CHAPTER 8

Cross-Platform and Embedded Systems Development

As you have already learned in Chapter 3, development systems consist of many tools. The discussion in Chapter 3 was about native software development, which means that the host system on which the software is developed and the target system on which the software runs are the same. This chapter provides information about embedded and cross-platform software development where the host system and the target system are different. The issues in this type of development are slightly different although basic programming methods and techniques remain the same.

In this chapter I have used a single board computer from Arcom (www.arcomcontrols.com) and a processor mezzanine card from Artesyn (www.artesyn.com). I am thankful to both of these vendors. The Arcom board is based upon a x86 compatible CPU and has four serial ports, an Ethernet port and a number of other ports. The Artesyn board is PowerPC based and it can be plugged into a Compact PCI career card. Information about these boards can be obtained from the respective company web site. I have listed some features at the end of this chapter.

Before reading this chapter, you should become familiar with a few concepts related to cross-platform development. The most important terms used in this chapter are *cross* and *native*. The term cross is used for tools that run on one type of machine but are used to carry out some service for another type of machine. For example, a cross compiler is a compiler that creates executable code for a machine which is different than the machine on which the compiler is running. Similar is the case with cross assemblers, cross debuggers and so on. On the other hand, a native tool provides a service to the same machine on which it is running. For example, a native compiler generates output code that is executable on the same machine on which the compiler runs.

8.1 Introduction to the Cross-Platform Development Process

You need a general-purpose computer for software development. This computer hosts development tools. In most cases, the computer also hosts some debugging tools. The executable code may be generated for the same computer on which you develop software or for some other machine that is not capable of running development tools (editors, compilers, assemblers, debuggers etc.). Most of the time host and target machines differ so that programs built on the host machine can't be run on the target machine. These differences may take many forms. Some common differences between host and target machines are different operating system, different system boards or a different CPU.

It's necessary to understand some basic concepts and terminology related to cross-platform development. These basic concepts are explained in the following subsections.

8.1.1 Host Machine

The host machine is the machine on which you write and compile programs. All of the development tools are installed on this machine. Most of the time this is your PC or workstation. Your compiler is built to run on this machine. When you are satisfied with your code, the executables are built and transferred to the target machine. If the host and target machines are different, the final executable can't be executed on the host machine where you built it.

8.1.2 Target Machine

A target machine may be another general-purpose computer, a special-purpose device employing a single-board computer or any other intelligent device. Usually the target machine is not able to host all the development tools. That is why a host computer is used to complete development work.

Debugging is an important issue in cross-platform development. Since you are usually not able to execute the binary files on the host machine, they must be run on the target machine. The debugger, which is running on the host machine, has to talk to the program running on the target machine. GNU debugger is capable of connecting to remote processes. Some CPUs have a JTAG or BDM connector, which can also be used to connect to processes on remote systems.

8.1.3 Native and Cross Compilers

A native compiler generates code for the same machine on which it is running. A cross compiler generates code for a machine different from the one on which it is running.

8.1.4 Cross Platform Development Cycle

The cross-platform development cycle consists of at least four stages. These stages are:

- Writing code
- Cross compilation
- Transferring binaries to the target
- Debugging

The steps may be different in some cases depending upon the type of product you are building. Figure 8-1 shows these processes and machines on which a particular process is carried out.

Let us take a closer look on these steps.

8.1.4.1 Writing Code

When you write code for cross compilation, you have to keep few things in mind. The CPU architecture of the target machine may be different from that of the host machine. This means that you have to consider word lengths, byte order and so on. For example, length of integer may be different on the target machine. Similarly, organization of memory may play an important role. As an example, the organization of bytes inside a 32-bit integer is different in Intel CPUs as compared to Motorola CPUs. This is a typical network-byte-order problem and must be considered when developing network protocols.

Some libraries' routines may be different for cross compilers than for native compilers.

8.1.4.2 Building on the Host Machine

After writing the code, you need to compile it on the host machine. In addition to a cross compiler and cross assembler that can generate code for the target machine, you need libraries that are linked to your code. These libraries are usually specific to the target machine and are cross-compiled.

If you are writing code according to some standard, your porting process is usually very quick. For example, if you have written libraries strictly using the ANSI standard, it will be relatively easy for you to port these libraries from one platform to another. In most large software development projects, it is important that you stick to some standard. This is because ultimately you may wish to port and sell the product for many platforms.

You have learned the difference between static and dynamic linking earlier in this book. Usually in cross compiling, you have to build the executable code with static linking because you may not have shared libraries available on the target system. This is especially true for embedded systems where you usually don't have a large secondary storage device.

