AttachingtoanAlreadyRunningProcess;¬Attaching to an Already Running Process

Sometimes, a problem occurs only when you launch an application outside the debugger. When you launch it inside **gdb**, the problem disappears. If this happens , launch the application using the launch button and use the **gdb** command **attach** to hook up to it:

```
(gdb) attach pid
```

pid is the process ID of the process you want to debug.

attach immediately stops the application. When you use **attach**, you can debug the process just like you normally would: by setting breakpoints, modifying storage, and so on.

When you're finished debugging, use **detach** (which takes no arguments) to detach the debugger from the process. The process resumes executing. You'll kill the process if you try to quit **gdb** or if you try to start the program from the beginning. (**gdb** asks for confirmation before it allows you to do these things.)

If you're having trouble attaching to a process before the errant code is executed, send your program a stop signal as one of the first messages:

```
[NSThread sleepUntilDate:[NSDate distantFuture]];
```

This indefinitely suspends execution of the application. Once you attach in **gdb**, click the continue button to go on from there.

370776_TableRule.eps ¬

Settingbreakpointsondata;¬Setting breakpoints on data

Sometimes you want to stop the program whenever the value of a variable changes, no matter which part of your code is doing the changing. To do this, use a watchpoint . To set a watchpoint:

```
(gdb) watch expr
```

where *expr* is any expression or variable.

gdb treats watchpoints and breakpoints the same. Anything you can do to a breakpoint, you can also do to a watchpoint (see ªCool Breakpoint Stuffº in this chapter ;DebuggingConcepts.rtfd;CoolBreakpointStuff;¬). The Breakpoints display of the Task Inspector provides information on both breakpoints and watchpoints.

When a watchpoint is set, your program runs much more slowly that if you had set a normal breakpoint, so use watchpoints sparingly. (One alternative is to set a conditional breakpoint, described in ªCool Breakpoint Stuff.º) However, watchpoints are sometimes the only way to catch an error when you don't know where the error occurs.

370776_TableRule.eps ¬

CoolBreakpointStuff;¬Cool Breakpoint Stuff

Using **gdb** commands, you can add more power to your breakpoints and make debugging a breeze. For complete information on **gdb** breakpoints, see the *OPENSTEP Development Tools Reference* manual. ;/NextLibrary/Documentation/NextDev/Reference/DevTools/Debugger/Debugger.rtf;;¬ Here are some highlights.

Setting Breakpoints in Dynamically Loaded Code

gdb doesn't know about symbols in dynamically loaded code (such as code inside frameworks or loadable bundles) because it's not laoded until run time. This means you can't set a breakpoint in a framework until after you start the program that uses it. This is pretty frustrating when the framework is what you want to debug. However, you can set a future breakpoint when the framework isn't loaded yet. To do this, use the **future-break** command:

```
(gdb) future-break address
```

(*address* can be a method name, a function name, a file name and line number, and so on.) When you enter

this command, **gdb** checks the loaded symbols for a symbol matching *address*. If one is found, it resolves the breakpoint. If not, it holds on to it. Then, whenever a dynamic shared library is loaded, **gdb** checks the breakpoint against the newly loaded symbols until it can resolve the symbol in the breakpoint. (If the symbol can never be resolved, the **future-break** just sits around doing nothing.)

When you quit the program, **gdb** unloads all of the breakpoints set in dynamic shared libraries. These breakpoints are converted into future breakpointsÐwhen the library is loaded again, the breakpoints are resolved again.

Future breakpoints are just like normal breakpoints in every other respect; you can add commands to them, disable them, enable them, and so on. In the Breakpoints display of the Task Inspector, they are listed as ªunloaded.º

Conditional Breakpoints

If you only want a breakpoint to stop when a certain condition is true, use the **condition** command:

```
(gdb) condition bnum expression
```

expression is any Boolean expression and it's associated with breakpoint number *bnum*. (The Breakpoints view of the Task Inspector tells you the breakpoint number.) From now on, this breakpoint will stop the program only if the value of *expression* is true. To remove a condition from a breakpoint, enter **condition** with no *expression*.

