Getting Started with GNU:

A Tutorial Introduction Using the ARM Evaluator-7T

William Gatliff

Table of Contents

Looking for a low-cost way to explore GNU tools for embedded development? Here's one approach.

If you have been following any news related to embedded systems development recently, you will have heard that GNU tools are growing increasingly popular for mainstream embedded systems development. To say that they are just now becoming popular for all types of embedded development would be untrue--- GNU tools have been used to produce embedded applications since their creation, but haven't seen widespread use in the greater embedded community until recently.

This article provides you an opportunity to evaluate GNU tools for embedded development in a low-cost setting using either the GNU debugger's ARM instruction set simulator, or the ARM Evaluator-7T single board computer. The Evaluator-7T is a good choice for an introductory GNU experience, because the ARM family of chips is well-supported by GNU, and this inexpensive board in particular comes ready-to-play with the entire GNU development tool chain, including the GNU debugger.

The following paragraphs discuss some of the advantages and disadvantages of using GNU tools for embedded development, then dive right in and show you how to set up and explore your own GNU development environment using a simple *"hello, world!"* program. I then run the program on the GNU debugger's ARM instruction set simulator, and on the Evaluator-7T board. Finally, I conclude with links to sources of pre-configured commercial GNU tool distributions that support lots of other hardware platforms, and a list of resources you can turn to for even more information.

A version of this article appeared in the December 2001 Issue of Circuit Cellar Ink (http://www.circuitcellar.com) Magazine.

Copyright

Why GNU?

One of the obvious advantages of the GNU family of development tools is their price. By virtue of their distribution license, source code for GNU tools like the compiler, linker and debugger can be had at virtually no cost by simply downloading them from the Free Software Foundation (http://www.fsf.org) (FSF)'s GNU website.

What you will find if you go this route, however, may disappoint you. The source code distribution packages do not come preconfigured for any of GNU's supported targets, so you must configure and build them yourself. Furthermore, the base tool distributions do not include a graphical *Integrated Development Environment* (IDE) to facilitate point-and-click code generation or debugging. Such tools certainly do exist, but they are not part of the base GNU tool distributions. You will see the explanation for this later on, when we discuss the GNU debugger in detail.

Once you have the GNU development tools configured and installed, you will discover that you have at your disposal a powerful and flexible foundation of products that can be used to deliver just about any kind of embedded solution, regardless of how arcane or extreme your needs might be. GNU tools are highly adaptable, and can be tailored to provide solutions that few competitive commercial offerings can:

debugging over CAN and other atypical communications links, quick replacement of internal libraries like floating point emulation, and fine-grained, precise control over linker memory assignments are but a few of the possibilities.

GNU tool flexibility extends beyond the target domain, however. No other embedded toolchain in existence, proprietary or otherwise, supports as many combinations of host and target environments as GNU. Whether your development workstation is a Linux PC, a Solaris Sparcstation, or even a Win32 machine, GNU probably runs on it--- and runs exactly the same as on any other host. Heterogenous development environments are especially attractive for GNU, because build management and automated debugging scripts will move consistently and transparently between systems with minimal or no modifications whatsoever.

Perhaps the most surprising thing of all, however, is how well GNU tools are supported. An active embedded GNU user base keeps in almost contact via a collection of mailing lists, frequently-asked-questions documents and websites, and in many cases the authors of the tools themselves offer valuable assistance by providing explanations helpful examples. To illustrate: I have consistently found that if I ask an intelligent question, I can expect to get an intelligent answer back in less than a day--- generally less than four hours. That's an impressive statistic for any help desk, but an especially impressive one for a help desk that costs nothing to use.

Why not GNU?

Alongside their long list of strengths, the GNU tools have a short list of weaknesses. The first two are that GNU tools aren't well documented for widespread embedded use, and they aren't "shrinkwrapped" to the point that new users can get started quickly. In response, a few vendors offer CDs containing preconfigured GNU setups that work with popular development hardware. There is also a growing list of consultants specializing in embedded GNU, who can provide reasonably-priced help during all phases of a development effort. The resources section at the end of this article includes a few links to this and other kinds of information.

Another drawback of the GNU world is that a switch to it from a commercial, closed development environment may incur some serious headaches or, even worse, be an all-or-nothing proposition. Many proprietary embedded development tool and library vendors do not publish specifications for their file formats, so you usually can't directly import outputs from their tools into a GNU system. In other words, if a vendor of a closed-source library of code won't provide the library in a format that the GNU linker understands, then you're sunk before you've even left the harbor. Ditto for the communications protocol used by your hardware emulator.

One much-hyped limitation of the GNU tools that isn't really a limitation at all is the notion that you cannot use GNU tools to produce proprietary, closed-source products. This is simply untrue, and in fact most embedded GNU users *are* using the tools in this way, and doing so legally. The real limitation, if

you want to call it that, is that you cannot incorporate libraries of code licensed under the GNU General Public License (http://www.gnu.org/licenses/licenses.html#GPL) (GPL) into applications that remain proprietary to the end user. For more on this, you should read the text of the GPL and related licenses as they appear on the Free Software Foundation's website. And while you are doing that, you should also read the licenses for the tools you are using today--- you may find that you're already agreeing to some terms that are just as surprising.

Keep in mind that the Free Software Foundation is not the only place to look for libraries of downloadable, GNU-compatible materials. The Internet is literally bursting with source code, released under licenses varying from "must use in Open Source applications only", to "come and get it." Graphics libraries, boot loaders, diagnostic tools, and even entire operating systems like Linux (http://www.kernel.org) and eCos (http://sources.redhat.com/ecos) can be easily found after just a few minutes with your favorite search engine.

If GNU tools are free, then why buy a CD?

It is important to understand that the Free Software Foundation doesn't exist to provide development tools at zero cost. Their mission is really to provide development tools and other code in a way that preserves the freedoms of the end users of that code--- the zero cost part is just one side effect of that. For example, their belief is that if you have purchased an operating system for your workstation, then you should get the source code for it as well, so that you may do with it as you wish.

A useful analogy is that of an automobile: when you buy a car, the automaker doesn't tell you where you may and may not drive with it. The FSF provides licenses like the GPL to enable software developers to pursue a similar philosophy if they so choose. The FSF website has a more detailed presentation of how their licenses and tools relate to their mission.

When you buy a CD containing FSF products, you demonstrate support of the FSF's goals. Your dollars go directly to the FSF if you purchase directly through them, or in some cases indirectly through the CD vendor in the form of a "portions of the proceeds" donation. But even if the CD vendor does not contribute financially to the FSF, the FSF still gets something out of the transaction: another satisfied GNU user, another supporter of their philosophy. Any way you slice it, the FSF wins--- as do you.

Purchasing a CD also gets you some support time with the vendor, which can come in handy if you have troubles with installation, or when you set out to adapt the tools to run on your own hardware. Usually this adaptation process is straightforward, but having someone around who knows the tools in detail is always an advantage.

Finally, buying a CD usually makes business sense. One vendor listed in the resources section at the end of this article offers a product containing preconfigured GNU tools for most of the targets that GNU supports. Despite its bulk, the price is very reasonable: about the same as a couple of hours of an

engineer’s salary. Unless you have a lot of prior GNU experience, creating a basic GNU installation by hand in a business setting will be more expensive than their CD.

So why aren’t we buying a CD?

Well, an article that says "just go buy a CD" would be really short, and probably not very interesting or compelling.

There is educational value in setting the tools up by hand, particularly if you can do it in a hobbyist’s environment where the business-case arguments aren’t applicable. The GNU development tool installation process is very similar to the process used to install other GNU and related tools, so going through the procedure manually a few times is a training exercise that pays dividends down the road, when you start using and adapting GNU tools for more specialized use.

