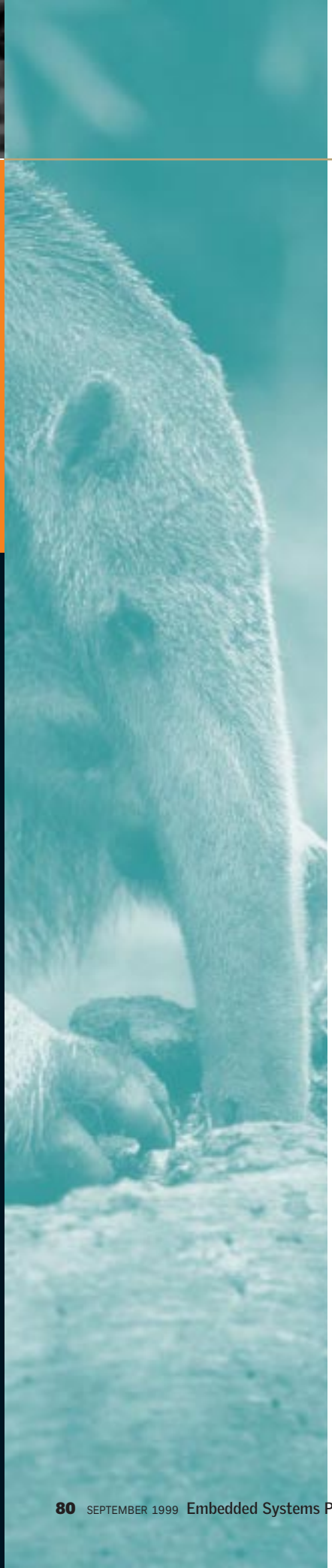


BILL GATLIFF

Embedding with GNU: GNU Debugger

Are GNU tools ready for prime-time embedded development? The author points out the strengths and weaknesses and then explains how to use the GNU debugger to debug firmware running on an embedded system.

The growth in popularity of the industry's best-known collection of free software development tools, the GNU toolkit, has reached the embedded marketplace. Well-known embedded vendors like Wind River Systems, Integrated Systems Inc., and others now openly advertise their compatibility with GNU, and the number of individual and corporate consultants claiming expertise in using GNU tools for embedded software development is growing at a noticeable rate.

But are the GNU tools really ready for prime-time embedded development? I think so, but only under the right circumstances. GNU certainly has its advantages: the tools are cheap (free!), stable, highly portable, and consistent across platforms. However, GNU also has drawbacks: its components were originally designed for developing desktop applications in a Unix-like environment, they can be a challenge to set up and use for embedded development, and the documentation occasionally lacks detail in areas important to embedded developers.

Even with these limitations, a motivated developer who's willing to invest the time to learn to use GNU will find the transition a rewarding and productive endeavor that lends tremendous stability and flexibility to the

I recently began using the GNU tools in an embedded project of my own, and the results have been so inspiring that GNU tools are now my first choice, even when a commercial alternative exists.

embedded development experience. I can say this with confidence because I recently began using the GNU tools in an embedded project of my own, and the results have been so inspiring that GNU tools are now my first choice, even when a commercial alternative exists.

In this article I'll explain how to use the GNU debugger, gdb, to debug firmware running on an embedded system connected to a PC by a serial cable.

What is gdb?

The GNU debugger, gdb, is an extremely powerful all-purpose debugger. Its text-based user interface can be used to debug programs written in C, C++, Pascal, Fortran, and several other languages, including the assembly language for every microprocessor that GNU supports.

