

Linux-Based Cross Development

Thomas E. Besemer

The cost of entry for commercial embedded operating systems can be prohibitive for many smaller organizations. Additionally, there are a significant number of embedded developers who prefer to “roll their own” embedded operating system. Linux provides a natural development platform for these groups of people: cost of configuration is minimal, and the GNU tools may be built to target a variety of microprocessor architectures. With care in design, it is possible to simulate a developer’s target hardware while executing within the Linux environment. Additionally, in certain cases, Linux works well for both the host-based development environment and the target runtime environment.

This article is an introduction to taking advantage of Linux for cross-based application software development. The GNU tools, which come with Linux distributions, are presented from the perspective of how to use them to generate ROM-based binary images, and concepts are shown on how to build embedded target applications from the GNU tools and Linux environment. The important concepts covered in this article center around the following key topics:

1. Understanding the GNU Memory Model and binary images.
2. Understanding how to use the GNU tools to manipulate the binary images.
3. Defining and constructing a runtime environment infrastructure for the embedded target.

Several important concepts are outlined in this article, and one of my goals is to show the reader how to use Linux to do cross work. Although this article contains numerous examples of code and tool usage, it will also help you understand the underlying development environment.

This article provides information for developers who will not work with commercial operating systems and who need an understanding of how to proceed with their own internally generated runtime environments. However, people using commercial environments such as VxWorks will gain a solid understanding of “what’s going on with memory” in such a way that their own debug efforts during development will be enhanced. All examples in this article were developed and tested under HardHat Linux from Monte Vista (<http://www.mvista.com>), running on a Pentium-grade processor. The cross target is iAPXx86, but it is important to note that all of the concepts discussed here apply to any target environment.

Listing 1 Simple C example — romCode.c

```
int          romIntData = 12345678;
static char  romCharData = 10;
int          romIntBss;
static char  romCharBss;
char         *textString = "my string";

void myPrintf( char *fmtStng, ... );

void romStart()
{
  int    localData = (romIntData * romIntBss);

  myPrintf( textString, localData );
}

void myPrintf( char *fmtStng, ... )
{
}
```

GNU Tools and Embedded Applications

In host-based development with GNU tools, the developer need not have a fluent understanding of memory usage, application startup and termination, or the object file format of the application code. These concepts and underlying technical details are the foundations of being able to use the GNU tools to cross develop an embedded application. An understanding of the memory model and how ROM-based code will operate is essential, however. The best way to understand these concepts is to work with a simple code example. Listing 1 is a simple C-based code example that will be used throughout this article, using each of the GNU tools to operate on and with it.

The most notable aspect about the source code in Listing 1 is that it has, for all practical purposes, all the types of information that will exist in a large and full-featured application: executable code (text), local data (resides on the stack), initialized data (data), and uninitialized data (bss). When writing code to embed in a system, understanding

where it all goes in memory is important. A simple trick to understanding where things are being placed in memory is to compile fragments (such as Listing 1) into assembly language. This concept works for small and large programs, and can often be a nice approach to gaining an understanding of the developer’s embedded application. Listing 2 was generated from the example using the following command:

```
cc -S romCode.c
```

This command generates a file called `romCode.s`, which provides a clear presentation of how C code is represented at the assembly language level. In reviewing Listing 2, each major section can be examined. Note that since the important concepts in this example are the memory segments, some of the original assem-

bler source has been deleted for clarity (the bulk of the assembly language code, as well as some additional information the compiler places in the assembler files, which is not needed for this discussion).

The Text Segment

All executable code is located in a segment called `text`. Listing 3 shows how the procedure `romStart()` is defined in assembly language. The segment directive `.text` states that all information from this point forward in the file (or until another segment directive is encountered) shall be located in the text segment. The directive `.align 4` ensures that the procedure is long-word aligned. The directive `.global` marks the procedure `romStart()` as public (as opposed to static), while the `.type` directive states the type of label `romStart`. Finally, `romStart:` is the label that marks the start of the procedure, followed by the first assembly language instruction of the procedure. Every procedure in a user's application will have a definition similar to this. The most important thing to note, for now, is how the procedure is defined. Later in this article, we will start evaluating object files, and an understanding of how the compiler generates code is essential.