The main reason we’re doing things by hand, however, is that there is no better way to get a thorough look at what the GNU tools are all about than by digging around and getting a little dirt under our fingernails, so to speak. So let’s get started already!

Building GNU Tools

In the next few sections, we’re going to set up an compiler, runtime library, assembler, linker and debugger that can be used to produce and run applications for the ARM Evaluator-7T development board. We’ll subsequently prove that by building and testing a simple, "hello, world!"-style application on the ARM microprocessor.

The first step in the process is to get the materials we need. The instructions will assume that you are working in one of two host environments: either a Linux or other unix-like workstation, or on a Win32 PC (preferably Windows NT) running a unix emulation environment called Cygwin (http://sources.redhat.com/cygwin). If you install Cygwin to try out this article, then Use the default installation setup.

Some commercial GNU tool distributions include a copy of Cygwin. If you have recently purchased a Windows-compatible GNU setup, check to make sure it doesn’t already include Cygwin before downloading another copy.

If you are trying to run Cygwin on Windows 95 or 98, expect to have some difficulty completing the procedure we’re about to start. Cygwin occasionally has difficulty providing a solid emulation in these environments, and when things start to look strange you will have to simply reboot and try again. Once you get the tools built, however, they run fine on these machines. Cygwin is very stable on Windows NT and Windows 2000.

You will need an ARM Evaluator-7T board (http://www.arm.com/devtools/evaluator_7t?OpenDocument) if you want to try out the "hello, world!" program on real hardware. The Evaluator-7T is available from ARM, Inc.'s website (http://www.arm.com).

If you don't want to invest in hardware just yet, then stick with me anyway: the GNU debugger comes with an excellent ARM instruction set simulator that can emulate much of what the Evaluator-7T can do, and in some cases even more, without any external hardware at all.

Table 1 lists the files you will need, and shows where to download them from.[1] There will probably be later versions of each of these packages by the time you get around to downloading them, but stick with the listed versions until you've completed the process at least once; updates occasionally change the installation procedure.

Note also that if you use Microsoft's Internet Explorer to download these, then you *must* right-click on the download link and select "Save Link As". Internet Explorer often has trouble with files in the GNU tar file format, and will corrupt them if you let it. If you are using Cygwin and cannot find the files after you download them, then move the downloaded files to a directory like `c:\cygwin\home\<adminstrator>` (replace *administrator* with your username, if necessary). This is Cygwin's "home directory", which should be right where you can find them when you type **ls** from the Cygwin shell's command prompt.

As you might imagine, the Free Software Foundation's website stays busy. If you have trouble getting a connection, use one of the mirror sites listed at:

http://www.gnu.org/order/ftp.html.

Table 1. What you need.

binutils (http://sources.redhat.com/binutils)	ftp://ftp.gnu.org/gnu/binutils/binutils-2.11.2.tar.gz
gcc (http://gcc.gnu.org)	ftp://ftp.gnu.org/gnu/gcc/gcc-2.95.3.tar.gz
newlib (http://sources.redhat.com/newlib)	ftp://sources.redhat.com/pub/newlib/newlib-1.9.0.tar.gz
gdb (http://sources.redhat.com/gdb)	ftp://sources.redhat.com/pub/gdb/releases/gdb-5.0.tar.gz
gdb patchfile (http://www.billgatliff.com/articles/gnu/gdb-5.0-rdi.patch)	http://www.billgatliff.com/articles/gnu/gdb-5.0-rdi.patch

The binutils package includes the GNU assembler and linker, and the gcc package contains the GNU

C/C++ compiler. Newlib is the C runtime library we will use to supply functions like `printf()`, and the gdb package contains the GNU debugger.

Setting up the build environment

Set up a directory structure in which to build and install the tools. Then, set up some environment variables that will save typing later and make sure that the arguments used during the configuration process are consistent throughout--- a source of common errors. Figure 1 describes the commands, just type them in as they appear there (omit the leading '$', which represents the Cygwin or unix command prompt).

Figure 1. Setting up the build environment.

```
$ cd
$ mkdir build-binutils
$ mkdir build-gcc
$ mkdir build-newlib
$ mkdir build-gdb
$ mkdir install
$ export TARGET=arm-elf
$ export PREFIX=‘pwd‘/install
$ export PATH=$PATH:$PREFIX/bin
```

In the `PREFIX` setting, the ticks surrounding the **pwd** command are backticks, often found on the tilde (~) key of a standard US-101 keyboard layout. You can see the result of this command by typing the following:

```
$ set | grep PREFIX
```

The value of `TARGET` is an argument that specifies the ARM microprocessor version of the tools, using the ELF debugging format. `PREFIX` specifies the topmost directory under which the GNU tools will be installed. The last command adds the GNU installation directory to the search path, so that we can run the tools after they're installed.

Next, decompress the source code for the tools themselves.

```
$ tar xzvf binutils-2.11.2.tar.gz
$ tar xzvf gcc-2.95.3.tar.gz
$ tar xzvf newlib-1.9.0.tar.gz
$ tar xzvf gdb-5.0.tar.gz
```

Building the binutils package

The commands to configure and install the assembler and linker are shown Figure 2. These commands do the following:

- Run the **configure** script included with the binutils package, to set up the source code to build for the ARM target.
- Invoke the **make** program to actually compile the and install binutils. The output from **make** is captured and stored in the file `make.log`.

When this process finishes, you will see a number of programs in `${PREFIX}/bin`. The assembler is **arm-elf-as**, and the linker is **arm-elf-ld**. **arm-elf-objcopy** is a utility that translates files to different formats (from ELF to S Record, for example), and **arm-elf-objdump** is a utility that can dissect the components of a file, to show you things like symbol addresses and a disassembly of the file's object code. We'll discuss these utilities later.

Figure 2. Instructions to build binutils

```
$ cd build-binutils
$ ../binutils-2.11.2/configure --target=$TARGET --prefix=$PREFIX
$ make all install 2>&1 | tee make.log
$ cd ..
```

Building a bootstrap cross compiler

Now that we have an assembler and linker, we can build and install the GNU C/C++ compiler. The initial step in the procedure yields a *bootstrap* compiler that can only be used to build runtime libraries and header files; we will use this compiler to build the arm-elf version of the newlib C runtime library. Once that's complete, we will rebuild the compiler completely, including internal libraries that need target-specific header files from newlib in order to be compiled properly.

The commands to make the boostrap compiler are shown in Figure 3. Do them now.

Figure 3. Instructions to build a bootstrap gcc.

```
$ cd build-gcc
$ ../gcc-2.95.3/configure --target=$TARGET --prefix=$PREFIX \
   --with-newlib --without-headers --with-gnu-as \
   --with-gnu-ld --disable-shared --enable-languages=c
```

```
$ make all-gcc install-gcc 2>&1 | tee make.log
$ cd ..
```

At this point we have a bootstrap compiler called **arm-elf-gcc**, located in `${PREFIX}/bin`. Now we will build newlib, then return to finish building gcc.

Building the newlib C runtime library

The procedure, shown in Figure 4, should seem pretty familiar by now.

Figure 4. Instructions to build newlib.

```
$ cd build-newlib
$ ../newlib-1.9.0/configure --target=$TARGET --prefix=$PREFIX
$ make all install 2>&1 | tee make.log
$ cd ..
```

Building a complete cross compiler

Now that we have ARM header files and libraries from newlib, we can build a complete cross compiler setup for C/C++ development. The steps are shown in Figure 5.

Figure 5. Instructions to build gcc.

```
$ cd build-gcc && rm -rf *
$ ../gcc-2.95.3/configure --target=$TARGET --prefix=$PREFIX \
   --with-gnu-as --with-gnu-ld --enable-languages=c,c++
$ make all install 2>&1 | tee make.log
$ cd ..
```

Building the debugger

Are we there yet? Almost. The last step in the setup process is to build and install the debugger. The procedure is shown in Figure 6.