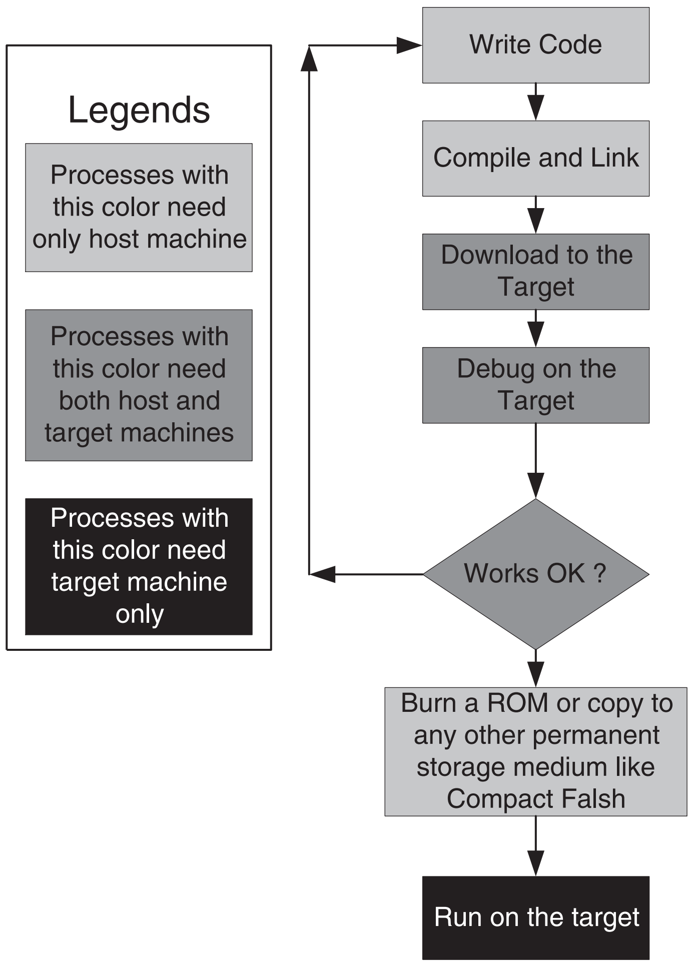
Legends
Processes with this color need only host machine
Processes with this color need both host and target machines
Processes with this color need target machine only
Write Code
Compile and Link
Download to the Target
Debug on the Target
Works OK ?
Burn a ROM or copy to any other permanent storage medium like Compact Falsh
Run on the target

Figure 8-1 Flow chart of cross-platform development.

During the development phase, most of the time you compile code with debug information enabled. The size of the output files is usually quite large but this is acceptable. A different approach is to create multiple files, one built without debug information or stripped and at least one other file containing all symbols information. You can load the latter file into the debugger before connecting the debugger to the executable.

8.1.4.3 Transfer to the Target Machine

After building executable code for a cross-platform development project, you have to transfer it to the target machine. There are several ways to do so depending upon the target machine. If the target machine has a JTAG interface, you can use that interface to transfer the file and then use the same interface to debug it. If the target does not have a JTAG interface, you can transfer using a serial or Ethernet link. Sometimes the target machine is already running an operating system and you have to transfer an application to the target machine. In some other circumstances you may have to build and transfer a single packaged file that contains the operating system as well as applications. If the target system is already running the operating system, you can transfer you application using a network interface very easily. If you have a single packaged file, you can have a boot routine on the target machine that may download this file from a TFTP server as well.

The bottom line is that there are many methods used by developers depending upon the nature of the files to be transferred, availability of special hardware on the target machine, availability of boot routines on the target machine and the operating system used. You can select a method that best suits your needs.

8.1.4.4 Remote Debugging

In the case of cross-platform development and embedded systems, you have to debug your code remotely. There are multiple reasons for that including:

- The target machine is different from the host machine so that you can’t execute the code on your host machine.
- The target system is connected to some special hardware and the software interacts with that special hardware. To debug in a real-world situation, you have to debug it remotely.
- The target system itself has some special hardware that is different from the host machine. For example, the target system may have the same CPU as the host machine but has different hardware interrupts.

There may be other reasons in addition to the ones mentioned here. In this chapter you will learn how to attach to a process running on a target machine using GNU debugger. You will use a local file that contains all of the symbol table information. Before you connect to the target machine, you will load this symbol table information into `gdb` and then start a debug session remotely on the target machine.

8.2 What are Embedded Systems?

There are many definitions of embedded system but all of these can be combined into a single concept. An embedded system is a special-purpose computer system that is used for a particular task. The special computer system is usually less powerful than general-purpose systems, although some exceptions do exist where embedded systems are very powerful and complicated. Usually a low power consumption CPU with a limited amount of memory is used in embedded systems. Many embedded systems use very small operating systems; most of these provide very limited operating system capabilities. However as memory and CPU power is becoming cheap, many modern operating systems are also being used in embedded systems these days. Embedded Linux products are becoming more and more popular.

Since embedded systems are used for special purpose tasks, there is usually no need to add new applications to the system once it is built. Therefore programs and operating systems are usually loaded only once into read-only memory. The read-only memory is available in many forms these days and includes some type of ROM, flash chips and flash cards. In case an upgrade is required, a new ROM chip replaces the old one.

