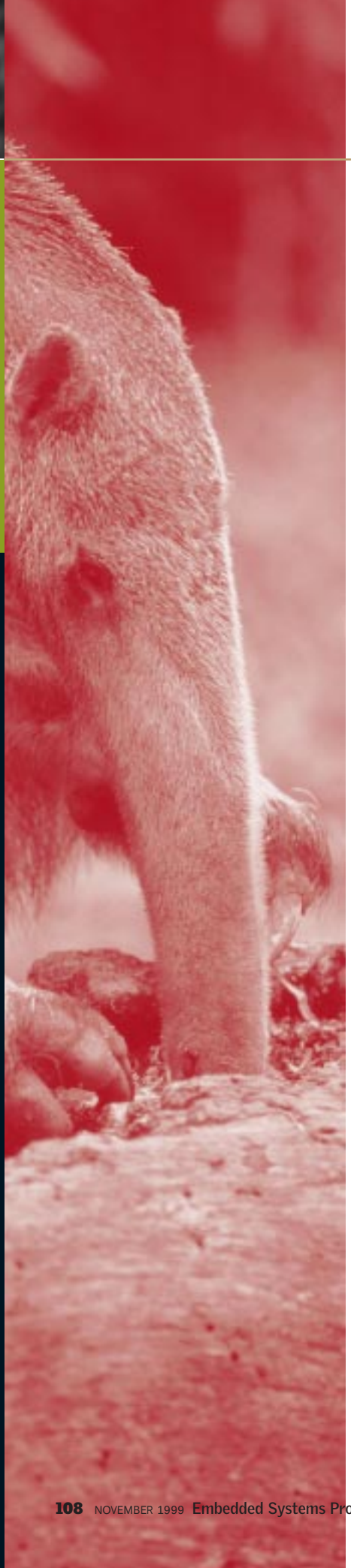

BILL GATLIFF

Embedding with GNU: the gdb Remote Serial Protocol

In this installment of a series on GNU-based embedded development, the author wraps up his discussion of using the GNU debugger, gdb, to debug embedded applications remotely.

In September, I introduced the topic of the GNU debugger, gdb.[1] I discussed how its remote debugging feature could be used to debug code running in an embedded system connected to a PC by a serial cable, network connection, or some other means. While commercial products with this capability are also available, in my opinion the freely available gdb is the preferred solution because it provides portable, sophisticated debugging over a broad range of embedded systems, including devices with communications interfaces or resource constraints too restrictive for general commercial support.

I also mentioned in that article that, to make remote debugging possible, gdb requires the services of a debugging stub—a small library of code in the debugging target that manages registers and memory, responds to breakpoints, and reports application status to gdb via the communications link. That article contained excerpts from a debugging stub for the Hitachi SH-2 microcontroller, but I didn't actually put enough of the pieces together to show how a complete debugging stub would work.

I'll fix that this month by presenting in detail the GDB Remote Serial Protocol—gdb's standard remote communications protocol. Once you are comfortable with how your processor handles breakpoint and other

In my opinion the freely available gdb is the preferred solution because it provides portable, sophisticated debugging over a broad range of embedded systems.

exceptions, then knowledge of a few basic Remote Serial Protocol messages is all you need to get your embedded system talking to gdb.

The protocol defined

The GDB Remote Serial Protocol (RSP) is a simple, ASCII message-based protocol suitable for use on serial lines, local area networks, or just about any other communications medium that can support at least half-duplex data exchange.

RSP packets begin with a dollar sign (`$`), followed by one or more ASCII bytes that make up the message being sent, and end with a pound sign (`#`) and two ASCII hex characters representing the message's checksum. For example, the following is a complete RSP packet:

```
$m4015bc,2#5a
```

The receiver of the packet responds immediately with either a "`+`" or a "`-`" to indicate that the message was received either intact or in error, respectively.

A typical transaction involves gdb issuing a command to a debugging target, which then responds with data, a simple acknowledgement, or a target-specific error code. If the latter is returned, gdb will report the code to the user and halt whatever activity is currently in progress.

The `console output` message, which debugging targets use to print text on the gdb console, is the lone exception to the typical command-response sequence. Except when another command is already in progress, this message can be sent from the debugging stub to gdb at any time.

The following paragraphs describe the RSP's essential commands. For the purposes of this tutorial, I have divided the messages into three categories: register- and memory-related commands, program control commands, and other commands.

Register- and memory-related commands

Here are the commands to read from and write to registers.

Read registers ("g")
Example: `$g#67`

The debugger will issue this command whenever it needs to know everything about the debugging target's current register state. An example target response would be:

```
+ $123456789abcdef0...#xx
```

(Register 0 is 0x12345678, register 1 is 0x9abcdef0, and so on.)

The response is an ordered stream of bytes representing register data ordered per the definition in the target's macro file, gdb/config/<arch>/tm-<arch>.h (for example, gdb/config/sh/tm-sh.h for the Hitachi SH).

