

- Disassembly window
- Examine window
- Stack window

Under the Maryland Windows interface, the screen can be divided into multiple windows, much like an X windows display. Interaction with the windows is performed using special keystrokes. You can raise, lower, move, and resize each window, move between windows, scroll and select text from a window, and edit the text on the command line.

You can set the geometry specifications for the windows of the Maryland Windows interface when invoking CXdb at the shell prompt. For more infomation on setting geometry specifications for windows, refer to the reference page for the `cxdb` command.

For a full description of the default key bindings used in the Maryland Windows interface, refer to the reference page for Maryland Windows or the reference page on the `bind` command.

The CXdb windows are described below.

- **Command window** — The primary window for entering commands in CXdb. The command window is the default viewport for command output and command error messages.
- **Source window** — Displays the source code of the current source file. The source window highlights the innermost source units associated with the program counter when process execution stops. Symbols are used to mark the location of eventpoints in the source code. A source window is brought up when an executable file is specified, when the `display routine` command is used, when you open a source window from a menu option, or when a new thread is created.
- **Process interface window** — The interaction window between you and your process. In CXwindows, the process interface window is a standard xterm window. In Maryland Windows, it is another subdivision of the screen.
- **Help window** — Displays the reference page for any topic in this manual. The window is brought up using the `help` command.
- **Display file window** — Displays the contents of a file. This window can display any ASCII text file for browsing only. The window is brought up using the `display file` command.

- **Disassembly window** (CXwindows only) — Displays a region of disassembly instructions for a thread of the process. From the disassembly window you can view the scalar, vector, and communication register sets, as well as the settings of the process status word (PSW) register, through popup windows. Symbols are used to mark the location of eventpoints in the disassembly code displayed in the disassembly window.
- **Examine window** (CXwindows only) — Displays a region of process memory for a thread of the process. You can select the memory unit (byte, half, word, long) and the memory format (such as octal, hexadecimal, decimal, or unsigned) used to display the region of memory.
- **Stack window** (CXwindows only) — Displays the current contents of the stack. The stack window displays the frames on the stack and can also display the arguments to each frame by clicking on the frame's description within the window.

Related Commands

```
bind                        cxdb
disassemble                 display disassembly
display examine             display file
display routine             display source
display source              examine
find memory backward        find memory forward
find window backward        find window forward
help                        info bind
```

Related Concepts

eventpoints
Maryland Windows
stepping
Xdefaults
logging
source units
viewports

Xdefaults

Description

The default X resource settings (Xdefaults) for the CXwindows interface of CXdb are specified in the file /usr/lib/X11/app-defaults/Cxdb.

To modify these resource settings, copy the default specifications into the file that contains your own X resource specifications (usually .Xdefaults or .Xresources). Then modify the specifications to suit your needs.

After modifying the resource specifications, you must enter the following command in your xterm window:

xrdb -merge ~/*resource_file*

resource_file is the name of the file that contains your resource specifications (usually .Xdefaults or .Xresources).

NOTE: If any of these resource specifications conflict with the settings used by your window manager, then the window manager may override your CXdb resource specifications.

The Xdefaults in the /usr/lib/X11/app-defaults/Cxdb file are organized into the following categories:

- General application resources
- Window geometry and sizing resources
- Resources that control interaction behavior
- Mouse and keyboard translations

General application resources — These Xdefaults define basic properties of all CXdb windows:

```
Cxdb*keyboardFocusPolicy:          pointer
Cxdb*foreground:                   black
Cxdb*background:                   white
Cxdb*genericRows:                  24
Cxdb*genericColumns:               80
Cxdb.autoCreate:                   True
```

Window geometry and sizing resources — These Xdefaults determine the size and placement of individual CXdb windows:

```
Cxdb.Command Window.geometry:              +0+0
Cxdb.Source Window.geometry:               +0-0
Cxdb.Stack Window.geometry:                +0-0
Cxdb.Disassembly Window.geometry:          -0+0
Cxdb.Examine Window.geometry:              -0+0
Cxdb.processWindowGeometry:                -0-0
!Cxdb.Scalar Registers.geometry:           -0-0
!Cxdb.Vector Registers.geometry:           -0-0
!Cxdb.Communication Registers.geometry:    -0-0
Cxdb.Display File Window.geometry:         +0-0
Cxdb.Help Window.geometry:                 -0+0
Cxdb.Help Window*XcRichText.width:         520
Cxdb*commandRows:                          20
Cxdb*commandColumns:                       80
Cxdb*sourceRows:                           20
Cxdb*sourceColumns:                        80
Cxdb*stackRows:                            24
Cxdb*stackColumns:                         80
Cxdb*disassemblyRows:                      24
Cxdb*disassemblyColumns:                   80
Cxdb*examineRows:                          24
Cxdb*examineColumns:                       80
Cxdb*displayFileRows:                      24
Cxdb*displayFileColumns:                   80
```

Resources that control interaction behavior — These Xdefaults determine how various windows interact with you. `AutoUpdate` resources determine whether or not the named window will update automatically during a debugging session. The other resources determine attributes of the command window, such as the primary and secondary prompt strings, and whether or not the expert buttons the command menu bar are displayed. These resources are:

```
Cxdb.sourceAutoUpdate:                     True
Cxdb.stackAutoUpdate:                      True
Cxdb.disassemblyAutoUpdate:                True
Cxdb.examineAutoUpdate:                    True
Cxdb.pswAutoUpdate:                        True
Cxdb.scalarAutoUpdate:                     True
Cxdb.communicationAutoUpdate:              True
Cxdb.vectorAutoUpdate:                     True
Cxdb*commandPrompt:                        (CXdb)
Cxdb*secondaryPrompt:                      (cxdb)
Cxdb*commandButtons:                       True
Cxdb*commandMenu:                          True
Cxdb*commandMenuImmediate:                 False
Cxdb*monitorCommandWindowLength:           False
Cxdb*maxCommandWindowLines:                1000
```

Mouse and keyboard translations — These Xdefaults define the mouse and keyboard functions you can use in various windows. The following translation tables apply to text fields in pop-up dialog boxes, text regions of windows other than the command window or pop-up dialog boxes, the command window, the source window, the disassembly window, and the stack window, respectively:

```
Cxdb*XmTextField*translations:  #override \
                !Ctrl<Key>a:    beginning-of-line() \n\
                !Ctrl<Key>b:    backward-character() \n\
                !Ctrl<Key>d:    delete-next-character() \n\
                !Ctrl<Key>e:    end-of-line() \n\
                !Ctrl<Key>f:    forward-character() \n\
                !Ctrl<Key>h:    delete-previous-character() \n\
                !Ctrl<Key>k:    delete-to-end-of-line() \n\
                !Meta<Key>b:    backward-word() \n\
                !Meta<Key>f:    forward-word()

Cxdb*XmText*translations:       #override \
                !Ctrl<Key>a:    beginning-of-line() \n\
                !Ctrl<Key>b:    backward-character() \n\
                !Ctrl<Key>d:    delete-next-character() \n\
                !Ctrl<Key>e:    end-of-line() \n\
                !Ctrl<Key>f:    forward-character() \n\
                !Ctrl<Key>h:    delete-previous-character() \n\
                !Ctrl<Key>k:    kill-to-end-of-line() \n\
                !Ctrl<Key>l:    redraw-display() \n\
                !Ctrl<Key>r:    redraw-display() \n\
                !Ctrl<Key>u:    beginning-of-line()kill-to-end-of-line()\n\
                !Ctrl<Key>v:    next-page() \n\
                !Ctrl<Key>y:    unkill() \n\
                !Meta<Key>b:    backward-word() \n\
                !Meta<Key>d:    kill-next-word() \n\
                !Meta<Key>f:    forward-word() \n\
                !Meta<Key>h:    kill-previous-word() \n\
                !Meta<Key>v:    previous-page()
```

```
Cxdb*CommandText*translations:  #override \
            !<Key>Tab:      complete() \n\
            !Ctrl<Key>i:    complete() \n\
            !Ctrl<Key>n:    down-history() \n\
            !Ctrl<Key>p:    up-history() \n\
            !Ctrl<Key>a:    beginning-of-command-line() \n\
            !Ctrl<Key>b:    backward-character() \n\
            !Ctrl<Key>d:    delete-next-character() \n\
            !Ctrl<Key>e:    end-of-command-line() \n\
            !Ctrl<Key>f:    forward-character() \n\
            !Ctrl<Key>h:    delete-previous-character() \n\
            !Ctrl<Key>k:    kill-to-end-of-line() \n\
            !Ctrl<Key>l:    redraw-display() \n\
            !Ctrl<Key>r:    redraw-display() \n\
            !ctrl<Key>u:    beginning-of-command-line()kill-to-end-of-line()\n\
            !Ctrl<Key>v:    next-page() \n\
            !Ctrl<Key>y:    unkill() \n\
            !Meta<Key>b:    backward-word() \n\
            !Meta<Key>d:    kill-next-word() \n\
            !Meta<Key>f:    forward-word() \n\
            !Meta<Key>h:    kill-previous-word() \n\
            !Meta<Key>v:    previous-page()

Cxdb.Source Window*SourceForm*XmText*translations:      #override \
            Ctrl<Key>v:                 next-page() \n\
            Meta<Key>v:                 previous-page() \n\
            <Key>c:                     submit-command(continue) \n\
            <Key>n:                     submit-command(next) \n\
            <Key>s:                     submit-command(step) \n\
            !<Btn1Down>:                grab-focus() \n\
            !<Btn1Motion>:              extend-adjust() \n\
            !<Btn1Up>:                  extend-end() \n\
            !<Btn2Down>:                grab-focus() \n\
            !<Btn2Motion>:              extend-adjust() \n\
            !<Btn2Up>:                  extend-end() info-expression() \n\
            !<Btn3Down>:                grab-focus() \n\
            !<Btn3Motion>:              extend-adjust() \n\
            !<Btn3Up>:                  extend-end() print-expression() \n\
            !Ctrl<Btn1Down>:            start-source-select() \n\
            !Ctrl<Btn1Motion>:          move-source-select() \n\
            !Ctrl<Btn1Up>:              end-source-select() \n\
            !Ctrl<Btn2Down>:            start-source-select() \n\
            !Ctrl<Btn2Motion>:          move-source-select() \n\
            !Ctrl<Btn2Up>:              end-source-select() info-expression() \n\
            !Ctrl<Btn3Down>:            start-source-select() \n\
            !Ctrl<Btn3Motion>:          move-source-select() \n\
            !Ctrl<Btn3Up>               end-source-select() print-expression() \n\
            !Meta<Btn1Down>:            start-source-select() \n\
            !Meta<Btn1Motion>:          move-source-select() \n\
            !Meta<Btn1Up>:              set-breakpoint() \n\
            !Meta<Btn2Down>:            start-source-select() \n\
```

```
            !Meta<Btn2Motion>:          move-source-select() \n\
            !Meta<Btn2Up>:              end-source-select() info-source() \n\
            !Meta<Btn3Down>:            grab-focus() \n\
            !Meta<Btn3Motion>:          extend-adjust() \n\
            !Meta<Btn3Up>:              extend-end() print-indirect() \n\
            !Ctrl Meta<Btn1Down>:       start-source-select() \n\
            !Ctrl Meta<Btn1Motion>:     move-source-select() \n\
            !Ctrl Meta<Btn1Up>:         run-to() \n\
            !Ctrl Meta<Btn2Up>:         info-line() \n\
            !Ctrl Meta<Btn3Down>:       start-source-select() \n\
            !Ctrl Meta<Btn3Motion>:     move-source-select() \n\
            !Shift Meta<Btn1Down>:      start-source-select() \n\
            !Shift Meta<Btn1Motion>:    move-source-select() \n\
            !Shift Meta<Btn1Up>:        set-tracepoint()

Cxdb.Disassembly Window*textareaSW*XmText*translations: #override \
            !<Btn3Down>:                grab-focus() \n\
            !<Btn3Motion>:              extend-adjust() \n\
            !<Btn3Up>:                  extend-end() print-expression() \n\
            !Meta<Btn1Down>:            start-source-select() \n\
            !Meta<Btn1Motion>:          move-source-select() \n\
            !Meta<Btn1Up>:              set-breakpoint() \n\
            !Shift Meta<Btn1Down>:      start-source-select() \n\
            !Shift Meta<Btn1Motion>:    move-source-select() \n\
            !Shift Meta<Btn1Up>:        set-tracepoint() \n\
            !Ctrl Meta<Btn1Down>:       start-source-select() \n\
            !Ctrl Meta<Btn1Motion>:     move-source-select() \n\
            !Ctrl Meta<Btn1Up>:         run-to() \n\
            <Key>c:                     submit-command(continue) \n\
            <Key>s:                     submit-command(step instruction) \n\
            <Key>n:                     submit-command(next instruction)

Cxdb.Stack Window*XmText*translations:  #override \
            <Key>0:                     submit-command(frame 0) \n\
            <Key>1:                     submit-command(frame 1) \n\
            <Key>2:                     submit-command(frame 2) \n\
            <Key>3:                     submit-command(frame 3) \n\
            <Key>4:                     submit-command(frame 4) \n\
            <Key>5:                     submit-command(frame 5) \n\
            <Key>6:                     submit-command(frame 6) \n\
            <Key>7:                     submit-command(frame 7) \n\
            <Key>8:                     submit-command(frame 8) \n\
            <Key>9:                     submit-command(frame 9) \n\
            <Key>t:                     submit-command(frame 0) \n\
            <Key>u:                     submit-command(frame +1) \n\
            <Key>d:                     submit-command(frame -1)
```

Examples

The following example sets several CXdb Xdefaults.

Assume that the following lines have been added to the .Xdefaults file of your home directory.

```
Cxdb.Command Window.geometry:    700x900+0+0
Cxdb.examineRows:                60
Cxdb*font:                       fixed
Cxdb.disassemblyAutoUpdate:      False
```

Adding the above lines to your ~/.Xdefaults file sets the command window geometry, sets the examine window height to 60 rows, specifies a "fixed" character font for all CXdb windows, and disables automatic screen updating in the disassembly window.

Related Commands

`clear autocreate` `set autocreate`

Related Concepts

Maryland Windows windows

Messages

2

This chapter contains reference pages that explain the CXdb messages. The messages are listed in order by number. Each explanation is divided into the following sections:

- **Message** — The exact text of the message. Variable parameters are enclosed in angle brackets (<>) and are shown in italic type. For example, *<char>* is a variable.
- **Type** — The message type, which can be one of the following:
  - `INFO` — A condition that is not normal or expected, but it is not severe enough to cause an error.
  - `ERROR` — A condition that prevents completion of the CXdb command.
  - `FATAL` — A condition that prevents further execution of the process being debugged.
- **Explanation** — A more detailed explanation of the message, including possible actions to correct the situation.

1 Message: Illegal character in alias identifier: <*char*>
Type: ERROR
Explanation: The alias identifier contains an illegal character that would prohibit parsing. The illegal character is indicated in the text of the error message. Please select a different name for your alias.

2 Message: Alias <*alias name*> exists already, please try a new identifier
Type: ERROR
Explanation: This alias currently exists. Therefore, you should provide a new name in the alias definition.

3 Message: An alias must have a unique identifier
Type: ERROR
Explanation: An alias must be given a name in order to be created.

4 Message: Alias <*alias name*> does not exist
Type: ERROR
Explanation: An alias with this name does not exist.

5 Message: Corrupt executable file for <*filename*>
Type: ERROR
Explanation: The executable file is corrupt. That is, one or more of the internal data structures have been corrupted.

