Go to the previous, next section.

Examining Data

The usual way to examine data in your program is with the `print` command (abbreviated `p`), or its synonym `inspect`. It evaluates and prints the value of an expression of the language your program is written in (see section Using GDB with Different Languages).

`print` *exp*
`print /`*f* *exp*

exp is an expression (in the source language). By default the value of *exp* is printed in a format appropriate to its data type; you can choose a different format by specifying `` `/f' ``, where *f* is a letter specifying the format; see section Output formats.

`print`
`print /`*f*

If you omit *exp*, GDB displays the last value again (from the *value history*; see section Value history). This allows you to conveniently inspect the same value in an alternative format.

A more low-level way of examining data is with the `x` command. It examines data in memory at a specified address and prints it in a specified format. See section Examining memory.

If you are interested in information about types, or about how the fields of a struct or class are declared, use the `ptype` *exp* command rather than `print`. See section Examining the Symbol Table.

Expressions

`print` and many other GDB commands accept an expression and compute its value. Any kind of constant, variable or operator defined by the programming language you are using is valid in an expression in GDB. This includes conditional expressions, function calls, casts and string constants. It unfortunately does not include symbols defined by preprocessor `#define` commands.

Because C is so widespread, most of the expressions shown in examples in this manual are in C. See section Using GDB with Different Languages, for information on how to use expressions in other languages.

In this section, we discuss operators that you can use in GDB expressions regardless of your programming language.

Casts are supported in all languages, not just in C, because it is so useful to cast a number into a pointer so as to examine a structure at that address in memory.

GDB supports these operators in addition to those of programming languages:

`@`
: `` `@' `` is a binary operator for treating parts of memory as arrays. See section Artificial arrays, for more information.

`::`
: `` `::' `` allows you to specify a variable in terms of the file or function where it is defined. See section Program variables.

`{type} addr`
: Refers to 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 binary operators, just as in a cast). This construct is allowed regardless of what kind of data is normally supposed to reside at *addr*.

Program variables

The most common kind of expression to use is the name of a variable in your program.

Variables in expressions are understood in the selected stack frame (see section Selecting a frame); they must either be global (or static) or be visible according to the scope rules of the programming language from the point of execution in that frame. This means that in the function

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

you can examine and use the variable `a` whenever your program is executing within the function `foo`, but you can only use or examine the variable `b` while your program is executing inside the block where `b` is declared.

There is an 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 (in different source files). If that happens, referring to that name has unpredictable effects. If you wish, you can specify a static variable in a particular function or file, using the colon-colon notation:

```
file::variable
function::variable
```

Here *file* or *function* is the name of the context for the static *variable*. In the case of file names, you can use quotes to make sure GDB parses the file name as a single word--for example, to print a global value of `x` defined in `` `f2.c' ``:

```
(gdb) p 'f2.c'::x
```

This use of `` `::' `` is very rarely in conflict with the very similar use of the same notation in C++. GDB also supports use of the C++ scope resolution operator in GDB expressions.

> *Warning:* Occasionally, a local variable may appear to have the wrong value at certain points in a function--just after entry to a new scope, and just before exit.

You may see this problem when you are stepping by machine instructions. This is because on most machines, it takes more than one instruction to set up a stack frame (including local variable definitions); if you are stepping by machine instructions, variables may appear to have the wrong values until the stack frame is completely built. On exit, it usually also takes more than one machine instruction to destroy a stack frame; after you begin stepping through that group of instructions, local variable definitions may be gone.

Artificial arrays

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

You can do this by referring to a contiguous span of memory as an *artificial array*, using the binary operator `` `@' ``. The left operand of `` `@' `` should be the first element of the desired array, as an individual object. The right operand should be the desired length of the array. The result is an array value whose elements are all of the type of the left argument. The first element is actually the left argument; the second element comes from bytes of memory immediately following those that hold the first element, and so on. Here is an example. If a program says

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

you can print the contents of `array` with

```
p *array@len
```

The left operand of `` `@' `` must reside in memory. Array values made with `` `@' `` in this way behave just like other arrays in terms of subscripting, and are coerced to pointers when used in expressions. Artificial arrays most often appear in expressions via the value history (see section Value history), after printing one out.