If you look at your surroundings, you will find tiny computers everywhere. They are present even in places where you rarely notice them. Some of the most common embedded systems used in everyday life are:

- Refrigerators
- Microwave ovens
- TV, VCR, DVD players
- Cameras
- Cars (Antilock brakes, engine control, sun roof, climate control, automatic transmission and so on)
- Personal digital assistants
- Printers

Complicated and more sophisticated embedded systems include routers and switches where high performance CPUs are used.

8.2.1 Embedded Systems and Moving Parts

Most of the embedded systems are supposed to be used for a long period of time and without any maintenance. This is the very nature of embedded systems applications, to manufacture a system and then leave it to run independently for its intended life. For this reason, most embedded systems usually don't have moving components because any moving or mechanical part experiences wear and tear and needs replacement periodically. Typical examples of moving components include fans, disk drives and so on. Fans are used for cooling in power supplies and CPU heat sinks in desktop computers. Disk drives (both floppy and hard disk drives) are used as a storage medium for operating systems, applications and data.

In embedded systems, ROM is used for storage of operating system and application data to eliminate the need for disk drives. These days, flash cards are also common and you can update a system just by replacing the flash memory card. In embedded systems you have to use special types of power supplies that don't need any fan for cooling. Many different types of power supplies are available on the market that don't heat up when used continuously for long periods of time.

8.2.2 Embedded Systems and Power Consumption

Power consumption is also an important issue in embedded systems. This is also related to the moving parts issue in a sense. Components that consume more power need cooling, so some type of fan must be attached to the system. For this reason people always prefer CPUs and other components that use less power and generate less heat. It is also beneficial for systems that run on batteries to use less power. High power-consuming components drain batteries quickly.

8.2.3 Embedded Operating Systems

Since embedded systems are usually not general-purpose systems, they need not be as sophisticated as commercial general-purpose systems. These operating systems are smaller in size. Most of these operating systems also offer real-time capabilities, which are missing in general-purpose systems. Although there are hundreds of operating systems available for the embedded market, some of the most commonly used ones are:

- VxWorks
- pSOS
- Embedded Linux
- QNX
- Windows CE

When you use Linux as an embedded operating system, you can take out many parts of the kernel that are required in the general-purpose Linux systems. Which parts should be excluded from the kernel depends upon the requirements of a particular system. For example, you can take out the entire networking code if the embedded system is not going to be networked. Similarly, you can take out most of the secondary storage related drivers and file systems support. Kernel parts that are the most likely candidates for removal from the embedded Linux are:

- Disk drivers
- CD-ROM drivers
- Most of the networking code. Even if an embedded system is intended to be networked, you may not need all of the routing code and drivers for all types of network adapters.
- Sound and multimedia related drivers and components. However you may need these components if the embedded system is used as a computer game.
- Most of the file system support
- Any other thing that is not required. You can go through the kernel configuration process to determine what is not necessary.

The trimmed version of the Linux kernel will be much smaller compared to the kernels you find in most Linux distributions.

8.2.4 Software Applications for Embedded Systems

Embedded applications are tightly integrated with each other and often with the operating system as well. In many cases you will find only one big file that is loaded by the system at the time you power it up that contains all of the operating and applications data. Since resources are sparse on embedded systems, applications are usually highly optimized to take the least amount of memory and storage space. Embedded systems programmers don't have the luxury of even one extra line of code. In addition, embedded software needs to be solid because there is nobody to restart a crashed application. Watchdog timers are used to recover from any system lock.

8.3 How Development Systems Differ for Embedded Systems

Development for embedded systems is different from common practices in many ways. For new developers in the embedded systems world, there is a learning curve to understand which conventional practices are no longer valid in this new environment. To be an embedded systems developer, you need to know many things about the hardware on which your software will be executed. Often embedded systems are connected to some sort of control system (wheel rotation in an automobile anti-lock braking system, for example) and the developer also needs knowledge of that system as well. If the CPU and/or the operating system are different on the target embedded platform, you have to do cross-platform development which has its own issues. There are different testing techniques as well because in most of the embedded systems you don't have a monitor screen where you can display error messages or test results. All of these issues make embedded systems development much more complicated than writing a program on a UNIX machine and then executing it.

This section provides information on some of these issues but should not be considered comprehensive. This is just a starting point to give you an idea that you should expect new challenges when you start embedded systems programming.

8.3.1 Knowledge of Target System Hardware

The embedded system is usually different from the system on which you develop software (the host machine). For example, usually the target computer has many input and output lines, different interrupts and probably a different type of CPU. It may also have memory constraints because these systems usually have a low amount of RAM and no virtual memory because of the absence of secondary storage media. To develop software for the target, you need to know all of these things.

As mentioned earlier, most of the embedded systems are connected to some other sort of hardware. The main job of these systems is to control the connected hardware or systems using some policy. As an embedded system programmer, if you don't have knowledge of these additional system or hardware, you will not be able to program the embedded system well. Usually embedded systems have some sort of digital and analog I/O connected to this external hardware where you have to exchange data.