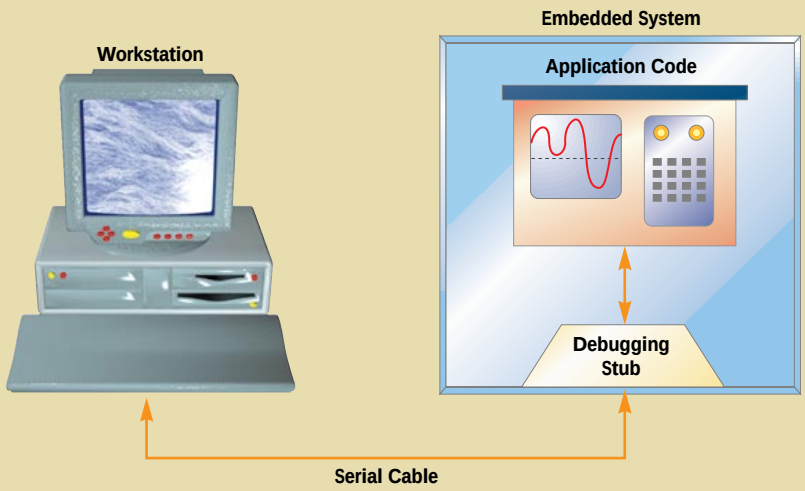
Among gdb's many noteworthy features is its ability to debug programs "remotely," in a setup where the platform running gdb itself (the host) is connected to the platform running the application being debugged (the target) via a serial port, network connection, or some other means. This capability is not only essential when porting GNU tools to a new operating system or microprocessor, but it's also useful for developers who need to debug an embedded system based on a processor that GNU already supports.

The remote debugging capability of gdb, when it has been properly integrated into an embedded system, allows a developer to step through code, set breakpoints, examine memory, and interact with the target in ways that rival the capabilities of most commercially available debugging kernels—and even some low-end emulators.

How gdb works, from an embedded perspective

When debugging a remote target, gdb depends on the functionality provided by a debugging stub, a small piece of code in the embedded system that serves as the intermediary between the host running gdb and the application being debugged. This relationship is depicted graphically in Figure 1.

FIGURE 1 A typical embedded GNU debugger setup



The debugging stub (the technological equivalent of a ROM monitor) and gdb communicate via the gdb Remote Serial Protocol, an ASCII, message-based protocol containing commands to read and write memory, query registers, run the program, and so forth. Since most embedded developers write their own stubs so that they can make the best use of their specific hardware's features and limitations, a clear understanding of how gdb uses the Remote Serial Protocol is very important.

To set breakpoints, gdb uses memory reading and writing commands to nondestructively replace a source instruction with a TRAP or similar opcode.[1] This causes control to transfer to the debugging stub when that instruction is encountered. The debugging stub's job at that point is to communicate the event to gdb (via Remote Serial Protocol messages), and then accept commands from gdb telling it what to do next.

To illustrate, Listing 1 is a TRAP exception handler for the Hitachi SH-2 microprocessor. When the processor encounters a TRAP instruction placed as a breakpoint by gdb, this function sends the processor context to a function called `gdb_exception()`, which subsequently communicates it to gdb. Eventually, the target invokes `gdb_return_from_exception()`, which restores the processor context and returns control to the application.

The Remote Serial Protocol's `step` command is a bit more challenging, especially when the target processor doesn't provide a "trace bit" or similar

LISTING 1 A gdb TRAPA handler for the Hitachi SH2

```
/* An example TRAPA #32 handler for the Hitachi SH2.
/*
/*Stores current register values on the stack, then calls gdb_exception.
*/
asm("
.global _gdb_exception_32
_gdb_exception_32:

/* push the stack pointer and r14 */
mov.l r15, @-r15
mov.l r14, @-r15

/* the sh2 stacks the pc and sr automatically when performing a trap
/* exception, so we have to adjust the stack pointer value we give to gdb to
/* account for this extra data. In other words, gdb wants to see the stack
/* pointer value as it was BEFORE the trap was taken, and not what its value
/* is right now. So, subtract eight (pc and sr are four bytes each) from the
/* sp value we just pushed onto the stack
*/
mov.l @(4,r15), r14
add #8, r14
mov.l r14, @(4,r15)

/* push other register values onto the stack */
mov.l r13, @-r15
mov.l r12, @-r15
mov.l r11, @-r15
mov.l r10, @-r15
mov.l r9, @-r15
mov.l r8, @-r15
mov.l r7, @-r15
mov.l r6, @-r15
mov.l r5, @-r15
mov.l r4, @-r15
mov.l r3, @-r15
mov.l r2, @-r15
mov.l r1, @-r15
mov.l r0, @-r15
sts.l macl, @-r15
sts.l mach, @-r15
stc vbr, r7
stc gbr, r6
sts pr, r5

/* call gdb_exception, pass it exception=32 */
mov.l _gdb_exception_target, r1
jmp @r1
mov #32, r4

.align 2
_gdb_exception_target: .long _gdb_exception
");
```

functionality.[2] In these cases, the only alternative is for the stub to disassemble the instruction about to be executed so that it can determine where the program is going to go next.