Sometimes the artificial array mechanism is not quite enough; in moderately complex data structures, the elements of interest may not actually be adjacent--for example, if you are interested in the values of pointers in an array. One useful work-around in this situation is to use a convenience variable (see section Convenience variables) as a counter in an expression that prints the first interesting value, and then repeat that expression via `RET`. For instance, suppose you have an array `dtab` of pointers to structures, and you are interested in the values of a field `fv` in each structure. Here is an example of what you might type:

```
set $i = 0
p dtab[$i++]->fv
RET
RET
...
```

Output formats

By default, GDB prints a value according to its data type. Sometimes this is not what you want. For example, you might want to print a number in hex, or a pointer in decimal. Or you might want to view data in memory at a certain address as a character string or as an instruction. To do these things, specify an *output format* when you print a value.

The simplest use of output formats is to say 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 format letters supported are:

`x`
: Regard the bits of the value as an integer, and print the integer in hexadecimal.

`d`
: Print as integer in signed decimal.

`u`
: Print as integer in unsigned decimal.

`o`
: Print as integer in octal.

`t`
: Print as integer in binary. The letter `` `t' `` stands for "two". (1)

`a`
: Print as an address, both absolute in hexadecimal and as an offset from the nearest preceding symbol. You can use this format used to discover where (in what function) an unknown address is located:

```
(gdb) p/a 0x54320
$3 = 0x54320 <_initialize_vx+396>
```

`c`
: Regard as an integer and print it as a character constant.

`f`
: Regard the bits of the value as a floating point number and print using typical floating point syntax.

For example, to print the program counter in hex (see section Registers), type

```
p/x $pc
```

Note that no space is required before the slash; this is because command names in GDB cannot contain a slash.

To reprint the last value in the value history with a different format, you can use the `print` command with just a format and no expression. For example, `` `p/x' `` reprints the last value in hex.

Examining memory

You can use the command `x` (for "examine") to examine memory in any of several formats, independently of your program's data types.

`x/`*`nfu addr`*
`x `*`addr`*
`x`
: Use the `x` command to examine memory.

n, *f*, and *u* are all optional parameters that specify how much memory to display and how to format it; *addr* is an expression giving the address where you want to start displaying memory. If you use defaults for *nfu*, you need not type the slash `` `/' ``. Several commands set convenient defaults for *addr*.

n, the repeat count
: The repeat count is a decimal integer; the default is 1. It specifies how much memory (counting by units *u*) to display.

f, the display format
: The display format is one of the formats used by `print`, or `` `s' `` (null-terminated string) or `` `i' `` (machine instruction). The default is `` `x' `` (hexadecimal) initially, or the format from the last time you used either `x` or `print`.

u, the unit size
: The unit size is any of

  `b`
  : Bytes.

`h`
: Halfwords (two bytes).

`w`
: Words (four bytes). This is the initial default.

`g`
: Giant words (eight bytes).

Each time you specify a unit size with `x`, that size becomes the default unit the next time you use `x`. (For the `` `s' `` and `` `i' `` formats, the unit size is ignored and is normally not written.)

- *addr*, starting display address *addr* is the address where you want GDB to begin displaying memory. The expression need not have a pointer value (though it may); it is always interpreted as an integer address of a byte of memory. See section Expressions, for more information on expressions. The default for *addr* is usually just after the last address examined--but several other commands also set the default address: `info breakpoints` (to the address of the last breakpoint listed), `info line` (to the starting address of a line), and `print` (if you use it to display a value from memory).

For example, `` `x/3uh 0x54320' `` is a request to display three halfwords (`h`) of memory, formatted as unsigned decimal integers (`` `u' ``), starting at address `0x54320`. `` `x/4xw $sp' `` prints the four words (`` `w' ``) of memory above the stack pointer (here, `` `$sp' ``; see section Registers) in hexadecimal (`` `x' ``).

Since the letters indicating unit sizes are all distinct from the letters specifying output formats, you do not have to remember whether unit size or format comes first; either order works. The output specifications `` `4xw' `` and `` `4wx' `` mean exactly the same thing. (However, the count *n* must come first; `` `wx4' `` does not work.)