8.3.2 Is the Target System Real-Time?

If the target system is running a real-time operating system, you have to keep in mind many other issues as well. Usually these issues don't come into the picture when you are developing code for general-purpose machines. System scheduling is important and you need to know what type of scheduling is available on the target system. Almost all real-time systems have priority-based scheduling. However, if two processes have the same priority, then scheduling may be different depending upon the operating system. Some systems implement round-robin scheduling for equal priority tasks. On the other hand some systems run a task to completion. You need to know what happens when a process or task is scheduled on a particular real-time scheduler. Time slices for running equal priority processes on round-robin schedulers can also be changed on some real-time schedulers.

To complicate the situation, some systems also allow processes to increase or decrease their priority levels at run time.

Interrupts and their service routine are usually critical in real-time systems. If your embedded target platform is real-time, you should keep in mind limitations in this area and tolerance of critical tasks in constrained environment. You should not miss a critical event in a real-time system because that may result in disaster. If a packet is dropped in a router, it may be tolerated, but if a robot reacts to some event late, it may result in major loss. Even dropped packets aren't particularly acceptable in modern routers.

8.3.3 Testing Methodology

The testing methodology is also different in the case of embedded systems. In typical cases, an embedded system is connected to some other device or system for control purposes. These connections are not available on the host system where you are developing the software. This is true even if the operating system and CPU on the host and target systems are the same. For example, a Linux-based embedded system running an x86 family of processors may be controlling some security system. It is difficult to simulate that security system on a PC and test your software. So even if the platform is the same, you have to test and debug your programs on the target system.

In some cases testing of embedded systems may be quite complicated as you need to get results of the functionality testing sent or stored somewhere. In small embedded systems, not connected to a network or a user-friendly output device, getting these results is difficult. In such circumstances you may need to build special testing techniques or special code in your embedded system to produce test results. I would recommend a thorough knowledge of how to write testable code.

8.4 Cross Compilations

Cross compilation tools are very important for successful product development. Selection of these tools should be made based upon the embedded system itself as well as features to test and debug software remotely. Since you are developing software on a Linux platform, your cross-platform development tools should be compatible with Linux as a host machine. Depending upon CPU family used for the target system, your toolset must be capable of generating code for the target machine. In the case of GNU development tools, you need to have a number of things to work together to generate executable code for the target. For example, you must have at least the following available on your Linux machine.

- Cross compiler
- Cross assembler
- Cross linker
- Cross debugger
- Cross-compiled libraries for the target host.
- Operating system-dependent libraries and header files for the target system.

These components will enable you to compile, link and debug code for the target environment. In this section, you will learn how to develop a cross-compilation environment on your Linux machine.

You have already built a native GCC compiler in Chapter 3. Conceptually, building a cross compiler is not much different from building a native compiler. However there are additional things that you should keep in mind while building a cross compiler. The most significant of these is the command line listing of the target when running the configure script. The whole process may be slightly different depending upon targets. The best resource to find out detailed information about how to build a cross compiler is the CrossGCC FAQ and mailing list. The FAQ is found at http://www.objsw.com/CrossGCC/ and using information in this FAQ, you can subscribe to and search the CrossGCC mailing list. I would suggest getting a cross compiler from a vendor instead of building it yourself. Many companies, including RedHat (http://www.redhat.com) and Monta Vista Inc. (http://www.mvista.com), provide development tools for embedded Linux development of different CPU families.

8.4.0.1 Cross Debugging

There are multiple methods of debugging that you can use in the cross-platform development environment. Most commonly you can use emulators, JTAG, and remote debugging over serial or network links. Some of these methods are discussed in this section.

8.4.1 Software Emulators

Software emulators are software tools that can emulate a particular CPU. Using a software emulator you can debug through your code and find out CPU register values, stack pointers and other information without having a real CPU. Software emulators are useful when you don’t have the real hardware available for testing and debugging and want to see how the CPU will behave when a program is run on it. Keep in mind that these are not very useful tools to simulate real hardware because there are many other hardware components that are built onto the target machine.

8.4.2 In-circuit emulators

In-circuit emulators are devices that are used to emulate a CPU. You can plug in this device in place of a CPU on your circuit board. The device has the same number of pins and the same package type as the CPU and fits into the CPU socket. The other side is connected to a PC or a workstation where a debugger is running. Using an in-circuit emulator, you can start and stop execution of a program as you wish. This is in contrast to a software emulator, where the CPU emulator is a program running on your machine, isolated from the circuit board on which you want to test your software. An in-circuit emulator provides you access to real hardware for testing purpose.

Usual arrangement of in-circuit emulator is shown in Figure 8-2. The emulator is plugged into the actual embedded system on one side and connected to your PC on the other side.

