4 Stopping and Continuing Your Program

This chapter describes how to stop and restart programs running under the iWarp debugger. The chapter includes the following:

- using breakpoints
- stepping through a program

Breakpoints

A breakpoint makes your program stop whenever a certain point in the program is reached. You set breakpoints explicitly with debugger commands, specifying the place where the program should stop by line number, function name, or exact address in the program. You can add various other conditions to control whether the program will stop. The debugger allows you to set multiple breakpoints at the same place in the program.

Each breakpoint is assigned a number when it is created. These numbers are successive integers starting with 1. In many of the commands for controlling various features of breakpoints you must use the breakpoint number to indicate which breakpoint you want to change. Each breakpoint may be enabled or disabled. If disabled, it has no effect on the program until you enable it again.

To get information about existing breakpoints, use the `info break` command. This command provides the following information about each breakpoint:

- breakpoint number
- location in program
- special features in use
- whether the breakpoint is enabled or disabled

You can also include a breakpoint number as an argument to the `info break` command, and the command will only print information about the specified breakpoint.

Setting Breakpoints

Breakpoints are set with the `break` command. The `break` command has the following syntax:

```
break argument
```

argument can be any of the following:

`+offset`	Sets a breakpoint *offset* lines forward from the position at which execution stopped in the currently selected frame.
`-offset`	Sets a breakpoint *offset* lines backward from the position at which execution stopped in the currently selected frame.
`[filename:]linenum`	Set a breakpoint at line *linenum* in source file *filename*. If *filename* is not specified, *filename* becomes the current source file.

`[filename:]function` Sets a breakpoint at entry to *function* found in file *filename*. If *filename* is not specified, *filename* becomes the current source file.

`*address` Sets a breakpoint at *address*. You can use this to set breakpoints in parts of the program which do not have debugging information or source files, or to specify a point within a source line containing many object instructions.

If you do not include an argument with the `break` command, the debugger sets a breakpoint at the next instruction to be executed in the selected stack frame. In any selected frame but the innermost, this will cause the program to stop as soon as control returns to that frame. This is equivalent to a `finish` command in the frame inside the selected frame. If this is done in the innermost frame, the debugger will stop the next time it reaches the current location. This may be useful inside loops.

The debugger normally ignores breakpoints when it resumes execution, until at least one instruction has been executed. If it did not do this, you could not proceed past a breakpoint without first disabling the breakpoint. This rule applies whether or not the breakpoint already existed when the program stopped.

The debugger allows you to set breakpoints for only one stop by using the `tbreak` command. The `tbreak` command has the following syntax:

```
tbreak argument
```

argument can be any of the arguments listed for the `break` command.

The `tbreak` command sets a breakpoint at the specified location, enables it for one stop, and then disables the breakpoint the first time it is hit.

Break Conditions

The simplest sort of breakpoint breaks every time the program reaches a specified place. You can also specify a condition for a breakpoint. A condition is a boolean expression that is evaluated each time the program reaches the breakpoint, and the program stops only if the condition is true.

Break conditions may have side effects and may even call functions in your program. It is usually more convenient and flexible however, to use breakpoint commands to perform side effects when a breakpoint is reached.

Setting Breakpoint Conditions

Break conditions can be specified when a breakpoint is set, by using the following `break` command syntax:

```
break argument if condition
```

argument can be any of the arguments listed for the `break` command.

condition is a boolean expression. The program stops at the breakpoint only if *condition* is true.

Break conditions can also be changed at any time with the `condition` command. The `condition` command has the following syntax:

```
condition bnum expression
```

bnum is a breakpoint number.

expression is the break condition for the specified breakpoint. The breakpoint will stop the program only if the value of *expression* is true. *expression* is not evaluated at the time the `condition` command is given.

You can also remove a condition from a breakpoint by using the `condition` command as follows:

```
condition  bnum
```

bnum is the breakpoint number.