Fortunately for me, several suggestions on how to implement the `step` command are supplied with gdb. For the Hitachi SH-2, the function `doSStep()` in gdb/sh-stub.c is illustrative, as are the similarly-named functions in the files gdb/i386-stub.c, gdb/m68k-stub.c, and others.

Other things gdb can do

The debugger can also evaluate arbitrary C expressions entered at the console, including ones containing function calls on the remote target. So you can type commands like:

```
print foo( sci[x]->smr.brg )
```

and gdb will happily report the results.

Of course, gdb can also disassemble code, and it does a good job of supplementing magic numbers with equivalent symbol information whenever possible. For example, the following output:

```
jmp 0x401010 <main + 80>
```

is gdb's way of saying that the address shown is equivalent to an offset of 80 bytes from the start of the function `main()`.

The Remote Serial Protocol messages exchanged between gdb and the debugging stub running on the target can be displayed, as well as logged to a file. These features are extremely useful both for debugging a new stub, and for understanding how gdb uses the Remote Serial Protocol to implement user requests for data, program memory, function calls, and so forth.

The GNU debugger has a scripting language that permits automated target setup and testing. The language is target microprocessor-independent, so scripts can be reused when the target application changes from one microprocessor to another.

LISTING 1, cont'd. A gdb TRAPA handler for the Hitachi SH2

```
/* An example on how to return control to an application from a debugging
/* stub, for the Hitachi SH2.
/*
/* If this were written in C, the prototype would be:
/* void gdb_return_from_exception( gdb_sh2_registers_T registers );
/*
/* In general, we can simply pop registers off the stack in the same fashion
/* as gdb_exception_nn puts them on. However, usually the return stack pointer
/* isn't the same as ours, so if we pop r15 before we copy the pc and sr back
/* onto the return stack, we lose them.
*/
asm("
.global _gdb_return_from_exception
_gdb_return_from_exception:

/* restore some registers */
lds r4, pr
ldc r5, gbr
ldc r6, vbr
lds r7, mach
lds.l @r15+, macl
mov.l @r15+, r0
mov.l @r15+, r1
mov.l @r15+, r2
mov.l @r15+, r3
mov.l @r15+, r4
mov.l @r15+, r5
mov.l @r15+, r6
mov.l @r15+, r7
mov.l @r15+, r8
mov.l @r15+, r9
mov.l @r15+, r10
mov.l @r15+, r11
mov.l @r15+, r12

/* pop pc, sr onto application's stack, not ours */
mov.l @(8,r15), r14
mov.l @(16,r15), r13
mov.l r13, @-r14
mov.l @(12,r15), r13
mov.l r13, @-r14

/* finish restoring registers */
mov.l @r15+, r13
mov.l @r15+, r14
mov.l @r15, r15

/* adjust application's stack pointer to account for pc, sr */
add #-8, r15

/* ... and return to the application */
rte
nop
");
```

And finally, gdb provides tracepoints, a way to record information about a running program with minimal interruption of the program to collect the data. Tracepoints require significant debugging stub support to implement, so they'll be the subject of a future article.

Information on all of these features is provided with gdb. You can type `help` at the gdb console, or you can use a GNU utility called "info" to review documentation contained in the gdb installation package. The debugger also comes with a preformatted quick-reference card in the file gdb/doc/refcard.ps, and typeset user's manuals are available online at several sites that mirror gdb source code distributions.

Installing gdb

GNU tools like gdb are generally distributed as source code, archived, and compressed into a single file using the tar and gzip utilities (available from GNU at *www.gnu.org*). Once the source code is in hand, the user typically decompresses, configures, compiles, links, and installs the programs in a manner most compatible with their workstation setup, target environment, and other factors. Some gracious members of the GNU community provide precompiled binary versions of the most popular tools; the availability of such releases is usually advertised in Internet newsgroups.