6 Message: Corrupt location range table for <*filename*>
Type: ERROR
Explanation: A location range table could not be opened. Therefore, check to see if it exists. If it does, then possibly some of the internal data representations have been corrupted.

7 Message: Corrupt section table for <*filename*>
Type: ERROR
Explanation: The section table for this program is corrupt.

8 Message: Corrupt source file map for <*filename*>
Type: ERROR
Explanation: The source file map for this program is corrupt.

9 Message: Corrupt source table for <*filename*>
Type: ERROR
Explanation: The source table for this program is corrupt.

No.		Description
10	Message:	Corrupt source range table for *<filename>*
	Type:	ERROR
	Explanation:	A source range table could not be opened. Therefore, check to see if it exists. If it does, then possibly some of the internal data representations have been corrupted.
11	Message:	Corrupt source unit table for *<filename>*
	Type:	ERROR
	Explanation:	A source unit table could not be opened. Therefore, check to see if it exists. If it does, then possibly some of the internal data representations have been corrupted.
12	Message:	Corrupt TSI table for *<filename>*
	Type:	ERROR
	Explanation:	A TSI table could not be opened. Therefore, check to see if it exists. If it does, then possibly some of the internal data representations have been corrupted.
13	Message:	Corrupt variable table for *<filename>*
	Type:	ERROR
	Explanation:	A variable table could not be opened. Therefore, check to see if it exists. If it does, then possibly some of the internal data representations have been corrupted.
14	Message:	Syntax Error - *<expecting or missing>*
	Type:	ERROR
	Explanation:	A syntax error was encountered in the parsing of your command. You should review what you typed, and make sure that it is a valid command. If you are uncertain about the syntax for a given command, you can use the on-line help system to get more information on specific commands.
15	Message:	Syntax Error - *<expecting or missing>*
	Type:	ERROR
	Explanation:	The Fortran language expression you entered contained syntax errors. Please refer to the FORTRAN Guide for details on Fortran sytax.
16	Message:	Display format '*<format specifier>*' may not be associated with memory size '*<size specifier>*'.
	Type:	ERROR
	Explanation:	You have tried to associate a display format with an incompatible memory size. Due to size and machine representations for specific types, not all memory sizes can be displayed in all formats.

17 Message: Invalid event-id,*<event-id>*
Type: ERROR
Explanation: The event number is either negative or not a currently defined event.

18 Message: Invalid thread-id,*<thread-id>*
Type: ERROR
Explanation: The thread number is either negative or larger than the number of threads allowed by the architecture.

19 Message: No such thread-id,*<thread-id>*, for process [#*<process number>*].
Type: ERROR
Explanation: The thread number is either negative or larger than the number of threads currently active for the indicated process.

20 Message: Unable to select thread-id,*<thread-id>*, for process *<process-id>*.
Type: ERROR
Explanation: CXdb was unable to select the indicated thread as the current thread within the process shown. This will probably lead to more errors.

21 Message: Process number *<process number>* is invalid.
Type: ERROR
Explanation: The process number you specified is either negative or not a currently defined process. Use the 'info cxdb' or 'info process' command to determine which process numbers are valid.

22 Message: Invalid directory pathname,*<file specifier>*
Type: ERROR
Explanation: The file specifier is not a directory.

23 Message: IOCTL error on file descriptor *<descriptor number>*
Type: ERROR
Explanation: An error ocurred while performing an ioctl system call on the indicated file descriptor. This may lead to additional errors. Other error messages should indicate the reason the ioctl was being performed if the failure of the ioctl was considered harmful to that operation.

24 Message: IO read/write error on file descriptor *<descriptor number>*
Type: ERROR
Explanation: An error occurred while trying to read or write on the specified file descriptor. This may lead to additional errors. Other error messages should indicate the reason the ioctl system call was being performed if the failure of the ioctl was considered harmful to that operation.

No.		Description
25	Message:	Macro <*macro-name*> already exists; please try a new identifier
	Type:	ERROR
	Explanation:	This macro currently exists. Therefore, you should provide a new name in the macro definition.
26	Message:	A macro must have a unique identifier
	Type:	ERROR
	Explanation:	A macro must be given a name in order to be created.
27	Message:	Macro <*macro-name*> does not exist
	Type:	ERROR
	Explanation:	A macro with this name does not exist.
28	Message:	Macro parameters beyond position <*count*> are ignored.
	Type:	INFO
	Explanation:	CXdb imposes a limit on the number of parameters allowed in a macro invocation. The remainder are ignored.
29	Message:	Macro invocations nested more than <*count*> levels.
	Type:	ERROR
	Explanation:	CXdb imposes a limit on the level of nesting of macro invocations. This limit has been exceeded.
30	Message:	Dynamic memory exhausted.
	Type:	FATAL
	Explanation:	No more virtual memory is available. CXdb has used up all the dynamic memory available to it. This can come from many causes. This is generally a fatal error.
31	Message:	Process [#<*process-number*>] is not stopped.
	Type:	ERROR
	Explanation:	To execute the command, the process must be stopped.
32	Message:	Process [#<*process-number*>] has no running image.
	Type:	ERROR
	Explanation:	The command you entered operates on the image of an executable. The process you specified does not have an active image. Images may be created by executing the 'debug proc', 'run', 'attach', or 'core' commands.
33	Message:	Parameter <*name*> already exists
	Type:	ERROR
	Explanation:	A parameter with that name already exists.

34 Message: Macro parameter name is missing.
Type: ERROR
Explanation: The name of a parameter in a macro definition has been omitted. Every formal parameter in a macro definition must have a name.

35 Message: An error occurred during a read or write on descriptor <*descriptor number*>
Type: ERROR
Explanation: An error occurred during a read, write, or seek system call. This may lead to additional errors. Other error messages should indicate the reason the ioctl was being performed if the failure of the ioctl was considered harmful to that operation.

36 Message: Syntax error in the following command:
<*command line*>
Type: ERROR
Explanation: You have typed something incorrectly. The CXdb parser cannot continue. Please read the command summary to resolve the problem.

37 Message: Error accessing system dependent information
Type: ERROR
Explanation: Error encountered during the getsysinfo system call. This is a fatal error.

38 Message: Unable to open <*filename*>'s executable file
Type: ERROR
Explanation: The executable file cannot be opened. No debugging information can be inferred.

39 Message: Unable to find CDI location range table for <*filename*> in search path; use 'add path' command.
Type: ERROR
Explanation: A location range table was not found. This file is generated by the compiler to inform CXdb about symbolic information. Therefore, check to see if it exists. If it does, then make sure that it is within one of the directories in the search path. Reference the 'add path' command for adding another directory to your search path.

40 Message: Unable to CDI open source file map for <*filename*>; use 'add path' command.
Type: ERROR
Explanation: The source file map for this map cannot be opened. This file is generated by the compiler to inform CXdb about symbolic information.

No.		Description
41	Message:	Unable to find source file for <*filename*> in search path; use 'add path' command.
	Type:	ERROR
	Explanation:	A source file was not found. Therefore, check to see if it exists. If it does, then make sure that it lies within the search path. Reference the 'add path' command for adding another directory to your search path.
42	Message:	Unable to find CDI source range table for <*filename*> in search path; use 'add path' command.
	Type:	ERROR
	Explanation:	A source range table was not found. This file is generated by the compiler to inform CXdb about symbolic information. Therefore, check to see if it exists. If it does, then make sure that it is within one of directories in the search path. Reference the 'add path' command for adding another directory to your search path.
43	Message:	Unable to find CDI source unit table for <*filename*> in search path; use 'add path' command.
	Type:	ERROR
	Explanation:	A source unit table could not be found. This file is generated by the compiler to inform CXdb about symbolic information. Therefore, check to see if it exists. If it does, then make sure that it is within one of directories in the search path. Reference the 'add path' command for adding another directory to your search path.
44	Message:	Unable to find CDI type scope information for <*filename*> in search path; use 'add path' command.
	Type:	ERROR
	Explanation:	A type-scope table could not be found. This file is generated by the compiler to inform CXdb about symbolic information. Therefore, check to see if it exists. If it does, then make sure that it is within one of directories in the search path. Reference the 'add path' command for adding another directory to your search path.
45	Message:	Unable to find CDI variable table for <*filename*> in search path; use 'add path' command.
	Type:	ERROR
	Explanation:	A variable table could not be found. This file is generated by the compiler to inform CXdb about symbolic information. Therefore, check to see if it exists. If it does, then make sure that it is within one of directories in the search path. Reference the 'add path' command for adding another directory to your search path.

46 Message: Unmatched parentheses
Type: ERROR
Explanation: There are unmatched parentheses in the command. Please retype the command correctly so that all of the parentheses are balanced.

47 Message: Internal Error: *<reason for internal error>*
Type: FATAL
Explanation: An internal error in the compiler has occurred. Please submit a problem report.

48 Message: Informational: *<reason for internal error>*
Type: INFO
Explanation: An internal error in the compiler has occured. Please submit problem report.

49 Message: Unable to locate your home directory.
Type: INFO
Explanation: None of the environment variables $DOTDIR, $LOGDIR, or $HOME are defined. Unable to locate your home directory to process any CXdb initialization file.

50 Message: Pathname for *<use of pathname>* is too long.
Type: ERROR
Explanation: The pathname constructed is longer than the maximum allowed by the system. Contact your system administrator.

51 Message: Couldn't *<operation>* file *<filename>*.
System error text: *<errno description>*
Type: ERROR
Explanation: An operation on a file failed. The error message contains the text of the error code returned by the failing operation.

52 Message: The initialization file *<init filename>* is not owned by either you or root. Skipped.
Type: INFO
Explanation: The initialization file specified is neither owned by you nor root. This is a potential security problem, so the file is being skipped.

53 Message: Executable file *<filename>* not found.
Type: ERROR
Explanation: The file you specified as an executable image could not be accessed. Either you do not have permissions to access the directories leading to the file, or the file doesn't exist.

54 Message: File <*filename*> is not marked executable.

Type: ERROR

Explanation: The file you specified exists but does not have the execute permission bit set for you. It is either not executable, or you do not have the permissions to execute it.

55 Message: Multiple process/thread specifier pairs not allowed.
All but first ignored.

Type: ERROR

Explanation: You have specified more than one process/thread specifier pair on a command. Only one pair is allowed. All specifier pairs except the first have been ignored. A common mistake is to enter ':t1 :p0' to specify thread 1 of process 0. The ordering of the process/thread specifiers is important. The process must always come first. The correct specification would be ':p0 :t1'. Starting a process/thread pair with a thread specifier means to process the specified thread in all processes. In version 1.0 this is always a single process.

56 Message: You may not specify threads on this command. Specifiers ignored.

Type: INFO

Explanation: You have specified specific threads on a command that only operates at the process level (for example, the 'kill process' command only operates on processes, not individual threads). The thread specifiers have been ignored.

57 Message: You may not specify processes or threads on this command.
Specifiers ignored.

Type: INFO

Explanation: You have specified specific processes or threads on a command which does not operate on processes (for example, 'info cxdb' and all forms of the 'debug' command do not operate on specific processes). The specifiers were ignored.

58 Message: No process object exists.

Type: ERROR

Explanation: You have tried to execute a command that operates on a process object, and there are no processes defined. Use a form of the 'debug' command to create a process object.

59 Message: Process object <*process number*> vanished!

Type: ERROR

Explanation: A process which was previously verified to be on the global process list has disappeared.

60 Message: Event object <*event number*> vanished!
Type: ERROR
Explanation: An event which was previously verified to be on the global event list has disappeared.

61 Message: No such offset type.
Type: ERROR
Explanation: An offset type has been used which does not exist.

62 Message: Frame specified is out of range.
Type: ERROR
Explanation: You have specified a frame, either by absolute or relative number, which does not exist on the stack. You can use the 'backtrace' command to see the frames currently on the stack.

63 Message: Environment variable <*variable name*> not found.
Type: INFO
Explanation: The environment variable you specified does not exist within the environment being operated upon. Many environments are maintained: CXdb's default environment table and each process' environment.

64 Message: Process ID <*process-id*> is invalid.
Type: ERROR
Explanation: The process ID you specified is invalid. It must be positive and less than 30000.

65 Message: Only one process object may be attached to a running process.
Type: ERROR
Explanation: You tried to specify multiple process objects on the 'attach' command. Only a single CXdb process object can be attached to a running process at a time.

66 Message: Unable to attach to process <*process-id*>.
Type: ERROR
Explanation: CXdb was unable to attach to a child process. During the normal startup process for a target process CXdb will attach to several processes. If any of these fail, the startup will fail. Also, if you specified a process to attach to with the 'debug proc' or 'attach' commands and you don't have the correct permissions to attach to it, the attach may fail.

No.		Description
67	Message:	The system call 'sigaction' failed! Reason: <*failure reason*>.
	Type:	ERROR
	Explanation:	CXdb tried to perform a 'sigaction' system call to alter the way it handles signals. This failed for the indicated reason. This is nearly always a fatal error.
68	Message:	This command may not be run asynchronously. Ignoring asynch operator.
	Type:	INFO
	Explanation:	You specified that the command should run asynchronously with the '&' operator. However, the command you entered can not be executed asynchronously. Only commands that control the execution of a process are allowed to run asynchronously. This includes operations such as run, attach, continue, step, next, stop, and kill.
69	Message:	Process [#<*process number*>] already has a running image.
	Type:	ERROR
	Explanation:	You tried to start the execution of or attach to a process when the process object already has a running image. If you want to restart the process, you should first terminate the current running image with the 'kill process' or 'detach' command and then issue the 'run' command again.
70	Message:	Process [#<*process number*>] has no executable specified for it.
	Type:	ERROR
	Explanation:	You tried to start the execution of a process and no executable file has been specified. You can associate an executable file with an existing process object with the 'executable' command.
71	Message:	Couldn't open a pty connection for process [#<*process number*>].
	Type:	ERROR
	Explanation:	In trying to create the target process, a pty channel is created to communicate with it. This pty connection failed. Previous error messages should indicate the reason for failure. Typically it is because the system is out of pty's.
72	Message:	Unable to block receipt of <*signal name*> signal.
	Type:	FATAL
	Explanation:	CXdb tried to block the receipt of a signal, and the system call failed.
73	Message:	Unable to unblock receipt of <*signal name*> signals.
	Type:	FATAL
	Explanation:	CXdb tried to unblock the receipt of a signal, and the system call failed.

74 Message: Unable to fork to create process [#<*process number*>].
Type: ERROR
Explanation: CXdb tried to fork a new process to create the image to be debugged, and the fork system call failed.

75 Message: Unable to set fixed scheduling on process [#<*process number*>].
Type: INFO
Explanation: CXdb tried to set fixed scheduling on the target process, but the setpattr system call failed.

76 Message: Unable to clear PIXINHERIT process [#<*process number*>].
Type: INFO
Explanation: CXdb uses a process mode called PIXINHERIT to obtain the processes as they start up. This mode must be cleared once the target process is started. CXdb was unable to clear this mode.

77 Message: Unable to set SEQ and SQS modes for process [#<*process number*>].
Type: ERROR
Explanation: CXdb failed to set the Sequential Store (SQS) and Sequential Execution (SEQ) bits in the target process' PSW. Processes can not be debugged if these modes are not set.

78 Message: Unable to determine the parent process for process <*process-id*>.
Type: ERROR
Explanation: CXdb was unable to determine the parent process for a process that it had attached to during the target startup handling. The process was discarded.

