Go to the first, previous, next, last section, table of contents.

Setting breakpoints

Breakpoints are set with the `break` command (abbreviated `b`). The debugger convenience variable `` `$bpnum' `` records the number of the breakpoint you've set most recently; see section Convenience variables, for a discussion of what you can do with convenience variables.

You have several ways to say where the breakpoint should go.

`break` *`function`*
: Set a breakpoint at entry to function *function*. When using source languages that permit overloading of symbols, such as C++, *function* may refer to more than one possible place to break. See section Breakpoint menus, for a discussion of that situation.

`break` *`+offset`*
`break` *`-offset`*
: Set a breakpoint some number of lines forward or back from the position at which execution stopped in the currently selected *stack frame*. (See section Stack frames, for a description of stack frames.)

`break` *`linenum`*
: Set a breakpoint at line *linenum* in the current source file. The current source file is the last file whose source text was printed. The breakpoint will stop your program just before it executes any of the code on that line.

`break` *`filename`*`:`*`linenum`*
: Set a breakpoint at line *linenum* in source file *filename*.

`break` *`filename`*`:`*`function`*
: Set a breakpoint at entry to function *function* found in file *filename*. Specifying a file name as well as a function name is superfluous except when multiple files contain similarly named functions.

`break` *`*address`*
: Set a breakpoint at address *address*. You can use this to set breakpoints in parts of your program which do not have debugging information or source files.

`break`
: When called without any arguments, `break` sets a breakpoint at the next instruction to be executed in the selected stack frame (see section Examining the Stack). In any selected frame but the innermost, this makes your program stop as soon as control returns to that frame. This is similar to the effect of a `finish` command in the frame inside the selected frame--except that `finish` does not leave an active breakpoint. If you use `break` without an argument in the innermost frame, GDB stops the next time it reaches the current location; this may be useful inside loops. GDB normally ignores breakpoints when it resumes execution, until at least one instruction has been executed. If it did not do this, you would be unable to proceed past a breakpoint without first disabling the breakpoint. This rule applies whether or not the breakpoint already existed when your program stopped.

`break ... if` *`cond`*
: Set a breakpoint with condition *cond*; evaluate the expression *cond* each time the breakpoint is reached, and stop only if the value is nonzero--that is, if *cond* evaluates as true. `` `...' `` stands for one of the possible arguments described above (or no argument) specifying where to break. See section Break conditions, for more information on breakpoint conditions.

`tbreak` *`args`*
: Set a breakpoint enabled only for one stop. *args* are the same as for the `break` command, and the breakpoint is set in the same way, but the breakpoint is automatically deleted after the first time your program stops there. See section Disabling breakpoints.

`hbreak` *`args`*
: Set a hardware-assisted breakpoint. *args* are the same as for the `break` command and the breakpoint is set in the same way, but the breakpoint requires hardware support and some target hardware may not have this support. The main purpose of this is EPROM/ROM code debugging, so you can set a breakpoint at an instruction without changing the instruction. This can be used with the new trap-generation provided by SPARClite DSU and some x86-based targets. These targets will generate traps when a program accesses some data or instruction address that is assigned to the debug registers. However the hardware breakpoint registers can take a limited number of breakpoints. For example, on the DSU, only two data breakpoints can be set at a time, and GDB will reject this command if more

than two are used. Delete or disable unused hardware breakpoints before setting new ones (see section Disabling breakpoints). See section Break conditions.

`thbreak` *`args`*
: Set a hardware-assisted breakpoint enabled only for one stop. *args* are the same as for the `hbreak` command and the breakpoint is set in the same way. However, like the `tbreak` command, the breakpoint is automatically deleted after the first time your program stops there. Also, like the `hbreak` command, the breakpoint requires hardware support and some target hardware may not have this support. See section Disabling breakpoints. See also section Break conditions.

`rbreak` *`regex`*
: Set breakpoints on all functions matching the regular expression *regex*. This command sets an unconditional breakpoint on all matches, printing a list of all breakpoints it set. Once these breakpoints are set, they are treated just like the breakpoints set with the `break` command. You can delete them, disable them, or make them conditional the same way as any other breakpoint. The syntax of the regular expression is the standard one used with tools like `` `grep' ``. Note that this is different from the syntax used by shells, so for instance `foo*` matches all functions that include an `fo` followed by zero or more os. There is an implicit `.*` leading and trailing the regular expression you supply, so to match only functions that begin with `foo`, use `^foo`. When debugging C++ programs, `rbreak` is useful for setting breakpoints on overloaded functions that are not members of any special classes.

`info breakpoints [`*`n`*`]`
`info break [`*`n`*`]`
`info watchpoints [`*`n`*`]`
: Print a table of all breakpoints, watchpoints, and catchpoints set and not deleted, with the following columns for each breakpoint:

  Breakpoint Numbers
  Type
  : Breakpoint, watchpoint, or catchpoint.
  Disposition
  : Whether the breakpoint is marked to be disabled or deleted when hit.
  Enabled or Disabled
  : Enabled breakpoints are marked with `` `y' ``. `` `n' `` marks breakpoints that are not enabled.
  Address
  : Where the breakpoint is in your program, as a memory address.
  What
  : Where the breakpoint is in the source for your program, as a file and line number.

  If a breakpoint is conditional, `info break` shows the condition on the line following the affected breakpoint; breakpoint commands, if any, are listed after that. `info break` with a breakpoint number *n* as argument lists only that breakpoint. The convenience variable `$_` and the default examining-address for the `x` command are set to the address of the last breakpoint listed (see section Examining memory). `info break` displays a count of the number of times the breakpoint has been hit. This is especially useful in conjunction with the `ignore` command. You can ignore a large number of breakpoint hits, look at the breakpoint info to see how many times the breakpoint was hit, and then run again, ignoring one less than that number. This will get you quickly to the last hit of that breakpoint.

GDB allows you to set any number of breakpoints at the same place in your program. There is nothing silly or meaningless about this. When the breakpoints are conditional, this is even useful (see section Break conditions).

GDB itself sometimes sets breakpoints in your program for special purposes, such as proper handling of `longjmp` (in C programs). These internal breakpoints are assigned negative numbers, starting with `-1`; `` `info breakpoints' `` does not display them.

You can see these breakpoints with the GDB maintenance command `` `maint info breakpoints' ``.

`maint info breakpoints`
: Using the same format as `` `info breakpoints' ``, display both the breakpoints you've set explicitly, and those GDB is using for internal purposes. Internal breakpoints are shown with negative breakpoint numbers. The type column identifies what kind of breakpoint is shown:

`breakpoint`
: Normal, explicitly set breakpoint.

`watchpoint`
: Normal, explicitly set watchpoint.

`longjmp`
: Internal breakpoint, used to handle correctly stepping through `longjmp` calls.

`longjmp resume`
: Internal breakpoint at the target of a `longjmp`.

`until`
: Temporary internal breakpoint used by the GDB `until` command.

`finish`
: Temporary internal breakpoint used by the GDB `finish` command.

`shlib events`
: Shared library events.

Go to the first, previous, next, last section, table of contents.