5 Examining Your Program

This chapter describes how to examine aspects of your program to detect bugs. This includes the following:

- examining the stack
- examining source files
- examining data
- examining memory
- examining the symbol table

Examining the Stack

When your program has stopped, the first thing you need to know is where it stopped and how it got there. Each time your program performs a function call, the information about where in the program the call was made from is saved in a block of data called a stack frame. The stack frame also contains the arguments of the call and the local variables of the function that was called. All the stack frames are allocated in a region of memory called the call stack. When your program stops, the debugger commands for examining the stack allow you to see all of this information.

One of the stack frames is selected by the debugger and many debugger commands refer implicitly to the selected frame. In particular, whenever you ask the debugger for the value of a variable in the program, the value is found in the selected frame. There are special debugger commands to select whichever frame you are interested in.

When the program stops, the debugger automatically selects the currently executing frame and prints a brief description of the frame.

Stack Frames

The call stack is divided up into contiguous pieces called stack frames. Each frame contains the data associated with one call to one function. The stack frame contains the following:

- the arguments given to the function
- the function's local variables
- the address at which the function is executing

When your program is started, the stack has only one frame, that of the function `main`. This is called the initial frame or the outermost frame. Each time a function is called, a new frame is made. Each time a function returns, the frame for that function invocation is eliminated. If a function is recursive, there can be many frames for the same function. The frame for the function in which execution is actually occurring is called the innermost frame. This is the most recently created of all the stack frames that still exist.

Inside your program, stack frames are identified by their addresses. A stack frame consists of many bytes. Each stack frame has its own address. This address is usually kept in a register called the frame pointer register while execution is going on in that frame.

The debugger assigns numbers to all existing stack frames, starting with zero for the innermost frame, one for the frame that called it, and so on. These numbers do not exist in your program. They simply give you a way to reference stack frames in debugger commands.

Many debugger commands refer implicitly to one stack frame. The debugger records a stack frame that is called the selected stack frame. You can select a frame using debugger commands, and then operate on that frame with other commands. When your program stops, the debugger automatically selects the innermost frame.

Selecting a Frame

Most commands for examining the stack and other data in the program work on whichever stack frame is selected at the moment. You can select a stack frame with the following commands:

- `frame`
- `up`
- `down`

All of these commands print the following information about the frame selected:

- frame number
- function name
- arguments
- source file
- line number of execution in the frame
- text of the source line

The `frame` Command

The `frame` command has the following syntax:

`frame [`*`argument`*`]`

argument can be any of the following:

n	Selects frame number *n*. Frame zero is the innermost (currently executing) frame, frame one is the frame that called the innermost frame, and so on. The highest-numbered frame is the frame for `main`.
addr	Selects the frame at address *addr*. This is useful mainly if the chaining of stack frames has been damaged by a bug, making it impossible for the debugger to assign numbers properly to all frames.

If you do not include an argument with the `frame` command, the debugger prints a brief description of the current stack frame.

The `up` Command

The `up` command has the following syntax:

`up  [`*`n`*`]`

The `up` command selects the frame *n* frames up from the frame currently selected. For positive numbers *n*, this advances toward the outermost frame, to higher frame numbers from frames that have existed longer. If *n* is not specified, *n* becomes one.

The `down` Command

The `down` command has the following syntax:

`down [`*n*`]`

The `down` command selects the frame *n* frames down from the frame currently selected. For positive numbers *n*, this advances toward the innermost frame, to lower frame numbers from frames that were created more recently. If *n* is not specified, *n* becomes one.

Getting Frame Information

You can use the `info` command to print information about the selected stack frame by using the following syntax:

`info` *`argument`*

argument can be any of the following:

`frame`	Prints a verbose description of the selected stack frame, including the following: • address of the frame • address of the next frame in (called by this frame) • address of the next frame out (caller of this frame) • address of the frame's arguments • address of execution in the caller frame • registers saved in the frame. The verbose description is useful when something has gone wrong that has made the stack format fail to fit the usual conventions.
`frame` *`addr`*	Prints a verbose description of the frame at address *addr*, without selecting that frame. The selected frame remains unchanged.
`args`	Print the arguments of the selected frame, each on a separate line.
`locals`	Prints the local variables of the selected frame, each on a separate line.

