Technical Note TN2030

GDB for MacsBug Veterans

CONTENTS

This technote shows how to translate your experience with MacsBug into a working knowledge of the GNU source-level debugger (GDB) as used on Mac OS X. The focus is on using GDB as an assembly-level tool for debugging programs for which you don't have source (for example, debugging crashes inside Carbon itself) or for which the source is inaccessible (for example, debugging a crash that only occurs at a customer's site). However, many of the tricks you learn here will also be useful for more mundane source-level debugging.

This technote is primarily directed at application developers (and other developers of user space code) who are familiar with MacsBug and want a quick introduction to GDB. If you're not already familiar with MacsBug, you probably should skip this technote and go straight to the GDB manual. If you're developing kernel code (for example, an IOKit driver) you will find most of this technote useful but you must also learn the two-machine debugging techniques described in other Apple documentation.

[Oct 18 2001]

Introduction

The first step in a successful transition from MacsBug to GDB is...

Don't Panic

If you're used to MacsBug, learning GDB won't be that hard. And while you'll miss some of MacsBug's nicer features (the `showpath` 'dcmd' for example) you'll be impressed by some of the cool things that GDB can do for you (future break is one of my favorites). So, let's get started.

About GDB

The GNU source-level debugger (GDB) is the primary debugging tool on Mac OS X. Whenever you debug on Mac OS X, you are using GDB in one form or another.

- GDB is available as a standard command-line tool (`gdb`) that you can run from Terminal.
- Project Builder provides a graphical front end to GDB's commonly-used debugging facilities. However, there are cases where you might want to enter GDB commands directly, either by starting GDB from the command line or by entering commands at Project Builder's GDB prompt.
- The CodeWarrior source-level debugger is currently layered on top of GDB (although there is no direct access to GDB from within CodeWarrior).

This technote concentrates on using GDB from the command line. This is in line with the overall goal of exploring GDB as an assembly-level debugging tool.

While it is possible to define macros to make GDB behave like MacsBug (for an amazing example, check out the MacsBug GDB plug-in), that's not a good long-term choice in my opinion. Mac OS X is the long-term direction of Apple, and GDB will be the core debugger on Mac OS X for the foreseeable future. It behooves you to learn GDB now rather than later.

GDB, as its full name suggests, is primarily a source-level debugger. This has its pros and cons. There's no denying that source-level debugging is easier and its absence in MacsBug has always been sorely missed. However, GDB's focus on source-level debugging means that it isn't a great assembly-level debugger. It is, however, good enough to get the job done.

Conventions Used In This Technote

Throughout this technote the following styles have special meaning.

- `Monospace text` indicates a GDB or MacsBug command, or text output by the computer.
- **`Bold monospace text`** indicates commands entered by the user.
- *`Italic monospace text`* indicates comments.
- A backslash (\) at the end of a line indicates that the line is continued on the next line.
- `^X` indicates that you should enter a control character (for example, the notation ^C means you should hold down the control key and press the C key).
- In debugger command descriptions, meta variables are shown in angle brackets. For example, `DM <addr>` indicates that you should substitute an address for `<addr>`.

GDB Basics

When you break into MacsBug on traditional Mac OS you stop the entire computer. This is not the case with GDB on Macs OS X. When you break into GDB you stop only the process that you're debugging. This allows you to look up source code and documentation while you're debugging.

There are two basic ways of targeting the process you want to debug.

- Attach to the process after it's started running
- Run your program from within GDB.

The first approach has the advantage that you can start debugging a program after it's been executing for some time. You don't need to know in advance that this is a debugging session. You can start debugging when you notice the program misbehaving. The disadvantage of this approach is that you have to jump through some hoops to see the program's output to `stdout` and `stderr`. The technique for doing this is explained later in this document.

The second approach lets you see your program's output to `stdout` and `stderr` easily. You should use it when you're actively debugging a problem.

Attaching to a Running Process

Listing 1 shows how to attach GDB to a running process. You start by launching Terminal and running GDB from the command line (**`gdb`**). Once you've started GDB you can either attach by name or by BSD process ID. First you should try attaching by name (**`attach BBEdit`**). This can fail for a variety of reasons--in this example it failed because the application is a CFM binary--in which case you should attach by BSD process ID. You can find the BSD process ID using GDB's `shell` command to execute the `ps` command line tool. This example uses `grep` to search the output (which would be rather large otherwise). The first column of numbers is the BSD process ID. You can use that BSD process ID to attach to the process (**`attach 795`**).

Listing 1. Attaching to a running process

```
[localhost:~] quinn% gdb
[... output omitted ...]
(gdb) attach BBEdit
Unable to locate process named "BBEdit".
(gdb) shell ps auxww | grep BBEdit
quinn   823   [...] -csh -c ps auxww | grep BBEdit (tcsh)
quinn   795   [...] /Volumes/YipYip/MyApplications/BBEdit [...]
(gdb) attach 795
[... output omitted ...]
(gdb)
```

IMPORTANT:
A BSD process ID (PID) is the BSD way of identifying a process. It is not the same as a Carbon Process Serial Number (PSN), although there are routines to map between the two in "Processes.h".

Note:
To attach to a running application either it must by owned by you or you must have root privileges.

Note:
The `attach` command supports Tab completion. If you type a partial process name and press Tab, GDB will complete the full name.

Running Your Program From Within GDB

Running a program from within GDB requires different actions depending on the program's binary format. For Mach-O programs you can simply enter the path to the executable on the command line as you launch GDB. This is shown in Listing 2. GDB will stop after reading the program's symbols. You can then set breakpoints, or just run the program (**`r`**).