Using the Breakpoint Ignore Count

A special case of a breakpoint condition is to stop only when the breakpoint has been reached a certain number of times. Every breakpoint has an ignore count, which is an integer. The default ignore count is zero, which has no effect on the program. If the program reaches a breakpoint whose ignore count is positive, instead of stopping, the debugger decrements the ignore count by one and continues. As a result, if the ignore count value is *n*, the breakpoint will not stop the next *n* times it is reached. To set the ignore count of a breakpoint, use the following command:

```
ignore bnum count
```

bnum is a breakpoint number.

count is an integer.

This command sets the ignore count of breakpoint number *bnum* to *count*. The next *count* times the breakpoint is reached, it will not stop. To make the breakpoint stop the next time it is reached, specify a count of zero.

You can also set the ignore count of a breakpoint when the program has stopped at that breakpoint by using the following command:

```
cont count
```

count is an integer.

This command continues execution of the program and sets the ignore count of the breakpoint that the program stopped at to *count* minus one. The program will not stop at this breakpoint again until the breakpoint is reached *count* times. This command is allowed only when the program stopped due to a breakpoint. At other times, the argument to `cont` is ignored.

If a breakpoint has a positive ignore count and a condition, the condition is not checked. Once the ignore count reaches zero, the condition will be checked.

Removing Breakpoints

It is often necessary to remove a breakpoint once it has been reached and you no longer want the program to stop there. You can use the following commands to clear breakpoints:

`clear`	delete breakpoints according to where they are in the program
`delete`	delete individual breakpoints by specifying their breakpoint numbers

The `clear` Command

The `clear` command has the following syntax:

```
clear [argument]
```

argument can be any of the following:

`[filename:]function`	deletes any breakpoints set at entry to *function* found in file *filename*

`[`*`filename:`*`]`*`linenum`* deletes any breakpoints set at or within the code of the specified line found in file *filename*

If you do not include any arguments with the `clear` command, the debugger deletes any breakpoints at the next instruction to be executed in the selected stack frame. You can delete a breakpoint the program just stopped at by selecting the innermost stack frame.

The **`delete`** Command

The `delete` command has the following syntax:

`delete` *`bnums`*

bnums are the numbers of any breakpoints to be deleted.

Disabling Breakpoints

Rather than deleting a breakpoint, you might want to disable it. This makes the breakpoint inoperative just as if it had been deleted, but the debugger retains the information on the breakpoint so you can enable it again later.

A breakpoint can have any of four different states of enablement:

- enabled — The breakpoint will stop the program. A breakpoint made with the `break` command starts out in this state.
- disabled — The breakpoint has no effect on the program.
- enabled once — The breakpoint will stop the program, but when it does so it will become disabled. A breakpoint made with the `tbreak` command starts out in this state.
- enabled for deletion — The breakpoint will stop the program, but immediately after it does so it will be deleted permanently.

You disable breakpoints with the `disable` command. The `disable` command has the following syntax:

```
disable bnums
```

bnums are the numbers of any breakpoints to be disabled. You can use the `info break` command to print a list of breakpoints if you don't know which breakpoint numbers to use. A disabled breakpoint has no effect, but all options such as ignore counts, conditions, and commands are remembered in case the breakpoint is enabled again.

Enabling Breakpoints

The `enable` command enables disabled breakpoints. The `enable` command has the following syntax:

```
enable argument bnums
```

bnums are the numbers of any breakpoints to be enabled.

argument can be any of the following:

`once`	Enables the specified breakpoints temporarily. Each will be disabled again the next time it stops the program.
`delete`	Enables the specified breakpoints to work once and then be removed. Each of the breakpoints will be deleted the next time it stops the program.