Backtraces

A backtrace is a summary of how a program got where it is. A backtrace shows the frame number, function name, and program counter value for each frame listed in the backtrace. Each frame is listed on a separate line.

If the function is in a source file whose symbol table data has been fully read, the backtrace shows the source file name and line number, as well as the arguments to the function. The program counter value is omitted if it is at the beginning of the code for that line number.

If the source file's symbol data has only been scanned, the extra information is replaced with an ellipsis. You can force the symbol data for that frame's source file to be read by selecting the frame.

You can use the `backtrace` command to generate a backtrace. The `backtrace` command has the following syntax:

`backtrace [`*`argument`*`]`

argument can be any of the following:

`n`	prints only the innermost *n* frames of the stack
`-n`	prints only the outermost *n* frames of the stack

If you do not include an argument with the `backtrace` command, the debugger prints a backtrace of the entire stack, one line per frame for all frames in the stack. You can stop the backtrace at any time by typing the system interrupt character, normally `Control-C`.

The following commands perform the same function as the `backtrace` command:

- `where`
- `info stack`

The following is an example of a backtrace. It was generated with the command `backtrace 3`, so it shows the innermost three frames.

```
cell01: #0 rtx_equal_p (x=(rtx) 0x8e58c, y=(rtx) 0x1086c4)
        (/gp/rms/cc/rtlanal.c line 337)
cell01: #1 0x246b0 in expand_call (...) (...)
cell01: #2 0x21cfc in expand_expr (...) (...)
        (More stack frames follow...)
```

The example shows that the functions `expand_call` and `expand_expr` are in a file whose symbol details have not been fully read. Full detail is available for the function `rtx_equal_p`, which is in the file `rtlanal.c`. Its arguments, named `x` and `y`, are shown with their typed values.

Examining Source Files

The debugger knows which source files your program was compiled from, and can print parts of their text. When your program stops, the debugger prints the line where the program stopped. Likewise, when you select a stack frame, the debugger prints the line in which execution in that frame has stopped. You can also print parts of source files by explicit command.

Printing Source Lines

To print lines from a source file, use the `list` command. The `list` command has the following syntax:

`list [`*`argument`*`]`

argument can be any of the following:

`[`*`filename:`*`]` *`linenum`*	Prints ten lines centered around line number *linenum* in the file *filename*. If *filename* is not included, the file is the current source file.
`[`*`filename:`*`]`*`function`*	Prints ten lines centered around the beginning of function *function* in the file *filename*. If *filename* is not included, the file is the current source file
`+`	Prints ten lines following the lines last printed.
`-`	Prints ten lines before the lines last printed.
`+` *`offset`*	Prints ten lines centered around the line *offset* lines after the last line printed. When used as a *last* specifier, this argument specifies the line *offset* lines down from *first*.

`-` *`offset`*	Specifies the line *offset* lines before the last line printed. When used as a *first* specifier, this argument specifies the line *offset* lines up from *last*.
`*`*`address`*	Prints ten lines centered around the line containing the program address *address*. *address* can be any expression.
`first``,`*`last`*	Prints lines from *first* to *last*. *first* and *last* can be specified by using any of the following: • a line number • a function name • an offset from the last line printed • an address
`first``,`	Prints ten lines starting with *first*.
`,`*`last`*	Prints ten lines ending with *last*.

If you do not include any arguments with the `list` command, the debugger prints ten lines. If the last lines printed were printed with a `list` command, the debugger prints ten lines following the last lines printed. If the last line printed was a single line printed as part of displaying a stack frame, the debugger prints ten lines centered around that line.

Repeating a `list` command with `RETURN` discards the argument, so it is equivalent to typing just `list`. An exception is made for the argument `-`. This argument is preserved so that each repetition moves up in the file.