79 Message: Parent process <*process-id*> for process <*process-id*> is unknown.
Type: ERROR
Explanation: CXdb determined the parent process for a process that it had attached to during the target startup handling, but it was not any of the processes that CXdb is currently controlling. The new process was discarded.

80 Message: Parent process <*process-id*> for process <*process-id*> has unknown type.
Type: ERROR
Explanation: CXdb determined the parent process for a process that it had attached to during the target startup handling, but its type was unknown. This error will cause the process startup phase to fail.

No.		Description
81	Message:	Parent process <*parent process-id*> for process <*child process-id*> is the initial process.
	Type:	ERROR
	Explanation:	CXdb determined the parent process for a process that it had attached to during the target startup handling, but it was the initial process that CXdb forked during the startup phase. The initial process may have forked another process before performing the exec. This error will cause the process startup phase to fail.
82	Message:	The xterm process <*xterm process-id*> forked process <*child process-id*>.
	Type:	ERROR
	Explanation:	During the startup phase for a target process under the X windows system, CXdb starts an xterm to invoke the process to be debugged. However, the xterm has forked a second process for some reason. This error will cause the process startup phase to fail.
83	Message:	The shell process <*shell process-id*> forked process <*child process-id*>.
	Type:	ERROR
	Explanation:	During the startup phase for a target process CXdb starts a shell to invoke the process to be debugged. However, the shell has forked a second process for some reason. This error will cause the process startup phase to fail.
84	Message:	CXdb has inherited target's child <*process-id*>. It was released.
	Type:	INFO
	Explanation:	For some reason, CXdb has inherited one of the child processes belonging to the target process. It was detached so that it can run normally.
85	Message:	Process <*parent process-id*> is unattached.
	Type:	FATAL
	Explanation:	CXdb determined that a process it received from wait() was not already pattached. This condition should never arise and is a fatal error.
86	Message:	Unexpected SIGTRAP code <*SIGTRAP subcode*> in process <*process-id*>.
	Type:	INFO
	Explanation:	The process indicated received a SIGTRAP signal, but the subcode indicated is unkown. The SIGTRAP was ignored.
87	Message:	Unable to clear the Trace Trap PSW bit in process <*process-id*>.
	Type:	INFO
	Explanation:	CXdb was unable to clear the Trace Trap bit in one of the processes it is managing. This is only a warning but will probably lead to other errors later. If this error occurs during target startup, the startup is terminated.

88 Message: Unknown process <*process-id*> exec'ed.

Type: ERROR

Explanation: One of the processes that CXdb is controlling has an unknown state, and it performed an exec. The process has been detached. If this error occurred during target startup, the startup is terminated.

89 Message: Process [#<*process number*>] exec'ed.

Type: INFO

Explanation: The target process you are debugging has performed an exec. This will probably make all the debugging information that CXdb has about the process obsolete. You should issue the 'executable' command to inform CXdb about the executable the process is now executing. CXdb has removed all your PC based eventpoints from this process because they now contain invalid locations and data.

90 Message: Command [#<*command number*>] is pending on process [#<*process number*>].

Type: ERROR

Explanation: You tried to execute a command on the indicated process, but there was already another command pending (running asynchronously) on that process. You can not execute another command on that process until the previous command completes.

91 Message: The shell controlling process [#<*process number*>] exited.

Type: INFO

Explanation: The shell process that CXdb started to control the execution of the target process exited unexpectedly. This will cause the target to be terminated.

92 Message: The xterm controlling process [#<*process number*>] exited.

Type: INFO

Explanation: The xterm process that CXdb started to control the execution of the target process exited unexpectedly. This will cause the target to be terminated.

93 Message: Thread [#<*process number*>/<*thread-id*>] is already running.

Type: ERROR

Explanation: You entered a command which would continue the execution of one or more threads within a process. At least one of these threads is already running. Your command can not be performed until all of the threads it affects are stopped.

94 Message: Floating point mode not recognized.
Type: ERROR
Explanation: You have referenced a floating point mode which is either unknown or not allowed with this command.

95 Message: No debugging information available.
Type: ERROR
Explanation: You have tried to execute a command that requires the full debugging information for a program. There is currently no debugging information available. You can specify an executable to retrieve debugging information using the 'executable' command.

96 Message: Source file <*file name*> not found.
Type: ERROR
Explanation: The source file you specified could not be found within the debugging information associated with this process. You may have misspelled it, or the directory search path for the process may not include the directory where that source file resides. You can check the search path with the 'info process' command.

97 Message: No source statements found.
Type: ERROR
Explanation: The location you specified does not contain any source statements. Possibly no source statements start at the line you specified or the region you indicated with the cursor contains no source statements.

98 Message: No debugging information available for stepping thread <*thread-id*>. Continuing until routine exit.
Type: INFO
Explanation: There is no debugging information available for the current location of the thread indicated. Stepping by source units is not possible. The execution of the thread will be continued until it returns from the current routine.

99 Message: Unable to clear bits <*bit names*> in PSW.
Type: ERROR
Explanation: The indicated bits could not be cleared in the current PSW or on the stack. This will probably lead to more cascading errors. There is no real way to recover from this.

100 Message: Unable to reset memory occupied by breakpoint at *<address>*.
Type: ERROR
Explanation: The original value of memory currently occupied by a breakpoint could not be restored. This will probably lead to incorrect operation of you program. You will probably not be able to continue the execution of any thread past this address.

101 Message: Unable to replace breakpoint at *<address>*.
Type: ERROR
Explanation: The breakpoint at the indicated address could not be reinstated. This will probably lead to incorrect operation of your program. It is probable that the current stepping operation will fail and further execution may also fail.

102 Message: Source unit *<source unit index>* does not exist within file *<filename>*.
Type: ERROR
Explanation: The source unit index you specified does not exist within the indicated source file. You can use the 'info source' command to get the source unit indices for all source units on a given source file line.

103 Message: *<operation>* only works with the X Window System interface.
Type: ERROR
Explanation: This operation's interface requires CXdb's X Window System interface to function.

104 Message: Eventpoint *<event number>* also has a breakpoint at address *<address>*.
Type: INFO
Explanation: You have created an eventpoint that placed a breakpoint at the same location another eventpoint placed a breakpoint. The last eventpoint created will take precedence over the previous one(s). You may wish to disable or remove the previous event by using the 'disable event' or 'remove event' commands.

105 Message: CXdb variable *<variable name>* not found.
Type: ERROR
Explanation: The CXdb variable you specified does not exist. CXdb variables are created with optional command parameters in some commands, or when used as the storage location of a value in an assignment expression.

106 Message: CXdb variable *<variable name>* cannot be deleted.
Type: ERROR
Explanation: The CXdb variable you specified is predefined by CXdb and cannot be deleted. Only variables that you create can be deleted.

107 Message: CXdb variable <*variable name*> cannot have its value set.

Type: ERROR

Explanation: The CXdb variable you specified is predefined by CXdb to be read-only.

108 Message: Invalid command in eventpoint handler. Handler aborted.

Type: ERROR

Explanation: The handler for the eventpoint indicated contained a command which is not allowed within eventpoint handlers. No form of process execution may be started from within an eventpoint handler. This includes all forms of the 'step', 'next', 'continue', 'finish', 'return', 'signal', 'stop', 'kill', 'run', 'attach', and 'detach' commands. You may resume the execution of the process with the 'resume' command. Execution of the eventpoint handler was terminated.

109 Message: Command not valid interactively. It may only be used in eventpoint handlers.

Type: ERROR

Explanation: The command you entered is not allowed interactively. It may only be used within an eventpoint handler. Commands of this type, 'resume' for example, have no meaning outside of an eventpoint handler and cannot be entered interactively.

110 Message: Ambiguous file <*filename*>, use a qualified pathname.

Type: ERROR

Explanation: The given source file name is ambiguous. Please provide the pathname for the specified source file. That is the file name with one of the directory names in source file search list prepended to it.

111 Message: Source file <*name*> is out of date: last modified <*modified date*>, compiled <*compiled date*>.

Type: INFO

Explanation: The source file is out of date with respect to the version used for compilation. This is only an informational message; the modified version of the source file will be used.

112 Message: Cannot get the value for <*symbol*>

Type: ERROR

Explanation: CXdb could not determine the value for the given symbol. Thus, evaluation cannot continue.

113 Message: File <*name*> is not a core file.
Type: ERROR
Explanation: The file you specifed was not a core file. Please be certain that you typed the name correctly. The operation was aborted.

114 Message: Address <*hex address*> is not within a defined region of the process image.
Type: ERROR
Explanation: The address you specified was not within any of the section maps contained in the current process image. Either you entered an invalid address or the process image is not from the executable you are debugging.

115 Message: A process image already exists! Use 'kill process' or 'detach' first.
Type: ERROR
Explanation: You tried to associate a new image with a process object that already has an image. You must remove the existing image with either the 'kill process' or 'detach' command and then try again.

116 Message: Vector register <*name*> doesn't have enough space.
Type: INFO
Explanation: The vector register you specified does not have enough space left for the operation you requested. Verify that the number of bytes you are trying to read/write exist in the register being accessed from the location you specified.

117 Message: Your SHELL setting, '<*shell path*>', is unsupported. Defaulting to '/bin/sh'.
Type: INFO
Explanation: Your SHELL environment variable specifies a unknown shell. CXdb only supports the use of 'sh', 'csh', 'ksh', and 'tcsh'. Your SHELL setting will not be honored, and '/bin/sh' will be used instead.

118 Message: Illegal regular expression, <*value*>
Type: ERROR
Explanation: This is an illegal regular expression. Please reference the ed(1) man page to determine the proper syntax of the regular expression. If certain special characters are part of the identifier, you may need to escape those characters.

119 Message: System call failed: *<syscall name> - <failure reason>*

Type: ERROR

Explanation: The indicated system call failed. The reason for the failure is also shown. The ramifications of this failure are entirely dependent upon what you were doing when the error occurred.

120 Message: Error reading scalar registers for [#*<process>*/*<thread>*].

Type: ERROR

Explanation: An error has been encountered while trying to read the scalar registers. This will almost certainly cascade into other errors. Depending on when this error was encountered, CXdb may not be able to determine the state of your process correctly.

121 Message: Error reading vector registers for [#*<process number>*/*<thread-id>*].

Type: ERROR

Explanation: An error has been encountered while trying to read the vector registers. This is not a fatal error, but CXdb will not be able to display the contents of these registers, nor will you be allowed to modify them.

122 Message: Error reading communication registers for [#*<process number>*/*<thread-id>*].

Type: ERROR

Explanation: An error has been encountered while trying to read the communication registers. This is not a fatal error, but CXdb will not be able to display the contents of these registers, nor will you be allowed to modify them.

123 Message: Invalid command! You may not resume the execution of a core file.

Type: ERROR

Explanation: You attempted to perform some type of process excution (such as 'step' or 'next') while examining a core file. Core files can not be executed in any fashion. You may start a new process image with the 'run' command, but you can not continue the execution of a core file.

124 Message: Unable to get trap addresses for thread *<thread id>*.

Type: ERROR

Explanation: During process creation CXdb was unable to read the user mode trap addresses from the process. This probably indicates a more serious problem, and the process object will be unusable.

125 Message: Read of address <*address*> to get <*datum*> failed.
Type: ERROR
Explanation: The read operation on the process image failed. The address being read and the data wanted were listed in the error message. The operation that required the read will fail. Further debugging may not be possible depending on the cause of the failure.

126 Message: Write of address <*address*> to put <*datum*> failed.
Type: ERROR
Explanation: The write operation on the process image failed. The address being written to and the data being written were listed in the error message. The operation that required the write will fail. Further debugging may not be possible depending on the cause of the failure.

127 Message: Unable to set bits <*bit names*> in PSW.
Type: ERROR
Explanation: The indicated bits could not be set in the current PSW or on the stack. This will probably lead to more cascading errors. There is no real way to recover from this.

128 Message: There is currently no default source file, please specify one.
Type: ERROR
Explanation: You entered a command that requires a file name specification. You did not enter a file name, and there is currently no default file. The default file is set each time you access a file from the source window. Until the source window is created, there is no default source file.

129 Message: Error in constructing a frame [#<*process number*>/<*thread-id*>].
Type: ERROR
Explanation: An error was encountered in trying to construct a stack frame for the thread of the process. It was impossible either to read the current registers, or to read from memory.

130 Message: Evaluation of the language expression failed.
Type: ERROR
Explanation: The evaluation of the language expression you entered failed for the indicated reason(s). How to resolve the problem is dependent on the reason for failure.

131 Message: Unable to locate main routine! No default source file set.
Type: INFO
Explanation: When CXdb loads an executable, it tries to locate the main routine for the application and then uses that to set up the default source file name and source language. CXdb was unable to determine the location of the main routine. This usually means that either the main routine was not compiled with the '-cxdb' option, or it is written in a language that CXdb doesn't support. No default source file has been set and the default language has been set to C.

132 Message: There is no location in memory for <*identifier*>.
Type: ERROR
Explanation: There is currently no location in memory for the referenced variable. Although the identifier exists, there is no corresponding location in memory. The identifier may be a compile time construct. If the identifier does represent a variable, the references to it were optimized away.

133 Message: Too many threads selected (<*thread count*>). Only 1 allowed.
Type: ERROR
Explanation: You have tried to select more than one thread for an operation that only allows a single thread. Please use the ':t' focus specifier to select a single thread and try the command again.

134 Message: Formal parameter <*identifier*> is never used in macro body.
Type: INFO
Explanation: Formal parameter was not used in the macro body. This is only an informational message, and the macro definition will persist.

135 Message: A Source Window could not be created.
Type: INFO
Explanation: A Source Window could not be created because there was either no executable known to the compiler-debugger interface, or because there is no stack frame in the current state of the process which contains compiler-debugger interface information, or because no source file for any stack frame containing compiler-debugger interface information could be found.

136 Message: An incomplete geometry specification was given.
Type: INFO
Explanation: An incomplete geometry specification was given. Either the height, X origin, or Y origin were missing. The format for a geometry specification is (width)x(height)+(x-origin)+(y-origin).

137 Message: Bad width field '*<width>*' was given in the geometry specification.

Type: INFO

Explanation: An invalid width was given in the geometry specification for a window. Either the width was the only part of the geometry given, or it was non-numeric. The format for a geometry specification is (width)x(height)+(x-origin)+(y-origin).

138 Message: Bad height field '*<height>*' was given in the geometry specification.

Type: INFO

Explanation: A invalid height was given in the geometry specification for a window. Either the height was the only part of the geometry given, or it was non-numeric. The format for a geometry specification is (width)x(height)+(x-origin)+(y-origin).

139 Message: Bad width and height separator '*<separator>*' was given in the geometry specification.

Type: INFO

Explanation: A invalid width and height separator was given in the geometry specification for a window. Either the width was the only part of the geometry given, or it was not an 'x' character. The format for a geometry specification is (width)x(height)+(x-origin)+(y-origin).

140 Message: Bad origin separator '*<separator>*' was given in the geometry specification.

Type: INFO

Explanation: A invalid origin separator was given in the geometry specification for a window. Either the width and height or width, height, and X origin were the only part of the geometry given, or the separator was not a '+' character. The format for a geometry specification is (width)x(height)+(x-origin)+(y-origin).

141 Message: Bad origin '*<origin>*' was given in the geometry specification.

Type: INFO

Explanation: A invalid origin value was given in the geometry specification for a window. Either an incomplete geometry specification was given, the origin value was non-numeric, or an origin value was given that extends beyond the edge of the physical CRT screen. The format for a geometry specification is (width)x(height)+(x-origin)+(y-origin).