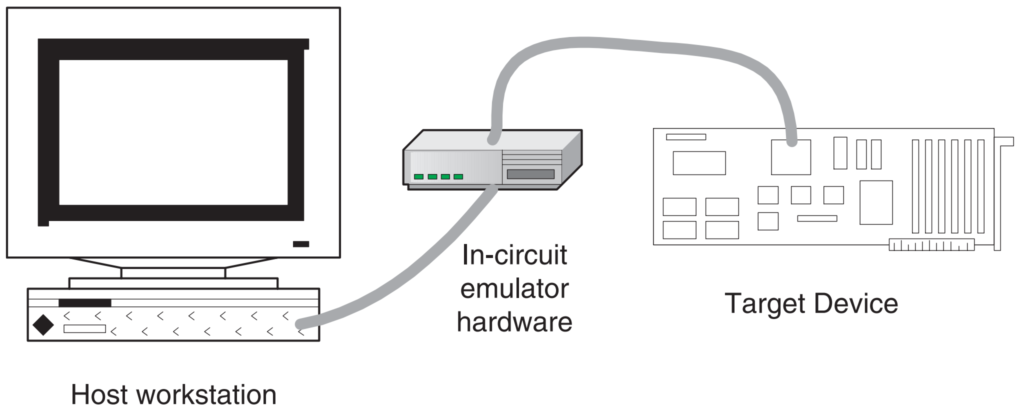

Figure 8-2 In-circuit emulator connection diagram.

8.4.3 Introduction to JTAG and BDM

JTAG or Joint Test Action Group is a method of accessing memory and CPU resources without having an application running on the target. BDM or background debugging mode also achieves this objective. Many CPUs provide JTAG or BDM port/connection to connect a serial or parallel port on your host to the target CPU board using a cable. There are a few open source projects that give you information, utilities and hardware diagrams about the procedure.

Using JTAG or BDM, you don’t need an in-circuit emulator and a debugger running on your host machine can connect to a target board. But keep in mind that not all CPUs provide this facility. Most Motorola controllers and PowerPC CPUs have this interface.

8.4.3.1 Building Cross Debugger

Building a cross debugger is similar to the building process of a native debugger. When you configure `gdb`, you need to mention the target CPU on the command line of the configure script. You also need the required libraries for the target.

8.5 Connecting to Target

Using `gdb`, there are two basic methods of connecting to the target machine when you debug a program. These are:

- Using `gdbserver` program
- Using a stub

In this section we shall discuss the `gdbserver` method in detail because it is the same for all platforms. The stub method differs from platform to platform and you need to create stub file(s) for your target and tailor these into your code.

8.5.1 Using `gdbserver` with GNU Debugger

The `gdbserver` utility comes with GNU debugger. I have used `gdb` version 5.1.1 and you can compile the `gdbserver` after you compile your debugger. If the host machine is different from the target machine, you must cross-compile the `gdbserver` program. This utility can run a program on a remote target and can communicate to `gdb` running on host.

When you build `gdb` from the source code, the `gdbserver` is not compiled during this process. This is what happened when I compiled `gdb` version 5.1.1 on my RedHat 7.1 machine. You have to compile it separately by going into the `gdb/gdbserver` directory. You can run the following two commands to build and install it:

```
make
make install
```

You may choose to build a static binary of the `gdbserver` program if you don't have dynamic libraries installed on your target board. For the purpose of this book, I used the Arcom board, which is introduced later in this chapter, and which has dynamic libraries installed. After compiling, you can strip down symbols from the binary to reduce size because embedded systems usually have small storage areas.

The `gdbserver` program may display some error messages when you connect to the server as it needed `/etc/protocols` files on the target Arcom board, which I copied manually.

The step-by-step process of using `gdbserver` is as follows:

- Compile a program on the host machine using the `–g` option.
- Transfer a stripped version of the output to the target machine. You can use a stripped version because you don't need symbols on the target.

- Run gdbserver on the target machine. It will control your application program. It is explained later in this section.
- Run gdb on the host machine.
- Connect gdb to the remotely running gdbserver process.
- Start debugging.

You can connect gdb on the host to the gdbserver running on the target over the network or using the serial port. Both of these methods are described here.

8.5.1.1 Running gdbserver On Target Using TCP/IP

When you have transferred the executable binary to the target, you can start the gdbserver process. The example below shows that the program being debugged is /tmp/sum, IP address of the host machine is 192.168.1.3 and we are using TCP port 2000 for connection between the host and the target machine. Note that the debugger will be running on the host machine.