Listing 2. Running a Mach-O program under GDB

```
[localhost:~] quinn% gdb /Applications/TextEdit.app/\
Contents/MacOS/TextEdit
[... output omitted ...]
(gdb) r
```

IMPORTANT:
You must enter the path to the executable, not the package. For example, enter "/Applications/TextEdit.app/Contents/MacOS/TextEdit" and not "/Applications/TextEdit.app".

If your program is built as a CFM binary you must do a little more work. Rather than pointing GDB at the application itself, you must point it at the CFM loader program (LaunchCFMApp) that is responsible for loading and running the CFM application. Listing 3 shows how this is done. You must enter the path to the CFM binary (again, this is the executable, not the package) as an argument to the run (`r`) command. This tells LaunchCFMApp which CFM application you want to launch.

Listing 3. Running a CFM program under GDB

```
[localhost:~] quinn% gdb /System/Library/Frameworks/\
Carbon.framework/Versions/A/Support/LaunchCFMApp
[... output omitted ...]
(gdb) r /Applications/Internet\ Explorer/\
Internet\ Explorer.app/Contents/MacOS/Internet\ Explorer
```

Breaking into GDB and Entering Commands

To stop your program in the debugger, just switch to the appropriate Terminal window and enter ^C. All the threads in your program will stop and GDB will present you with the `(gdb)` prompt. Your program will immediately stop responding to events (if you trying to click on one of its windows you will get the spinning rainbow beach ball cursor of death™). You can continue executing your program with the continue command (**`c`**). Listing 4 shows this entire process.

Listing 4. Stopping and continuing a program

```
^C
Program received signal SIGINT, Interrupt.
0x700009a8 in mach_msg_overwrite_trap ()
(gdb) c
Continuing.
```

When you see the GDB prompt you can enter GDB commands. GDB accepts commands in much the same way as MacsBug. You type the name of the command followed by any command arguments and then press Return to execute the command. Like MacsBug, GDB supports command line editing and history. You can also press Return to re-execute the previous command. However, there are two significant differences from the basic MacsBug command loop.

- MacsBug commands tend to be short and cryptic. GDB commands, on the other hand, are long and explanatory. Fortunately, you don't have to type the entire command. Any unambiguous abbreviation of the command will do. For example, the full command to see the registers is `info registers`. A more succinct abbreviation is `info reg`. However, if you really hate typing you can abbreviate this all the way down to `i r`. GDB has special logic that allows one letter commands to work properly even if they are ambiguous. For example, even though `c` is the first letter of many commands, it is always interpreted as (the very commonly used) `continue` command. Finally, pressing Tab will autocomplete an unambiguous command and pressing Tab twice will list alternatives for an ambiguous command.
- In MacsBug you can type an expression on the command line and MacsBug will evaluate it and display the result. GDB requires you to do something with the expression. If you want to see its value, simply print it using the `print` command (or `p` for short). Unlike MacsBug, the print command only displays the results in one format at a time. To control that format, append a '/' and a format specifier ('o' for octal, 'x' for hex, 'd' for decimal, 'u' for unsigned decimal, 't' for binary, 'f' for float, 'a' for address, 'c' for char). You can also append a size qualifier ('b' for 1 byte, 'h' for 2 bytes, 'w' for 4 bytes, 'g' for 8 bytes).

MacsBug	GDB
`command-power`	`^C` [1]
`G` or `command-G`	`c` [2]
`HELP <command>`	`help <command>`
	`apropos <regex>` [3]
`<expr>`	`p/x <expr>` [4]

Notes:

1. You must enter `^C` in the Terminal window in which you're running GDB.
2. `c` is short for `continue`.
3. `apropos` will print all GDB help entries that match the regular expression. This is really useful if you're totally lost.

For example, lets say you want change the default radix for numbers input and printed by GDB. `help radix` is no help at all, but `apropos radix` is very useful.

4. `p` is short for `print`, while `/x` specifies that the number should be printed in hex. You can use any of the format specifiers listed above.

Stopping Debugging

You can stop debugging in a variety of ways.

- Quit your program -- GDB continues to run but is no longer attached to your program
- Detach from your program (using GDB's `detach` command) -- GDB will keep running but the connection between it and your program is broken.
- Force quit your program (using GDB's `kill` command) -- This is akin to the MacsBug `ES` command. Your program dies but GDB keeps running.
- Quit GDB (using GDB's `quit` command) -- If you quit GDB while it's still attached to your program, GDB will either force quit it (if you ran the program from within GDB) or detach from it (if you attached to it) before quitting.

MacsBug	GDB
`ES`	`kill`

Back to top

MacsBug To GDB Reference

The following sections describe the GDB equivalent for various MacsBug constructs. The sections are structured along similar lines to the MacsBug online help.

Editing

GDB supports command-line editing in much the same way as MacsBug, but of course all the key bindings are different. You can configure the GDB key bindings--see the GDB manual for more information on this--but the default configuration is described below.

MacsBug	GDB
`left arrow`	`left arrow` or `^B`
`right arrow`	`right arrow` or `^F`
`option-left arrow`	`escape B`
`option-right arrow`	`escape F`
`option-delete`	`escape delete`
`command-left arrow`	`^A`
`command-right arrow`	`^E`
`command-delete`	`^U`
`command-V`	`up arrow`
`command-B`	`down arrow`

Selecting Procedure Names

Like MacsBug, GDB is smart enough to find some symbols even if the program wasn't compiled with debugger symbols. You can learn more about accessing symbols in the Symbols section later in this document.