142 Message: Too wide geometry width '*<width>*' was given in the geometry specification.

Type: INFO

Explanation: The value for the width in the geometry specification for the window is wider than the physical CRT screen.

No.		Description
143	Message:	Too high geometry height '*<height>*' was given in the geometry specification.
	Type:	INFO
	Explanation:	The value for the height in the geometry specification for the window is higher than the physical CRT screen.
144	Message:	The geometry specification of the X origin plus the width ('*<X origin + width>*') would extend past the edge of the physical screen.
	Type:	INFO
	Explanation:	The values for the X origin and width in the geometry specification for the window would make the window extend past the edge of the physical CRT screen.
145	Message:	The geometry specification of the Y origin plus the height ('*<Y origin + height>*') would extend past the bottom of the physical screen.
	Type:	INFO
	Explanation:	The values for the Y origin and height in the geometry specification for the window would make the window extend past the bottom of the physical CRT screen.
146	Message:	Object file *<name>* compiled with early version of compiler.
	Type:	INFO
	Explanation:	An object file used to create the executable being debugged was created with a compiler which is older than the current version supported by this version of CXdb. Object files must be created with the following two compilers: CONVEX Fortran 6.1.1 or CONVEX C 4.0, or greater. Recompiling the indicated file will allow CXdb to provide complete debugging support.
147	Message:	Line *<line number>* is out of bounds. Valid range is 1 to *<max line number>*.
	Type:	ERROR
	Explanation:	You have specified a line number that is not within the valid range for a source file. The valid range is included in the error message text. You should retry your command using a line number within the valid range.
148	Message:	Source unit *<number>* is out of bounds. Valid range is 0 to *<maximum>*.
	Type:	ERROR
	Explanation:	You have specified a source unit number which is not within the valid range for a source file. The valid range is included in the error message text. You should retry your command using a source unit number within the valid range.

149 Message: Line <*line number*> has been optimized away.
Type: ERROR
Explanation: You have specified a line number that, due to optimizations performed by the compiler, has no object code associated with it. It is impossible to place eventpoints on this line.

150 Message: Source unit <*source unit number*> has been optimized away.
Type: ERROR
Explanation: You have specified a source unit number that, due to optimizations performed by the compiler, has no object code associated with it. It is impossible to place eventpoints on this source unit.

151 Message: A process object already exists. Try the '<*command name*>' command.
Type: ERROR
Explanation: You have used a command which would create another process object. Version 1.0 of CXdb only operates on a single process. Each of the commands that create process objects have equivalents that modify an existing process object. For 'debug exec' use 'executable', for 'debug core' use 'core', and for 'debug proc' use 'attach'.

152 Message: Address <*address*> is an invalid PC setting for process [#<*process*>/<*thread*>].
Type: ERROR
Explanation: You have specified an invalid address with the 'goto' command. It is either outside the address regions of the program, or read permissions are not set for the page containing the address. The PC for this process and thread will not be modified.

153 Message: <*source unit or line*> <*file*>:<*index*> has multiple entry points.
Type: ERROR
Explanation: You have specified a source unit or line number with the 'goto' command which has multiple entry points. Because this is an ambiguous situation, the PC of the thread has not been modified. You can use the 'info sourceunit' or 'info line' command to see the entry point information and then try the 'goto address' command with a specific address from that list.

154 Message: Multiple threads selected, evaluating expression in thread <*thread-id*>
Type: INFO
Explanation: Expression evaluations must be made in the context of a process image. If that you selected more than 1 thread in the command focus, then one of the threads will be selected to be used as the context for performing the expression evaluation. This may be incorrect if the threads selected are in different contexts (or scopes). If this is the case, you should retry the command selecting only a single thread using the ':t' focus specifier.

155 Message: Expression has invalid type for use as an address.
Type: ERROR
Explanation: The expression you entered as part of a command that required an address evaluated to a type which could not be converted to an address. When an address is required, CXdb will only convert from integer types. All other types are invalid. Please try the command again using a different expression or, if you are in C, try casting the result to an integer type.

156 Message: Conversion of expression result to <*type name*> failed.
Type: ERROR
Explanation: The expression you entered could not be converted to the type indicated. Please try the command again using a different expression or, if you are in C, try casting the result to a type that can be converted to the desired type.

157 Message: Invalid signal number <*signal number*>.
Type: ERROR
Explanation: The signal number you entered is invalid. Signal numbers must be in the range 0 - 31. Please enter the command again with a valid signal number.

158 Message: Unable to push <*byte count*> bytes onto the process stack.
Type: ERROR
Explanation: CXdb was unable to push the indicated number of bytes onto the process' stack. This operation is done in preparation for calling a function within the target process (as in evaluating an expression which includes a function call). Because the data could not be placed on the stack, the call will not be performed. Further errors may result.

159 Message: CXdb is assuming the screen contains <*screen rows*> rows.
Type: INFO
Explanation: The shell from which CXdb was invoked did not provide a valid number of rows for the terminal screen. CXdb is assuming a default height as indicated.

160 Message: CXdb is assuming the screen contains <*screen cols*> columns.
Type: INFO
Explanation: The shell from which CXdb was invoked did not provide a valid number of columns for the terminal screen. CXdb is assuming a default width as indicated.

161 Message: File <*name*> doesn't exist. Can't append with NOCLOBBER set.
Type: ERROR
Explanation: Noclobber is currently set, and you specified an append operation in a viewport specification on a file that doesn't exist. Because noclobber is set, CXdb will not create this file. You can clear noclobber mode and try again, or you can use the override redirection operator '>>!' to force the file creation.

162 Message: File <*name*> exists. Can't overwrite with NOCLOBBER set.
Type: ERROR
Explanation: Noclobber is currently set, and you specified an overwrite operation in a viewport specification on a file that already exists. Because noclobber is set, CXdb will not truncate this file. You can clear noclobber mode and try again, or you can use the override redirection operator '>!' to force the file truncation.

163 Message: Window ID <*id*> is invalid for use as a viewport target.
Type: ERROR
Explanation: Noclobber is currently set, and you specified an overwrite operation in a viewport specification on a file that already exists. Because noclobber is set CXdb will not truncate this file. You can clear noclobber mode and try again or you can use the override redirection operator '>!' to force the file truncation.

164 Message: Eventpoint <*eventpoint id*> expression evaluation failed.
Type: ERROR
Explanation: The evaluation of the language expression you entered failed while testing the evenpoint's conditional for the indicated reason(s). How to resolve the problem is dependent on the reason for failure.

165 Message: Relational expression already TRUE. Eventpoint not created.
Type: ERROR
Explanation: The language expression you entered currently evaluates to TRUE. Because a relation eventpoint halts execution when a relation becomes true, the eventpoint was not created. Please make sure that you entered the expression correctly.

166 Message: File '<*name*>' not found within current search path.
Type: ERROR
Explanation: The file you specified could not be located anywhere within the directories currently in your search path. Check to be sure that you typed the file name correctly and that you have the necessary directories in your search path.

167 Message: Reading compiler data files...
Type: INFO
Explanation: When reading the compiler data files, an unusually long response time may occur. This message informs the user that everything is fine.

168 Message: Unable to read previous stack frame.
Type: ERROR
Explanation: CXdb was attempting to read a stack frame and the attempt failed. This error will almost certainly lead to additional errors. In the case of creating relation eventpoints, the eventpoint will not be created.

169 Message: Relation eventpoint <*eventpoint #*> now out of scope. Disabled.
Type: INFO
Explanation: Relation type events are tied to a specific frame on the stack. When this frame is popped off the stack during normal program execution, any relation eventpoints that were associated with that frame are automatically disabled.

170 Message: File <*name*> is <*desired kind*>.
Type: ERROR
Explanation: An operation on a file failed. The error message contains the reason why the file operation failed.

171 Message: Unknown scope path, <*scope path*>
Type: ERROR
Explanation: An unknown scope path was used.

172 Message: Evaluating expression in each thread context
Type: INFO
Explanation: Expression evaluations must be made in the context of a specific thread within the process. The evaluation of the expression you entered will be evaluated within the context of EACH thread processed. It is possible that errors may result from evaluation within one thread and not in others. This can happen if the threads are currently within different scopes in which all the same identifiers are not visible.

173 Message: Invalid offset <*number*>
Type: ERROR
Explanation: The offset you entered is invalid. It must be a non-zero positive integer. Additionally, it is an error if the offset specified would place the ending address outside the memory allocated to the process.

174 Message: Invalid address range <*start*>..<*end*>

Type: ERROR

Explanation: The address range you specified is invalid. Both addresses specified must reside within the address range occupied by the process, and the starting address of the range must be less than the ending address.

175 Message: Data region lies on stack. Eventpoint will be disabled when frame is popped.

Type: INFO

Explanation: The address range you specified lies within the bounds of the stack. When the stack frame containing this region is popped off the stack, the eventpoint will be automatically disabled.

176 Message: Debugger variable '<*name*>' is not <*object*> specifier

Type: ERROR

Explanation: Debugger variables have a type associated with them. You used a debugger variable in a context that required it to have a specific type and it didn't have that type. For example, when an eventpoint is created, a debugger variable may be associated with that eventpoint. Later, that variable may be used to reference that eventpoint in other commands. However, it couldn't be used in commands that operate on processes because it references an eventpoint.

177 Message: Stack popped region being monitored by eventpoint <*eventpoint #*>. Disabled.

Type: INFO

Explanation: The data region being monitored by the eventpoint indicated resided within the stack. The stack has now been popped to a point that no longer contains this region. Any data stored there 'technically' no longer exists. To prevent inaccurate results, the evenpoint has been disabled.

178 Message: Expression not valid for use as an address.

Type: ERROR

Explanation: The language expression entered is not usable as an address specifier. You can use the ':' or '..' notation to construct explicit address ranges or you can use the address operators (%LOC in Fortran and '&' in C) to obtain the address of an identifier. When using the address operator form, the size of the region will be determined from the identifier referenced.

179 Message: Cycle detected in alias translation with alias *<alias name>*

Type: INFO

Explanation: A cycle has been detected in alias translation. The command line is still processed, but alias translation ceases. Only the alias itself may expand with itself as part of the name. The given name is one of the aliases within the cycle. Note, by definition, there will be several other aliases within the cycle.

180 Message: Syntax Error - *<expecting or missing>*

Type: ERROR

Explanation: The C language expression you entered contained syntax errors. Please refer to a C Guide for details on C syntax.

181 Message: Identifier not visible from the current lexical scope, '*<identifier>*'

Type: ERROR

Explanation: The specified identifier is not visible from the current lexical scope. There are two principal causes for this message. This message can be caused either by a misspelled identifier or the use (or omission) of a scope prefix. If a scope prefix is used, the message indicates the identifier is not visible in the specified scope. If the scope prefix is omitted, the message indicates the identifier is not visible from the present lexical scope established from the current PC (or selected frame).

182 Message: Floating point overflow

Type: ERROR

Explanation: This message occurs when you apply an operator to a floating point operand(s) which yields a result too large to be represented by the resulting type.

183 Message: Mixed mode operation - left operand of '*<operator>*' is mixed mode

Type: INFO

Explanation: The left operand of the indicated operator has an internal floating point representation that is not compatible with the floating point mode established by the 'set evalopts fpmode' command. Try changing the fpmode of the evaluator to match the fpmode of the left operand. Refer to the 'set evalopts fpmode' command for more information.

184 Message: Mixed mode operation - right operand of '*<operator>*' is mixed mode

Type: INFO

Explanation: The right operand of the indicated operator has an internal floating point representation, which is not compatible with the floating point mode established by the 'set evalopts fpmode' command. Try changing the fpmode of the evaluator to match the fpmode of the right operand. Refer to the 'set evalopts fpmode' command for more information.

185 Message: Divide by zero
Type: ERROR
Explanation: The operation's divisor has a value of zero, which causes the operation to produce a mathematically undefined result. Verify the origin of the divisor and make the necessary changes to ensure the divisor is non-zero, then retry the operation.

186 Message: Subscript truncated to integer
Type: INFO
Explanation: The precision of the integral data type used in a subscript expression is greater than the precision supported by the hardware architecture. The subscript expression has been converted (truncated) to the precision supported by the hardware architecture.

187 Message: Subscript not integer type
Type: ERROR
Explanation: Subscript expressions must result in an integral data type. Try converting this expression using the INT() Fortran intrinsic or by using the C cast operator (int).

188 Message: Too many subscripts specified
Type: ERROR
Explanation: The number of subscripts specified in the array expression outnumber the number of subscripts defined by the referenced array. Refer to the 'info expression' command to obtain the array definition of the array expression.

189 Message: Too few subscripts specified
Type: ERROR
Explanation: The number of subscripts specified in the array expression are fewer than the number of subscripts defined by the referenced array. Refer to the 'info expression' command to obtain the array definition of the array expression.

190 Message: Left Hand Side of '.' is not a structure/union type
Type: ERROR
Explanation: The selection operator (.) is being applied to an expression that does not yield a structure or union type. Refer to the 'info expression' command to obtain the type yielded by the expression.

191 Message: Subscript required
Type: ERROR
Explanation: The array reference requires a subscript. Refer to the 'info expression' command to obtain the number of subscripts required by the type yielded by the array reference.

192 Message: Illegal indirection
Type: ERROR
Explanation: An attempt has been made to reference an object through a nonpointer expression. Refer to the 'info expression' command to obtain the type yielded by the expression.

193 Message: Illegal address reference
Type: ERROR
Explanation: An attempt has been made to take the address of an operand which is not a function designator, an lvalue which designates an object which is a bit field or has been declared with register storage class.

194 Message: Cannot dereference a pointer to 'void'
Type: ERROR
Explanation: The address expression refers to a non-existent object (void). Dereferencing the address expression is prohibited. Try casting the address expression to a type which refers to a legal object type.

195 Message: Illegal pointer/integer combination
Type: ERROR
Explanation: An attempt has been made to take the address of either (1) an operand that is not a function designator, or (2) an lvalue that designates an object which is a bit field, or (3) an object that has been declared with register storage class.

196 Message: Variable's storage is not available, *<identifier>*
Type: ERROR
Explanation: The referenced variable's storage is not available. There are many reasons that may cause this. The variable may neither be read nor written by the program due to program logic. If it is a stack local variable, its frame may not be active. If its frame is active, the variable may be logically dead because it is no longer referenced past this point in the routine. In fact, it may have been replaced by a temporary variable introduced by the compiler in the case of induction variables.

197 Message: CXdb variable undefined
Type: ERROR
Explanation: The CXdb variable has not been previously defined.

198 Message: Loader symbol not found
Type: ERROR
Explanation: The specified loader symbol does not appear in the nlist for the executable assigned to the specified process. Check the spelling of the symbol. Note that global Fortran symbols have underscores prepended and appended to the symbol root. C symbols have underscores prepended to the symbol root. Assembler symbols appear exactly as defined in the assembly source.

199 Message: Variable has not yet become active
Type: ERROR
Explanation: The variable specified in the expression is visible from the current scope. However, at present there are no active locations available for use in the evaluation. This can occur if the process has not been activated with the 'run' command. Start the process and try evaluating the expression again when the process stops as a result of an eventpoint such as a breakpoint.

200 Message: Incomplete type
Type: ERROR
Explanation: An attempt has been made to evaluate an expression in which the definition of the object designated by the expression is not visible. An example of this situation frequently occurs when a pointer to a derived type is declared in the source file, yet the source never references the object except to assign (or access) a value to the pointer. In this case, try prefixing the derived type's tag identifier with a scope path in which the type's definition is visible in a cast expression.