You can also use the `info` command to map source lines to program addresses as follows:

`info line` *`linenum`*

This command prints the starting and ending addresses of the compiled code for source line *linenum*.

Searching Source Files

There are two commands for searching through the current source file for a regular expression.

- `forward-search` *`regexp`* — checks each line, starting with the one following the last line listed, for a match for *regexp* and lists the line found.
- `reverse-search` *`regexp`* — checks each line, starting with the one before the last line listed, for a match for *regexp* and lists the line found.

Specifying Source Directories

Executable programs do not record the directories of the source files from which they were compiled, just the names. The debugger remembers a list of directories to search for source files. This list is called the source path. Each time the debugger looks for a source file, it searches all the directories in the list, in the order they are present in the list, until it finds the desired file.

When you start the debugger, the source path contains only the current working directory. To add other directories, use the `directory command.` The `directory` command has the following syntax:

`directory  [`*`dirname`*` ...]`

dirname is a directory name.

The `directory` command adds directory *dirname* to the end of the source path. Multiple directory names can be specified with this command. The names must be separated by blank space or a colon.

If no arguments are included with the `directory` command, the debugger resets the source path to include only the current working directory of the debugger. Since this command deletes directories from the search path, it may change the directory in which a previously read source file is found. Consequently, this command also clears out the tables the debugger maintains about the source files it has already found when it resets the source path.

Because the `directory` command adds to the end of the source path, it does not affect any file that the debugger has already found. If the source path contains directories that you do not want, do the following:

1. Choose the directory you want at the beginning of the source path, and use the `cd` command to make it the current working directory.
2. Use the `directory` command with no arguments to reset the source path to include only that directory.
3. Use the `directory` command with arguments to add any other directories you want in the source path.

You can use the `info` command to see which directories are on the source path as follows:

```
info directories
```

Examining Data

The usual way to examine data in your program is with the `print` command. The `print` command evaluates and prints the value of any valid expression of the source language. The `print` command has the following syntax:

```
print exp
```

exp is any valid expression. The value of *exp* is normally printed in a format appropriate to its data type. Although the debugger normally prints all values according to their data types, you can control how the debugger prints values by using output formats.

Output formats

The simplest use of output formats is to specify how to print a value already computed. This is done by starting the arguments of the `print` command with a slash and a format letter. The following format letters are supported:

`x`	regards the bits of the value as an integer, and prints the integer in hexadecimal
`d`	prints as integer in signed decimal
`u`	prints as integer in unsigned decimal
`o`	prints as integer in octal
`a`	prints as an address, both absolute in hex and then relative to a symbol defined as an address below it
`c`	regards the bits of the value as an integer and prints as a character constant

`f`	regards the bits of the value as a floating point number and prints using typical floating point syntax

For example, to print the program counter in hexadecimal, type

```
print/x $pc
```

No space is required before the slash, since debugger command names cannot contain a slash.

To reprint the last value in the value history with a different format, use the `print` command with only a format. For example, `print/x` reprints the last value in hexadecimal.

Expressions

Many different debugger commands accept an expression and compute its value. Any kind of constant, variable, or operator defined by your programming language is valid in a debugger expression. This includes the following:

- conditional expressions
- function calls
- casts
- string constants

It does not include symbols defined by preprocessor `#define` commands.

The debugger supports three operators in addition to those included in your programming language:

`@`	A binary operator for treating parts of memory as arrays. This technique is described in the section *Artificial Arrays*.
`::`	Allows you to specify a variable in terms of the file or function in which it is defined.
`type addr`	Specifies an object of type *type* stored at address *addr* in memory. *addr* may be any expression whose value is an integer or pointer, but parentheses are required around nonunary operators, just as in a cast. This construct is allowed regardless of what kind of data resides at *addr*.