Efforts are underway to produce ports of GNU tools for Windows 95, 98, and NT hosts. At the present time, however, gdb and other tools necessary for embedded development cannot be easily built on Microsoft hosts, although they will run fine when properly cross-compiled on another host. To get the latest information about GNU-Microsoft compatibility, check out the Cygwin Project at *http://sourceware.cygnus.com/cygwin/*.

The most recent release of gdb is the file gdb-4.18.tar.gz, which contains both the gdb source code and some configuration scripts that help automate its compilation and installation. This file is available from the Cygnus Solutions gdb Web site, *http://sourceware.cygnus.com/gdb/*.

To install gdb, first decompress gdb-4.18.tar.gz:

```
tar xzvf gdb-4.18.tar.gz
```

Next, study gdb-4.18/README and gdb-4.18/gdb/News, as these files contain important information on how to install gdb, the current state of gdb development, new features, and so forth.

Per the installation instructions in README, configure gdb for your host machine and debugging target:

```
mkdir gdb-build
cd gdb-build
../gdb-4.18/configure
    -- target=sh-hitachi-hms
```

The configure script will study the host machine's setup and create the proper local environment in which to build gdb. The `target` parameter tells gdb what microprocessor your embedded applications will run on. You'll find a list of supported targets in the file gdb/ChangeLog.

Finally, tell gdb to compile and install itself:

```
make all install
```

When you're done, you'll end up with an executable called `sh-hitachi-hms-gdb` (or whatever prefix you supplied in the `target` parameter during configuration). By default, this file is placed in /usr/local/bin. To run it, simply type the executable file name followed by the name of your application file:

```
sh-hitachi-hms-gdb a.out
```

Source code for a gdb debugging stub

Despite the target-specific nature of remote software debugging, it is possible to create a highly portable debugging stub that can be reused with minimal modification on several different embedded microprocessors.

I have posted my attempt at a "portable" debugging stub on *ESP*'s Web site, at *www.embedded.com/code.htm*. This source code has been compiled and tested for the Hitachi SH-2, and ports are underway for the Hitachi H8, Power PC, and other processors.

The processor-specific code is contained in files with processor-specific file names, like gdb_sh2*.c. You should first copy these files to ones with names related to your microprocessor (such as gdb_m68k*.c), and replace the contents with code that works for your machine.

If you do succeed in porting my code to another processor, or desire assistance in doing so, please let me know. In addition to hearing your feedback, I would like to make your modifications available to the rest of the embedded community.

A typical gdb session

Now that we've covered the gdb's functionality in general terms and I've shown how to install it, let's observe gdb in action. Table 1 is a transcript of a typical gdb debugging session, during which gdb initiates communications with a remote target running a debugging stub, downloads a program, sets a breakpoint, and then runs the program. The debugging stub informs gdb when the breakpoint is encountered, and gdb then displays the appropriate source line to the user. The user subsequently displays a variable, steps one instruction, and then exits gdb.

The left column of the example shows a portion of the gdb console, where the user types commands and views data. The right columns show some of the GDB Remote Serial Protocol messages exchanged between the host machine and the embedded device. I've included supplemental information in square brackets. For a detailed explanation of

these messages, see the sidebar, "The GDB Remote Serial Protocol."

Note that Table 1 is not what users see when they use gdb—they see a terminal display in English, complete with source code, displayed variables, and the like. Instead, the transcript shown illustrates what happens behind the scenes when the user types the listed commands.

Ideas on adapting gdb to solve specific problems

The modular implementation architecture used by gdb makes it straightforward to change aspects of gdb's behavior that don't suit your needs. For example, if your product has only one communications port that is already dedicated to a non-gdb communications protocol, then it's possible to modify gdb so that debugger messages fit inside of packets that your product already understands. Likewise, if your product lacks a serial port but has some other kind of communications interface—like a CAN port, for instance—then you could enhance gdb's remote communications strategy to work with an off-the-shelf serial-to-CAN or parallel-to-CAN bridge.

You can also modify gdb's behavior to make it more compatible with other software in your embedded application. For example, if you were already using TRAPA #32 for something non-gdb-related, you could either change the opcode gdb uses for a breakpoint, or you could have gdb produce a new message—one that signaled your target to turn on instruction tracing or enable on-chip breakpoint-generating hardware, for example.