201 Message: Illegal sizeof expression, sizeof(void) not permitted
Type: ERROR
Explanation: It is illegal to use the sizeof operator on an object of type void. The void type is an incomplete type that cannot be completed, hence no size can be computed.

202 Message: Illegal sizeof expression, sizeof(function) not permitted
Type: ERROR
Explanation: It is illegal to use the sizeof operator on an object that designates a function.

No.		Description
203	Message:	Illegal sizeof expression, sizeof(bitfield) not permitted
	Type:	ERROR
	Explanation:	It is illegal to use the sizeof operator on an object that designates a bit field.
204	Message:	Illegal sizeof expression, 'type' is incomplete
	Type:	ERROR
	Explanation:	An attempt has been made to compute the size of an object whose definition is not visible from the present scope. This situation frequently occurs when a pointer to a derived type is declared in the source file, yet the source never references the object except to assign (or access) a value to the pointer. In this case, try prefixing the derived type's tag identifier with a scope path in which the type's definition is visible in a cast expression and re-apply the sizeof operator.
205	Message:	Illegal type combination
	Type:	ERROR
	Explanation:	The type name specification includes conflicting type specifiers. An example of this might be a type name specification that includes both "float" and "struct foo" type specifiers.
206	Message:	Storage class specifier ignored
	Type:	INFO
	Explanation:	The storage class specifier identified is not applicable in the expression and is therefore ignored. The evaluation is not affected by its specification.
207	Message:	Illegal type combination, both 'signed' and 'unsigned' specified
	Type:	ERROR
	Explanation:	The type name specification includes conflicting type modifiers. The type modifiers "signed" and "unsigned" semantically conflict.
208	Message:	Illegal type combination, both 'short' and 'long' specified
	Type:	ERROR
	Explanation:	The type name specification includes conflicting type modifiers. The type modifiers "short" and "long" semantically conflict.
209	Message:	Illegal type combination, both 'short' and 'char' specified
	Type:	ERROR
	Explanation:	The type name specification includes conflicting type modifiers and specifiers. The type modifier "short" cannot be applied to the type "char".

210 Message: Illegal type combination, both 'long' and 'char' specified
Type: ERROR
Explanation: The type name specification includes conflicting type modifiers and specifiers. The type modifier "long" cannot be applied to the type "char".

211 Message: Illegal type combination, 'volatile' specified more than once
Type: ERROR
Explanation: The type qualifier "volatile" has been specified more than once.

212 Message: Illegal type combination, 'const' specified more than once
Type: ERROR
Explanation: The type qualifier "const" has been specified more than once.

213 Message: 'volatile' qualifier ignored
Type: INFO
Explanation: The "volatile" type qualifier has no effect on the evaluation of the expression, therefore it has been ignored.

214 Message: 'const' qualifier ignored
Type: INFO
Explanation: The "const" type qualifier has no effect on the evaluation of the expression, therefore it has been ignored.

215 Message: Type redeclared 'volatile'
Type: ERROR
Explanation: The type name has already been declared "volatile" and is now being redeclared as such. A type may be qualified "volatile" only once.

216 Message: Type redeclared 'const'
Type: ERROR
Explanation: The type name has already been declared "const" and is now being redeclared as such. A type may be qualified "const" only once.

217 Message: Illegal type in cast expression
Type: ERROR
Explanation: The type name specified in the cast expression is not a scalar or void type. Examples of scalar types are char, int, float, and double, where each may be further modified by short/long or signed/unsigned and qualified by const and/or volatile. A pointer to one of the above simple types or to a derived type may be specified. All derived types (except pointer) are disallowed.

218 Message: Illegal type in cast expression, cannot cast to 'function' type
Type: ERROR
Explanation: It is illegal to cast an expression to "function" type.

219 Message: Operand of 'cast' has incompatible type
Type: ERROR
Explanation: The operand in the specified cast expression is not a scalar. Examples of scalar types are char, int, float, and double, where each may be further modified by short/long or signed/unsigned and qualified by const and/or volatile. A pointer to one of the above simple types or to a derived type may be specified. All derived types (except pointer) are disallowed.

220 Message: Invalid expression, constant expression required
Type: ERROR
Explanation: This is a syntax error. The syntax for the expression requires a constant expression. That is an expression consisting of literals only.

221 Message: Illegal subscript value, must be greater than zero
Type: ERROR
Explanation: The type name definition requires that the constant expression evaluate to an integral value greater than zero.

222 Message: 'void' is not a legal array element type
Type: ERROR
Explanation: An array type may not be derived from the void type. The void type is an incomplete type that cannot be completed.

223 Message: 'function' is not a legal array element type
Type: ERROR
Explanation: An array type may not be derived from a function type.

224 Message: Functions cannot return array types
Type: ERROR
Explanation: The specified type name defines a function that returns an array type as its result. This is an illegal type constructor.

225 Message: Functions cannot return function types
Type: ERROR
Explanation: The specified type name defines a function that returns a function type as its result. This is an illegal type constructor.

226 Message: Illegal array dimension, dimension is null
Type: ERROR
Explanation: The type name definition requires that the constant expression describing the bounds of the array evaluate to an integral value greater than zero. This expression evaluates to a zero-sized dimension.

227 Message: Bad type name, specified tag does not identify a struct type
Type: ERROR
Explanation: The type tag specified in the structure type name designates a type that is not a structure type.

228 Message: Bad type name, specified tag does not identify a union type
Type: ERROR
Explanation: The type tag specified in the union type name designates a type that is not a union type.

229 Message: Bad type name, specified tag does not identify an enumeration type
Type: ERROR
Explanation: The type tag specified in the enumeration type name designates a type that is not a enumeration type.

230 Message: Incomplete type, struct/union type has not been defined
Type: ERROR
Explanation: A type definition corresponding to the specified type tag cannot be found from the present scope. Try prefixing the derived type's tag identifier with a scope path in which the type's definition is visible.

231 Message: Parameter's type is expected to be an array type
Type: INFO
Explanation: Parameter type mismatch. The called function expects an array type as its actual parameter. This warning does not inhibit the evaluation of the function call.

232 Message: Parameter's type is expected to be a function type
Type: INFO
Explanation: Parameter type mismatch. The called function expects a function type as its actual parameter. This warning does not inhibit the evaluation of the function call.

233 Message: Actual and formal are different sized objects
Type: INFO
Explanation: This message results when the called subroutine or function declares a dummy argument of one type and/or precision, and the caller passes an actual argument with a different type and/or precision.

234 Message: Too many actual parameters specified
Type: INFO
Explanation: The function or subroutine is being called with more actual parameters than the formal definition has declared. This is an informational message only, and its purpose is to provide information which might expose a potential programming error.

235 Message: Too few actual parameters specified
Type: INFO
Explanation: The function or subroutine is being called with fewer actual parameters than the formal definition has declared. This is an informational message only, and its purpose is to provide information which might expose a potential programming error.

236 Message: Operand truncated to integer
Type: INFO
Explanation: The integral result of the subexpression has been truncated to an integer with a precision consisting of four bytes. This occurs as a result of constraints imposed by the hardware architecture. The resulting value of this expression will be used to obtain an address in the process' virtual memory space.

237 Message: Incompatible formal and actual parameter types
Type: ERROR
Explanation: The type of the actual parameter is incompatible with the declaration of the formal parameter. Try referring to the function declarator (prefixing the declarator with an explicit scope path leading to its definition, such as c$foo (extern) or c$file'foo (static)) in an 'info expression' command to obtain its definition.

238 Message: 'const' qualifier ignored
Type: INFO
Explanation: The side effect produced by evaluating the expression has been permitted to occur in an effort to permit a more flexible debugging environment. This sole purpose of this message is to report the actions of the evaluator.

239 Message: Scope block not found, '*<block>*'
Type: ERROR
Explanation: The specified block is not visible from the current lexical scope. There are two main causes for this message. The first cause may be the result of a typing error in which the specified block is misspelled. The second cause arrises as a result of the use of a scope prefix. When a scope prefix is used, the message indicates the block is not visible in the specified scope chain.

240 Message: Syntax Error - must be constant
Type: ERROR
Explanation: The real and imaginary components of a COMPLEX constant must be constant expressions.

241 Message: Syntax Error - precision of constant too high
Type: ERROR
Explanation: The precision of the real and imaginary components of a COMPLEX constant must be either single or double precision. Specifying quad (REAL*16) precision causes this error.

242 Message: Syntax Error - REAL or INTEGER constant required
Type: ERROR
Explanation: The real and imaginary components of a COMPLEX constant must be of type INTEGER or type REAL.

243 Message: Merged scope blocks, disambiguate identifier specification, *<identifier>*
Type: ERROR
Explanation: This message occurs as a result of compiler optimizations. The compiler has merged two or more scope blocks into a single scope block. As a result of optimization, the identifier which would normally shadow an identifier with the same name in an outer scope block, now becomes visible. To disambiguate between these two (or more) identifiers, a scope prefix must be used. CXdb preserves the lexical scope, thereby permitting a scope prefix to address the desired identifier.

244 Message: No address returned after successful lookup of loader symbol: *<symbol>*
Type: ERROR
Explanation: No address returned after successful lookup of loader symbol. This is an internal error. This should be reported to the CONVEX Technical Assistance Center.

No.		Description
245	Message:	Illegal scope - scope paths are not used with the 'loader$' prefix
	Type:	ERROR
	Explanation:	This message results when a scope path has been specified in conjunction with the loader or assembler language specifier (such as l$, loader$, or asm$). Loader symbols do not have "lexical" scope. Remove the the portion of the scope path delimited by the language specifier and the target symbol to correct the specification.
246	Message:	Floating point modes conflict: CXdb [*<fpmode>*] Process [*<fpmode>*]
	Type:	INFO
	Explanation:	This message reports the condition when the evaluator's floating point mode and the process' floating point mode are incompatible. This is an informational message only; evaluation proceeds as directed.
247	Message:	Floating point hardware not available, evaluation aborted
	Type:	ERROR
	Explanation:	This is a fatal evaluation error. The floating point mode assigned to the evaluator (refer to the 'set evalopts fpmode' command) requires IEEE floating point hardware which is not present.
248	Message:	Unacceptable type conversion - cannot make '*<target type>*' from '*<source type>*'
	Type:	ERROR
	Explanation:	This is an internal evaluation error. This should be reported to the CONVEX Technical Assistance Center.
249	Message:	Mixed mode conversion, '*<source type>*' to '*<target type>*'
	Type:	INFO
	Explanation:	This is an informational evaluation error. This message occurs as a result of the formation of an expression that contains floating point operand(s) defined in one floating point mode but used in a different mode. Evaluation proceeds as directed.
250	Message:	Incompatible type - operand of '*<unary operator>*' has incompatible type
	Type:	ERROR
	Explanation:	The operand of the unary operator has a type that is incompatible with unary operation.
251	Message:	Constant too large, specification of '*<constant>*' too large to be represented
	Type:	ERROR
	Explanation:	This message occurs if an integer constant specification is too large to be represented by the integral data types available in the language.

252 Message: Invalid use - '*<identifier>*' is not an array/function type
Type: ERROR
Explanation: This message occurs as a result of an illegal use of the identifier in a routine call or array expression. The identifier is not a routine or array designator.

253 Message: Subscript out of range - Value: *<subscript>* Range: *<lower bound>..<upper bound>*
Type: INFO
Explanation: The value of the subscript is not within the bounds of the dimension defined for this array.

254 Message: Invalid member - left operand of '.' does not have a member '*<field name>*'
Type: ERROR
Explanation: The expression on the left hand side of the selector operator designates a structure or union object that does not have a member identified by "field name".

255 Message: Incomplete type - size of type is unknown
Type: ERROR
Explanation: The referenced object has an incomplete type. No size information is available to permit evaluation to continue. This can occur as a result of a forward declaration in which the definition is not visible from the specified scope.

256 Message: Bad argument type - argument to intrinsic function has incorrect type
Type: ERROR
Explanation: Argument to the intrinsic function has a type that is inappropriate for use by the intrinsic function. Refer to the CONVEX Fortran Language Reference manual for a description of the intrinsic function.

257 Message: Too many arguments - too many arguments to intrinsic function
Type: ERROR
Explanation: Too many arguments have been passed to the intrinsic function. Refer to the CONVEX FORTRAN Guide for a description of the intrinsic function.

258 Message: Invalid CHARACTER length - arguments to ICHAR() must be of length 1
Type: ERROR
Explanation: The CHARCTER argument to the ICHAR() intrinsic function has a precision greater than one. Refer to the CONVEX FORTRAN Guide for a description of the intrinsic function.

No.		Description
259	Message:	Too few arguments - too few argments to intrinsic function
	Type:	ERROR
	Explanation:	Too few arguments have been passed to the intrinsic function. Refer to the CONVEX FORTRAN Guide for a description of the intrinsic function.
260	Message:	Bad argument type - cannot take the address of a CXdb variable
	Type:	ERROR
	Explanation:	An attempt to obtain the address of a CXdb debugger variable has been discovered. CXdb variables do not exist in the virtual memory space of the debugged process.
261	Message:	Undefined result - mathematical result is undefined
	Type:	ERROR
	Explanation:	The formation of the expression produces a result that is mathematically undefined. Verify the values of the components in the expression to ensure the integrity of the result.
262	Message:	Incompatible type - operands of '*<operator>*' point to incompatible types
	Type:	ERROR
	Explanation:	The operands of the specified operator refer to incompatible types.
263	Message:	Unsupported feature: *<message>*
	Type:	ERROR
	Explanation:	This functionality has been omitted from this release of CXdb.
264	Message:	CXdb type information unavailable
	Type:	INFO
	Explanation:	The evaluator is unable to obtain the necessary type information to provide assistance in further debugging. Recompile the file in the present scope (if possible) using the "-cxdb" flag to permit the evaluator to gain access to the missing type information.
265	Message:	Invalid scope specification - descending scope path specification requires a descendent block
	Type:	ERROR
	Explanation:	To look "forward" in scope, one must choose the desired scope block (descendent) from which the target identifier is visible.
266	Message:	Incompatible type - left operand of '(:)' is not a CHARACTER type
	Type:	ERROR
	Explanation:	The substring operator requires its left operand to have CHARACTER type. Refer to the CONVEX FORTRAN Guide for a description of the operation.

267 Message: Incompatible type - operands of '*<operator>*' have incompatible types
Type: ERROR
Explanation: The operands of the reported operator have incompatible types. Refer to either the CONVEX FORTRAN Guide or the CONVEX C Guide for a description of the operation.

268 Message: Lvalue required
Type: ERROR
Explanation: The operation requires that the left operand of the assignment operator designate a modifiable object.

269 Message: No such scalar register < *register number*>.
Type: ERROR
Explanation: An attempt has been made to reference a nonexistent scalar register.

270 Message: No such vector register < *register number*>.
Type: ERROR
Explanation: An attempt has been made to reference a nonexistent vector register.

271 Message: No such address register < *register number*>.
Type: ERROR
Explanation: An attempt has been made to reference a nonexistent address register.

272 Message: Can't reference CXdb variable < *variable name*>.
Type: ERROR
Explanation: An error has been encountered in the referencing of a CXdb variable.

273 Message: Error in accessing vector registers.
Type: ERROR
Explanation: An address was encountered that is outside the range of CXdb's cache of vector register variables.

274 Message: Invalid repeat count: <*count*>
Type: ERROR
Explanation: The repeat count specified in the command was invalid. Repeat counts for all 'step', 'next', and 'continue' commands must be greater than zero.