The Data Segment

All initialized data is located in a segment called `data`. Listing 4 shows how the integer and character data, called `romIntData` and `romCharData`, respectively, are defined. Like the procedure shown in the text segment, the segment is identified, the alignment specified, the data type specified (object, in this case), and labels exist. The big difference between this segment and the procedure definition is that following the label `romIntData:` is the actual data, defined by `.long 12345678`. The label `romCharData` shows byte-wide data (also initialized), and since it was declared static, a `.global` directive does not exist for it.

It is important to note that C application code may write to initialized data, altering its initial compiled-in value. This characteristic gives the developer two important concepts to keep in mind when building ROM-based code:

1. The data segment cannot be contained in ROM; it must be writable.
2. The user-developed "loader", which starts the ROM-based application code, must copy the initialized data from a location in ROM into RAM every time the system starts, or when the application is invoked for the first time.

The `bss` Segment

All un-initialized data is located in a segment called `bss`. Listing 5 shows how integer (`romIntBss`) and character-wide data (`romCharBss`) are defined and allo-

Listing 2 Simple C example — Generated assembly language

```
.globl romIntData
.data
    .align 4
    .type    romIntData,@object
    .size    romIntData,4
romIntData:
    .long 12345678
    .type    romCharData,@object
    .size    romCharData,1
romCharData:
    .byte 10
.globl textString
.section    .rodata
.LC0:
    .string "my string"
.data
    .align 4
    .type    textString,@object
    .size    textString,4
textString:
    .long .LC0
.text
    .align 4
.globl romStart
.type    romStart,@function
romStart:
    pushl %ebp
    # Much more code in here
    leave
    ret
.Lfe1:
    .size    romStart,.Lfe1-romStart
    .align 4
.globl myPrintf
.type    myPrintf,@function
myPrintf:
    pushl %ebp
    movl %esp,%ebp
.L2:
    leave
    ret
.Lfe2:
    .size    myPrintf,.Lfe2-myPrintf
    .comm   romIntBss,4,4
    .local  romCharBss
    .comm   romCharBss,1,1
```

Listing 3 Procedures and the text segment

```
.text
    .align 4
.global romStart
    .type    romStart,@function
romStart:
    pushl %ebp
```

cated. This is similar to how `romIntData` and `romCharData` are defined in the data segment, but lacks a segment directive — the directive that allocates uninitialized data is called `.comm`, and local (static) data is identified with a `.local` directive. No initialization occurs. Note that when a program is loaded by the Linux loader for execution, the loader will zero all of bss. When building ROM-based code, it is essential that the user's "home grown" loader zero bss. This will be shown later in the article. Also important is that bss is writable; this segment of application code must be located in RAM.

Text Strings

Listing 6 shows how the string `textString` is allocated. Note that the string itself is located in a segment called `.rodata` (read-only data), while the pointer to the string is located in the data segment. The compiler assigns an internally defined label, `.LC0`, to the string itself, and then assigns (initializes) the address of `.LC0` to the pointer labeled `textString`.

Listing 4 Initialized data and the data segment

```
.global romIntData
.data
    .align 4
    .type    romIntData,@object
    .size    romIntData,4
romIntData:
    .long 12345678
    .type    romCharData,@object
    .size    romCharData,1
romCharData:
    .byte 10
```

Building a Binary

Working at the assembly language level enables the programmer to understand how the compiler allocates C-source information. The next step lies in understanding object files, and how to generate binary images. Listing 7 shows a basic makefile that compiles the C source, and then links it into a linked and located ELF executable file.

The source C code must first be compiled into an ELF object file suitable for linkage. This is shown with the command:

```
cc -c romCode.c
```

This command generates the object file `romCode.o`. This object file is considered relocatable, in that none of the internal labels have physical memory addresses assigned to them. When building images for ROM, it is necessary to link all of the application object files into a single object file. This object file must have symbols-assigned addresses based on a

physical address in memory where the application will execute.

ld — The GNU Linker

Just as when doing native Linux application development, the linker (`ld`) is used to perform final linkage and location of the application. However, when generating binary images suitable for placing in ROM and executing in a defined physical location on the target, there are many significant differences in how the linker is used, and how the application binary is constructed.