Commands Executed on Breaking

You can give any breakpoint a series of commands to execute when the program stops at that breakpoint. For example, you might want to print the values of certain expressions or enable other breakpoints. To give a breakpoint a series of commands, use the `commands` command as follows:

```
commands bnum
```

bnum is a breakpoint number. Enter the commands to be executed at the breakpoint on the lines that follow the `commands` command. Type a line containing only `end` to terminate the commands. If no breakpoint number is specified, `commands` refers to the last breakpoint set.

To remove all commands from a breakpoint, enter `commands` and follow it immediately with `end.`

The following example illustrates the use of breakpoint commands to print the value of `x` at entry to `foo` whenever it is positive.

```
break foo if x>0
commands
silent
echo x is\040
output x
echo \n
cont
end
```

One application for breakpoint commands is to correct one bug so you can test another. Put a breakpoint just after the erroneous line of code, give it a condition to detect the case in which something erroneous has been done, and give it commands to assign correct values to any variables that need them. Start with the `silent` command so no output is produced, and end with the `cont` command so the program does not stop. The following example illustrates this application:

```
break 403
commands
silent
set x = y + 4
cont
end
condition 5  (x = y + 4), 0
```

Continuing

After your program stops, you will probably want to continue if the bug you are looking for has not yet occurred. To continue execution of a stopped program, use the `cont` command as follows:

```
cont
```

This command resumes program execution at the place where the program stopped.

If the program stopped at a breakpoint, the place to continue running is the address of the breakpoint. You might expect that continuing would immediately stop at the same breakpoint, but the `cont` command moves the program though the breakpoint. You do not need to delete the breakpoint to proceed through it.

Stepping

Stepping means setting your program in motion for a limited time, so that control will return automatically to the debugger after a specified number of lines of code or machine instructions. Breakpoints are active during stepping, and the program will stop for them even if it has not gone as far as the stepping command specifies.

A typical technique for using stepping is to put a breakpoint at the beginning of the function or program section in which a problem is believed to exist, and then step through the suspect area, examining the relevant variables until the problem occurs. The `cont` command can be used after stepping to resume execution until the next breakpoint or signal.

There are several commands that allow you to step through a program. These include:

- `step`
- `next`
- `finish`
- `until`
- `stepi`
- `nexti`

The `step` Command

The `step` command has the following syntax:

```
step [count]
```

The `step` command causes the debugger to continue running the program until *count* source lines have been executed, then stop the program and return control to the debugger. If *count* is not specified, *count* becomes one. If a breakpoint is reached or a signal not related to stepping occurs before *count* steps, stepping stops right away.

The **`next`** Command

The `next` command has the following syntax:

```
next [count]
```

The `next` command is similar to `step`, but any function calls appearing within the line of code are executed without stopping. Execution stops when control reaches a different line of code at the stack level which was executing when the `next` command was given.

The **`finish`** Command

The `finish` command has the following syntax:

```
finish
```

The `finish` command causes the debugger to continue running the program until the selected stack frame returns, or until there is some other reason to stop, such as a fatal signal or a breakpoint.

The **`until`** Command

The `until` command has the following syntax:

```
until [location]
```

location is a memory location. The `until` command continues running the program until either the specified location is reached, or the current (innermost) stack frame returns. If *location* is not specified, `until` uses single-instruction stepping.

The `until` command is used to avoid single stepping through a loop more than once. It operates like the `next` command, except when `until` encounters a jump, it automatically continues execution until the program counter is greater than the address of the jump. When you reach the end of a loop after single stepping though it, `until` causes the program to continue execution until the loop is exited.

The `until` command always stops the program if it attempts to exit the current stack frame.

The **`stepi`** Command

The `stepi` command has the following syntax:

```
stepi [count]
```

The `stepi` command executes *count* machine instructions, then stops the program and returns control to the debugger. If *count* is not specified, *count* becomes one.

The **`nexti`** Command

The `nexti` command has the following syntax:

```
nexti [count]
```

The `nexti` command is similar to `stepi`, but any function calls appearing within the line of code are executed without stopping.