Even though the unit size *u* is ignored for the formats `` `s' `` and `` `i' ``, you might still want to use a count *n*; for example, `` `3i' `` specifies that you want to see three machine instructions, including any operands. The command `disassemble` gives an alternative way of inspecting machine instructions; see section Source and machine code.

All the defaults for the arguments to `x` are designed to make it easy to continue scanning memory with minimal specifications each time you use `x`. For example, after you have inspected three machine instructions with `` `x/3i addr' ``, you can inspect the next seven with just `` `x/7' ``. If you use `RET` to repeat the `x` command, the repeat count *n* is used again; the other arguments default as for successive uses of `x`.

The addresses and contents printed by the `x` command are not saved in the value history because there is often too much of them and they would get in the way. Instead, GDB 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.

Automatic display

If you find that you want to print the value of an expression frequently (to see how it changes), you might want to add it to the *automatic display list* so that GDB prints its value each time your program stops. Each expression added to the list is given a number to identify it; to remove an expression from the list, you specify that number. The automatic display looks like this:

```
2: foo = 38
3: bar[5] = (struct hack *) 0x3804
```

This display shows item numbers, expressions and their current values. As with displays you request manually using `x` or `print`, you can specify the output format you prefer; in fact, `display` decides whether to use `print` or `x` depending on how elaborate your format specification is--it uses `x` if you specify a unit size, or one of the two formats (`` `i' `` and `` `s' ``) that are only supported by `x`; otherwise it uses `print`.

`display` *`exp`*
: Add the expression *exp* to the list of expressions to display each time your program stops. See section Expressions.

  `display` does not repeat if you press `RET` again after using it.

`display/`*`fmt exp`*
: For *fmt* specifying only a display format and not a size or count, add the expression *exp* to the auto-display list but arrange to display it each time in the specified format *fmt*. See section Output formats.

`display/`*`fmt addr`*
: For *fmt* `` `i' `` or `` `s' ``, or including a unit-size or a number of units, add the expression *addr* as a memory address to be examined each time your program stops. Examining means in effect doing `` `x/fmt addr' ``. See section Examining memory.

For example, `` `display/i $pc' `` can be helpful, to see the machine instruction about to be executed each time execution stops (`` `$pc' `` is a common name for the program counter; see section Registers).

`undisplay` *`dnums`*`...`
`delete display` *`dnums`*`...`
: Remove item numbers *dnums* from the list of expressions to display.

  `undisplay` does not repeat if you press `RET` after using it. (Otherwise you would just get the error `` `No display number ...' ``.)

`disable display` *`dnums`*`...`
: Disable the display of item numbers *dnums*. A disabled display item is not printed automatically, but is not forgotten. It may be enabled again later.

`enable display` *`dnums`*`...`

Enable display of item numbers *dnums*. It becomes effective once again in auto display of its expression, until you specify otherwise.

`display`
Display the current values of the expressions on the list, just as is done when your program stops.

`info display`
Print the list of expressions previously set up to display automatically, each one with its item number, but without showing the values. This includes disabled expressions, which are marked as such. It also includes expressions which would not be displayed right now because they refer to automatic variables not currently available.

If a display expression refers to local variables, then it does not make sense outside the lexical context for which it was set up. Such an expression is disabled when execution enters a context where one of its variables is not defined. For example, if you give the command `display last_char` while inside a function with an argument `last_char`, GDB displays this argument while your program continues to stop inside that function. When it stops elsewhere--where there is no variable `last_char`---the display is disabled automatically. The next time your program stops where `last_char` is meaningful, you can enable the display expression once again.

Print settings

GDB provides the following ways to control how arrays, structures, and symbols are printed.

These settings are useful for debugging programs in any language:

`set print address`
`set print address on`
GDB prints memory addresses showing the location of stack traces, structure values, pointer values, breakpoints, and so forth, even when it also displays the contents of those addresses. The default is `on`. For example, this is what a stack frame display looks like, with `set print address on`:

```
(gdb) f
#0  set_quotes (lq=0x34c78 "<<", rq=0x34c88 ">>")
    at input.c:530
530         if (lquote != def_lquote)
```

`set print address off`
Do not print addresses when displaying their contents. For example, this is the same stack frame displayed with `set print address off`:

```
(gdb) set print addr off
(gdb) f
#0  set_quotes (lq="<<", rq=">>") at input.c:530
```

```
530         if (lquote != def_lquote)
```

You can use `` `set print address off' `` to eliminate all machine dependent displays from the GDB interface. For example, with `print address off`, you should get the same text for backtraces on all machines--whether or not they involve pointer arguments.