Figure 6. Instructions to build gdb.

```
$ patch -p0 < gdb-5.0-rdi.patch
$ cd build-gdb
$ ../gdb-5.0/configure --target=$TARGET --prefix=$PREFIX
$ make all install 2>&1 | tee make.log
$ cd ..
```

This procedure is slightly different from what you have already seen, in that it includes a **patch** step. This step corrects a minor source code problem that prevents gdb from building properly for the arm-elf target.

Like most GNU programs, the GNU debugger is under constant development, and sometimes is released with minor issues that affect one or two of its many supported targets. When such problems are resolved, a *patchfile* is usually released that contains just the differences between the updated version and the original version; this lets you get the improvement by downloading just a few bytes of data, instead of downloading the whole package again.

The patchfile is plaintext; you can easily read it with a text editor to see the changes it contains. In fact, that's exactly what the **patch** program does.

If you run out of disk space

All told, you will need about 500MB of free disk space in order to build a complete GNU cross development environment. But you don't that much all at once: if space is tight, you can delete the files left at the end of each step in the procedure before you move on to the next step. For example, once you finish building the cross assembler and linker, you can delete the contents of the `build-binutils` directory.

The installed tool set requires less than 100MB of disk space.

Compiling and Analyzing a Program

Now that we have shiny new tools, let's try them out. Using the text editor of your choice, create a file `hello.c` that looks like Figure 7.

Figure 7. The `hello.c` program.

```
#include <stdio.h>

int a_global = 10;
```

```
int foo ( int arg )
{
  static int foo_counts = 0;

  foo_counts++;
  return foo_counts + arg;
}

int main ( void )
{
  int nloops;
  char* charstr = "hello, world!\n";


  for( nloops = 0; nloops < 10; nloops++ ) {
    printf( "%s", charstr );
    foo( nloops );
  }

  return 0;
}
```

Compiling a program

You compile `hello.c` into an ARM executable file called `hello`, like this:

```
$ arm-elf-gcc -Wall -g -o hello hello.c
```

or, like this:

```
$ arm-elf-gcc -v -Wall -g -o hello hello.c
```

The first command quietly compiles, assembles and links the `hello` application; the second command is more verbose, which is useful when you are trying to find out where **gcc** is looking for header files or linker command files. The `-Wall` option causes gcc to emit warnings for almost every kind of suspicous code construct it finds, and the `-g` option includes debugging information in the output file.

If you are using the arm-elf tools under some versions of Cygwin, the compiler's output file is called `hello.exe`, instead of just `hello`. The file itself is still an ARM executable, however, and you can confirm that using the **file** command. You can also rename `hello.exe`, using the **mv** command.

```
$ file hello.exe
hello.exe: ELF 32-bit LSB executable, Advanced RISC Machines ARM, version 1,
```

```
statically linked, not stripped

$ mv hello.exe hello
```

Some other useful variations on how to invoke **arm-elf-gcc** are:

```
$ arm-elf-gcc -S hello.c
```

to compile to assembly language and then stop (the output file is `hello.s`), and:

```
$ arm-elf-gcc -O3 -o salutations -Wl,-Tlinker.cmd hello.c goodbye.c
```

to combine the modules `hello.c` and `goodbye.c` into a single, highly optimized application, linked using instructions found in a linker command file called `linker.cmd`. The resulting executable is named **salutations**.

You will want to avoid supplying optimization commands like `-O3` during compilation of source code modules you intend to debug. The GNU compiler is an aggressive optimizer, and if you try to step through optimized code you will find that some source lines have moved, some have changed, and some have gone away entirely. Optimized code still produces the results you expect it to, but not necessarily in the way you expect it to; this makes it nearly impossible to debug using linear techniques like line-by-line stepping.

Using objdump

The utilities **objdump** and **objcopy** are two of many hidden gems of the GNU tool family. **Objdump** cracks apart the contents of object and executable files to display their contents; **objcopy** translates object and executable files to different formats.

To get an overall view of the structure of our new **hello** program, invoke **objdump** like this:

```
$ arm-elf-objdump --headers hello
```

The output, shown in Figure 8, packs a lot of information into a small space. It starts by stating that **hello** is an ARM executable file, coded using the ELF file format in little-endian mode.

Figure 8. Output from objdump --headers.

```
hello:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
```

```
 0 .text         00008880  00008000  00008000  00008000  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
 1 .rodata       00000230  00010880  00010880  00010880  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
 2 .data         0000085c  00010bb0  00010bb0  00010bb0  2**2
                 CONTENTS, ALLOC, LOAD, DATA
 3 .ctors        00000008  0001140c  0001140c  0001140c  2**2
                 CONTENTS, ALLOC, LOAD, DATA
 4 .dtors        00000008  00011414  00011414  00011414  2**2
                 CONTENTS, ALLOC, LOAD, DATA
 5 .sbss         00000000  0001141c  0001141c  0001141c  2**0
                 CONTENTS
 6 .bss          000000e0  0001141c  0001141c  0001141c  2**2
                 ALLOC
 7 .debug_aranges 00000540  00000000  00000000  0001141c  2**0
                 CONTENTS, READONLY, DEBUGGING
 8 .debug_pubnames 00000af5  00000000  00000000  0001195c  2**0
                 CONTENTS, READONLY, DEBUGGING
 9 .debug_info   00020938  00000000  00000000  00012451  2**0
                 CONTENTS, READONLY, DEBUGGING
10 .debug_abbrev 00003098  00000000  00000000  00032d89  2**0
                 CONTENTS, READONLY, DEBUGGING
11 .debug_line   0000a400  00000000  00000000  00035e21  2**0
                 CONTENTS, READONLY, DEBUGGING
12 .stack        00000000  00080000  00080000  00040221  2**0
                 CONTENTS
```

Next, **objdump** lists the contents of all the memory sections in the file. The first section is called *.text*, and contains the machine instructions that make up the program. The section is 882CH (hex, or 34860 decimal) bytes long, and starts at address 8000H. The section's Virtual Memory Address and Load Memory Address are the same, which means that the linker placed the contents of the section at the physical address in which the code was intended to run. This will almost always be the case for .text sections of programs that do not implement or use virtual memory.

The next line states that the linker marked the .text section with the attributes `CONTENTS`, `ALLOC`, `LOAD`, `READONLY` and `CODE`. Put simply, this means that if an operating system tries to load this file into memory, it must allocate memory for the .text section and copy the contents of this part of the **hello** file into it.

The *.rodata* section contains the program's read-only data. In addition to the "hello, world!" text, this section also includes some string and constant data defined by the runtime library. The `.data` section holds the program's initialized global variables, which includes `a_global` and, again, some declarations from the runtime library.

The *.ctors* and *.dtors* sections are lists of function pointers to global C++ object constructors and destructors. If **hello** had been a C++ application with global objects, these sections would be larger. As it is, they only include a single entry: a null pointer that tells the runtime library's startup code that there are no constructors or destructors to run.

The *.sbss* and *.bss* sections hold the program's uninitialized global data. They lack the LOAD attribute because there is nothing in the **hello** file to load: an runtime program loader only needs to allocate initialized memory at the section's address for the program to use.

With the exception of the *.stack* section, the remaining sections contain debugging information. They do not affect the size of the executable image, although they do significantly affect the size of the hello file itself.

The *.stack* section could be used to reserve memory for the program's runtime stack. In **hello**, however, this section is empty--- apparently the program loader or application's startup code allocates its stack in some other way than by having the linker reserve memory for it.