Program Variables

The most common type of expression used is the name of a variable in your program. Variables in expressions are understood in the selected stack frame. They must either be global or static, or be visible from the point of execution in that frame. For example, in the following function, the variable `a` is usable whenever the program is executing within the function `foo`, but the variable `b` is visible only while the program is executing inside the block in which `b` is declared.

```
foo (a)
 int a;
{
  bar (a);
  {
  int b = test ();
  bar (b);
```

```
    }
  }
```

As a special exception, you can refer to a variable or function whose scope is a single source file even if the current execution point is not in this file. But it is possible to have more than one such variable or function with the same name if they are in different source files. In such a case, it is not defined which one is accessed. If you wish, you can specify one of them using the colon-colon construct as follows:

```
block:: variable
```

block is the name of the source file that contains the variable you want to access.

Artificial Arrays

It is often useful to print out several successive objects of the same type in memory, such as a section of an array, or an array of dynamically determined size for which only a pointer exists in the program.

This can be done by constructing an artificial array with the binary operator @, using the following syntax:

```
first@length
```

first is the first element of the desired array.

length is the length of the array.

The result is an array whose elements are all of the type of *first*. *first* is the first element. The second element comes from bytes of memory immediately following *first*, and so on. For example, if a program has the following declaration,

```
int *array = (int *) malloc (len * sizeof (int));
```

you can print the contents of `array` with the following command:

```
print *array@len
```

The left operand of `@` must reside in memory. Artificial arrays created with `@` behave just like other arrays in terms of subscripting, and elements are coerced to pointers when used in expressions.

Format options

You can control how arrays and structures are printed by using the `set` command as follows:

```
set [argument]
```

argument can be any of the following:

`array-max` *elements* — Stops printing after it has printed the number of elements set by *elements*. This limit also applies to the display of strings.

`prettyprint on` — Prints structures in an indented format with one member per line, as in the following example:

```
$1 = {
  next = 0x0,
  flags = {
    sweet = 1,
```

```
      sour = 1
    },
    meat = 0x54 ”Pork”
  }
```

`prettyprint off`	Prints structures in a compact format, as in the following example: `$1 = {next = 0x0, flags ={sweet = 1, sour = 1}, meat = 0x54 ”Pork”  }` `prettyprint off` is the default format.
`unionprint on`	Prints unions which are contained in structures. `unionprint on` is the default format.
`unionprint off`	Does not print unions which are contained in structures.

You can use the info command to display the current settings for the format options as follows:

```
info format
```

Examining Memory

You can use the `x` command to examine memory without reference to the program's data types. With the `x` command, you explicitly specify the format in which you wish to examine memory. The allowable formats are a superset of the formats used with the `print` command. The `x` command has the following syntax:

`x/`*`format addr`*

addr is an expression specifying an address. The expression need not have a pointer value. It is used as an integer, as the address of a byte of memory. If no address is specified, the debugger examines the address following the last address examined.

format includes an integer and two letters. The integer is optional, and specifies how many memory units to examine. The first letter specifies the size of the memory units to examine. The second letter specifies the format for printing the output. If either the memory unit size or the output format are not specified, the default is to use the last specification. The size is specified by any of the following letters:

`b`	Examine individual bytes.
`h`	Examine halfwords (two bytes each).
`w`	Examine words (four bytes each).
`g`	Examine double words (8 bytes).

The output format is specified by any of the following letters:

`x`	Print as integers in unsigned hexadecimal.
`d`	Print as integers in signed decimal.
`u`	Print as integers in unsigned decimal.
`o`	Print as integers in unsigned octal.
`a`	Print as an address, both absolute in hex and then relative to a symbol defined as an address below it.

`c` Print as character constants.

`f` Print as floating point. This works only with sizes `w` and `g`.

`s` Print a null-terminated string of characters. The specified unit size is ignored; instead, the unit is the number of bytes needed to reach a null character (including the null character).

i Print a machine instruction in assembler syntax. The specified unit size is ignored.

For example, the following command prints the four words of memory above the stack pointer in hexadecimal.

```
x/4xw $sp
```

When you use `RETURN` to repeat an `x` command, it examines successive locations in memory rather than the same ones.