Write registers ("G")
Example: `$G123456789abcdef0...#xx`
(Set register 0 to 0x12345678, register 1 to 0x9abcdef0, and so on.)

This message is the complement to the `read registers` command. With this command, gdb supplies an ordered stream of bytes representing data to be stored in the target processor's registers immediately before program execution resumes. An example target response:

```
+ $OK#9a
```

Write register N ("P")
Example: `$P10=0040149c#b3`
(Set register 16 to the value 0x40149c.)

When it wants to set the value of only one or two registers, gdb sends this command instead of sending a complete register set to the debugging target. The register numbering is the same as that used in the `read registers` and `write registers` commands. An example target response:

```
+ $OK#9a
```

Below are the commands to read from and write to memory.

Read memory ("m")
Example: `$m4015bc,2#5a`
(Read two bytes, starting at address 0x4015bc.)

A `read memory` command is sent by gdb to determine the values of local and global variables, the value of an opcode about to be replaced by a breakpoint instruction, and any other kind of information the user requests. The debugger generally is aware of any endian issues present in the debugging target, so the target need only return the result as a simple stream of bytes; gdb will reformat them as appropriate.

Debugging stubs on targets that are sensitive to data widths should optimize the implementation of the `write memory` and `read memory` commands as the target architecture dictates. For example, certain peripheral configuration registers in the Hitachi SH-2 processor family can only be properly read and written in 16-bit or 32-bit units, so a debugging stub for this target should use 16-bit or 32-bit accesses whenever possible. An example target response:

```
+ $2f86#06
```

Write memory ("M")
Example: `M4015cc,2:c320#6d`

CORBIS/TOM BRAKEFIELD

(Write the value 0xc320 to address 0x4015cc.)

This command is the complement to the `read memory` command. An example target response:

```
+ $OK#9a
```

Program control commands

Program control commands are messages that gdb uses to control the behavior of the application being debugged. As such, these commands are somewhat more complicated to implement than the more basic register- and memory-related commands we've already covered.

Get last signal ("?")
Example: `$?#3f`

This command is used to find out how the target reached its current state. The response is the same as the "Last signal" response documented below.

Step ("s")
Example: `$s#73`

When it wants the target to execute exactly one assembly language instruction, gdb issues a `step` command to the debugging target. The debugger sends this command when the user types `stepi` or `step` at the gdb console. An example target response follows the `continue` command description.

Continue ("c")
Example: `$c#63`

A `continue` command is issued when gdb releases the application to run at full speed, as happens when the user enters a `continue` command at the gdb console. An example target response follows.

Responses to the step and continue commands. A debugging stub does not immediately respond to the `step` or `continue` commands, other than to send the "+" that signifies proper reception of the packet. Instead, the stub provides a response when the next breakpoint is reached, the requested instruction has been executed (in the case of the `step` command), an exception occurs, or the application exits.

There are two ways to respond to these commands: a brief "last signal" response, or a more useful "expedited response."

"Last signal" response ("S")
Example: `$S05#b8`

This is the minimum reply to the `last signal`, `step`, and `continue` commands. The "05" in the response can be any one of the signal values used in the standard POSIX `signal()` function call. For example, "5" is a breakpoint exception, "10" is a bus error, and so forth.

Expedited response ("T")
Example: `$T0510:1238;F:FFE0...#xx`

This message combines the information in a `last signal` response (the "05" in the example message) with key register values that gdb may be immediately interested in. Designed to improve gdb's performance during code stepping, this message allows gdb to avoid a `read registers` request if the values it needs (the target's program counter and status register, generally) are included in this message.

Registers are identified by the same numbering scheme used in the `read registers` and `write registers` commands; in the example provided, the value of register 16 (10 hex) is 0x1238, and register 15 (F hex) contains 0xffe0.

Other commands

Console output ("O")—*optional*
Example:

```
$O48656c6c6f2c20776f726c64210a#55
```

(Prints "Hello, world!\n" on the gdb console)

This command allows a debugging stub to send a text message to the gdb console. The text to be displayed is sent in its hex byte equivalent ('H' == 0x48) and gdb will queue successive messages until it sees a newline ('\n', 0x0a) character.

This message always originates in the debugging target; gdb never sends a `console output` message to the debugging target.

Empty response ("")
If a debugging stub encounters a command it doesn't support or understand, it should return an `empty response`. This allows gdb to select an alternate command if one is available.
Example: `<an unrecognized command>`
Target response: `+ $#00`

Error response ("E")
When a debugging stub encounters an error during command processing, it should send an error response back to gdb. Bus errors and/or illegal addresses during memory operations are examples of commands which may generate an error response from a debugging stub.
Example: `<a command that produces an error>`
Target reponse: `+$E01#xx`

There aren't any predefined error codes in gdb; when gdb receives an error message, it prints it on the console and halts the operation in progress.

Putting it all together

At this point, I've covered individually all the pieces necessary to get an embedded system talking to gdb. In the last article, I covered traps, single-stepping, and a few gdb features; and in the previous section I covered the communications protocol that gdb expects a debugging stub to use. All we have left to do now is to throw all of these pieces together, right?