Ignoring Breakpoints

You can disable a breakpoint for a specific length of time with **gdb** command **ignore**:

```
(gdb) ignore bnum count
```

This command ignores the breakpoint the next *count* times it is reached. (0 means the program stops the next time it's reached.) If the breakpoint is a conditional breakpoint, the condition isn't checked unless the ignore count is 0.

Executing Commands at a Breakpoint

You can give any breakpoint a series of commands to execute when the program stops at it. For example, if you want to know what the value of the variable x is whenever breakpoint 5 is hit, enter the following. (You must type **end** when you're through to make the **gdb** prompt return.)

```
(gdb) commands 5
> print x
> end
```

This brings up a handy trick for ignoring breakpoints. Often, you don't know how many times you want to ignore a breakpoint (making the **ignore** command useless), but you know that you want to ignore it until a specific point in a program is reached. For example, say you want to stop at a method named **setCurrent:** but only if the message is sent by the **processParagraph** method. In this case, you can do the following:

```
(gdb) break setCurrent:
Breakpoint #1 set
(gdb) break processParagraph
Breakpoint #2 set
(gdb) disable 1
(gdb) commands 2
> silent
> enable 1
> continue
> end
(gdb) continue
```

This example sets two breakpoints, one at the beginning of each method. Then, it disables the breakpoint at **setCurrent:**. When the breakpoint at **processParagraph** is reached, it enables the breakpoint at **setCurrent:** and continues executing. (**silent** is just a convenience. It means that **gdb** won't print the usual stopped at breakpoint message.)

370776_TableRule.eps ¬

GettingUsefulInformationFromPrint-object;¬Getting Useful Information From Print-object

The Print-object button (which invokes the **gdb** command **print-object**) sends the message **description** to the selected object. NSObject defines the **description** method, so all objects respond to it. By default, this method prints the object's class name and hexidecimal address:

```
<NSApplication: 0xbb5e4>
```

However, you can override this method in your classes to provide more useful data. Compared to dumping the contents of the underlying struct, an implementation of **description** can print out just the information that is helpful and use a more readable format. Your **description** method should return an NSString.

Many Foundation classes override **description.** For example, NSArrays, NSDictionaries, and NSStrings print their contents instead of their addresses.

370776_TableRule.eps ¬

FortheExperts:MoreonExaminingVariables;¬For the Experts: More on Examining Variables

Making Sure Variables Stick Around

When you build the program using the default build target (for example, app for Application projects), an optimized, debuggable executable results. This executable is helpful if a bug surfaces only in the optimized version; however, debugging optimized code sometimes gives surprising results. Control flow may change and variables may disappear without a trace. You ask **gdb** to print such a variable and even though the source clearly shows it is in scope, **gdb** replies:

```
(gdb) print num
No symbol "num" in current context
```

To ensure that a variable be available in the debugger even after optiumization, declare the variable **volatile**.

Value History

gdb maintains a value history for your session. This means that every expression you evaluate using the **print** command (or the Print, Print *, and PO buttons) is assigned a value number in the history, like this:

```
(gdb) print self
$7 = (struct NSApplication *) 0xbb5e4
```

You can refer to this value as $7 and use it in future expressions:

```
(gdb) print (char *) [$7 appName]
$8 = 0xb80cc "FunWithGDB"
```

Once a value is entered into the history, it doesn't change. The value is stored as $7, not the expression that generated it. This means that $7 doesn't change to hold the new value of **self** when your program enters a different scope.

Also, at any time, $ refers to the last value in the history and $$ to the next-to-last value.

The **output** command has the same semantics as the **print** command, but doesn't add the result to the value history. You can use this difference to avoid cluttering the value history with unimportant results. For more sophisticated printing needs, **gdb** provides a **printf** command similar to the C version that provides for formatted output. Like **output**, the results from **printf** are not entered into the value history.