The addresses and contents printed by the `x` command are not put in the value history because of their quantity. Instead, the debugger makes these values available for subsequent use in expressions as values of the convenience variables `$_` and `$__`. After an `x` command, the last address examined is available for use in expressions in the convenience variable `$_`. The contents of that address, as examined, are available in the convenience variable `$__`.

If the `x` command has a repeat count, the address and contents saved are from the last memory unit printed. This is not the same as the last address printed if several units were printed on the last line of output.

The specialized command `disassemble` is also provided to dump a range of memory as machine instructions. The disassemble command has the following syntax:

```
disassemble arg1 [arg2]
```

If only one argument is included, *arg1* is read as a program counter value, and the function surrounding this value is dumped. If two arguments are included, *arg1* is the bottom and *arg2* is the top of a range of addresses to be dumped. The default memory range is the function surrounding the program counter of the selected frame.

Automatic Display

If you find that you want to print the value of an expression frequently to see how it changes, you can add the expression to the automatic display list so the debugger prints its value each time the program stops. Each expression added to the list is given a number to identify it.

To add an expression to the automatic display list, use the `display` command. The `display` command has the following syntax:

```
display[/fmt] [argument]
```

fmt is a format specifier. Refer to the description of the `print` command for a list of format specifiers. If *fmt* is included, the expression being added to the automatic display list is displayed in the specified format.

argument can be any of the following:

`exp` adds the expression *exp* to the list of expressions to display each time the program stops

`addr` adds the expression *addr* as a memory address to be examined each time the program stops

If no arguments are included with the display command, the debugger displays the current values of the expressions on the list.

If an expression refers to local variables, the expression does not make sense outside the scope of the variable. Such an expression is printed only when execution is inside that scope. For example, if you give the command `display arg` while inside a function with an argument `arg`, this argument will be displayed whenever the program stops inside that function, but not when it stops elsewhere.

There are two commands to remove items from the automatic display list:

`undisplay` *`dnums`*

or

`delete display` *`dnums`*

Both commands remove item numbers *dnums* from the list of expressions to display.

You can also disable expressions on the automatic display list so they do not currently display. To disable items, use the `disable display` command as follows:

`disable display` *`dnums`* Disables the display of item numbers *dnums*. A disabled display item is not printed automatically, but it remains on the automatic display list. It can be re-enabled later.

To re-enable a disabled items, use the `enable display` command as follows:

`enable display` *`dnums`* Enables the display of item numbers *dnums*.

You can print the list of expressions on the automatic display list by using the info command as follows:

```
info display
```

The `info display` command lists each expression on the automatic display list each with its item number, but does not show the expression values. Disabled expressions are included, and are marked as disabled. The list also includes expressions that would not currently be displayed because they refer to automatic variables not currently available.

Value History

Every value printed by the `print` command is saved for the entire session in the debugger's value history so you can refer to it in other expressions.

The values printed are given history numbers for you to reference them. These numbers are successive integers starting with 1. The `print` command shows the history number assigned to a value by printing `$ num =` before the value.

To refer to any previous value, use `$` followed by the value's history number. Just `$` without a number refers to the most recent value in the history, and `$$` refers to the previous value.

For example, if you just printed a pointer to a structure and want to see the contents of the structure, type

```
print *$
```

If you have a chain of structures where the component `next` points to the next structure, you can print it by typing:

```
print *$.next
```

You can repeat this command many times by typing a carriage return.

NOTE

History records values, not expressions. If the value of `x` is 4 and you type the commands:

```
print x
set x=5
```

the value recorded in the value history by the `print` command remains 4 even though the value of `x` has changed.

You can get information about the value history by using the `info` command as follows:

`info values [`*`argument`*`]`

argument can be any of the following:

n	Prints ten history values centered on history item number *n*.
+	Prints ten history values following the values last printed.

If no argument is included with the `info values` command, the debugger prints the last ten values in the value history, with their item numbers.