To disassemble the executable contents of the hello file (the .text section), use objdump's `--disassemble` command. The output is long, naturally, so the example pipes the output through the **less** utility:

```
$ arm-elf-objdump --disassemble hello | less
```

Figure 9. Output from objdump --disassemble.

```
hello:     file format elf32-littlearm

Disassembly of section .text:

00008000 <_mainCRTStartup>:
    8000:       e3a00016        mov     r0, #22 ; 0x16
    8004:       e28f10c4        add     r1, pc, #196    ; 0xc4
    8008:       ef123456        swi     0x00123456
    800c:       e59f00bc        ldr     r0, [pc, #bc]   ; 80d0 <_mainCRTStartup+0xd0>
    8010:       e590d008        ldr     sp, [r0, #8]
    8014:       e590a00c        ldr     sl, [r0, #12]
...
```

The output, shown in Figure 9, says that there is a function called `_mainCRTSStartup` located at address 8000H, and its first instruction is `mov r0,#22`, whose raw binary representation is the byte string `E3A00016H`. Trying to be helpful, **objdump** translates the decimal value of the opcode's operand to hex and puts it in a comment after the instruction.

A few instructions later, objdump detects another opcode that may be difficult to decipher in its raw form. The operand for the `ldr` instruction at address `800CH` ends up being the value `80D0H`, which is the

same thing as an offset of D0H bytes from the start of `_mainCRTStartup`. Whether or not this additional information is helpful depends on what the programmer intended this instruction to do.

To see source code intermixed with assembly code, use the `--source` option:

```
$ arm-elf-objdump --source hello | less
```

Figure 10. Output from objdump --source.

```
...

    80d8:       000114fc        streqd  r1, [r1], -ip
    80dc:       00010bc0        andeq   r0, r1, r0, asr #23
    80e0:       000000ff        streqd  r0, [r0], -pc

000080e4 <foo>:
    80e4:       e1a0c00d        mov     ip, sp
    80e8:       e92dd800        stmdb   sp!, {fp, ip, lr, pc}
    80ec:       e24cb004        sub     fp, ip, #4      ; 0x4
    80f0:       e24dd004        sub     sp, sp, #4      ; 0x4

int a_global = 10;

int foo ( int arg )
{
    80f4:       e50b0010        str     r0, [fp, -#16]
  static int foo_counts = 0;

  foo_counts++;
    80f8:       e59f2028        ldr     r2, [pc, #28]   ; 8128 <foo+0x44>
    80fc:       e59f3024        ldr     r3, [pc, #24]   ; 8128 <foo+0x44>
    8100:       e59f2020        ldr     r2, [pc, #20]   ; 8128 <foo+0x44>
    8104:       e5921000        ldr     r1, [r2]
    8108:       e2812001        add     r2, r1, #1      ; 0x1
    810c:       e5832000        str     r2, [r3]
  return foo_counts + arg;
    8110:       e59f3010        ldr     r3, [pc, #10]   ; 8128 <foo+0x44>
    8114:       e5932000        ldr     r2, [r3]
    8118:       e51b1010        ldr     r1, [fp, -#16]
    811c:       e0823001        add     r3, r2, r1
    8120:       e1a00003        mov     r0, r3
    8124:       ea000000        b       812c <foo+0x48>
    8128:       00010cc4        andeq   r0, r1, r4, asr #25
}
    812c:       e91ba800        ldmdb   fp, {fp, sp, pc}
```

```
...
```

You will need to look a few hundred lines into the output, shown in Figure 10, before seeing any source code, because the beginning of the .text section contains instructions from the runtime library's startup code, which is written in assembly language.

For more about **objdump**, use its `--help` option:

```
$ arm-elf-objdump --help

...
arm-elf-objcopy: supported targets: elf32-littlearm elf32-bigarm
   elf32-little elf32-big srec symbolsrec tekhex binary ihex
```

Using objcopy

To translate our ELF hello file into a Motorola S Record file, use **objcopy**:

```
$ arm-elf-objcopy --output-target srec hello hello.srec
```

To see what other formats **objcopy** understands, use its `--help` option:

```
$ arm-elf-objcopy --help
```

Debugging a "Hello, World!" Program

Enough stalling already, let's run some code.

Using the ARM instruction set simulator

If you don't have the Evaluator-7T, or you just don't want to set it up yet, you can use the GNU debugger's ARM instruction set simulator to do all the following without any ARM hardware at all. The simulator is so complete, in fact, that even `printf()` works properly--- the output shows up on the GDB console.

Typical of GNU tools, the debugging procedure is almost exactly the same whether you are using the instruction set simulator or real hardware. I will provide two sets of instructions in places when there is a difference.

Using the ARM Evaluator-7T board

If you intend to use real hardware, unwrap your new Evaluator-7T board and plug it in now. Connect the serial cable to the DEBUG port on the board, and connect the other end to a serial port on your PC. Note the serial port you connect it to, as the debugger will use that port in a moment to control the board. Don't do anything with the CDs included with the Evaluator-7T packaging, they're ARM's proprietary (albeit highly rated) development environment and aren't needed here.

Running gdb

Change to the directory containing the `hello` executable, and use the following command to launch the GNU debugger from the command prompt on your workstation.

```
$ arm-elf-gdb hello
```

If you are using the ARM instruction set simulator, then tell gdb to establish a connection to it:

```
(gdb) target sim
```

If you are using the Evaluator-7T board, tell gdb to establish a connection to it instead. The command used here will vary depending on what kind of host machine you are using, and what serial port you've plugged into. For Cygwin hosts, the commands will look something like this:

```
(gdb) set remotebaud 57600
(gdb) target rdi com1 (or com2, or ...)
```

For a Linux or other unix-like host, the commands will look something like this:

```
(gdb) set remotebaud 57600
(gdb) target rdi /dev/ttyS0 (or /dev/ttyS1, or ...)
```

If you are using a unix host and you get a permission error on the serial port, then you need to change either your userid's group membership, or the permissions on the serial port device node. Contact your system administrator if you have one, or simply log in as the root user and use the **chmod** command:

```
$ su -
<password>
# chmod 777 /dev/ttyS0 /dev/ttyS1
# exit
```

You may see an "unimplemented message" error when gdb establishes the connection to the target board. This is normal--- it just means that the board's firmware has a few low-level commands that gdb doesn't

recognize, or that the target board lacks some commands that gdb understands. During the connectio process, gdb will negotiate with the target board until the two agree on the set of commands that are available.

Once gdb has established a connection with the instruction set simulator or Evaluator board--- referred to uniformly as the *debugging target*--- tell it to download your application to the target in preparation for debugging:

```
(gdb) load
```

Set a breakpoint at `main()`:

```
(gdb) break main
Breakpoint 1 at 0x8140: file hello.c, line 14.
```

Now launch the application. If you are running the instruction set simulator, use the **run** command. If you are using the Evaluator-7T, use the **continue**:

```
(gdb) run
```

or,

```
(gdb) continue
```

The debugger should stop when you reach `main()`:

```
Continuing.

Breakpoint 1, main(), at hello.c:14
14      {
(gdb)
```

This display says that gdb encountered the breakpoint we set, and is currently stopped at source line 14 in `hello.c`. Gdb displays the source line, which only contains a "{".

Type **list**, to list a few lines from `main()`:

```
(gdb) list main
9         foo_counts++;
10        return foo_counts + arg;
11      }
12
13      int main ( void )
14      {
15        int nloops;
```

```
16        char *charstr = "hello, world!\n";
17
18
```

In general, gdb repeats your last command if you just press **ENTER**. So if you press **ENTER** here a few times, listing will continue beyond `main()` if there is anything beyond it to list.