MacsBug	GDB
command-D or command-;	Tab Tab [1]
	info func <regex> [2]

Notes:

1. The first Tab tries to complete the symbol; if there is no unambiguous completion, the second Tab lists the alternatives.
2. This lists all the functions whose names match the <regex>. Using an overly general regular expression (for example, `info func .*`) can take a very long time if your program uses large system frameworks (for example, the Carbon framework).

Values

For an experienced MacsBug user GDB values are hard to get used to. You must remember to use $ in front of any register names you use. By default a numeric constant is in decimal (not hex), and you must prefix hex numbers with 0x and not $ ($ is used for registers, the value history and GDB convenience variables). There's also no support for unit suffixes (K, M, G, and ß). Finally, $_ behaves much the same as "`.`" in MacsBug (it's the last address examined by "`X`"), while $__ is like "`.^`".

One nice feature of GDB is the value history. Every time you print an expression GDB prints the value of the expression along with a dollars sign ($) and a number. You can use this dollars sign and number combination to recall the value in subsequent expressions. You can see an example of this in Listing 5.

GDB also supports convenience variables, which are discussed in more detail later in this technote.

MacsBug	**GDB**
`<register>`	`$<register>`
`$<hexnum>`	`0x<hexnum>`
`#<decnum>`	`<decnum>`
`K`, `M`, `G` and `ß` (postfix)	not supported
`.`	`$_`
… (option-;)	not supported
`:`	not supported

Note:
GDB does let you change the default input and output radix. Type `apropos radix` for details.

Expressions and Operators

GDB uses a C-like syntax for its expressions, so it shouldn't be hard for you to pick up. One big difference between this and MacsBug is that each GDB expression has an associated type. For example, GDB knows that the type of `argv` is a pointer to a pointer to a character. If you use it in an expression, GDB can work out the type of the result--GDB knows that `**argv` is a character. Thus, MacsBug's naive implementation of the indirection operation (@ or ^) is replaced by the GDB's C-like operator (*) which carries type information with it. This means that you usually don't need MacsBug's size postfix operators (`.B`, `.W`, and `.L`) because GDB already knows the right size of an indirection. If necessary you can override GDB's type information by using a cast. Some examples of this are shown in Listing 5.

Listing 5. Using casts in expressions.

```
 $r3 is an integer, but you can cast it.
(gdb) p/x *(char *)$r3
$6 = 0x0
(gdb) p/x *(short *)$r3
$7 = 0x3
(gdb) p/x *(int *)$r3
$8 = 0x3e360

 You can also use the brace notation when printing.
(gdb) p/x {char}$r3
$9 = 0x0
(gdb) p/x {short}$r3
$10 = 0x3
(gdb) p/x {long}$r3
$11 = 0x3e360

 Accessing the value history.
(gdb) p/x $11
$12 = 0x3e360
```

Note:
GDB's indirection operation (*) is more lenient that an ANSI C compiler in that it will let you treat an integer value as a pointer. So an expression like `*0x01000` works in GDB, but wouldn't work in C without a cast.

Note:
GDB's expression syntax is actually dependent on the language you're debugging. However, when you're debugging without source the default C-like syntax applies.

MacsBug	GDB
+	+
-	-
*	*
/ or ÷	/
MOD	%
AND or &	& or && [1]
OR or |	| or || [1]
NOT or !	~ or ! [1]
XOR	^
<<	<<
>>	>>
= or ==	==
<> or != or &ne;	!=
<	<
>	>
<=	<=
>=	>=
@ (prefix) or ^ (postfix)	* (prefix)
.B	not supported [2]
.W	not supported [2]
.L	not supported [2]

Notes:

1. MacsBug makes no distinction between bitwise and logical operators (AND and & are identical). GDB supports both types, with the standard C syntax.
2. You can achieve the same effect with a cast, as shown in Listing 5.

Flow Control

GDB has much the same flow control facilities as MacsBug. Note that the commands with a trailing 'i' (the 'i' stands for instruction) work at the assembly level and step one instruction at a time. You must use these variants when debugging without source. When stepping one instruction at a time, you might want to use the `display` command (as shown in Listing 6) to display the current instruction after each command. Alternatively, activate the MacsBug GDB plug-in.

Listing 6. Using `display` when stepping through instructions

```
Breakpoint 1, 0x70268f8c in HLock ()
(gdb) display/i $pc
2: x/i $pc  0x70268f8c <HLock>: mflr    r0
(gdb) si
0x70268f90 in HLock ()
2: x/i $pc  0x70268f90 <HLock+4>:       stw     r0,8(r1)
(gdb) si
0x70268f94 in HLock ()
2: x/i $pc  0x70268f94 <HLock+8>:       stwu    r1,-64(r1)
(gdb)
```

MacsBug	GDB
G or command-G	continue or c
G <addr>	jump *<addr> [1]
GG	delete [2] continue
GTP <addr>	tbreak *<addr> continue
S or command-S	step or s
	stepi or si [3] [4]
SO or T or command-T	next or n
	nexti or ni [3] [4]
MRP	finish [5]
	tbreak *$lr continue [6]

Notes:

1. Because GDB's primary focus is source-level debugging, the arguments to commands like `jump` are normally a function name, or a line number within a source file. If you want to use an address, you have to prefix it with '*'. This convention is used by a number of commands.
2. `delete` removes all breakpoints. You have to confirm this action unless you turn off confirmation using `set confirm off`.
3. Commands ending with 'i' operate one instruction at a time.
4. You should use `si` for stepping through normal instructions. Only use `ni` for instructions that specifically call a function (typically the `bl` instruction). Using `ni` for other instructions can have weird side effects; for example, if the current PC is at a `mflr` instruction, `ni` mistakenly works like a magic return [2787251].
5. `finish` works properly only if you have source for the current routine.
6. This assumes that the return address is still in LR.