275 Message: Address <*hex address*> is not within the text region of the process: [<*text start*>..<*text end*>]

Type: ERROR

Explanation: The address you specified is not within the defined text region of the process image. The 'info objectmap' command can be used to determine the object file regions that make up the text segment.

276 Message: Find pattern too long (<*length*>), max is <*max length*>

Type: ERROR

Explanation: The pattern you specified to the 'find' command was too long. The maximum length is indicated in the message. Please try the command again with a shorter pattern.

277 Message: Invalid find pattern length (<*length*>), must be an even number.

Type: ERROR

Explanation: The pattern you specified to the find memory command was an odd number of characters. The 'find memory' command only works on whole bytes, so you must specify an even number of hex characters. Please try the command again with a pattern specifying an even number of characters.

278 Message: Invalid find pattern character (<*char*>), must be a hex digit

Type: ERROR

Explanation: The pattern you specified to the 'find memory' command contains an invalid character. The pattern may only be composed of an even number of hexadecimal digits (representing a byte pattern in memory). The character that was invalid was included in the error message. Please try the command again with a pattern specifying only hex digits.

279 Message: Corrupt name space table for <*filename*>

Type: INFO

Explanation: A name space table could not be opened. Therefore, check to see if it exists. If it does, then possibly some of the internal data representations have been corrupted.

280 Message: Unable to find CDI name space table for <*filename*> in search path; use 'add path' command.

Type: INFO

Explanation: A name space table could not be found. This file is generated by the compiler to inform CXdb about symbolic information. Therefore, check to see if it exists. If it does, then make sure that it is within one of directories in the search path. Reference the 'add path' command for adding another directory to your search path.

281 Message: <*This operation*> only works with the CXdb Maryland Windows interface.
Type: ERROR
Explanation: This operation's interface requires CXdb's Maryland Window interface to function. If you are using the 'bind' or 'info bind' commands, they are not available in batch mode and they are set via the X resources interface for the X Window System.

282 Message: Ambiguous specification - '<*keyword prefix*>' is not unique among:<*keyword list*>
Type: ERROR
Explanation: The abbreviated command keyword prefix matches more than one possible keyword. Examine the list of possible keywords to determine a specification that uniquely specifies the keyword desired.

283 Message: Process memory read failure, address: <*address*> Cause: <*errno description*>
Type: ERROR
Explanation: An attempt to read from the virtural memory space of the debugged process failed. The exact cause is derived from the failure code returned by errno which is used to obtain the text associated with error. Refer to the errno.h(3) man page for further details describing the failure code.

284 Message: Process memory write failure, address: <*address*> Cause: <*errno description*>
Type: ERROR
Explanation: An attempt to write to the virtual memory space of the debugged process failed. The exact cause is derived from the failure code returned by errno which is used to obtain the text associated with error. Refer to the errno.h(3) man page for further details describing the failure code.

285 Message: Examine does not support <*format*>
Type: ERROR
Explanation: An attempt was made to specify a memory size and format type combination that is not supported.

286 Message: Conflicting floating point mode and format type.
Type: ERROR
Explanation: An attempt was made to specify both a floating point mode and a nonfloating point format.

287 Message: Bad floating point precision specification.
Type: ERROR
Explanation: An attempt was made to specify an unsupported or incorrect floating point width and precision specifier.

No.		Description
288	Message:	Incomplete type - pointer expression references an incomplete object type
	Type:	ERROR
	Explanation:	Pointer expression references an object of type 'void' or an incomplete type. Refer to the 'info expression' command to obtain the type referred to through the pointer expression.
289	Message:	Symbolic location read failure - cause: *<errno description>*
	Type:	ERROR
	Explanation:	An attempt to read from the symbolic location of the program data specified at line/column in the debugged process failed. The exact cause is derived from the failure code returned by errno which is used to obtain the text associated with error. Refer to the errno.h(3) man page for further details describing the failure code.
290	Message:	Symbolic location write failure - cause: *<errno description>*
	Type:	ERROR
	Explanation:	An attempt to write to the symbolic location of the program data specified at line/column in the debugged process failed. The exact cause is derived from the failure code returned by errno which is used to obtain the text associated with error. Refer to the errno.h(3) man page for further details describing the failure code.
291	Message:	*<feature>* only available on C2 architectures.
	Type:	ERROR
	Explanation:	The feature indicated in the error message text is only available on the C2 Series architecture. Either the core file you are debugging is not from a process that was running on a C2, or the currrently running process is not on a C2.
292	Message:	Ambiguous specification - '*<field name prefix>*' is not unique among:*<field list>*
	Type:	ERROR
	Explanation:	The abbreviated structure or union field prefix matches more than one possible field name. Examine the list of possible field names to determine a specification that uniquely specifies the field desired.

293 Message: Permanent memory allocations can't be done without 'sbrk' linked in the target

Type: ERROR

Explanation: You tried to perform an operation that required permanently allocating memory within the target process' address space. For example, this can arise when performing assignments of character strings to 'char *' types in expressions. However, the target image does not contain the 'sbrk' library function. Memory allocations can not occur without this routine. You might consider relinking your application to include this library function, or not performing the operation that requires memory allocation.

294 Message: Allocation of <*byte count*> bytes in the target process failed.

Type: ERROR

Explanation: You tried to perform an operation that required permanently allocating memory within the target process' address space. For example, this can arise when performing assignments of character strings to 'char *' types in expressions. However, the operation to allocate this memory failed. This could be due to several reasons: swap space is exhausted, the program has exceeded its data size limit (as set by setrlimit(2), or the maximum possible size of a data segment (compiled into the system) was exceeded.

295 Message: Process [#<*process number*>] is already stopped.

Type: INFO

Explanation: You tried to stop the execution of a process when it was already stopped. Use the 'info process' command to determine the complete status of the process.

296 Message: No process image - a process image must be available to evaluate string literals

Type: ERROR

Explanation: All string literals are allocated in the debugged processes memory space. The evaluator cannot evaluate the literals in the expression without a process image. Refer to either the 'run', 'rerun', or 'attach' command for details on obtaining a process image.

297 Message: Memory Allocation failure - cannot allocate memory in the process' memory space

Type: ERROR

Explanation: All string literals are allocated in the debugged process' memory space. An attempt to allocate memory in the process' virtual memory space failed.

No.		Description
298	Message:	Window [#*<window id>*] not found.
	Type:	ERROR
	Explanation:	The window ID you specified does not correspond to any currently active window. Please check the window ID and try the command again.
299	Message:	Window [#*<window id>*] is not a source window
	Type:	ERROR
	Explanation:	You specified a window that is not a source window in a command which only operates on source windows. Please check the window ID and try the command again.
300	Message:	No command matching '*<text>*' found
	Type:	ERROR
	Explanation:	The text you specified in the 'recall' command did not match any of the commands currently in the command history. Please check the text you used and try the command again.
301	Message:	Operation not available in batch mode.
	Type:	ERROR
	Explanation:	You requested an operation that is not supported in batch mode. Consider using one of the interactive modes if you must use the feature.
302	Message:	Process has multiple threads. Terminated.
	Type:	ERROR
	Explanation:	The process you were debugging spawned additional threads. This release of CXdb does not support multi-threaded applications. To prevent incorrect debugger behavior, the process was terminated.
303	Message:	On-line help directory '*<directory name>*' not found.
	Type:	ERROR
	Explanation:	The online help directory specified does not exist. Either the online help database was not installed properly or it was deleted.
304	Message:	There is no previous command in the command history.
	Type:	ERROR
	Explanation:	The command history is empty, so there is no previous command to recall.

305 Message: The environment provided to CXdb is corrupt. The first invalid entry is '*<string>*'.

Type: ERROR

Explanation: When CXdb is started, it is provided with a set of environment variables by the shell that starts it. This environment becomes the default environment in CXdb. Each entry in the environment is of the form 'name=string'. The environment provided to CXdb when it was started contains at least one entry not of the proper form.

306 Message: Viewport '*<string>*' is not on the viewport list.

Type: ERROR

Explanation: An attempt was made to remove a file viewport which did not exist on the viewport list specified. Check the current value of the viewport list with the 'info cxdb' command.

307 Message: Viewport window '*<id>*' is not on the viewport list.

Type: ERROR

Explanation: An attempt was made to remove a window viewport which did not exist on the viewport list specified. Check the current value of the viewport list with the 'info cxdb' command.

308 Message: Viewport stream '*<string>*' is not on the viewport list.

Type: ERROR

Explanation: An attempt was made to remove a stream viewport which did not exist on the viewport list specified. Check the current value of the viewport list with the 'info cxdb' command.

309 Message: A null viewport file name was passed internally.

Type: ERROR

Explanation: An internal error was discovered when a NULL viewport file name was passed. The command was aborted.

310 Message: The invalid viewport window ID '*<id>*' was passed internally.

Type: ERROR

Explanation: An internal error was discovered when an invalid viewport window ID was passed. The command was aborted.

311 Message: A null viewport stream was passed internally.

Type: ERROR

Explanation: An internal error was discovered when a NULL viewport stream was passed. The command was aborted.

No.		Description
312	Message:	Xterm process <*process-id*> exec'ed.
	Type:	ERROR
	Explanation:	The xterm process that CXdb used to start the target process has performed an exec system call. This makes it impossible for CXdb to continue properly controlling the target process, so it has been terminated.
313	Message:	Shell process <*process-id*> exec'ed.
	Type:	ERROR
	Explanation:	The shell process that CXdb used to start the target process has performed an exec system call. This makes it impossible for CXdb to continue properly controlling the target process, so it has been terminated.
314	Message:	An unknown FormatCode was specified.
	Type:	ERROR
	Explanation:	An internal error was discovered during the formatting of a piece of data. This should be reported to the CONVEX Technical Assistance Center.
315	Message:	An unknown type descriptor was specified.
	Type:	ERROR
	Explanation:	An internal error was discovered during the formatting of a piece of data. This should be reported to the CONVEX Technical Assistance Center.
316	Message:	REAL*16 cannot be represented in IEEE floating point mode.
	Type:	INFO
	Explanation:	REAL*16 data types are only supported using native floating point mode.
317	Message:	Operand incompatible with /C format.
	Type:	INFO
	Explanation:	An attempt was made to format data as a complex floating point number. This failed either because the size of the data made it inappropriate for use as a complex floating point number, or because the data itself was incompatible with being formatted as a complex floating point.
318	Message:	Mixed mode conversion from '<*source type*>' to '<*target type*>'
	Type:	INFO
	Explanation:	This message occurs as a result of a specification of a floating point mode which is different from the default floating point mode. Evaluation proceeds as specified.
319	Message:	Operand incompatible with /i format.
	Type:	INFO
	Explanation:	In order to format data as an instruction, the data must be at least two bytes in length.

320 Message: The pattern '<*pattern*>' was not found.
Type: ERROR
Explanation: The pattern given to the 'find window' command was not found in the specified source window.

321 Message: Syntax Error - <*expecting or missing*>
Type: ERROR
Explanation: A lexical error was encountered in the parsing of your command. If at all possible, this message will try to provide the missing lexical element. After reviewing your command entry and this error message, if you are unable to rectify your specification, please refer to the online help (type "help") for this command to obtain a presentation of its syntax.

322 Message: All references to synthesized variable < *id*> are optimized away
Type: INFO
Explanation: A front end synthesized variable has been optimized away and so the original value is used. Front end synthesized variables are created so that language semantics are strictly adhered to. In most cases, it is used to retain an old value of the variable so that the variable may be updated later. An example in FORTRAN is when a parameter's value is saved upon entry into the routine. The synthesized variable may be optimized away later when there are no assignments to the actual parameter.

323 Message: Scope Path required - descending scope specification is missing a block specifier
Type: ERROR
Explanation: Descending scope specifications (those that look forward in the scope chain) require at least one intervening block specifier before specifying the target identifier.

324 Message: Invalid Scope Specification - program scope has not yet been established
Type: ERROR
Explanation: Descending scope specifications (those that look forward in the scope chain) require the process to have an active scope. This message can occur if the process has not yet been started and therefore has undefined scope. Use the 'run' or 'rerun' command to start the process combined with an eventpoint to stop the running process to establish an active scope, or, provide a complete scope path by prefixing it with a block specifier which is global to the entire program and descending from it.

325 Message: Operand incompatible with format.
Type: INFO
Explanation: Data which require sixteen bytes of storage, such as REAL*16 and COMPLEX*16, cannot be formatted as an integral type.

326 Message: Data file < *file name*> is out of date: last compiled <*date*>, created <*data file date*>

Type: ERROR

Explanation: A data file can become out of date with respect to an executable, if the data file is regenerated (by recompiling the source file) after the executable has been linked. Relinking the executable should correct the problem. If not, make sure the search path is properly set.

327 Message: Incompatible type - left operand of '()' does not identify a function

Type: ERROR

Explanation: The operand of '()' does not have function type. This message may appear as a result of the operand not being visible from the current scope, a typing error, or an illegal use of an operand which does not have type "function returning ...".

328 Message: Data cannot be formatted as floating point.

Type: INFO

Explanation: An attempt was made to format data as a floating point number. This failed either because the size of the data made it inappropriate for use as a floating point number, or because the data itself was incompatible with floating point format.

329 Message: Syntax Error - illegal slice range specifier

Type: ERROR

Explanation: A slice range has been recognized in a context to which it is not applicable. Slice ranges are only applicable in array expressions. A slice range is used in place of a subscript in an array expression to define the subset of the array's bounds.

330 Message: Specified thread <*thread-id*> ignored.

Type: INFO

Explanation: A thread that was specified on the command line has been ignored in the processing of this command. The specified thread is not currently active.

331 Message: Invalid upper bound

Type: ERROR

Explanation: The upper bound of the specified slice range must either be equal to or greater than the specified lower bound.

332 Message: No process image; accesses to memory require a process image.
Type: ERROR
Explanation: The commanded operation cannot continue without a process image. Refer to either the 'run' or 'attach' commands for details on obtaining a process image.

333 Message: The Specified Region (<*region size*>) is too large.
Type: ERROR
Explanation: The region you specified in a watchpoint or modify eventpoint is too large. CXdb must make a copy of the region being monitored to determine when it changes. The acquisition of the memory to hold this copy failed. You must specify a smaller region to monitor.

334 Message: Invalid Slice Expression - left operand of '[]' is not an array type
Type: ERROR
Explanation: Slice expressions require the left operand of '[]' to have type "array of ...". Either the operand is not an array type, or its type has been defined with assumed bounds (the operand is possibly a pointer). Cast pointer types to array of ..., defining the bounds of the array.

335 Message: All threads selected in Process [#<*process number*>] are exiting.
Type: ERROR
Explanation: All the threads that you selected for execution within the process indicated are currently in the process of exiting (due to a join, wfork, or idle instruction). These threads can't be executed by themselves. At least one non-exiting thread must be executed along with these threads.

336 Message: Invalid type - precision of integer type too small or too large
Type: ERROR
Explanation: Integer constants (and loader symbols) used as function designators, require a precision of four bytes. The result of the subexpression to be used as the function designator is either greater than or is less than this precision. You can obtain the precision the evaluator has arrived at by using the 'info expression' command and passing it the function designator's expression, that is the left operand of the ()'s.

337 Message: Illegal pointer arithmetic
Type: ERROR
Explanation: Function pointers are not permitted in arithmetic expressions. For more information, refer to the CONVEX C Guide manual.