In the following session, the gdbserver waits for connection from the debugger after starting up. When you step through the program on the debugger side, you can enter values of two numbers on the target machine. I used a Telnet session to open a window on the target. This window was used to start gdbserver and enter values of two numbers.

```
root@SBC-GXx /bin# gdbserver 192.168.1.3:2000 /tmp/sum
Process /tmp/sum created; pid = 154
Remote debugging using 192.168.1.3:2000
Enter first number : 5
Enter second number : 7

The sum is : 12
Killing inferior
root@SBC-GXx /bin#
```

Note that the gdbserver terminates when you disconnect the debugger on the host side. Source code of the program sum.c used in this example is as follows:

```
#include <stdio.h>
main ()
{
  int num1, num2, total ;

  printf("Enter first number : ");
  scanf("%d", &num1);
  printf("Enter second number : ");
  scanf("%d", &num2);

  total = num1 + num2;

  printf("\nThe sum is : %d\n", total);
}
```

It is the same program that was used in Chapter 5 for different examples.

8.5.1.2 Running gdb on Host Using TCP/IP

After starting `gdbserver` on the target, now you can start the GNU debugger on host. The host and target must be connected over a TCP/IP network for this example. In the following session on the host machine, file `sum` is a non-stripped version of the program that you uploaded to the target machine as it provides the symbol table to the debugger. After starting the debugger, you use the "`target remote 192.168.1.10:2000`" command to connect to the `gdb-server` running on the target machine 192.168.1.10. After that you can continue with normal debugging process.

```
[rrehman@desktop 5]$ gdb sum
GNU gdb 5.1.1
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public
License, and you are
welcome to change it and/or distribute copies of it under
certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty"
for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb) target remote 192.168.1.10:2000
Remote debugging using 192.168.1.10:2000
0x40001930 in ?? ()
(gdb) break main
Breakpoint 1 at 0x8048496: file sum.c, line 6.
(gdb) continue
Continuing.

Breakpoint 1, main () at sum.c:6
6         printf("Enter first number : ");
(gdb) n
7         scanf("%d", &num1);
(gdb) n
8         printf("Enter second number : ");
(gdb) n
9         scanf("%d", &num2);
(gdb) n
11        total = num1 + num2;
(gdb) n
13        printf("\nThe sum is : %d\n", total);
(gdb) n
14      }
(gdb) n
warning: Cannot insert breakpoint 0:
Cannot access memory at address 0x1
```

```
(gdb) **quit**
The program is running.  Exit anyway? (y or n) y
[rrehman@desktop 5]$
```

Note that after each `scanf` statement, you have to enter the input value on the target. This is because the program is actually running on the target machine. Since both host and target are connected over a network, you can open a Telnet window on the target for input and output.

8.5.1.3 Running `gdbserver` on Target Using Serial Port

You can also use `gdbserver` to connect a host and target machine over a serial port link. In the following example, you start `gdbserver` to communicate to the debugger over serial port `/dev/ttyS1`. GNU debugger uses its own serial port protocol to exchange information between host and target machines. The program being debugged is stored as `/tmp/sum` on the target machine. This example session is completed on an Arcom single-board computer running embedded Linux.

```
root@SBC-GXx /bin# gdbserver /dev/ttyS1 /tmp/sum
Process /tmp/sum created; pid = 180
Remote debugging using /dev/ttyS1
Enter first number : 3
Enter second number : 8

The sum is : 11
Killing inferior
root@SBC-GXx /bin#
```

Note that if you are running any other program on the serial port, it should be stopped. In most cases a `getty` process is running on serial ports. You can edit the `/etc/inittab` file to stop the `getty` process from using the serial port.

8.5.1.4 Running `gdb` on Host Using Serial Port

To connect to a target over a serial port `/dev/ttyS0`, you can use "`target remote /dev/ttyS0`" command after starting GNU debugger. A typical session to debug the `sum.c` program is shown below.

```
[root@desktop /root]# gdb sum
GNU gdb 5.1.1
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public
License, and you are
welcome to change it and/or distribute copies of it under
certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty"
for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb) break main
Breakpoint 1 at 0x8048496: file sum.c, line 6.
```

```
(gdb) target remote /dev/ttyS0
Remote debugging using /dev/ttyS0
0x40001930 in ?? ()
(gdb) continue
Continuing.

Breakpoint 1, main () at sum.c:6
6         printf("Enter first number : ");
(gdb) n
7         scanf("%d", &num1);
(gdb) n
8         printf("Enter second number : ");
(gdb) n
9         scanf("%d", &num2);
(gdb) n
11        total = num1 + num2;
(gdb) n
13        printf("\nThe sum is : %d\n", total);
(gdb) n
14      }
(gdb) n
warning: Cannot insert breakpoint 0:
Cannot access memory at address 0x1
 (gdb) quit
The program is running.  Exit anyway? (y or n) y
[root@desktop /root]#
```

You do not use the `run` command in this session because the program has already been started by the `gdbserver` running on the target machine.

8.5.1.5 Serial Cable

You can use a null modem cable to connect serial ports on the host and target machines. Connections for the null modem cable for a DB-9 connector are as shown in Table 8-1. However, I hope that most people can simply go out and buy a null modem cable, without worrying about pinouts.