`show print address`
: Show whether or not addresses are to be printed.

When GDB prints a symbolic address, it normally prints the closest earlier symbol plus an offset. If that symbol does not uniquely identify the address (for example, it is a name whose scope is a single source file), you may need to disambiguate. One way to do this is with `info line`, for example `` `info line *0x4537' ``. Alternately, you can set GDB to print the source file and line number when it prints a symbolic address:

`set print symbol-filename on`
: Tell GDB to print the source file name and line number of a symbol in the symbolic form of an address.

`set print symbol-filename off`
: Do not print source file name and line number of a symbol. This is the default.

`show print symbol-filename`
: Show whether or not GDB will print the source file name and line number of a symbol in the symbolic form of an address.

Another situation where it is helpful to show symbol filenames and line numbers is when disassembling code; GDB shows you the line number and source file that corresponds to each instruction.

Also, you may wish to see the symbolic form only if the address being printed is reasonably close to the closest earlier symbol:

`set print max-symbolic-offset` *`max-offset`*
: Tell GDB to only display the symbolic form of an address if the offset between the closest earlier symbol and the address is less than *max-offset*. The default is 0, which means to always print the symbolic form of an address, if any symbol precedes it.

`show print max-symbolic-offset`
: Ask how large the maximum offset is that GDB prints in a symbolic address.

If you have a pointer and you are not sure where it points, try `` `set print symbol-filename on' ``. Then you can determine the name and source file location of the variable where it points, using `` `p/a `` *`pointer`*`'`. This interprets the address in symbolic form. For example, here GDB shows that a variable `ptt` points at another variable `t`, defined in `` `hi2.c' ``:

```
(gdb) set print symbol-filename on
(gdb) p/a ptt
$4 = 0xe008 <t in hi2.c>
```

Warning: For pointers that point to a local variable, `` `p/a' `` does not show the symbol name and filename of the referent, even with the appropriate `set print` options turned on.

Other settings control how different kinds of objects are printed:

`set print array`
`set print array on`
Pretty-print arrays. This format is more convenient to read, but uses more space. The default is off.

`set print array off`
Return to compressed format for arrays.

`show print array`
Show whether compressed or pretty format is selected for displaying arrays.

`set print elements` *`number-of-elements`*
If GDB is printing a large array, it stops printing after it has printed the number of elements set by the `set print elements` command. This limit also applies to the display of strings. Setting the number of elements to zero means that the printing is unlimited.

`show print elements`
Display the number of elements of a large array that GDB prints before losing patience.

`set print pretty on`
Cause GDB to print structures in an indented format with one member per line, like this:

```
$1 = {
  next = 0x0,
  flags = {
    sweet = 1,
    sour = 1
  },
  meat = 0x54 "Pork"
}
```

`set print pretty off`
Cause GDB to print structures in a compact format, like this:

```
$1 = {next = 0x0, flags = {sweet = 1, sour = 1}, \
meat = 0x54 "Pork"}
```

This is the default format.

`show print pretty`

Show which format GDB is using to print structures.

`set print sevenbit-strings on`
: Print using only seven-bit characters; if this option is set, GDB displays any eight-bit characters (in strings or character values) using the notation `\`*nnn*. This setting is best if you are working in English (ASCII) and you use the high-order bit of characters as a marker or "meta" bit.

`set print sevenbit-strings off`
: Print full eight-bit characters. This allows the use of more international character sets, and is the default.

`show print sevenbit-strings`
: Show whether or not GDB is printing only seven-bit characters.

`set print union on`
: Tell GDB to print unions which are contained in structures. This is the default setting.

`set print union off`
: Tell GDB not to print unions which are contained in structures.

`show print union`
: Ask GDB whether or not it will print unions which are contained in structures.