Breakpoints, A-Trap Breaks, and TVector Breaks

GDB has comprehensive support for breakpoint. In addition to the MacsBug equivalent commands listed below, you also have the `fb` (future break) command which allows you to set a breakpoint on a symbol that isn't yet loaded. The breakpoint will be set when the code containing the symbol is loaded. This is a very useful technique for setting a breakpoint in a dynamically loaded plug-in.

You can create thread-specific breakpoints. These breakpoints only trigger if a particular thread executes the specified code. See the section on threads later in this technote for an example of how to do this.

Another cool GDB feature is that you can enable and disable breakpoints without actually deleting them (using the `enable` and `disable` commands). This lets you run your program for a time without disrupting your breakpoint state.

A final cool feature of GDB breakpoints is the ability to save breakpoints to a file (using the `save-breakpoints` command) and restore them later (using the `source` command). This makes it easy to retain your breakpoint state between GDB sessions.

GDB has no support for A-traps because there's no trap dispatcher in Mac OS X. Furthermore, there's no support for TVector breaks because TVectors are a CFM construct and GDB knows very little about CFM. This isn't a problem though: you can use normal breakpoints instead of A-trap and TVector breaks.

MacsBug	GDB
`BRP <function>`	`b <function>` [1]
`BRP <addr>`	`b *<addr>` [2]
`BRP <addr> <condition>`	`b *<addr> if <condition>` [3]
`BRP <addr> <count>`	`b *<addr>` `ignore $bpnum <count>` [4]

`BRP <addr> ; <statement>`	`b *<addr>`
	`command $bpnum`
	`> <statement>`
	`> end`

`BRC <addr>`	`clear *<addr>` [5]
`BRC`	`delete`
`BRD`	`info break`

Notes:

1. If you break at the beginning of a function you can use `tbreak $lr` followed by `c` to step out of the function.
2. As described in the previous section, if you want to set a breakpoint on a specific address, you have to use the "*" syntax. For example, to break on the address $12345678, you use the command `b *0x12345678`.
3. You can use the `condition` command to change the condition associated with a breakpoint at any time.
4. `$bpnum` is a convenience variable that returns the number of the last breakpoint created. Convenience variables have no direct analogue in MacsBug. They are explained in detail in the GDB manual. Convenience variables can be helpful in places where you'd use MacsBug's `playmem`.
5. It's normally easier to clear a breakpoint by number (using `delete <x>`, where `<x>` is the breakpoint number) than by address. If you can't remember the breakpoint number, the `info break` command prints it for each breakpoint.

Watch Points

MacsBug	GDB
`WP`	not supported [1]
`WPC`	not supported [1]
`WPD`	not supported [1]

Notes:

1. GDB does support watchpoints (use `apropos watch` to learn about the commands) but this support does not currently work on Mac OS X [2762873]. Then again, my experience with watchpoints under MacsBug is that they were kind of flaky. Until GDB's watchpoint support is fixed you have to find random memory corruption the hard way (by applying conditional breakpoints to get a rough idea of who's responsible and then stepping through code).

Disassembly

For a MacsBug user, GDB's `disassemble` command is hard to get used to: if you only supply one argument, it disassembles the entire function containing the address specified by the argument. To disassemble something smaller than a function you have to use two arguments to specify a range of addresses. Alternatively, if you just want to disassemble a few instructions, the examine (`x`) command with the instruction format modifier (`/i`) is easier.

MacsBug	GDB
`ILP <addr>`	`x/8i <addr>`
`ILP <addr> <len>`	`disas <addr> <addr>+<len>`
`IPP PC`	`disas $pc-20 $pc+20`
	`x/20i $pc-40`
`IRP <function>`	`disas <function>`
`DHP <addr>`	not supported; you can use `playmem` to work around this

Emulator

What emulator?

Heaps

There is no direct support for debugging heap zones in GDB. However, you can call heap checking functions in various memory management libraries. For example, Listing 7 shows how to call some of the debug functions in the System framework memory library.

IMPORTANT:
This technique is not without its perils. See The Dangers of Calling Functions for details.