No.		Description
338	Message:	Inappropriate value <*maxarray*> for printopts ignored.
	Type:	INFO
	Explanation:	An inappropriate value was specified for the 'printopts maxarray' setting. Maxarray must be greater than zero and less than 2147483648.
339	Message:	File < *name*> compiled with prerelease compiler, recompile
	Type:	INFO
	Explanation:	The source file was compiled with a prerelease version of the FORTRAN compiler. Please recompile the specfic module in order to have access to debugging information.
340	Message:	Wildcard specifications are not allowed in this command.
	Type:	ERROR
	Explanation:	You specified a wildcard character (*) in a command that does not allow it. Please refer to the reference guide or online help system for the correct syntax of the command.
341	Message:	Memory request failure
	Type:	ERROR
	Explanation:	The evaluation of array slice expressions require heap allocation in the inferior process. This request for space has been denied for one of two reasons; either there is no additional heap space available, or the allocation request will exhaust the heap.
342	Message:	Character format converted memory size to byte
	Type:	INFO
	Explanation:	The character format is available only in byte size. If any other memory size was chosen, it was ignored.
343	Message:	Encountered <*a large number of*> <*args or locals*>. Processing <*a default number*>.
	Type:	INFO
	Explanation:	A large number of arguments or locals was encountered. This number is ignored in the processing of the command. However, the first few arguments or locals are accessed and processed.
344	Message:	There are no text lines in < *this file*>.
	Type:	ERROR
	Explanation:	There are no lines of text in this file, although the source file is known to exist. Try updating the source search path so that the correct source file is found.

345 Message: There are no source units for < *this file*>.

Type: ERROR

Explanation: There are no source units in this file, although the source file is known to exist. Try updating the source search path so that the correct data files are found.

346 Message: Reading data file <*name*>, <*type*>.

Type: INFO

Explanation: Data files are read incrementally throughout the user's debugging session. Currently, a large data file is in the process of being read into memory. Therefore, an unusual delay will occur in this command.

347 Message: Expression too complex - expr: <*expression*>

Type: ERROR

Explanation: The language expression is too complex to be parsed completely. This condition can occur if: the expression is arbitrarily large; the expression is nested too deeply; or the expression is the result of successively expanding a recursive macro. The entire preprocessed version of the expression being parsed is provided so that the effect of preprocessing can be observed.

348 Message: Command line too complex

Type: ERROR

Explanation: The CXdb command line is too complex to be parsed completely. This condition can occur if a CXdb command is nested too deeply or the command has become arbitrarily too complex as a result of successively expanding a recursive macro (or alias). The entire preprocessed version of the command line being parsed is provided so that the effect of preprocessing can be observed.

349 Message: Initializer expression type is not compatible with '<*memory type*>'.

Type: ERROR

Explanation: The data type obtained from the initializer expression is not compatible with the optional memory type specified on the command. CXdb cannot safely determine how to initialize the memory region specified when the data type of the initializer is incompatible with the specified optional memory type. If the expression syntax is C, try casting the initializer expression to match the precision of the memory type specified. If FORTRAN, try invoking one of CXdb's built-in intrinsics to convert the initializer to the precision of the specified memory type (such as INT(), REAL(), QEXT(), CMPLX(), DCMPLX(), etc...). Use the 'info expression' command to determine the data type and precision of the initializer expression if needed.

350 Message: Precision of COMPLEX initializer is not compatible with '*<memory type>*'.

Type: ERROR

Explanation: The precision of the COMPLEX data type obtained from the initializer expression is not compatible with the optional memory type specified on the command. CXdb cannot safely determine how to initialize the memory region specified when the data type of the initializer is incompatible with the specified optional memory type. Try invoking one of CXdb's built-in FORTRAN intrinsics to convert the initializer to the precision of the specified memory type (such as INT(), REAL(), QEXT(), CMPLX(), DCMPLX(), etc...). Use the 'info expression' command to determine the data type and precision of the initializer expression if needed.

351 Message: Precision of floating point initializer is not compatible with '*<memory type>*'.

Type: ERROR

Explanation: The precision of the floating point data type obtained from the initializer expression is not compatible with the optional memory type specified on the command. CXdb cannot safely determine how to initialize the memory region specified when the data type of the initializer is incompatible with the specified optional memory type. If the expression syntax is C, try casting the initializer expression to match the precision of the memory type specified. If FORTRAN, try invoking one of CXdb's built-in FORTRAN intrinsics to convert the initializer to the precision of the specified memory type (such as INT(), REAL(), QEXT(), CMPLX(), DCMPLX(), and so on). Use the 'info expression' command to determine the data type and precision of the initializer expression if needed.

352 Message: Partial initialization, *<starting address>..<ending address>* (*<memory range>* bytes) too small for initializer (*<initializer>* bytes).

Type: INFO

Explanation: The specified memory region to be initialized does not contain enough space to permit an even multiple of stores. The last element cannot be written without exceeding the region specified. Therefore, these remaining bytes will not be initialized.

353 Message: Precision of initializer exceeds memory region, initializer: *<element size>* memory region: *<memory region>*.

Type: ERROR

Explanation: The precision of the initializer expression is greater than the capacity of the specified memory region. If the expression syntax is C, try casting the initializer expression to match the precision corresponding to the capacity of the specified memory region. If FORTRAN, try invoking one of CXdb's built-in FORTRAN intrinsics to convert the initializer to a precision corresponding to the capacity of the memory region (such as INT(), REAL(), QEXT(), CMPLX(), DCMPLX(), and so on).

354 Message: Invalid evalopts precision setting, must be 4 or 8.

Type: ERROR

Explanation: The 'evalopts' precision setting you entered is invalid. The values of 'iprecision' and 'rprecision' must be either 4 or 8. These values are used to determine the default size of integer and floating point constants in expression evaluation.

355 Message: Invalid ignore count: *<count>*

Type: ERROR

Explanation: The ignore count you specified is invalid. It must be greater than or equal to zero. If you used a debugger variable to specify the ignore count, rounding or truncation affects may have caused this result. The debugger variable value is implicitly converted to an integer before it is used.

356 Message: Too many statements, input: *<command>*

Type: ERROR

Explanation: The CXdb command line has too many statements. This condition is usually the result of an alias or macro expansion. If an alias or macro definition contains multiple statements, and the last statement is another macro or alias that is directly or indirectly recursive, the preprocessor and parser can loop indefinitely. This message is displayed if the parser believes this to be the case. The entire preprocessed version of the command line being parsed is displayed so that the effect of preprocessing can be observed.

357 Message: The '*<command string>*' command expects a *<composition type>*, but received a *<composition type>*

Type: ERROR

Explanation: The command being completed with the composition expected a specific type, but received another. The composition display tells what kind of type is expected.

No.		Description
358	Message:	Process [#*<process>*] has invalid state (*<routine>*): *<state>*
	Type:	ERROR
	Explanation:	This is an internal error within CXdb. It should be reported to the Technical Assistance Center immediately. In general, CXdb has detected that the state of the process control information is inconsistent. This will almost assuredly lead to further errors.
359	Message:	PIXRUN error on process [#*<process>*]: *<error text>*
	Type:	ERROR
	Explanation:	This is an internal error within CXdb. It should be reported to the Technical Assistance Center immediately. CXdb was attempting to restart the execution of the target process, and the system call failed. The reason for failure is included in the error message.
360	Message:	Error trying to *<get/set>* process attributes on process [#*<process>*]: *<error text>*
	Type:	ERROR
	Explanation:	This is an internal error within CXdb. It should be reported to the Technical Assistance Center immediately. CXdb was attempting to get or set the process attributes of the process indicated, and the system call failed. The reason for failure is included in the error message.
361	Message:	Error trying to *<clear/set>* the inherit mode on process [#*<process>*]: *<error text>*
	Type:	ERROR
	Explanation:	This is an internal error within CXdb. It should be reported to the Technical Assistance Center immediately. CXdb was attempting to clear or set the inherit mode of the process indicated, and the system call failed. The reason for failure is included in the error message.
362	Message:	Error trying to detach process [#*<process>*]: *<error text>*
	Type:	ERROR
	Explanation:	This is an internal error within CXdb. It should be reported to the Technical Assistance Center immediately. CXdb was attempting to detach from the process indicated, and the system call failed. The reason for failure is included in the error message.
363	Message:	Error trying to get signal subcode on thread [#*<thread-id>*]: *<error text>*
	Type:	ERROR
	Explanation:	This is an internal error within CXdb. It should be reported to the Technical Assistance Center immediately. CXdb was attempting to signal subcode of the thread indicated, and the system call failed. The reason for failure is included in the error message.

364 Message: Error trying to <*step/continue*> thread [#<*thread-id*>]: <*error text*>

Type: ERROR

Explanation: This is an internal error within CXdb. It should be reported to the Technical Assistance Center immediately. CXdb was attempting to step or continue the execution of the thread indicated, and the system call failed. The reason for failure is included in the error message.

365 Message: Error trying to select thread [#<*thread-id*>]: <*error text*>

Type: ERROR

Explanation: This is an internal error within CXdb. It should be reported to the Technical Assistance Center immediately. CXdb was attempting to select the thread indicated for read and write access, and the system call failed. The reason for failure is included in the error message.

366 Message: Error trying to get CWD for process [#<*process*>]: <*error text*>

Type: ERROR

Explanation: This is an internal error within CXdb. It should be reported to the Technical Assistance Center immediately. CXdb was attempting to get the current working directory for the process indicated, and the system call failed. The reason for failure is included in the error message.

367 Message: Error trying to get proc structure for process [#<*process*>]: <*error text*>

Type: ERROR

Explanation: This is an internal error within CXdb. It should be reported to the Technical Assistance Center immediately. CXdb was attempting to get the process control structure for the process indicated, and the system call failed. The reason for failure is included in the error message.

368 Message: Error trying to get thread count for process [#<*process*>]: <*error text*>

Type: ERROR

Explanation: This is an internal error within CXdb. It should be reported to the Technical Assistance Center immediately. CXdb was attempting to get the number of threads in the process indicated, and the system call failed. The reason for failure is included in the error message.

369 Message: Process [#<*process*> with pid <*pid*> is terminated.

Type: ERROR

Explanation: During the process start-up phase CXdb tried to handle a process for the first time and it was found terminated. This will usually lead to a general start-up failure, and the target process will probably not start. This may also leave the process control data in an inconsistent state. Use the 'kill process' command to try to resolve the problem.

370 Message: Process [#<*process*> with pid <*pid*> in unknown wait state.
Type: ERROR
Explanation: During the process start-up phase CXdb tried to handle a process for the first time and it had un unknown state as indicated by the wait system call return value. This usually leads to a general start-up failure, and the target process will probably not start. This can also leave the process control data in an inconsistent state. Use the 'kill process' command to try to resolve the problem.

371 Message: Error trying to get uarea for process [#<*process*>]: <*error text*>
Type: ERROR
Explanation: This is an internal error within CXdb. It should be reported to the Technical Assistance Center immediately. CXdb was attempting to get the user area structure for the process indicated, and the system call failed. The reason for failure is included in the error message.

372 Message: Error trying to pattach process with pid <*pid*>: <*error text*>
Type: ERROR
Explanation: This is an internal error within CXdb. It should be reported to the Technical Assistance Center immediately. CXdb was attempting to pattach to the process indicated, and the system call failed. The reason for failure is included in the error message.

373 Message: Event <*event-id*> is already disabled
Type: INFO
Explanation: The eventpoint indicated in the message is already disabled. Trying to disable it again has no real effect.

374 Message: Event <*event-id*> is already enabled
Type: INFO
Explanation: The eventpoint indicated in the message is already enabled. Trying to enable it again has no real effect.

375 Message: Descending slice (or substring) range - lower bound: <*lower bound*> upper bound: <*upper bound*>
Type: ERROR
Explanation: The lower bound must be less than or equal to the upper bound for the slice (or substring) expressions.

376 Message: Window functions can not be rebound to different keys.
Type: ERROR
Explanation: An attempt was made to bind a window function to a key. Only command line editing functions may be rebound.

377 Message: '*<function name>*' is an unknown command line editing function.
Type: ERROR
Explanation: An unknown command line editing function was specified as an argument to the 'bind' command. Please see the CXdb reference page 'function-name' for a list of the available command line editing functions.

378 Message: The key name *<key name>* is unknown.
Type: ERROR
Explanation: An unknown key name was specified as an argument to the 'bind' command. Please see the CXdb reference page 'key-name' for a description of the valid key names.

379 Message: The command string given to 'recall' was NULL.
Type: ERROR
Explanation: The string given to the 'recall' command was NULL. A non-NULL string is required for the 'recall' command.

380 Message: The number of history elements '*<number>*' to print is invalid.
Type: ERROR
Explanation: The number of history elements to print is invalid. A zero or negative value was given.

381 Message: No object file object was passed to a source window screen update function.
Type: ERROR
Explanation: An internal error occurred when a compiler-debugger interface object file object was not given to a source window screen update function.

382 Message: There is no active object file at the current point of execution.
Type: INFO
Explanation: There is no compiler-debugger interface information for any of the frames currently on the stack, so the source window could not be updated to reflect the current point of execution.

383 Message: There is no source file corresponding to the current CDI object file.
Type: ERROR
Explanation: An internal error occurred because there is no source file corresponding to the currently active compiler-debugger interface object file.

384 Message: No source file object was passed to a source window screen update function.

Type: ERROR

Explanation: An internal error occurred when a compiler-debugger interface source file object was not given to a source window screen update function.

385 Message: The string given to the 'find window' command was NULL.

Type: ERROR

Explanation: A NULL string was given to the 'find window' command. The 'find window' command requires a search string.

386 Message: No source unit exists in an internal source unit tree.

Type: ERROR

Explanation: An internal error occurred when a source unit tree node did not contain a source unit when one was expected. The attempted operation failed.

387 Message: No source unit exists at line <*line number*> column <*column number*> of the file '<*source file*>'.

Type: ERROR

Explanation: An attempt was made to select a source unit at a source position that has no source units associated with it. Try selecting the source unit again at a source position with executable code.

388 Message: No parent could be found for the currently selected source unit.

Type: ERROR

Explanation: No parent source unit could be found because there is no source unit tree at the position of the currently selected source unit.

389 Message: The currently selected source unit has no parent source unit.

Type: ERROR

Explanation: The currently selected source unit has no parent source unit. The most likely cause is that the currently selected source unit is a routine.

390 Message: No child could be found for the currently selected source unit.

Type: ERROR

Explanation: No child source unit could be found because there is no source unit tree at the position of the currently selected source unit.

391 Message: The currently selected source unit has no child source unit.

Type: ERROR

Explanation: The currently selected source unit has no child source units. The most likely cause is that the currently selected source unit is an innermost source unit.

392 Message: The currently selected source unit has no sibling source unit.
Type: ERROR
Explanation: The currently selected source unit has no further sibling source units; however, it may have other siblings. These other sibling source units can be found by finding the current source unit's parent and then selecting the parent's child and further sibling source units.

393 Message: There is no stack frame with CDI information at the current point of execution.
Type: ERROR
Explanation: No routines currently on the stack at or above the currently selected stack frame have compiler-debugger interface information. This means that the view in the source window may be out of date.

394 Message: Highlighting failed from line <*line*>, column <*column*>, to line <*line*>, column <*column*>.
Type: ERROR
Explanation: An internal error occurred when the range for highlighting was out of the bounds on the window being operated on.

395 Message: Unhighlighting failed from line <*line*>, column <*column*>, to line <*line*>, column <*column*>.
Type: ERROR
Explanation: An internal error occurred when the range for unhighlighting was out of the bounds on the window being operated on.