Set a breakpoint at the `printf()` function call on line 20, then continue running the program:

```
(gdb) b 20
(gdb) c
```

When you reach the next breakpoint, examine the value of `nloops`:

```
Breakpoint 2, main() at hello.c:20.
20          printf( "%s", charstr );
(gdb) print nloops
$1 = 0
```

or,

```
(gdb) display nloops
1: nloops = 0
```

The former displays `loops` only once; the latter will display it every time the debugger stops the program when it is in scope. In either case, the value of `nloops` is currently `0`. The `$1` text is a mnemonic that can be used in place of `nloops` to save some typing later, or for scripting:

```
(gdb) print $1
$1 = 0
```

Now, step over and into a few lines of code:

```
(gdb) next
hello, world!
21          foo( nloops );
(gdb) step
foo (arg=35704) at hello.c:6
```

The output of the **next** command is the result of the `printf()` function call, which we did not step into. The **step** command steps into the `foo()` function call, reports the arguments to `foo()`, and reports its location in the source code.

Remember that if you just press **ENTER**, gdb will repeat your previous command. So if you want to step over several lines, you can just do this:

```
(gdb) step
(gdb) <ENTER>
(gdb) <ENTER>
```

To see a disassembly of a function, use the **disassemble** command:

```
(gdb) disassemble main
0x80e4 <main>:  mov     r12, sp
0x80e8 <main+4>:        stmdb   sp!, {r11, r12, lr, pc}
0x80ec <main+8>:        sub     r11, r12, #4    ; 0x4
0x80f0 <main+12>:       sub     sp, sp, #8      ; 0x8
0x80f4 <main+16>:       bl      0x8214 <__gccmain>
0x80f8 <main+20>:       ldr     r3, [pc, #3c]   ; 0x813c <main+88>
0x80fc <main+24>:       str     r3, [r11, -#20]
0x8100 <main+28>:       mov     r3, #0  ; 0x0
0x8104 <main+32>:       str     r3, [r11, -#16]
0x8108 <main+36>:       ldr     r3, [r11, -#16]
0x810c <main+40>:       cmp     r3, #9  ; 0x9
0x8110 <main+44>:       ble     0x8118 <main+52>
0x8114 <main+48>:       b       0x8134 <main+80>
0x8118 <main+52>:       ldr     r0, [pc, #20]   ; 0x8140 <main+92>
0x811c <main+56>:       ldr     r1, [r11, -#20]
```

To disassemble just the instruction the processor is about to execute, use the *examine memory* **(x)** command, with the *instruction* `(/i)` option:

```
(gdb) x/i $pc
0x8178 <main+72>:       ldr   r3, [r11,  -#16]
```

To step one processor instruction, use the **stepi** command. In situations where multiple processor instructions are associated with a single C statement, the program counter value shown at the far left of the output changes even though the source line displayed stays the same.

```
(gdb) stepi
0x817c  19   for( nloops = 0; nloops < 10; nloops++ ) {
(gdb) <ENTER>
0x8180  19   for( nloops = 0; nloops < 10; nloops++ ) {
```

When combined with the **display** command, you can use the **stepi** command to watch the processor step through assembly code:

```
(gdb) display/i $pc
```

```
2: x/i $pc  0x817c <main+76>: add r2, r3, #1 ; 0x1
(gdb) stepi
0x8180 19   for( nloops = 0; nloops < 10; nloops++ ) {
2: x/i $pc  0x8180 <main+80>: str r2, [r11, -#16]
(gdb) <ENTER>
2: x/i $pc  0x8184 <main+84>: b 0x8154 <main+36>
```

To see the value of a particular register, use the **print** command. To prevent clashes with similarly-named C objects, prefix register names with `$`.

```
(gdb) print $r3
```

To see information on *all* the registers, use the **info** command:

```
(gdb) info registers
```

To see the value that a call to `foo()` would return, simply invoke `foo()` from the debugger. The called function is actually run by the debugging target, so any side-effects produced will remain:

```
(gdb) print foo(10)
$4 = 12
(gdb) print foo(10)
$4 = 13
```

If you get lost in your application, and want to find out where you are, use the **backtrace** command:

```
(gdb) backtrace
#0 foo (arg=35704) at hello.c:6
#1 0x8178 in main() at hello.c:21
```

To find out about any commands you may have missed, use the **help** command:

```
(gdb) help
```

If you can only remember part of a command, or just the description of the command, use the **apropos** command to search for it:

```
(gdb) apropos print
```

Finally, when you are done, quit.

```
(gdb) quit
```

You will usually need to reset the Evaluator-7T board before starting another debugging session.

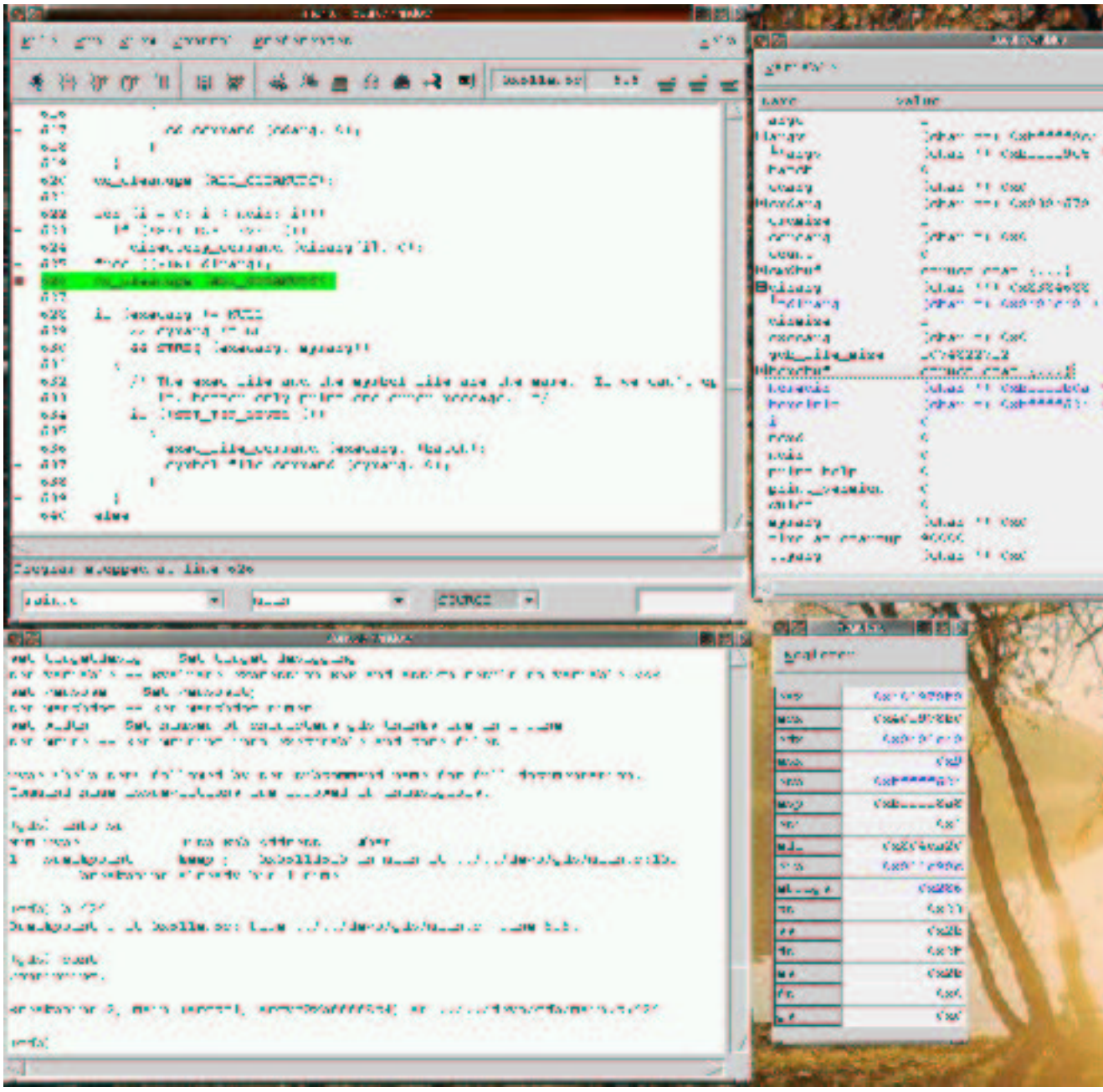
Graphical Debugging

By now you may have concluded that although gdb is functional, it isn't very pretty. And trying to debug code that interacts with complicated data structures using a command-line interface can be tedious at best. Is this the best that GNU can do?