Listing 7. Using `call` for heap debugging

```
 Prints the list of heap zones, ala MacsBug "HZ".
(gdb) call (void) malloc_zone_print(0, 0)
Scalable zone 0x5010: inUse=10056(2815KB) small=10027(1227KB) [...]
5 regions:
Region [0x5000-0x45000, 256KB]  In_use=2472
Region [0x1387000-0x13c7000, 256KB]     In_use=4110
Region [0x13f8000-0x1438000, 256KB]     In_use=1647
Region [0x154a000-0x158a000, 256KB]     In_use=1064
Region [0x16a3000-0x16e3000, 256KB]     In_use=734
Free in last zone 26898
Scalable zone 0xf4f010: inUse=645(85KB) small=645(85KB) [...]
1 regions:
Region [0xf4f000-0xf8f000, 256KB]       In_use=645
Free in last zone 158402

 Prints a detailed dump of each zone, ala MacsBug "HD". If you
just want to dump one zone, supply it at the first parameter.
(gdb) call (void) malloc_zone_print(0, 1)
Scalable zone 0x5010: inUse=10056(2815KB) small=10027(1227KB) [...]
5 regions:
Region [0x5000-0x45000, 256KB]  In_use=2472
        Sizes in use: 16[282] 32[1136] 48[344] 64[183] 80[130] [...]
Region [0x1387000-0x13c7000, 256KB]     In_use=4110
        Sizes in use: 16[360] 32[2041] 48[1088] 64[172] 80[228] [...]
Region [0x13f8000-0x1438000, 256KB]     In_use=1647
        Sizes in use: 16[316] 32[590] 48[211] 64[108] 80[75] [...]
Region [0x154a000-0x158a000, 256KB]     In_use=1064
        Sizes in use: 16[281] 32[330] 48[127] 64[65] 80[51] [...]
Region [0x16a3000-0x16e3000, 256KB]     In_use=734
        Sizes in use: 16[46] 32[225] 48[72] 64[26] 80[17] [...]
Free Sizes: >=1024[5] 592[1] 576[1] 400[1] 32[2] 16[8]
Free in last zone 26898
Scalable zone 0xf4f010: inUse=645(85KB) small=645(85KB)  [...]
1 regions:
Region [0xf4f000-0xf8f000, 256KB]       In_use=645
        Sizes in use: 16[1] 32[373] 48[122] 64[47] 80[34] [...]
Free Sizes: >=1024[3] 768[1] 160[1] 128[1] 80[4] 64[2] 16[23]
Free in last zone 158402

 Determines which zone a pointer is in.
(gdb) call (void) malloc_zone_print_ptr_info(0x16a3000)
ptr 0x16a3000 in registered zone 0x5010

 Checks all heap zones (or a specific zone if you supply
it as a parameter), ala MacsBug "HC ALL".
(gdb) call (int) malloc_zone_check(0)
$18 = 1

 Deliberately destroy our heap.
```

```
(gdb) call (void) memset(0x16a3000, 0, 256)

 Recheck the heap. Houston control, we have a problem.
(gdb) call (int) malloc_zone_check(0)
*** malloc[988]: invariant broken at region end: ptr=0x16a3010 [...]
*** malloc[988]: Region 4 incorrect szone_check_all() counter=3
*** malloc[988]: error: Check: region incorrect
$19 = 0
```

Note:
Remember that GDB might not be the right tool for debugging heap problems. Mac OS X has a number of really cool heap debugging tools that are independent of GDB. Check out Inside Mac OS X: Performance for details.

Note:
The zones shown in Listing 7 are not Memory Manager zones but a different concept (with the same name) supported by the System framework memory library. Memory Manager zones are not supported under Carbon.

Symbols

GDB, being a source-level debugger, has a gazillion commands to control how symbols are handled. Type `apropos symbol` to see them all. Their description is beyond the scope of this document. The GDB manual has all the gory details.

Stack

MacsBug	GDB
`SC` or `SC6`	`bt` [1] [2] [3]

Notes:

1. `bt` is short for `backtrace`.
2. GDB has no equivalent for SC7 but you rarely need it because virtually all Mac OS X code is written in a high-level language and thus `bt` works (almost) always.
3. By default `bt` crawls the current thread's stack. There are commands to work with different threads within your process. For example, you can use `thread apply all bt` to get a backtrace of all threads. See the discussion on threads later in this document for more details.

Memory

GDB's `x` (examine) command is the direct equivalent of MacsBug's `DM` command. `x` works much like `print` in that you can append a '/' and then a format specifier ('o' for octal, 'x' for hex, 'd' for decimal, 'u' for unsigned decimal, 't' for binary, 'f' for float, 'a' for address, 'c' for char) and optionally a size qualifier ('b' for 1 byte, 'h' for 2 bytes, 'w' for 4 bytes, 'g' for 8 bytes). The x command supports two additional format qualifiers that aren't supported by `print` ('s' for string, 'i' for instruction). It also supports a repeat counter. Listing 8 shows some examples of using the `x` command.

Listing 8. Some examples of using the `x` command.

```
 Stop at SetMenuItemText.  $r5 is the itemString parameter.
Breakpoint 1, 0x738087cc in SetMenuItemText ()

 Print itemString as a C string.  This usually works pretty well.
(gdb) x/s $r5
0xbffff1e8:      "\005Close"

 Print itemString as a sequence of characters. Note the use of
 'c' as a format qualifier and 5 as a repeat count.
(gdb) x/5c $r5+1
0xbffff1e9:     67 'C'  108 'l' 111 'o' 115 's' 101 'e'

 Print the first byte of itemString as binary.
(gdb) x/tb $r5
0xbffff1e8:     00000101

 And the first half word.
(gdb) x/th $r5
0xbffff1e8:     0000010101000011

 And the first word.
(gdb) x/tw $r5
0xbffff1e8:     00000101010000110110110001101111

 Note how both the format specifier and size qualifier
 are 'sticky'.
(gdb) x/t $r5
0xbffff1e8:     00000101010000110110110001101111
(gdb) x $r5
0xbffff1e8:     00000101010000110110110001101111

 Print the first word as hex.
(gdb) x/xw $r5
0xbffff1e8:     0x05436c6f

 Hitting return on the blank line repeats the previous command.
 Like MacsBug this continues dumping the next memory location.
(gdb)
0xbffff1ec:     0x73650000

 You can use the 'i' format specifier to disassemble.
 The repeat count is especially useful in this case.
(gdb) x/i $pc
0x738087cc <SetMenuItemText+20>:        stw     r3,120(r30)
(gdb) x/4i $pc
0x738087cc <SetMenuItemText+20>:        stw     r3,120(r30)
0x738087d0 <SetMenuItemText+24>:        mr      r0,r4
0x738087d4 <SetMenuItemText+28>:        stw     r5,128(r30)
0x738087d8 <SetMenuItemText+32>:        sth     r0,124(r30)
```