For example, given the declarations

```
typedef enum {Tree, Bug} Species;
typedef enum {Big_tree, Acorn, Seedling} Tree_forms;
typedef enum {Caterpillar, Cocoon, Butterfly}
              Bug_forms;

struct thing {
  Species it;
  union {
    Tree_forms tree;
    Bug_forms bug;
  } form;
};

struct thing foo = {Tree, {Acorn}};
```

with `set print union on` in effect `` `p foo' `` would print

```
$1 = {it = Tree, form = {tree = Acorn, bug = Cocoon}}
```

and with `set print union off` in effect it would print

```
$1 = {it = Tree, form = {...}}
```

These settings are of interest when debugging C++ programs:

`set print demangle`
`set print demangle on`
: Print C++ names in their source form rather than in the encoded ("mangled") form passed to the assembler and linker for type-safe linkage. The default is `` `on' ``.

`show print demangle`
: Show whether C++ names are printed in mangled or demangled form.

`set print asm-demangle`
`set print asm-demangle on`
: Print C++ names in their source form rather than their mangled form, even in assembler code printouts such as instruction disassemblies. The default is off.

`show print asm-demangle`
: Show whether C++ names in assembly listings are printed in mangled or demangled form.

`set demangle-style` *`style`*
: Choose among several encoding schemes used by different compilers to represent C++ names. The choices for *style* are currently:

`auto`
: Allow GDB to choose a decoding style by inspecting your program.

`gnu`
: Decode based on the GNU C++ compiler (`g++`) encoding algorithm.

`lucid`
: Decode based on the Lucid C++ compiler (`lcc`) encoding algorithm.

`arm`
: Decode using the algorithm in the *C++ Annotated Reference Manual*. **Warning:** this setting alone is not sufficient to allow debugging `cfront`-generated executables. GDB would require further enhancement to permit that.

- show demangle-style Display the encoding style currently in use for decoding C++ symbols.

- set print object
- set print object on When displaying a pointer to an object, identify the *actual* (derived) type of the object rather than the *declared* type,

using the virtual function table.

- set print object off Display only the declared type of objects, without reference to the virtual function table. This is the default setting.

- show print object Show whether actual, or declared, object types are displayed.

- set print vtbl
- set print vtbl on Pretty print C++ virtual function tables. The default is off.

- set print vtbl off Do not pretty print C++ virtual function tables.

- show print vtbl Show whether C++ virtual function tables are pretty printed, or not.

Value history

Values printed by the `print` command are saved in the GDB *value history* so that you can refer to them in other expressions. Values are kept until the symbol table is re-read or discarded (for example with the `file` or `symbol-file` commands). When the symbol table changes, the value history is discarded, since the values may contain pointers back to the types defined in the symbol table.

The values printed are given *history numbers* by which you can refer to them. These are successive integers starting with one. `print` shows you the history number assigned to a value by printing ``$*num* = '` before the value; here *num* is the history number.

To refer to any previous value, use ``$'` followed by the value's history number. The way `print` labels its output is designed to remind you of this. Just `$` refers to the most recent value in the history, and `$$` refers to the value before that. `$$*n*` refers to the *n*th value from the end; `$$2` is the value just prior to `$$`, `$$1` is equivalent to `$$`, and `$$0` is equivalent to `$`.

For example, suppose you have just printed a pointer to a structure and want to see the contents of the structure. It suffices to type

```
p *$
```

If you have a chain of structures where the component `next` points to the next one, you can print the contents of the next one with this:

```
p *$.next
```

You can print successive links in the chain by repeating this command--which you can do by just typing `RET`.

Note that the history records values, not expressions. If the value of `x` is 4 and you type these commands:

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

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

`show values`
: Print the last ten values in the value history, with their item numbers. This is like `` `p $$9' `` repeated ten times, except that `show values` does not change the history.

`show values n`
: Print ten history values centered on history item number *n*.

`show values +`
: Print ten history values just after the values last printed. If no more values are available, produces no display.

Pressing `RET` to repeat `show values n` has exactly the same effect as `` `show values +' ``.

Convenience variables

GDB provides *convenience variables* that you can use within GDB to hold on to a value and refer to it later. These variables exist entirely within GDB; they are not part of your program, and setting a convenience variable has no direct effect on further execution of your program. That is why you can use them freely.

Convenience variables are prefixed with `` `$' ``. Any name preceded by `` `$' `` can be used for a convenience variable, unless it is one of the predefined machine-specific register names (see section Registers). (Value history references, in contrast, are *numbers* preceded by `` `$' ``. See section Value history.)