Of course it isn't. Like many GNU tools, gdb is a command line application because it is used as a foundation for other applications, including several graphical debugging interfaces. All the point-and-click you could ever want in a debugger is available, and from a variety of sources.

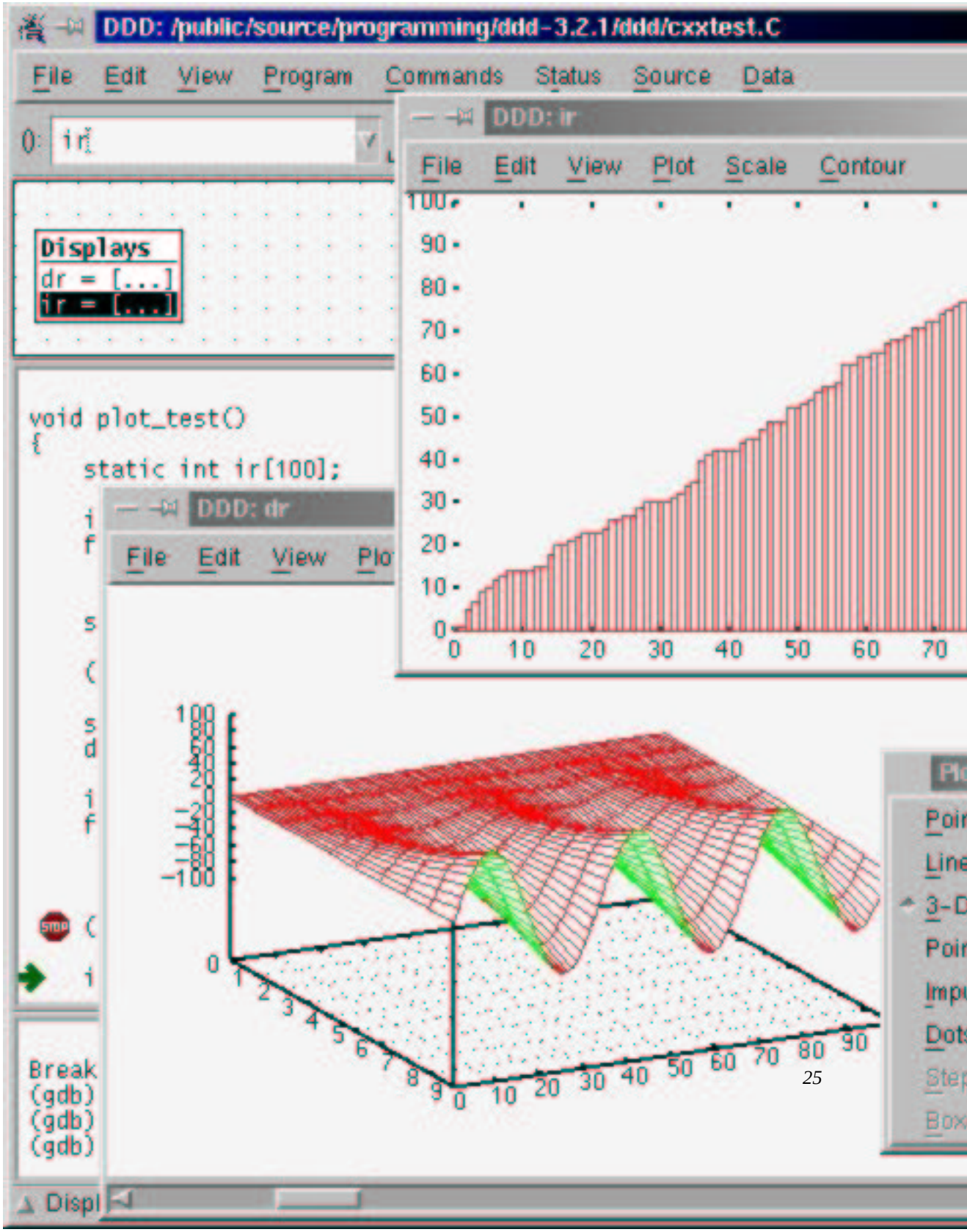
Screenshots for two popular graphical debugging frontends to gdb are shown in Figure 11 and Figure 12. The first is from a program called Insight (http://sources.redhat.com/insight), an enhanced gdb that can run on both Win32 (Microsoft) and X Windows (everything else) workstations. See the resources section for more information about Insight.

Figure 11. The Insight debugger.



The second screenshot is from an application called DDD (http://www.gnu.org/software/ddd/ddd.html), which uses a command-line gdb running in the background to do its work. This may sound clumsy, but it has tremendous benefit: the same binary DDD application works for all versions of the GNU debugger, regardless of the target chosen. In contrast, Insight must be configured and installed for each version of the debugger that you intend to use.

Figure 12. The Data Display Debugger.



DDD gets its graphics by way of X Windows, which means that it doesn't work on Win32 hosts without extra software. See the resources section for more information about DDD.

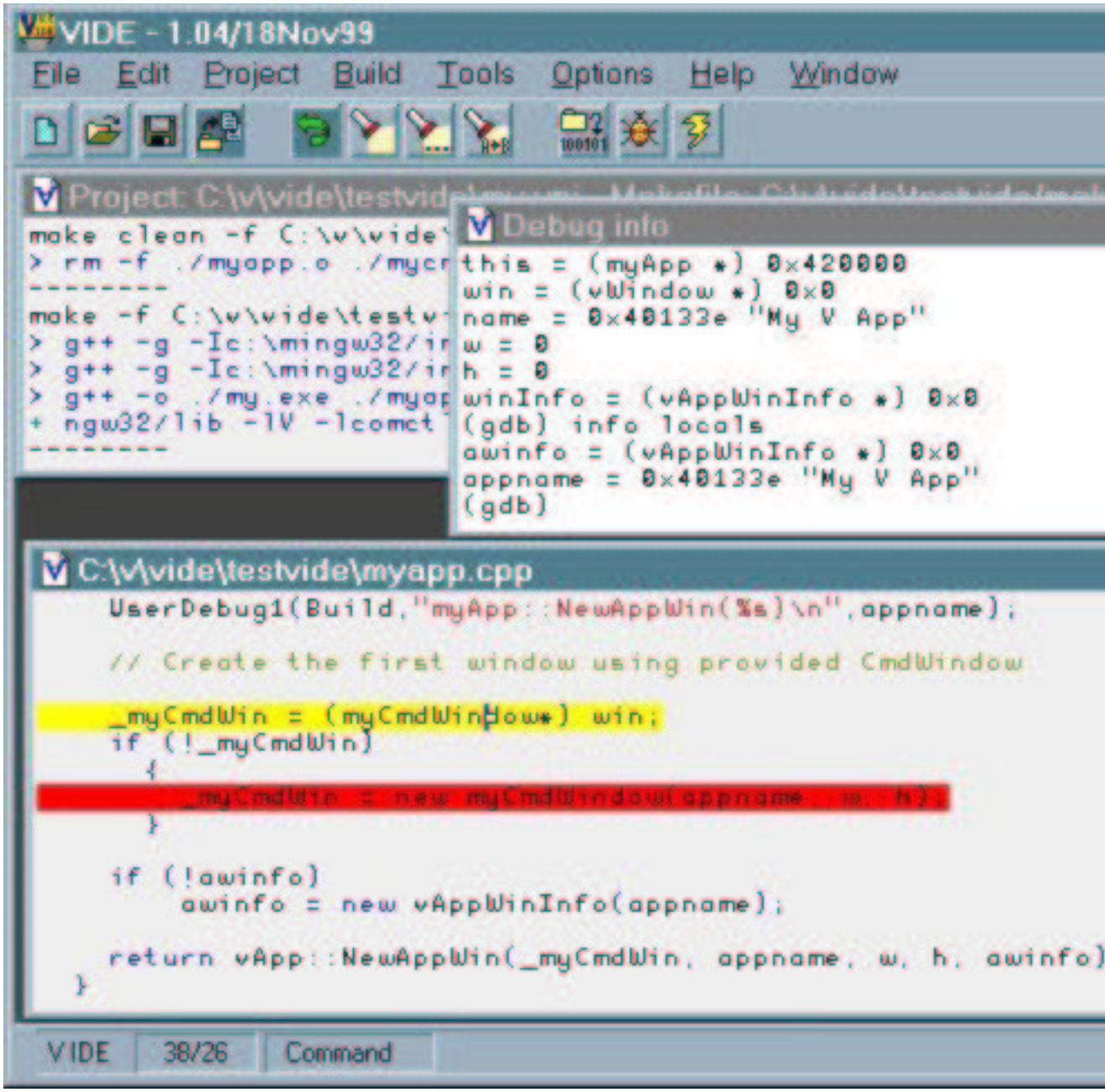
The approaches to graphical debugging taken by Insight and DDD each have their advantages and disadvantages, and you will probably find that you routinely move between all three options (Insight, DDD, and the plain-vanilla gdb) depending on what you are trying to accomplish at the time.