MacsBug	**GDB**
`DM <addr>`	`x/8xb <addr>`
`DMA <addr>`	`x/4c <addr>` [1]
`DM <addr> <type>`	`p {<type>}<addr>`
`DM <addr> pstring`	`x/s <addr>` [2]
`TMP <template>`	`info type <regex>`
`SB <addr> <val>`	`set *((char *) <addr>) = <val>`
`SW <addr> <val>`	`set *((short *) <addr>) = <val>`
`SL <addr> <val>`	`set *((int *) <addr>) = <val>`

Notes:

1. This is actually a pretty lame substitute for DMA. See the MacsBug GDB plug-in for a user-defined command that implements this properly.
2. This actually prints a C string, but it works pretty well most of the time for pstrings. `x/8c <addr>` is not a good substitute because it's hard to read the separate characters as a string. Probably the best way print a pstring is with a user-defined command.

Registers

MacsBug	GDB
`<register>=<value>`	`set $<register> = <value>`
`TD`	`info reg`
`TF` or `TV`	`info all-reg`

Macros

GDB supports user-defined commands that are much more powerful than MacsBug macros. User-defined commands can span multiple lines, and can include control flow statements like `if` and `while` (see the GDB manual for details on these). They can't, however, be used in expressions, which is sometimes inconvenient.

You can use the `source` command to run a file containing GDB commands as if you'd typed them in at the prompt. GDB automatically sources the ".gdbinit" file in your home directory and the current directory when you start it. This allows you to define persistent user-defined commands, much like adding a macro to your MacsBug "Debugger Prefs" file.

MacsBug	GDB

`MC <name> <expansion>`	`define <name>`
	`> <expansion>`
	`> end`

`®1` through `®9`	`$arg0` through `$arg9`, `$argc`
`MCC`	no equivalent
`MCD <name>`	`help user-defined` [1]
	`show user`
	`show user <name>`

Notes:

1. You can provide help for your user-defined commands using the `document` command. This is especially useful if you need help remember how to use some complex command you put in your ".gdbinit" file six months ago. See the GDB manual for details.

Miscellaneous

MacsBug	GDB
`ES`	`kill`
`WH <addr>`	`p/a <addr>`
	`info sym <addr>`

`FILL <addr> <num> \`
`<val>`
	`call (void) memset(<addr>, <val>, <num>)` [1]

`SHOW`	you can probably achieve the results you need with `display`
`DV`	`show version`
`STOPIF`	GDB supports real `if` and `while` statements, see the GDB manual
`HELP <command>`	`help <command>`
	`apropos <regex>`
`HELP <template>`	`ptype <type>`

Notes:

1. This technique is not without its perils. See The Dangers of Calling Functions for details.

DCMDs

IMPORTANT:
This section makes heavy use of GDB's ability to call arbitrary functions within a process. Many of these functions were specifically designed to support debugging. The list of functions changes over time. To see the latest list, search for all functions containing "DebugPrint" or "GDB" (use `info func DebugPrint` and `info func GDB`).

IMPORTANT:
This technique is not without its perils. See The Dangers of Calling Functions for details.

MacsBug	GDB
`AEDesc`	`call (void) GDBPrintHelpDebuggingAppleEvents()` [1]
`Evt`	`call (void) GDBPrintEventQueue()`
`Zap`	see discussion of `MallocScribble` environment variable in Inside Mac OS X: Performance
`Leaks`	see discussion of MallocDebug application and `leaks` command line tool in Inside Mac OS X: Performance
`Layers`	`call (void) DebugPrintAllWindowGroups()` `call (void) DebugPrintWindowGroup(<group>)` [2]
`THING`	`call (void) GDBComponentList()`
`FRAGS`	`info target`
	`info cfm`
	`info dyld`
	`info sharedlibrary`
`ECHO` or `PRINTF`	`echo`
	`output`
	`print`
	`printf`
`EBBE`	not need because $0 through $FFF are not mapped for normal applications
`MLIST`	`call (void) DebugPrintMenuList()` `call (void) DebugPrintMenu(<menu>)` [3] `call (void) DebugPrintMenuItem(<menu>, <item>)` [3]
`SCREAM`	see the discussion on threads later in this document

Notes:

1. This prints out information about how to display Apple event descriptors. The instructions vary depending on the system version.
2. You can determine `<group>` by looking through the list of groups displayed by `DebugPrintAllWindowGroups`.
3. You can determine `<menu>` by looking through the list of menus displayed by `DebugPrintMenuList`.

Other Cool Stuff

MacsBug	GDB

`WINDLIST`	`call (void) DebugPrintWindowList()`
	`call (void) DebugPrintPlatformWindowList()`

`DM <port> GrafPort`	`call (void) QDDebugPrintPortInfo(<port>)` `call (void) QDDebugPrintCGSInfo(<port>)` [1]
`DM <window> WindowRecord`	`call (void) DebugPrintWindow(<window>)`
`DM <control>^ ControlRecord`	`call (void) GDBShowControlInfo(<control>)`
	`call (void) GDBShowControlHierarchy(<window>)`
`PLAYMEM`	see below
`©<func>(<param>...)`	`call (<resulttype>) <func>(<param>...)`

Notes:

1. You can determine the current port by calling `GetQDGlobalsThePort`.

Back to top

Hints and Tips

This section contains miscellaneous hints and tips for using GDB as an assembly-level debugger.