The file gdb/remote.c contains gdb's implementation of the Remote Serial Protocol, and is a good starting point for studying how gdb's modular implementation permits you to quickly adapt it to meet the needs of a specific debugging target. Other files, like gdb/remote-hms.c and gdb/remote-e7000.c, use this modular framework to provide support for debuggers and emulators supplied by Hitachi, Motorola, and other vendors.

For the most up-to-date information on how gdb works, how to enhance gdb, and how to get your improvements into future gdb releases, type `info gdb-internals` after installing gdb.

TABLE 1 Transcript of a typical gdb session

What the user enters	What happens on the serial port gdb sends...	 target responds with...
host>gdb myprogram		
gdb> target remote /dev/ttyS0	+$Hc-1#09 +$qOffsets#4b +$?#3f +$g#67	+$OK#9a +$Text=0;Data=0;Bss=0#04 +$S05#b8 +$00001a00ffff81b200000020...
gdb> load	$M401054,10:004020240040 2024004020240040 2024#72 ... [lots more M messages]	+$OK#9a
gdb> breakpoint main	[nothing— gdb physically sets the breakpoint immediately before the continue command is sent]	
gdb> continue	+$M4015cc,2:c320#6d +$c#63 [gdb places a breakpoint opcode at main()]	+$OK#9a + [program runs until it reaches main()] $T050:00401400;1:00404850;2:00000001;3:00000030;4:ffffffff;5:00000000;6:00000010;7:00000010;8:0040161c;9:00002070;a:00404068;b:004015bc;c:ffffffff;d:fffffef;e:00404840;f:00404840;10:004015cc;11:004015cc;12:d04001e2;13:00401000;14:00000000;15:00ffffff;16:000000f0;#d1 [target stopped at main(), address 0x4015cc]
gdb> display foo	+$m4015bc,2#5a	+$2f86#06 [foo is at address 0x4015bc; its value is 0x2f86]
gdb> stepi	$s#73	+ [target executes one instruction] $T050:00401400;1:00404840;2:00000001;3:00000030;4:ffffffff;5:00000000;6:00000010;7:00000010;8:0040161c;9:00002070;a:00404068;b:004015bc;c:ffffffff;d:fffffef;e:00404840;f:00404840;10:004015ce;11:004015cc;12:d04001e2;13:00401000;14:00000000;15:00ffffff;16:000000f0;#d2 [PC is now 0x4015ce]
gdb> quit	$k#6b	+

tools I've listed don't suit your needs.

DDD: Data Display Debugger

The Data Display Debugger (*www.cs.tu-bs.de/softech/ddd/*) by Andreas Zeller (Software Technology Department, Technical University of Braunschweig) is a mature, high-quality X-Windows-based graphical gdb interface. Besides the usual features you would expect from any graphical debugger, DDD provides an easy-to-navigate graphical data display that allows sophisticated data structure visualization with just a few mouse clicks.

This application is well documented and easy to install. Its graphics-intensive interface extracts a slight penalty in runtime performance, but any host processor that can reasonably run other graphical GNU tools will probably work fine.

Code Medic

Code Medic (*www.cco.caltech.edu/~glenn/medic/*) is an elegant, X-Windows-based graphical interface to gdb's most important features. The interface was written by Glenn Bach (of the Physics, Mathematics, and Astronomy Division, California Institute of Technology), and is also part of a suite of integrated software development tools known as Code Crusader.

Code Medic provides drag-and-drop data management, context-sensitive highlighting, and a sophisticated display tailored for complicated data structure visualization. Furthermore, Code Medic uses a message-based interface to gdb, which improves performance significantly over DDD in both remote and self-hosted debugging setups.