What about an IDE?

There are several freely-available packages for complete point-and-click development environments. Like the strategy used to enhance gdb with graphical capabilities, all of these options use the standard GNU compiler and other tools behind the scenes--- which is why it is important to know how to set the basic tools up first.

One option is called VIDE (http://www.objectcentral.com/vide.htm). This package contains features you would expect in an integrated development environment, including a graphical editor, project files, and build management. A screenshot of VIDE is shown in Figure 13.

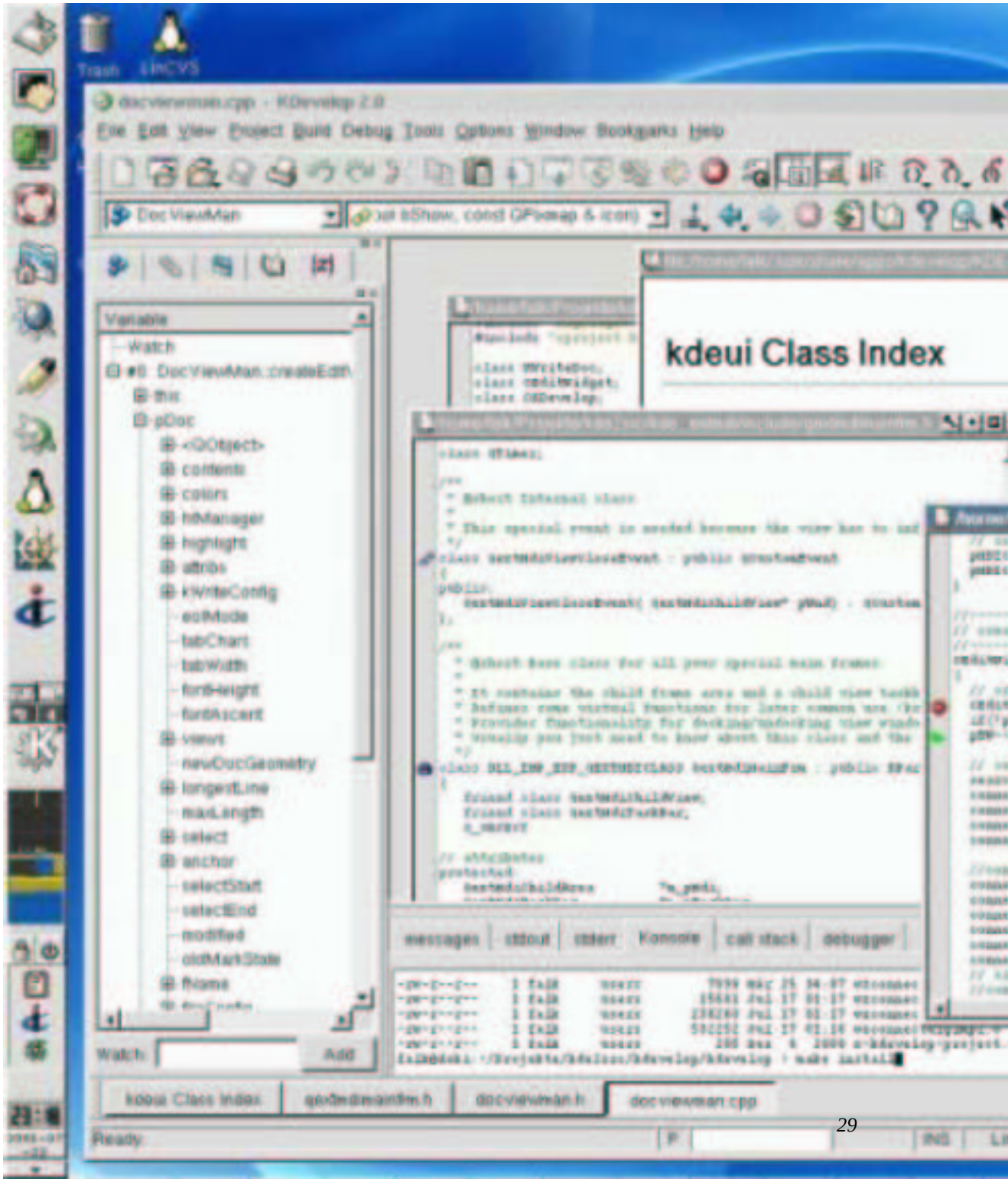
Figure 13. Screenshot of VIDE



Another option, albeit one that takes some work to set up for embedded use, is KDevelop (http://kdevelop.kde.org). This environment is currently in widespread use on Linux hosts to write and maintain applications for the KDE (http://www.kde.org) window manager, and is an extremely powerful and full-featured IDE for producing native applications. Because it uses GNU tools behind the scenes, however, it can be retasked for embedded duty by tweaking some configuration files.

A screenshot of KDevelop in action appears in Figure 14.

Figure 14. Screenshot of KDevelop



What Next?

There are several subjects you will want to learn more about after you have been introduced to GNU tools. Four of the important ones are the compiler's *inline assembly language* syntax, the linker's *command file* syntax, how to enhance the C runtime library's *support for functions like malloc() and printf()*, and how to construct a *debugging agent* that lets gdb talk to your own hardware. We'll hit the high points here, you can refer to the resources listed at the end of this article for more in-depth information.

Inline assembly language

To jam an assembly language instruction into the middle of a program flow, the GNU syntax looks like this:

```
for( x = 0; x < 10; x++ ) {
   foo();
   ___asm__("nop");
}
```

Inline assembly language has the risk of corrupting the program environment if register values are changed in unexpected ways. To avoid this danger, the GNU compiler provides an extended inline assembly language syntax that includes *operand constraints* that tell the compiler both how the instruction is using registers, and how those registers can be mapped to objects in C:

```
 int one;
 __asm__( "mov %0, #1", "=r" (one));
```

This instruction tells gcc to move the immediate value #1 into the register occupied by the integer variable `one`. If `one` isn't actually contained in a register (i.e. maybe it is actually found on the stack), then gcc will insert additional instructions before and after the inlined opcode to get the value #1 into whatever representation the compiler uses for `one`.