Seeing stdout and stderr After Attaching

If you attach GDB to a process (as opposed to starting the process from within GDB), you won't be able to see anything that the process prints to `stdout` or `stderr`. Programs launched by the Finder typically have `stdout` and `stderr` connected to "/dev/console", so the information they print goes to the console. You can view this by launching the Console application (in the Utilities folder), however, it's inconvenient to have to look in a separate window. Another alternative is to connect the process's `stdout` or `stderr` to the terminal device for GDB's Terminal window. Listing 9 shows how to do this.

Listing 9. Connecting `stdout` and `stderr` to GDB's terminal device.

```
(gdb) attach 795
[... output omitted ...]
(gdb) call (void) DebugPrintMenuList()
 No output )-:

 Close the stdout and stderr file descriptors.
(gdb) call (void) close(1)
(gdb) call (void) close(2)

 Determine the name of the terminal device for GDB itself.
(gdb) shell tty
/dev/ttyp1

 Reopen stdout and stderr, but connected to GDB's terminal.
 The function results should be 1 and 2; if not, something
 is horribly wrong.
(gdb) call (int) open("/dev/ttyp1", 2, 0)
$1 = 1
(gdb) call (int) open("/dev/ttyp1", 2, 0)
$2 = 2

 Try the DebugPrintMenuList again.
(gdb) call (void) DebugPrintMenuList()
 Yay output!
Index MenuRef     ID  Title
----- ---------- ---- -----
<regular menus>
00001 0x767725D3 -21629 Ed
00002 0x76772627 1128 <Apple>
00003 0x767726CF 1129 File
00004 0x76772567 1130 Edit
[... remaining output omitted ...]
```

Remote Debugging

One of the nice features of GDB is that you can debug your application across any TCP/IP network (including the public Internet). You do this by logging into the remote machine, running GDB on that machine, and then attaching to the application you want to debug. This has two key advantages.

- You can debug programs at remote sites; it no longer matters if you can't reproduce the problem in your office!
- You can debug things that are disrupted by the debugger's user interface. Two interesting examples of this are application suspend/resuming handling and Drag Manager callbacks.

Remote debugging has some caveats of which you should be aware. Firstly, you must enable remote login on the remote machine. The easiest way to do this is to check the Allow Remote Login checkbox in the Sharing panel of System Preferences. You can then log in to that machine by launching Terminal on your local machine and using the `ssh` command line program to log into the remote machine. Obviously you have to have the user name and password of an account on the remote machine. Once you have logged into the remote machine you can use GDB in much the same way as you would normally. Listing 10 shows an example of this.

Note:
Mac OS X 10.0 did not support `ssh`; you must use `telnet` instead. For more information on these commands, type `man ssh` and `man telnet` at the command line.

Listing 10. Using `ssh` to log in to a remote machine.

```
[localhost:~] quinn% ssh puppy.apple.com
quinn@puppy.apple.com's password:
Welcome to Darwin!
puppy> gdb
GNU gdb 5.0-20001113 (Apple version gdb-200) (Mon Sep  3 02:43:52 GMT 2001) (UI_OUT)
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "powerpc-apple-macos10".
(gdb)
```

Don't run a GUI application from within GDB when logged in remotely. A GUI application needs to connect to system services which aren't available if you run it from a remote login session. If you need to debug a GUI application, have the remote user launch the application from the Finder, then start GDB in your `ssh` session and attach to the running application.

DebugStr

Mac OS X supports the standard `DebugStr` and `Debugger` system calls. However, their default behavior is to just print a message to `stderr` and continue. You must take special steps if you want these calls to stop program execution. Specifically, these routines only stop in the debugger if you have set the `USERBREAK` environment variable to 1. You can set this environment variable in a number of ways.

- Command line -- Type the following shell command before running GDB: `setenv USERBREAK 1`. You must run your application from within GDB for this to be effective.
- Within GDB -- Enter the following GDB command before running your application: `set env USERBREAK 1`. You must run your application from within GDB for this to be effective.
- Programmatically -- You can set the `USERBREAK` environment variable from within your own program using the following C statement: `setenv("USERBREAK", "1", true)`. You must do this before any other part of your program calls `DebugStr` or `Debugger`. Also, this only works if you call the standard C library `setenv` routine (in System framework); for MSL CFM applications this means you should call the Mach-O version of this function via CFBundle.

Finally, the underlying mechanism used to stop your program in the debugger when you call `DebugStr` is the `raise` system call (part of System framework). You can explicitly cause your program to stop regardless of the setting of `USERBREAK` by calling `raise(SIGINT)`.

The Dangers of Calling Functions

GDB, like MacsBug, lets you call functions within the application you're debugging. Also like MacsBug, this technique has its dangers. You should be aware of the following things.

- The function executes within the context of your program. If your program has crashed, there's no guarantee that it will be in a fit state to execute the function.
- You have to be very careful if you call a function that has intricate dependencies. For example, if your program has a mutex that protects a critical data structure and you break into GDB while that mutex is held, don't try to call a function that needs to acquire the mutex.
- The function you call executes in the context of the current thread (the thread displayed by `info thread`). Other threads are not blocked while the function executes. You can block them explicitly using the `thread suspend` and `thread resume` commands, but this may cause more problems than it solves.
- You should be especially careful calling functions that are not preemptive safe. For example, calling `NewWindow` would be a mistake because the call will execute in the context of a preemptive thread and will access data structures that aren't preemptive safe.

MacsBug GDB Plug-in

No technote on GDB for MacsBug users would be complete without mentioning the MacsBug GDB plug-in. This plug-in extends GDB to support a subset of the MacsBug commands and a MacsBug-like user interface. The plug-in is installed as part of the Mac OS X 10.1 developer tools in the "/usr/libexec/gdb/plugins/MacsBug/" directory. You can learn more about the plug-in from the documentation in that directory.