Any name that begins with a $ can be used as the name of a **gdb** convenience variable. These variables are implicitly typed and created at first reference. Use **print** to get the value of a convenience variable and the **set** command to set or change the value. You can set the value to any valid C or Objective-C expression, including methods or functions:

```
(gdb) p $array = [NSArray array]
$24 = 793052
(gdb) p $num = 1230 % 4
$25 = 2
```

All registers have convenience variables associated with them. The **info registers** command dumps the contents of all registers so you can see the names associated with each register. The register convenience variables most often used are **$fp**, which holds the frame pointer, **$sp** for the stack pointer, and **$pc** for the program counter.

Locating Your Variables

To find out how a variable is stored, use this command:

```
(gdb) info address self
Symbol "self" is a variable in register a2.
```

info address tells you if the variable is stored on the stack or in a register. This command is useful to determine if optimizations are causing problems, particularly on RISC machines.

Examining Raw Memory

Use the command **x** (for ªexamineº) to examine memory without referencing the program's data types.

x is followed by a slash and an output format specification, followed by an expression for an address:

```
x/fmt addr
```

These *fmt* letters specify the size of unit to examine:

b Examine individual bytes.
h Examine halfwords (two bytes each).
w Examine words (four bytes each).
g Examine giant words (eight bytes).

These *fmt* letters specify how to print the contents:

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 and relative
c Print as character constants (this implies size b).
f Print as floating-point. This works only with sizes w and g.
s Print a null-terminated string of characters.
i Print a machine instruction in assembler syntax (or nearly).

Once you've entered **x** to see the value at an address, hit return to see the value at the next address.

370776_TableRule.eps ¬

IgnoringAutoreleaseErrors;¬Ignoring Autorelease Errors

You may want to debug the rest of your program first, saving the release problems until later. The **enableRelease:** convenience method defined in Foundation's NSAutoreleasePool class helps you ignore autorelease errors. NSAutoreleasePool defines the application's autorelease pool. When an object is autoreleased, it is added to the autorelease pool. At the top of the event loop, all objects in the pool are sent a **release** message, which decrements the reference count and potentially deallocates the object. NSAutoreleasePool allows you to control that pool.

If you receive messages from the debugger indicating that you are sending messages to deallocated objects, enter this command:

```
(gdb) call [NSAutoreleasePool enableRelease:NO]
```

This message disables the deallocation of all objects in your program, ignoring autorelease errors.

Your program must be started when you send this message. It's often useful to break on **main()** and send this message after the first line or two of the program.

370776_TableRule.eps ¬

CommonAutoreleaseMistakes;¬Common Autorelease Mistakes

Once you find the object with the autorelease error, look for the following:

- For every **autorelease** and **release** message in your application, make sure there is a corresponding **alloc**, **copy**, **mutableCopy**, or **retain** message sent to the same object. **autorelease** and **release** decrement an object's reference count. **alloc**, **copy**, **mutableCopy**, and **retain** increment the reference count. The number of increments and decrements for an object must be equal. Another way of thinking about this is: If you don't allocate, copy, or retain an object, you're not responsible for releasing it.
- When an NSArray, NSDictionary, or NSSet (known as the collection objects) is deallocated, the objects stored in the collection are released as well. If you need to access an object you stored in a collection after the collection is released, you must retain that object before you release the collection.
- Superviews retain subviews as you add them to the hierarchy and release subviews as you remove them from the hierarchy. If you swap views in and out of the hierarchy, you should retain the views that are not in the hierarchy.
- When you change a window's content view, the window releases the old content view and retains the new content view.
- Objects do not retain their delegates (to avoid retain cycles).
- **decodeValuesOfObjCTypes:** returns a retained object. **decodeObject** returns an autoreleased object. If you unarchive an instance variable with **decodeObject**, send it the **retain** or **copy** message.