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:

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

would save in `$foo` the value contained in the object pointed to by `object_ptr`.

Using a convenience variable for the first time creates it, 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, including structures and arrays, even if that variable already has a value of a different type. The convenience variable, when used as an expression, has the type of its current value.

`show convenience`
: Print a list of convenience variables used so far, and their values. Abbreviated `show con`.

One of the ways to use a convenience variable is as a counter to be incremented or a pointer to be advanced. For example, to print a field from successive elements of an array of structures:

```
set $i = 0
print bar[$i++]->contents
... repeat that command by typing RET.
```

Some convenience variables are created automatically by GDB and given values likely to be useful.

`$_`
: The variable `$_` is automatically set by the `x` command to the last address examined (see section Examining memory). Other commands which provide a default address for `x` to examine also set `$_` to that address; these commands include `info line` and `info breakpoint`. The type of `$_` is `void *` except when set by the `x` command, in which case it is a pointer to the type of `$__`.

`$__`
: The variable `$__` is automatically set by the `x` command to the value found in the last address examined. Its type is chosen to match the format in which the data was printed.

Registers

You can refer to machine register contents, in expressions, as variables with names starting with `` `$' ``. The names of registers are different for each machine; use `info registers` to see the names used on your machine.

`info registers`
: Print the names and values of all registers except floating-point registers (in the selected stack frame).

`info all-registers`
: Print the names and values of all registers, including floating-point registers.

`info registers` *`regname`* `...`
: Print the relativized value of each specified register *regname*. *regname* may be any register name valid on the machine you are using, with or without the initial `` `$' ``.

GDB has four "standard" register names that are available (in expressions) on most machines--whenever they do not conflict with an architecture's canonical mnemonics for registers. The register names `$pc` and `$sp` are used for the program counter register and the stack pointer. `$fp` is used for a register that contains a pointer to the current stack frame, and `$ps` is used for a register that contains the processor status. For example, you could print the program counter in hex with

```
p/x $pc
```

or print the instruction to be executed next with

```
x/i $pc
```

or add four to the stack pointer(2) with

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

Whenever possible, these four standard register names are available on your machine even though the machine has different canonical mnemonics, so long as there is no conflict. The `info registers` command shows the canonical names. For example, on the SPARC, `info registers` displays the processor status register as `$psr` but you can also refer to it as `$ps`.

GDB always considers the contents of an ordinary register as an integer when the register is examined in this way. Some machines have special registers which can hold nothing but floating point; these registers are considered to have floating point values. There is no way to refer to the contents of an ordinary register as floating point value (although you can *print* it as a floating point value with `` `print/f $regname' ``).

Some registers have distinct "raw" and "virtual" data formats. This means that the data format in which the register contents are saved by the operating system is not the same one that your program normally sees. For example, the registers of the 68881 floating point coprocessor are always saved in "extended" (raw) format, but all C programs expect to work with "double" (virtual) format. In such cases, GDB normally works with the virtual format only (the format that makes sense for your program), but the `info registers` command prints the data in both formats.

Normally, register values are relative to the selected stack frame (see section Selecting a frame). This means that 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 true contents of hardware registers, you must select the innermost frame (with `` `frame 0' ``).

However, GDB must deduce where registers are saved, from the machine code generated by your compiler. If some registers are not saved, or if GDB is unable to locate the saved registers, the selected stack frame makes no difference.

`set rstack_high_address` *`address`*
: On AMD 29000 family processors, registers are saved in a separate "register stack". There is no way for GDB to determine the extent of this stack. Normally, GDB just assumes that the stack is "large enough". This may result in GDB referencing memory locations that do not exist. If necessary, you can get around this problem by specifying the ending address of the register stack with the `set rstack_high_address` command. The argument should be an address, which you probably want to precede with `` `0x' `` to specify in hexadecimal.

`show rstack_high_address`
: Display the current limit of the register stack, on AMD 29000 family processors.

Floating point hardware

Depending on the configuration, GDB may be able to give you more information about the status of the floating point hardware.

`info float`
: Display hardware-dependent information about the floating point unit. The exact contents and layout vary depending on the floating point chip; on some platforms, `` `info float' `` is not available at all.

Go to the previous, next section.