Convenience Variables

The debugger provides convenience variables you can use to retain a value and refer to it later. These variables exist entirely within the debugger. They are not part of your program, and setting a convenience variable has no effect on further execution of your program.

Convenience variables have names starting with `$`. Any name starting with `$` can be used as a convenience variable, unless it is one of the predefined register names. You can save a value in a convenience variable with an assignment expression, just as you would set a variable in your program. For example, the following declaration would save the value contained in the object pointed to by `object_ptr` in the convenience variable `$foo`.

```
set $foo = *object_ptr
```

Using a convenience variable for the first time creates the variable, but its value is `void` until you assign a new value. You can alter the value with another assignment at any time. Convenience variables have no fixed types. You can assign a convenience variable any type of value, even if it already has a value of a different type. As an expression, the convenience variable has whatever type its current value has.

The following convenience variables are created automatically by the debugger:

`$_`	The variable `$_` is automatically set by the `x` command to the last address examined. Other commands which provide a default address for `x` to examine also set `$_` to that address. These commands include `info line` and `info breakpoint`.
`$_ _`	The variable `$__` is automatically set by the `x` command to the value found in the last address examined.

You can print a list of convenience variables and their values by using the info command as follows:

```
info convenience
```

Registers

You can refer to machine register contents in expressions as variables with names starting with `$`. The debugger always considers the contents of an ordinary register as an integer when the register is examined in this way. Register values are relative to the selected stack frame. This means you get the value that the register would contain if all stack frames farther in were exited and their saved registers restored. In order to see the real contents of all registers, you must select the innermost frame with the `frame 0` command.

For example, the following command prints the program counter in hexadecimal:

```
p/x $pc
```

The following command prints the next instruction to be executed:

```
x/i $pc
```

The following command adds four to the stack pointer:

```
set $sp += 4
```

The last example assumes that the innermost stack frame is selected. Setting `$sp` is not allowed when other stack frames are selected.

You can use the info command to see register names and values as follows:

`info registers [`*`regname`*`]`

This command prints the relative value of register *regname*. *regname* may be any valid register name, with or without the initial `$`. If *regname* is not included, the debugger prints the names and relative values of all registers.

Refer to Appendix B for a complete list of register names you can use in debugger commands.

Examining the Symbol Table

The following commands allow you to get information about the symbols defined in your program. Symbol information is found by the debugger in the symbol table. This information is inherent in the text of your program and does not change as the program executes.

`whatis [`*`exp`*`]`	Prints the data type of expression *exp*. *exp* is not actually evaluated, and any side-effect operations such as assignments or function calls inside it do not take place. If *exp* is not specified, the debugger prints the data type of the last value in the value history.
`ptype` *`typename`*	Prints a description of data type *typename*. *typename* can be the name of a type, or may have the form `struct` *`struct-tag`*, `union` *`union-tag`*, or `enum` *`enum-tag`*.
`printsyms` *`filename`*	Writes a complete dump of the debugger's symbol data into the file *filename*.
`info address` *`symbol`*	Describes where the data for *symbol* is stored. For a register variable, this prints which register it is kept in.

	For a non-register local variable, this prints the stack-frame offset at which the variable is always stored.
`info sources`	Prints the names of all source files in the program for which there is debugging information.
`info functions [`*regexp*`]`	Prints the names and data types of all defined functions whose names contain a match for regular expression *regexp*. For example, `info functions step` finds all functions whose names include `step`. `info functions ^step` finds those whose names start with `step`. If *regexp* is not specified, the debugger prints the names and data types of all defined functions.
`info variables [`*regexp*`]`	Prints the names and data types of all variables except local variables whose names contain a match for regular expression *regexp*. If *regexp* is not specified, the debugger prints the names and data types of all variables declared outside of functions.
`info types [`*regexp*`]`	Prints all data types defined in the program whose names contain a match for regular expression *regexp*. If *regexp* is not specified, the debugger prints all data types that are defined in the program.