FIGURE 2 Data Display Debugger

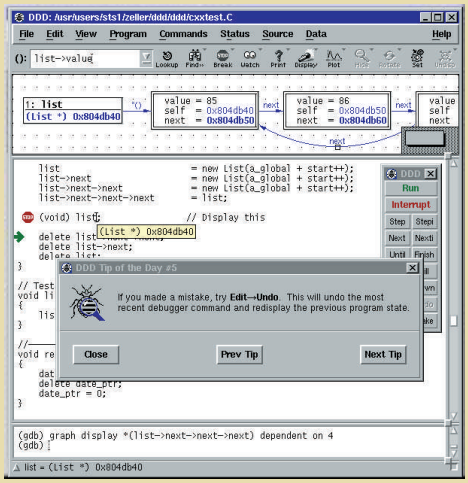


FIGURE 3 Code Medic

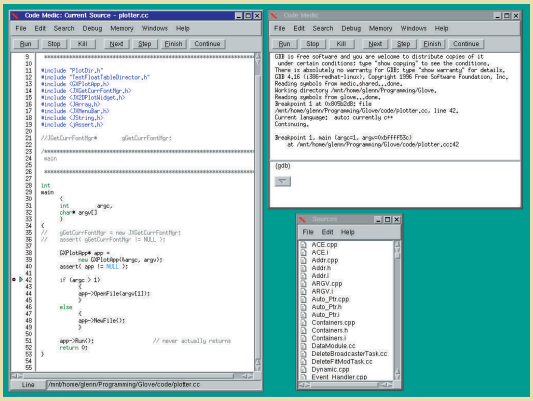
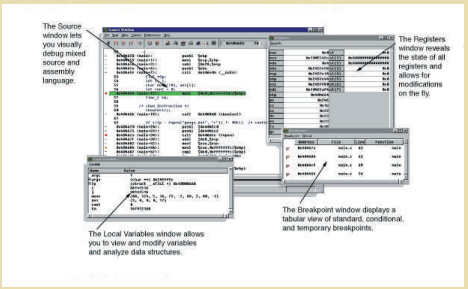


FIGURE 4 Insight



Insight

Insight (*http://sourceware.cygnus.com/gdb/)* is Cygnus Solution's own (and formerly proprietary) graphical enhancement to gdb. In contrast to DDD and Code Medic, Insight's graphics come by way of Tcl/Tk instead of X-Windows, which means it runs easily on Windows platforms. Furthermore, Insight is compiled into gdb (rather than it running gdb as a subprocess, the way CodeMedic and DDD do), which improves its performance and makes its communications with gdb more interactive. It also provides functionality to graphically configure remote connections, which the other front ends can only do by way of gdb command scripts.

But is gdb right for you?

I hope that by now you're convinced that gdb is a powerful and capable tool for embedded software debugging. If that were all that mattered, then I believe that gdb would be the most popular embedded debugger available today.

Unfortunately, tool selection is a trade-off among many factors—price, performance, compatibility, and support, to name a few—and when everything is considered, gdb has both its strengths and weaknesses (as does any tool available today at any price, free or otherwise).

As I've already mentioned, gdb's strengths (besides its cost) lie in its feature list, its uniform applicability across many host platforms and debug targets, and the degree to which its behavior can be tailored to meet the specific demands of an embedded debugging target.

In particular, gdb's consistency across hosts and targets presents the opportunity for considerable reuse of automated configuration, compilation and unit testing scripts, which is a value that should not be underestimated. This in itself may reward the pain of switching to GNU, particularly if proliferation of

your products across a variety of target processors is likely.

Although the GNU tools are generally well documented, the information occasionally lacks details that are important to embedded developers. For example, procedures to properly establish the C run-time environment—initialize global variables, zero out memory, and so forth—are only supplied as working code and as an occasional side note in the descriptions of a few linker commands; there is currently no single document that completely describes in detail a process that is critically important for C/C++-based embedded systems. (I'm certain that the GNU effort would welcome the creation of a tutorial on this subject.)

Finally, gdb works best when fed debugging information produced by the GNU compiler and linker, which presents compatibility problems for non-GNU legacy projects, or work involving third-party libraries that aren't available in a GNU-compatible format.

What about support?

One perceived drawback that's actually an advantage for GNU is the lack of available support from a central, controlling organization. Instead, people highly skilled in the use and development of the GNU tools (including the authors of the tools themselves) are available 24 hours a day via a variety of Internet news groups (gnu.gdb.bug is one) and mail servers (*www.sourceware.cygnus.com/gdb*, for example). In situations where I've requested help I usually received an answer in four hours or less—an impressive statistic for even the most expensive product support desks.