To activate the plug-in, take the following steps.

1. Start GDB from the command line.
2. Enter the command `source /usr/libexec/gdb/plugins/MacsBug/gdbinit-MacsBug`. You will find that most MacsBug commands are available (in addition to the standard GDB commands). Type *`help MacsBug`* for more online help on these commands.
3. To start the user interface, enter the command `mb on`. The user interface will display "Not Running" until you attach to the application you wish to debug. Use normal GDB commands to attach.

IMPORTANT:
The MacsBug-like user interface requires a terminal window of at least 44 rows and 80 columns. You may need to resize your Terminal window for the user interface to start. This and other terminal considerations are discussed in the plug-in's documentation.

Note:
If you want to always activate the plug-in, just add the above commands to the ".gdbinit" file in your home directory.

The plug-in is especially useful when doing assembly language debugger because it maintains a lot of useful state on the screen (the current registers, a disassembly of the instructions around the current PC, and so on).

Playmem

One nice feature of MacsBug is that it allocates a small region of memory (512 bytes) and sets the symbol `playmem` to point to it. You can use this memory for all sorts of nasty debugging hacks. GDB has no dedicated `playmem`, but you can easily create your own, as shown in Listing 11. These commands create a 512 byte block of zeroed memory and set the convenience variable `$playmem` to point to it.

Listing 11. Creating `playmem`.

```
(gdb) call (void *) calloc(1, 512)
$1 = (void *) 0x1ed11b0
(gdb) set $playmem = $
(gdb) p/a $playmem
$2 = 0x1ed11b0
```

Once you have a block of play memory you can use it for a variety of tricky things. For example, you can use it to simulate the MacsBug `DHP` command, as shown in Listing 11.

Listing 12. Simulating MacsBug's `DHP` command using `playmem`.

```
 Create a mydhp user-defined command.
(gdb) define mydhp
Type commands for definition of "mydhp".
End with a line saying just "end".
>set *(int *)$playmem = $arg0
>x/i $playmem
>end

 Use the newly created command.
(gdb) mydhp 0x4e800020
0x1ed11b0:      blr
```

This is just one example of the use of convenience variables. You can do many other cools things with them. To create a convenience variable, just name it (don't forget to prefix it with $) on the left of a set expression (as shown in Listing 11). To see the value of a convenience variable, just `print` it. To see all convenience variables, use the `show convenience` command. You can learn more about convenience variables in the GDB manual.

Threads

GDB provides a number of commands to support debugging threads. Listing 13 shows an example of their use.

Listing 13. Thread debugging commands.

```
 Show information about the current thread.
(gdb) info thread
Thread 0x1903 has current state "WAITING"
Thread 0x1903 has a suspend count of 0.

 Show a list of all threads.
(gdb) info threads
  3 process 665 thread 0x1b03  0x700009a8 in mach_msg_overwrite_trap ()
  2 process 665 thread 0x1a03  0x70059d58 in semaphore_wait_signal_trap ()
* 1 process 665 thread 0x1903  0x700009a8 in mach_msg_overwrite_trap ()

 Get information about a specific thread.
(gdb) info thread 3
Thread 0x1b03 has current state "WAITING"
Thread 0x1b03 has a suspend count of 1.

 Make another thread the current thread
 and then backtrace that thread, and then
 switch back to the previous thread.
(gdb) thread 3
[Switching to thread 3 (process 665 thread 0x1b03)]
#0  0x700009a8 in mach_msg_overwrite_trap ()
(gdb) bt
#0  0x700009a8 in mach_msg_overwrite_trap ()
#1  0x700058d4 in mach_msg_overwrite ()
#2  0x700279a0 in thread_suspend ()
#3  0x70027934 in _pthread_become_available ()
#4  0x70027658 in pthread_exit ()
#5  0x700150f8 in _pthread_body ()
#6  0x00000000 in ?? ()
(gdb) thread 1
[Switching to thread 1 (process 665 thread 0x1903)]
#0  0x700009a8 in mach_msg_overwrite_trap ()

 Apply a command to all threads. In this example,
 we "backtrace" every thread.
(gdb) thread apply all bt

Thread 3 (process 665 thread 0x1b03):
#0  0x700009a8 in mach_msg_overwrite_trap ()
#1  0x700058d4 in mach_msg_overwrite ()
#2  0x700279a0 in thread_suspend ()
[... remaining output omitted ...]

Thread 2 (process 665 thread 0x1a03):
#0  0x70059d58 in semaphore_wait_signal_trap ()
#1  0x70016300 in semaphore_wait_signal ()
#2  0x70016168 in _pthread_cond_wait ()
[... remaining output omitted ...]
```

```
Thread 1 (process 665 thread 0x1903):
#0  0x700009a8 in mach_msg_overwrite_trap ()
#1  0x70006e34 in mach_msg ()
#2  0x7017ab20 in __CFRunLoopRun ()
[... remaining output omitted ...]

                            Create a thread-specific breakpoint.
(gdb) break HandToHand thread 1
Breakpoint 1 at 0x7025e844
```

Back to top

Summary

GDB is your friend. Learn to love GDB. Soon you will forget your infatuation with MacsBug.

Back to top

References

Richard Stallman et al, Debugging with GDB, Free Software Foundation, March 2000

Apple Computer, MacsBug Reference and Debugging Guide, Addison-Wesley, 1990

Inside Mac OS X: Performance

projectbuilder-users Mailing List -- The Apple GDB team hang out on this mailing list.

Back to top

Downloadables

Acrobat version of this Note (size in bytes, size in K) Download

Back to top