Table 8-1 Null modem connection for DB-9 connector serial cable

DB-9 pin number (Host)	DB-9 pin number (Target)
TXD (3)	RXD (2)
RXD (2)	TXD (3)
RTS (7)	CTS (8)
CTS (8)	RTS (7)
Signal Ground (5)	Signal Ground (5)
DSR (6)	DTR (4)
DTR (4)	DSR (6)

For a DB-25 connector, the cable connections are as listed in Table 8-2.

Table 8-2 Null modem connection for DB-25 connector serial cable

DB-25 pin number (Host)	DB-25 pin number (Target)
Frame Ground (1)	Frame Ground (1)
TXD (2)	RXD (3)
RXD (3)	TXD (2)
RTS (4)	CTS (5)
CTS (5)	RTS (4)
Signal Ground (7)	Signal Ground (7)
DSR (6)	DTR (20)
DTR (20)	DSR (6)

8.5.2 Attaching to a Running Process Using `gdbserver`

Using the `gdbserver` program, you can also connect to a running process on the target. For example, the following command can be executed on the target to connect to a process with process ID equal to 375.

```
gdbserver /dev/ttyS1 375
```

You can also use the TCP/IP connection by replacing the `/dev/ttyS1` with hostname:port number as you have seen earlier.

8.5.3 Using Stubs with GNU Debugger

Stubs are software routines that you can link with your program. Using a stub, you can connect the process created by the program to the GNU debugger from a remote target, eliminating the need for the `gdbserver` program.

Stubs are different for different targets. The GNU debugger contains some stubs by default in the source code tree. For example, the `i386-stub.c` file can be used for targets that use Intel CPUs. However you have to customize your stub depending on the target. Some instructions are included in the example source code files for the stub.

8.5.4 Debugging the Debug Session

If you are not successful in connecting to the target machine using GDB, you can get help from the remote debug mode of GNU debugger. After turning this mode ON, you will see a lot on information on your screen when `gdb` establishes or tries to establish connection to a remote target. This information shows packets sent and received by `gdb`. The following session shows that we set the baud rate to 9600 using the `remotebaud` command, turn the remote debug mode ON and then connect to the remote target on serial port `/dev/ttyS0`.

```
[root@desktop /root]# gdb sum
GNU gdb 5.1.1
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public
License, and you are
welcome to change it and/or distribute copies of it under
certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty"
for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb) break main
Breakpoint 1 at 0x8048496: file sum.c, line 6.
(gdb) set remotebaud 9600
(gdb) set debug remote 1
(gdb) target remote /dev/ttyS0
Remote debugging using /dev/ttyS0
Sending packet: $Hc-1#09...Ack
Packet received: OK
Sending packet: $qC#b4...Ack
Packet received:
Sending packet: $qOffsets#4b...Ack
Packet received:
Sending packet: $?#3f...Ack
Packet received: T0508:30190040;05:00000000;04:f8fdffbf;
Sending packet: $m40001930,8#62...Ack
Packet received: 54e8520000005b89
Sending packet: $m40001930,7#61...Ack
```

```
Packet received: 54e8520000005b
0x40001930 in ?? ()
Sending packet: $qSymbol::#5b...Ack
Packet received:
Packet qSymbol (symbol-lookup) is NOT supported
(gdb) continue
Continuing.
Sending packet: $Z0,8048496,1#8a...Ack
Packet received:
Packet Z0 (software-breakpoint) is NOT supported
Sending packet: $m8048496,1#41...Ack
Packet received: 83
Sending packet: $X8048496,0:#65...Ack
Packet received:
binary downloading NOT supported by target
Sending packet: $M8048496,1:cc#21...Ack
Packet received: OK
Sending packet: $m4000e060,1#89...Ack
Packet received: 8b
Sending packet: $M4000e060,1:cc#69...Ack
Packet received: OK
Sending packet: $Hc0#db...Ack
Packet received: OK
Sending packet: $c#63...Ack
Packet received: T0508:97840408;05:a8fdffbf;04:90fdffbf;
Sending packet: $Hg0#df...Ack
Packet received: OK
Sending packet: $g#67...Ack
Packet received:
f83011409084040838 4f1140ac40114090fdffbfa8fdffbfa0b60040fcfdff
bf9784040886020000230000002b0000002b0000002b000000000000000000
00000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000
Sending packet: $P8=96840408#6c...Ack
Packet received:
Sending packet:
$Gf83011409084040838 4f1140ac40114090fdffbfa8fdffbfa0b60040fcfd
ffbf9684040886020000230000002b0000002b0000002b0000000000000000
00000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
```

```
00000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000#9
2...Ack
Packet received: OK
Sending packet: $M8048496,1:83#c6...Ack
Packet received: OK
Sending packet: $M4000e060,1:8b#3d...Ack
Packet received: OK

Breakpoint 1, main () at sum.c:6
6         printf("Enter first number : ");
(gdb)
```

All of this information is very useful to resolve problems in establishing connection between a host and a target machine.