Gcc's online documentation tells more about how to use operand constraints in inline assembly language constructs.

Linker command syntax

The essence of a typical linker command file looks something like the text in Figure 15. This excerpt starts loading the contents of the .text memory section at address 8000H, and includes in that section all the contents of the .text, .text.*, .stub and other sections from the application's object files. Any empty spaces in the section created by alignment adjustments are filled with zeros.

Figure 15. An excerpt from a linker command file.

```
...

SECTIONS
{
  . = 0x8000;
 .text      :
  {
    *(.text)
    *(.text.*)
    *(.stub)
    *(.gnu.warning)
    *(.gnu.linkonce.t.*)
    *(.glue_7t) *(.glue_7)
  } =0

  PROVIDE (__etext = .);
  PROVIDE (_etext = .);

...
```

The **PROVIDE** statement creates symbols that programs can use to find out about the application's memory layout at runtime. In the example, the symbols `__etext` and `_etext` are allocated at the end of the .text section.

This excerpt illustrates only a fraction of the GNU linker's capabilities. See its extensive online help documentation for more information.

Newlib stubs

The C runtime library we are using does not come equipped to talk to the Evaluator-7T's hardware. To add this functionality, newlib provides hooks that your application can connect to in order to catch the text coming out of a `printf()` call, detect math errors, and implement filesystem-like behavior for functions like `fopen()`.

There are seventeen such hooks in newlib, with names like `_write_r` and `_sbrk_r`. By supplying reentrant functions with these names in the application, you can more completely integrate newlib's capabilities into an embedded system.

Debugging stubs

Gdb works fine when the hardware is as ready-to-play as the Evaluator-7T. But what if you want to run a debugging session on custom hardware? To do this, you need a *debugging agent* in the target system that knows how to communicate with the debugger running on your workstation. In the GNU debugger world, this debugging agent is often called a *debugging stub*.

A few example stubs are included with the GNU debugger's source code, in files with names ending in "-stub". A more extensive and extensible list of stubs is housed in the gdbstubs (http://www.sourceforge.net/projects/gdbstubs) project. By modifying these stubs to suit your hardware, you enable gdb to talk to your application as it runs in your system.

Commercial GNU Offerings

I mentioned earlier that a handful of companies offer preconfigured and ready-to-go versions of GNU tools for all kinds of embedded processors. The following mentions two of them. Note that I am not an employee of either of these companies, and I hold no financial interests in them. Furthermore, the following is not a comprehensive list: I only describe products that I have current, favorable experience with.

GNUPro

At the high end, you can't get much better than Red Hat, Inc.'s (http://www.redhat.com) GNUPro (http://www.redhat.com/products/tools/) distribution. This product contains the latest-and-greatest versions of everything GNU has to offer, thoroughly tested and maintained by the company that employs more GNU developers (embedded and otherwise) than anyone else in the world. And because Red Hat has immediate access to such a large mindshare of the GNU community, they can--- and do--- offer a level of support that you simply cannot find anywhere else.

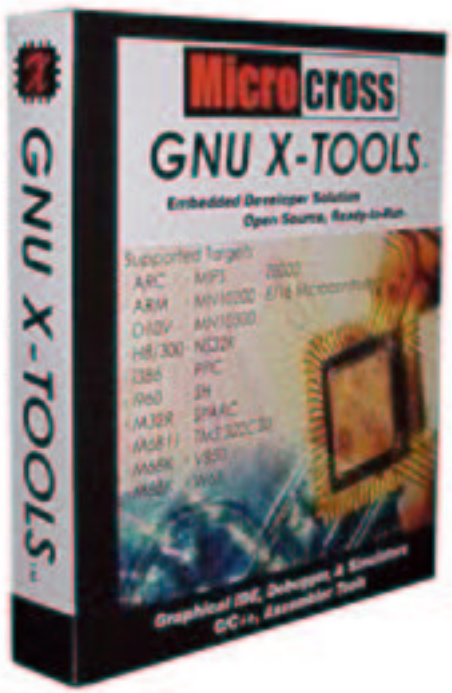
Figure 16. Red Hat, Inc.'s GNUPro GNU Distribution.

But this level of attention comes at a cost. A GNUPro distribution starts at about $20k USD, including unlimited support for up to ten users. For organizations contemplating aggressive customizations of GNU tools, this is a reasonable and pragmatic sum of money; for someone who just wants an introduction to GNU or who only has basic needs, however, the value package is a bit overkill.

X-Tools

Fortunately, there is an option on the low end. A recent, and somewhat unique offering is from a company called Microcross, Inc. (http://www.microcross.com) who, for about $499 USD, markets a single CD containing GNU tools for almost every target that GNU supports. In addition, they supply an informative and well-written manual, a graphical installer, PDF versions of all available GNU documentation, and a copy of the VIDE integrated development environment. To keep costs low, support and other services are sold seperately--- you don't pay for what you don't want.

Figure 17. Microcross, Inc.'s X-Tools GNU Distribution.



The Microcross CD, called GNU X-Tools (http://www.microcross.com/Products/products.html), makes sense for someone who wants an introductory look at using GNU tools for embedded work and doesn't want to get mired in the details of setting up the tools themselves. X-Tools is also a good choice for someone who knows they will be working with multiple target architectures, or someone who doesn't need any service or support now but thinks that they may at some point in the future. Finally, X-Tools is especially appropriate in situations where the development environment *must* be a Win32 host: X-Tools comes in both Windows and Linux flavors, which lets you avoid the pain of building GNU tools on suboptimal hosts like Windows 98.

Resources

- http://www.billgatliff.com

Additional information on GNU programming for embedded systems.

- http://sources.redhat.com/ml/crossgcc
  The CrossGCC mailing list archives.

- http://www.arm.com
  Information on the ARM7T microprocessor, and the Evaluator 7T single-board computer.

- http://sources.redhat.com/cygwin
  The Cygwin unix emulation package.

- http://sources.redhat.com/binutils
  The Binutils homepage.

- http://gcc.gnu.org
  The GNU Compiler Collection homepage.

- http://sources.redhat.com/gdb
  The GNU debugger homepage.

- http://sources.redhat.com/newlib
  The Newlib homepage.

- http://sources.redhat.com/insight
  The Insight homepage.

- http://www.gnu.org/software/ddd
  The DDD homepage.

- http://www.objectcentral.com/

  The developers of VIDE.

- http://kdevelop.kde.org

  The KDevelop homepage.

- http://www.microcross.com

  The Microcross, Inc. website; they make X-Tools.

- http://www.microcross.com/Support/Docs/docs.html

  Microcross's GNU documentation tree.

- http://www.redhat.com

  Red Hat, Inc.'s website; they make GNUPro.

- http://www.gnu.org

  The Free Software Foundation's GNU website.

- http://sourceforge.net/projects/gdbstubs

  Homepage for the gdbstubs project.

About the Author

Bill Gatliff is an independent consultant with almost ten years of embedded development and training experience. He specializes GNU-based embedded development, and in using and adapting GNU tools to meet the needs of difficult development problems. He welcomes the opportunity to participate in projects of all types.

Bill is a Contributing Editor for Embedded Systems Programming Magazine (http://www.embedded.com/), a member of the Advisory Panel for the Embedded Systems Conference

(http://www.esconline.com/), maintainer of the Crossgcc FAQ, creator of the gdbstubs (http://sourceforge.net/projects/gdbstubs) project, and a noted author and speaker.

Bill welcomes feedback and suggestions. Contact information is on his website, at http://www.billgatliff.com.

Notes

1. We aren't downloading RedHat products here, although what we are downloading goes into several RedHat products. RedHat provides a valuable service to the GNU community by hosting the web and FTP sites for many GNU tools.