396 Message: The address expression '<*expression*>' does not match any routine with CDI information.
Type: ERROR
Explanation: The address expression given for displaying a new routine in the source window did not evaluate to be within the range of any routine known to the compiler-debugger interface.

397 Message: The line number '<*line*>' is out of the bounds of the source file.
Type: INFO
Explanation: The line number given in the file name/line number specification was out of the range of line numbers for the given source file. The line number '1' was assumed.

398 Message: The file name '<*file*>' is an unknown source file.
Type: ERROR
Explanation: The file name given in the file name/line number specification was a source file unknown to the compiler-debugger interface.

No.		Description
399	Message:	Nested handler for eventpoint #<*event-id*> aborted.
	Type:	ERROR
	Explanation:	The eventpoint handler for the eventpoint indicated was invoked while another eventpointhandler was active. This is is possible when eventpoint handlers cause function calls to be evaluated. The execution of eventpoint handlers can not be nested so the indicated eventpoint handler was aborted.
400	Message:	Process [#<*process-id*>/<*thread*>] stopped by <*signal*> can't be restarted.
	Type:	ERROR
	Explanation:	There are hardware generated signals (SIGBUS, SIGSEGV, and SIGILL) which can not be recovered from. Once one of these signals is generated further execution of the thread is not possible. There is no way to work around this restriction.
401	Message:	Integer overflow on <*number*>.
	Type:	ERROR
	Explanation:	All numbers used in CXdb commands (for repeat counts, line numbers, ignore counts, etc.) must be less than 2147483647 (the maximum value of a 4-byte signed integer). The number you entered is larger than this. Please reenter the command with a valid number.
402	Message:	Return value expression has an invalid type: <*type*>
	Type:	ERROR
	Explanation:	You have specified a return value that is incompatible with the current function. Please verify the return type of the function and try the command again. The 'info expression' command can give you more details on both the function name and the expression you are trying to use as a return value.
403	Message:	The current function has void return type. Result ignored.
	Type:	INFO
	Explanation:	You have specified a return value for a function which has a void return type. The return value you specified is being ignored. The function will still be forced to return.
404	Message:	No CDI information on current function. Assuming <*type*> return type.
	Type:	INFO
	Explanation:	There is no CDI information pertaining to the current function. The return type specified has been assumed for conversion of the return value you specified.

405 Message: Initial process failed to exec; cause: <*reason*>

Type: ERROR

Explanation: During the start-up of the target process something went wrong. CXdb initially forked but was unable to exec the command to create the target process. The reason for the failure is indicated. This generally indicates something is wrong with the installation of CXdb.

406 Message: <*type-of*> process failed to exec. Startup terminated.

Type: ERROR

Explanation: During the startup of the target process something went wrong. One of the processes used to start the target process has exited for some reason. It normally comes from not being able to exec the image for some reason. Typical causes are the file is marked executable but is not in a format that the kernel will exec; the executable is not meant for the current architecture (such as running a C2 executable on a C1); or a system limit may have been exceeded. The startup process will be terminated.

407 Message: Unable to read line <*line number*> from file <*file*>

Type: ERROR

Explanation: While trying to read from the source file indicated, an error was encountered. This may be due to the file having been modified since it was compiled (which may lead CXdb to try to access portions of the file that no longer exist).

408 Message: Unable to find CDI expression table information for <*filename*> in search path; use 'add path' command.

Type: ERROR

Explanation: A synthesized variable table could not be found. This file is generated by the compiler to inform CXdb about symbolic information. Therefore, check to see if it exists. If it does, then make sure that it is within one of directories in the search path. Reference the 'add path' command for adding another directory to your search path.

409 Message: Corrupt synthesized variable table for <*filename*>

Type: ERROR

Explanation: A synthesized table could not be opened. Therefore, check to see if it exists. If it does, then possibly some of the internal data representations have been corrupted.

410 Message: Inconsistent induction value for < *name*>, using <*value*>, from the set of {<*list of identifiers*>}

Type: INFO

Explanation: Several equations are used to determine the loop induction value. Sometimes, these equations may become inconsistent. Therefore, all calculated values are displayed and the lowest value is arbitrarily chosen to be used in expressions.

411 Message: Inconsistent induction value for <*identifier*> from the set of {<*list of identifiers*>}

Type: ERROR

Explanation: Several equations are used to determine the loop induction value. Sometimes, one of these equations may become inconsistent. Therefore, all calculated values are displayed; however, the operation cannot proceed because of this inconsistency.

412 Message: Calculated induction value for following <*number of*> variable(s): <*list of identifiers*>

Type: INFO

Explanation: Calculated the induction value for the variable and stored the value in the above mentioned variables. That is, the induction variable was removed and replaced with the above-mentioned synthesized variables.

413 Message: Unique object file <*name*> is not found.

Type: ERROR

Explanation: The unique name of the file was not found. This may be a result of two reasons. One reason is that no object file has that name. Another reason is that more than one object file has that identical name.

414 Message: The file <*name*> is invalid: <*reason*>

Type: ERROR

Explanation: The file specified as an object file was invalid for the indicated reason. This is typically because the file is not in a supported object file format.

415 Message: Breakpoint address <*address*> not in any defined text region

Type: INFO

Explanation: The address that you specified as a breakpoint address does not lie within any of the defined text regions of the program. This may be an error unless you are somehow intending to execute code not within the defined text areas as can happen with dynamically loaded object code.

416 Message: alias id '*<id>*' is a prefix for alias '*<other id>*'
Type: ERROR
Explanation: The alias name that you tried to define is a prefix of an alias that is already defined. It is illegal to define an alias that is a prefix of another alias. For example, it is illegal to have aliases 'foo' and 'foo bar'. Please choose a new alias name.

417 Message: alias id '*<id>*' would be prefixed by alias '*<other id>*'
Type: ERROR
Explanation: There is already an alias defined that is a prefix of the alias name you just tried to define. It is illegal to define an alias which is a prefix of another alias. For example, it is illegal to have aliases 'foo' and 'foo bar'. Please choose a new alias name.

418 Message: Invalid automatic dynamic load data: *<reason>*
Type: ERROR
Explanation: CXdb will automatically process dynamically loaded object files whenever a function by the special name of '_cxdb_dynamic_load' is called. The arguments to this function are expected to contain the data necessary to load the CDI information for the object file. Some error was encountered while trying to process this data. The reason for the error is shown in the message.

419 Message: Assignment could not be performed on *<identifier>*
Type: ERROR
Explanation: The assignment operation could not be performed on the above variable. This variable either may not have storage assigned or may be a constant typed variable.

420 Message: Subscript not arithmetic type
Type: ERROR
Explanation: Subscript expressions must result in an arithmetic data type. Some examples of arithmetic types are: integer, logical, real and complex.

421 Message: Invalid operation! You may not write to a core file.
Type: INFO
Explanation: An operation has been attempted which would result in a write to a core file. Core files may not be written to, and the values of variables in a core file may not be modified.

No.		Description
422	Message:	Incompatible pointers for formal and actual parameter types
	Type:	INFO
	Explanation:	The type of the actual parameter is incompatible with the declaration of the formal parameter. However, these are both pointers and so execution continues.
423	Message:	All address expressions must have a size in the PUT command, < *parameters*> needs a size
	Type:	ERROR
	Explanation:	Every address expression in the 'put' command must have an identifiable size. This size may come from either the command line or default from the address expression. When it defaults from the variable, the address expression must be deterministic, like a pointer to an object or an array.
424	Message:	Cannot read/write to currently opened file.
	Type:	ERROR
	Explanation:	Can neither read nor write to the current file.
425	Message:	The file <*name*> is not a standard PUT/GET file.
	Type:	ERROR
	Explanation:	Only standard files created with the 'put' command may be read by the G command. This is required since the files are in a fixed binary format.
426	Message:	Cannot parse the expression stored in PUT file: < *file*>
	Type:	ERROR
	Explanation:	The textual expression is stored in the 'put/get' data file. After executing the GET command, the expression is reparsed. If the parsing environment is different, then a parser error may occur. Usually, it is always best to execute the GET command with the same macros and at the same program counter of the original PUT command. Otherwise, there is a good likelihood that parsing cannot be performed.
427	Message:	Difference in sizes for address expression: size from command line <*integer*>, size from data file <*integer*>
	Type:	INFO
	Explanation:	The size of the address expression is different than the size used to store the data in the data. The smaller of the two sizes will be used.
428	Message:	File '<*name*>' not found.
	Type:	ERROR
	Explanation:	The file you specified could not be located. The search path is not used in this context.

429 Message: The expression < *phrase*> has a bad size < *value*>
Type: ERROR
Explanation: The expression cannot have negative size. Therefore, redefine the size of the expression.

430 Message: Unable to find the following CDI files; use the "add path command" < *pathname*>
Type: ERROR
Explanation: CDI data files are incrementally loaded during the CXdb debugging session. The necessary file cannot be found and so you are informed about it. For CXdb to access the data file, you must issue the 'add path' command.

431 Message: Cannot have a file on this viewport list
Type: ERROR
Explanation: This command does not allow a file name on the viewport list.

432 Message: This expression cannot have its value determined
Type: ERROR
Explanation: This is the case when the value for the variable cannot be determined. There are several cases that cause this error message. For example, a dynamically bound FORTRAN string may have a bad value for the location of its size. Thus, the contents of the string cannot be determined.

433 Message: Could not create the disassembly window
Type: ERROR
Explanation: The disassembly could not be created. Currently, a disassembly window may only be created if there is a running process.

434 Message: Error reading from remote server: <*reason*>
Type: ERROR
Explanation: An error was encountered trying to read a packet from a remote server. This generally means that the connection to the remote server has been terminated or the network support services are failing for some reason. There is generally no recovery from this error.

435 Message: End-Of-File condition from remote server
Type: ERROR
Explanation: An EOF condition was encountered while trying to read a packet from a remote server. This generally means that the server has exited for some reason or the underlying network connection was terminated.

No.		Description
436	Message:	Malformed packet from remote server received, length = <*len*>
	Type:	ERROR
	Explanation:	A malformed packet was received from the remote server. This means that the communication between the server and the client has been corrupted or is out of sequence in some way. This condition will probably lead to more errors.
437	Message:	File <*name*> is not an executable file.
	Type:	ERROR
	Explanation:	The file you specifed was not in an executable file format that CXdb can recognize. Please be certain that you typed the name correctly. The operation was aborted.
438	Message:	Can't connect to remote host <*name*>.
	Type:	ERROR
	Explanation:	An error occurred in trying to make a connection to the remote host.
439	Message:	Cannot perform operation on command window <*id*>
	Type:	ERROR
	Explanation:	The selected operation cannot be performed on a command window.
440	Message:	Cannot create an examine window.
	Type:	ERROR
	Explanation:	An examine window could not be created. One cause of this problem is that an examine window may only appear as an X window. Maryland Windows does not support it.
441	Message:	Cannot create a stack window.
	Type:	ERROR
	Explanation:	A stack window could not be created. One cause of this problem is that a stack window may only appear as an X window. Maryland Windows does not support it.
442	Message:	There is no server associated with process <*pid*>
	Type:	ERROR
	Explanation:	An attempt was made to access a process through a remote server. There is currently no remote access to this process.
443	Message:	Error expanding command arguments: <*reason*>
	Type:	ERROR
	Explanation:	An error was detected while processing the arguments specified for the target process. The text displayed will indicate the exact nature of the error. Usually it stems from invalid variable references or syntax errors.

444 Message: The task priority < *value*> is out of range.
Type: ERROR
Explanation: The value given for an rtk task priority is out of range.

445 Message: Can't set task priority for task < *value*> of application <*value*>.
Type: ERROR
Explanation: An error was detected during the setting of the task priority for the indicated task of the indicated rtk application.

446 Message: Unknown remote file system specification for process <*pid*>.
Type: ERROR
Explanation: The remote file system specified is unknown.

447 Message: Error setting remote file system: <*file system*>.
Type: ERROR
Explanation: An error occurred while setting the remote file system.

448 Message: Error setting memory size for application <*value*>.
Type: ERROR
Explanation: An error occurred while setting the maximum memory available to a remote application.

449 Message: Error setting maximum number of tasks for application <*value*>.
Type: ERROR
Explanation: An error occurred while setting the maximum number of tasks allowed in a remote application.

450 Message: Error setting cpu affinity for application <*value*>.
Type: ERROR
Explanation: An error occurred while setting the CPU affinity for a remote application.

451 Message: Error setting default task priority for application <*value*>.
Type: ERROR
Explanation: An error occurred while setting the default task priority for a remote application.

452 Message: Error : ring <*value*> is not a valid ring number.
Type: ERROR
Explanation: An incorrect ring number was specified. An application may execute in either user or kernel mode.

No.		Description
453	Message:	Error setting ring mode for application <*value*>.
	Type:	ERROR
	Explanation:	An error occurred while setting the ring in which the initial task of a remote application may execute.
454	Message:	Error setting initial task's private data size for application <*value*>.
	Type:	ERROR
	Explanation:	An error occurred while setting the initial task's task-private data size.
455	Message:	Error setting initial task's stack size for application <*value*>.
	Type:	ERROR
	Explanation:	An error occurred while setting the initial task's stack size.
456	Message:	Error setting the name of application <*value*>.
	Type:	ERROR
	Explanation:	An error occurred while setting the application's name.
457	Message:	Bad dynamic size, < *the size*>
	Type:	ERROR
	Explanation:	The dynamic size stored in the processes memory was erroneous. Possibly, some corruption in memory has occured.
458	Message:	No existing process image.
	Type:	ERROR
	Explanation:	An attempt was made to access the process image. Currently, no process image exists.
459	Message:	Invalid command! You may not detach from an executable file.
	Type:	ERROR
	Explanation:	You attempted to execute the 'detach' command while the process image was derived from an executable file. You may start a new process image with the 'run' command, but you can not detach while the image originates with an executable file.
460	Message:	You may not execute a remote command on a local process.
	Type:	ERROR
	Explanation:	You have attempted to execute a command which is applicable only to remote processes on a local process.
461	Message:	Can't create a remote process.
	Type:	ERROR
	Explanation:	An error has occurred attempting to create a remote process.

462 Message: Can't read a remote process' state.
Type: ERROR
Explanation: An error has occurred attempting to read the state of a remote process. The process may not exist, or it may exist and be in a state in which it cannot be examined. For example, it may be running.

463 Message: No primary selection exists.
Type: ERROR
Explanation: You performed a GUI action that requires a primary selection. You should make a primary selection and then try the operation again.

464 Message: This is not a rooted directory, < *directory name*>
Type: ERROR
Explanation: The supplied directory name must be a rooted name. That is, no assumption is made about the current default directory in this case. Therefore, the name should begin with a "/".

465 Message: The directory < *directory name*> does not exist
Type: INFO
Explanation: This directory does not exist. In this case, it is not an error.

466 Message: The file < *name*> cannot be glob'ed
Type: ERROR
Explanation: This file name could be successfully globbed. The rules of the C shell (csh) are followed during file name expansion.

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

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

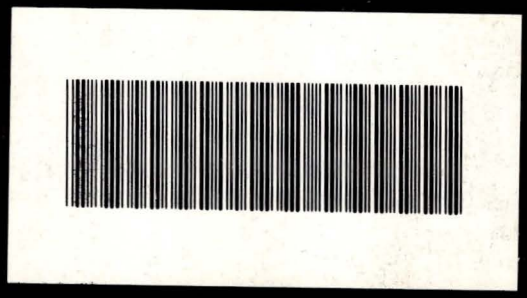
S

T

V

W

X



Order Number
DSW-478

Document Number
710-024130-000