8.6 Hardware Used for Cross-Platform and Embedded Systems Development

During development of this book, especially this chapter, I have used two single board computers. This was done to create examples and test functionality of code for cross and embedded systems development. Arcom (http://www.arcomcontrols.com) and Artesyn (http://www.artesyn.com) are the two companies that provided these systems for experimentation. Information about these two boards is provided here for reference.

8.6.1 Arcom SBC-GX1 Board

The Arcom people were very helpful in providing their SBC-GX1board development kit. This kit includes the SBC-GX1 board, cables, and adapter for supplying power to the board, documentation and software CDs. It also includes a PS/2 mouse that can be connected to the board during experimentation. Everything is enclosed in a box to carry the kit.

The CPU used on this board is an x86 compatible processor from National Semiconductor's Geode® GX1™ processor family. It is a low voltage, low power consumption CPU that can be used in application where long batter life and low power consumption are important factors. If you recall the earlier discussion in this chapter, both of these factors are very important in embedded systems applications. The CPU contains additional hardware components that are not part of x86 class CPUs. These components are:

- Integrated VGA controller, which has graphic acceleration features. Can be used to connect to CRT or TFT flat panel video devices.
- A PCI host controller.
- Additional features for graphics and audio applications.
- Integrated I/O lines.

A complete list of features can be found in the accompanying Arcom CD. The CPU on the board I received has a clock speed of 333 MHz with no fans or requirement of any other cooling method. This is a cool CPU board.

The Arcom board has the following features that make it suitable for many applications. All of this is integrated to a small sized board.

• Floppy disk controller
• Hard disk controller to which you can connect hard drives or CD/DVD drives.
• Sound Blaster
• Four serial ports
• Parallel port
• VGA connector for CRT monitors
• Interface for flat panel monitor
• PC/104 bus
• PCI bus
• 16 MB flash that can be used as a disk
• 64 MB RAM
• Ethernet controller to connect to a LAN
• PS/2 mouse and keyboard connectors
• Compact flash socket that can be used to increase storage space in case you need large applications

The board comes with Linux kernel 2.4.4 pre-installed with many features that are important for embedded systems. Arcom embedded Linux is optimized for space but has all of the commonly used applications and utilities. These include:

• A Telnet server and client
• NFS support
• FTP server and client that can be used to transfer files from host computer to the target computer
• HTTP server. You can start using the HTTP server right from the beginning. A default page is already present on the board.
• X-Windows support
• The `inetd` daemon that can be used to launch network applications on demand

In addition to these, Arcom embedded Linux has the following features that may be of interest to the developers of embedded applications.

• Kernel 2.2.4
• Journaling Flash File System (JFFS2), which is a compressed file system used on flash storage.

- Glibc 2.1.3, which allows dynamically linked programs to run on the embedded platform.
- X-Free version 4.0.2
- `Ncruses` library
- Bash shell
- Busybox which is a Swiss army knife of generally used UNIX utilities
- Networking suite and utilities
- Use management utilities
- Modules handling utilities
- Syslogd/klogd
- Terminal handling

Onboard flash memory is divided into three parts. Partition mounted on `/` is read-only at the boot time and contains Linux system. Two other file systems `/var` and `/var/tmp` are mounted as read-write at the boot time and can be used to store temporary files. In addition to that, you can also use the removable flash card to store files. Bundled utilities on accompanying disks can be used to re-install the operating system or add new components. In networked systems, you can also use a NFS server to add more storage space.

Arcom also provides RedHat 6.2 CD that you can use to install Linux on a PC. This PC will be used as the development system or a host machine. The Arcom embedded Linux board will act as the target system. This means that in addition to the single board computer with Embedded Linux installed on it, you also get the Linux operating system to use as a development platform.

8.6.2 Artesyn PM/PPC Mezzanine Card

The other embedded system used in the development of this book is Artesyn PM/PPC mezzanine card that can be plugged into a carrier card. Artesyn also supplied the carrier card for this purpose. It is a PowerPC 750 based card and provides the Ethernet and serial interfaces to the outside world directly from the PMC (PCI Mezzanine Card). It contains 64 Mbytes of RAM, I^2C bus, general purpose timers, real-time clock, LEDs and so on. The board also contains flash memory that can be used to store the operating system and utilities. It has a real-time clock and PLD used to setup memory map The CPU runs at 333 MHz. Boot flash is used to store the operating system. The optional JTAG connector can be used to download and debug code.

As mentioned earlier, you need a career card to power up the PMC and Artesyn provides you with a CC1000 PMC carrier card that can host up to two PMC. CC1000 can be placed into a Compact PCI chassis that allows 6U cards to be plugged in.

More information can be found on the Artesyn web site http://www.artesyncp.com or by sending an email to info@artesyncp.com.

8.7 References

1. Arcom web site at http://www.arcomcontrols.com
2. Artesyn web site at http://www.artesyncp.com
3. Embedded Linux web site at http://www.linux-embedded.org
4. Busybox for Embedded Linux at http://www.busybox.net/