Actually, we must deal with one more minor consideration before we're really ready to start debugging code: the chicken-and-egg problem of actually getting the debugging stub into the embedded system for the first time, so that gdb has something with which to communicate when we turn on the power.

Several methods of attacking this problem exist.[2] To me, the best way always seems to be to place a minimal stub into some kind of nonvolatile memory in the target, and use that code to both boot the system and help gdb download the rest of the application into RAM. Once gdb starts the application, debugging control then transfers to a second debugging stub linked with the application itself.

The biggest advantage of this approach is that it allows you to continue developing the application-linked debugging stub without needing to reprogram the resident stub in the target's nonvolatile memory, which is a real bonus if the only nonvolatile memory available is one-time-programmable ROM. Furthermore, because the resident stub needs to know only the simplest commands—`read memory`, `write memory`, `write register N`, and `continue`—the likelihood of a serious bug being present in this code is fairly low.

Another approach is to place a complete debugging stub into the target's nonvolatile memory, and use it for all debugging activities. This eliminates the need to link a debugging stub with the target application, but may make it more difficult to change the stub if errors are found or new features are added.

If you're using a commercial microprocessor evaluation board, you might not need to provide a debugging stub at all—gdb may already support the vendor's protocol, or you may be able to add support to gdb after spending a few minutes reverse-engineering the protocol with a serial-port analyzer. Check the vendor's license agreement if you take this approach: you may need to sign a non-disclosure agreement first, and you will definitely need to seek permission before releasing your gdb improvements to the community at large.

Testing a debugging stub

Once you think you have a debugging stub ready to go, you'll want to test it as thoroughly as possible before putting it to work. Remember, the only thing worse than a buggy application is a buggy development tool.

The following is a test procedure I recommend you use whenever you make changes to your debugging stub, to make sure things are working the way they should.

First, for convenience, add the following lines to your .gdbinit file:

```
set remotedebug 1
set remotelogfile gdb_logfile
```

These commands make gdb display all RSP messages exchanged between the host and the target, as well as log them to the file gdb_logfile.

Next, connect gdb to the remote target, and enter the `target remote [port]` command. As gdb makes the connection, watch the messages exchanged to make sure that your stub provides the proper responses to gdb's requests.

During startup, your debugging stub should load values into each of the target processor's registers. Use gdb's `info registers` command to confirm that gdb can properly receive and display these values.

Next, use gdb's `set` command to change a few register values, and make sure that the stub both properly responds to gdb's `write register` command, and returns the correct results in subsequent `read registers` commands initiated when you type `info registers`.

Next, do the same thing for a few memory locations—have the stub set the locations at startup, and then verify that gdb can read and write these locations on request. For example, try the following:

```
print *(long*)0x1234
set *(long*)0x1234=5678
print *(long*)0x1234
```

If everything works so far, you're ready to try out gdb's `load` command. The console will get extremely noisy as gdb displays the multitude of `write memory` commands it uses to transfer a test application to the target. You'll want to refer to the log file then, to ensure that everything happened as expected. Once this has been completed, check a few memory locations in the application's code space to confirm that the expected values are there.

The test application should contain some gdb console output if your stub

supports this command. Enter `continue` at the gdb console, and see that the output appears as you expected.

Reset the target, and reload the test application. Set a breakpoint immediately before the line that performs the console output. Enter `continue` again, and verify both that the application stops where it's supposed to and that the console output appears properly when execution resumes.

Next (and this can be combined with the above test), reload the application, set a breakpoint, and step through a few source lines with the `step` and `stepi` commands. Make sure that local variables (viewed with the `display` command) change as expected, and that they don't change unexpectedly, particularly when stepping through opcodes that branch or jump. Also, monitor the program counter, stack pointer, and other register values, to make sure they change as your code dictates they should.

At this point, your stub is ready to go.

Final thoughts

In my opinion, gdb has no equal in the debugging tool community, free or otherwise. In addition to its stability and long list of useful features, gdb provides the flexibility that today's embedded developers need at a price that's hard to beat. For those willing to invest the time necessary to develop a debugging stub, the rewards are both a thorough understanding of their embedded target's architecture, and the services of a powerful, extensible debugger. **esp**

Bill Gatliff is a freelance embedded developer and senior design engineer with Komatsu Mining Systems, Inc. in Peoria, IL, and is a semi-regular presenter at the Embedded Systems Conferences. He can be reached at bgat@usa.net.

References

1. Gatliff, Bill, "Embedding with GNU: GNU Debugger," *Embedded Systems Programming*, September 1999, p. 80.
2 Actually, there are no fewer than seven different ways of accomplishing this, depending on which services the target can provide itself vs. which services it needs external help with. For a more thorough presentation on this subject, see the *1999 Embedded Systems Conference Proceedings* for a paper entitled "gdb: An Open Source Debugger for Embedded Development," by Stan Shebs, PhD, of Cygnus Solutions.