On the other hand, if you truly need the level of support that only money can buy, or you need extensive modifications to one or more GNU tools in support of a very specialized situation, then there are companies and individuals around that can provide those kinds of services. The best way to find them is to subscribe to and read the Internet news groups and mail specific to the GNU tools you need help with.

A rewarding experience

The ability of gdb to adapt to the specific needs of a debugging target

The GDB Remote Serial Protocol

The GDB Remote Serial Protocol (RSP) is the lingua franca between gdb and a remote target. It defines messages for reading and writing data, controlling the application being debugged, and reporting application status. The host side is implemented in gdb's remote.c source file.

The following are short descriptions of the RSP's most important commands. A complete summary of each command is available on the *ESP* Web site, *www.embedded.com/code.htm*.

Data exchanged between gdb and a remote target with the GDB Remote Serial Protocol uses plain ASCII characters. Messages begin with a dollar sign ($) and end with a gridlet (#) and eight-bit checksum. In other words, each message looks like this:

```
$ <data> # CKSUM_MSN CKSUM_LSN
```

where <data> is typically a string of ASCII hex [0-9,a-f,A-F] characters.

CKSUM_MSN and CKSUM_LSN are ASCII hex representations of an eight-bit checksum of <data>. Only the hex digits 0 to 9 and a to f are allowed.

When a message is sent, the receiver responds with either:

+ if the received checksum was correct, and the receiver is ready for the next packet
– the received checksum was incorrect, and the message needs to be retransmitted

A target can respond to a message from gdb with either data or an OK (depending on the message), or a target-defined error code. When gdb receives an error code, it reports the number to the user via the gdb console.

Definitions for <data> are shown below.

TABLE A Register-related commands

Command name	<data>definition	Description
read registers	g	Return the values of all registers
write registers	GXX..XX	Set registers to XX..XX
write register nn	Pnn=XX..XX	Set the value of register NN

TABLE B Memory-related commands

Command name	<data> definition	Description
read memory	mAA..AA,LL..LL	Read values from memory
write memory	MAA..AA,LL..LL:XX..XX	Write values to memory

TABLE C Target infromation commands

Command name	<data> definition	Description
query section offsets	qOffsets	Return section offset information

TABLE D Program control commands

Command name	<data> definition	Description
set thread	Hc	Set current program thread
step	sAA..AA	Execute one assembly instruction
continue	cAA..AA	Resume application execution
last signal	?	Report last signal
kill	k	Terminate application

TABLE E Target status messages (responses from the target)

Message name	<data> definition	Description
last signal response	Snn	Minimal reply to the *last signal* command
expedited response	Tnnr...:v...r...:v...;	The last signal reported, plus key register values
console output	Ovvvvvvvv...	Sends text from the target to gdb's console

(memory usage, communications media, and so forth) often makes it the only choice available for on-target debugging, particularly given the growing popularity of single-chip, highly-integrated, and IP-based embedded products. The complexity of today's embedded devices is increasing at an alarming rate, and as the technology choices available for new designs continue to diverge, finding a commercial development tool vendor with a product that fits your needs becomes less likely every day.

A switch to GNU might be in your best long-term interests, even if you're using conventional technology in your embedded designs, because GNU applications are becoming increasingly popular as educational tools. So a recent engineering graduate would probably be more familiar with them than with any specific equivalent commercial products.

And finally, GNU's support of a variety of popular embedded processors means that you reduce the risk of needing to find a new tool vendor because your current one doesn't support the processor you'd like to use in your next design.

With all these advantages, I have no doubt that a motivated developer who is willing to master the GNU tools will find their product development a more stable, flexible, and rewarding experience. **esp**

Source code for gdb is available at http://sourceware.cygnus.com/gdb/. *This site also includes links to gdb-specific mailing lists, graphical front ends, and sources for precompiled gdb binaries.*

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. An assumption here is that the application being debugged is located in RAM. With a smart enough debugging stub, proper hardware support, and/or compiled-in breakpoints, however, this need not be true.
2. For example, Motorola 683xx processors contain the ability to trap on instruction execution and/or changes in program flow; this feature is controlled by the "trace enable" bits, T1 and T0, in the processor's status register.