Listing 5 Uninitialized data and the bss segment

```
.comm  romIntBss,4,4
    .local  romCharBss
    .comm  romCharBss,1,1
```

Libraries

Applications that execute under Linux can take advantage of the native Linux libraries, such as `glibc`. These libraries provide such basic functions as `printf()`. The function `printf()` provides a good example of the differences between host-based applications and embedded applications: if the linker encounters a symbol such as `printf()`, which is

external to the object files being linked, it will search native libraries to resolve the symbol. In doing such, the linker will automatically link in a substantial foundation for I/O in the system. This I/O system allows the simple function `printf()` to send messages to the console. In an X Window environment, this pulls in object files that are able to send messages to the console, or any file descriptor that the console may be redirected to.

None of these support mechanisms exist in a bare-bones target execution environment, and it is up to the developer to design and implement an underlying target support package for doing even the simplest of functions, such as `printf()`. These libraries are considered Board Support Packages. If the developer uses a commercial operating system, such as VxWorks from Wind River Systems, an extensive set of libraries are supplied by the vendor to support functions like `printf()`. In a VxWorks (or any other commercial embedded operating system environment), this extensive library package allows the user a clean means of tying the runtime to the hardware through a Board Support Package. If the reader is developing his own “home-grown” embedded operating system for a target, there are many options on getting a head start when using Linux as the host for development. The biggest option is the availability of source code. The source code for the basic libraries, such as `printf()`, `strcpy()`, `vsprintf()`, exists. In Linux, these functions will call I/O functions such as `open()`, `close()`, `read()`, and `write()`. Functions like `vsprintf()` are generic in nature. They have no underlying ties to the operating environment, whereas calls such as `open()` have extensive ties to a native operating system like Linux. However, with all the source code available, developers may examine the Linux `open()` call to gain an understanding of how to structure their own libraries. A good starting point for developing a target library environment can be with the `glibc` sources. When doing this, it’s important for the developer to understand the GNU licensing model when it comes to modifying source code that is public domain. The Free

Software Foundation allows developers to modify source code and redistribute it. The basic rule is that developers who modify the source code must place it back in public domain for others to use. It is important that developers read and understand the licensing rules for GNU and the FSF.

Application Startup and Entry Point

When developing applications under Linux, `main()` is the entry point that all developers are familiar with. Yet applications in Linux, after the final stage of linkage, have a significant amount of Linux-specific startup code linked in that is executed prior to `main()` being called. This Linux-specific startup code is pulled from native Linux libraries. Startup code performs many functions. A basic example is allocating memory for a stack pointer, and initializing the stack pointer. When cross-developing to a target, using Linux as the host, the developer is responsible for implementation of startup code. Typically, this startup code does the following type of functions:

Listing 6
Text string allocation

```
.section      .rodata
.LC0:
    .string "my string"
.data
    .align 4
    .type   textString,@object
    .size   textString,4
textString:
    .long .LC0
```

Listing 7
Makefile

```
<<>code<<>
romCode:      romCode.o makefile
              ld -Ttext 2000 -Tdata 4000 -e romStart -o romCode <<>
              romCode.o <<>
              -Map romCode.map

romCode.o:    romCode.c
              cc -c romCode.c
<<>text<<>
```

1. Hardware initialization, such as setting up SDRAM, configuring I/O, and initializing peripherals — In almost every case there will need to be a short assembly language program that is jumped to coming out of reset, and this program will then, after doing basic initialization, pass control to C-based startup code.
2. Performing power-on self-tests.
3. Often copying the application out of ROM and into RAM for execution.
4. Allocating a stack pointer and setting the appropriate processor register with the address of the stack pointer .
5. Processor-specific, register-specific initialization, such as base pointers pointing to the base of the application's data segment. In this case, we use a broad brush stroke to define the application. It is the user's home-grown operating system (if one exists) as well as the user-functional application, the code that gives their embedded target its functionality.

Controlling Linkage

When cross targeting, there are several important concepts to keep in mind when performing final linkage:

1. No libraries are pulled in that have ties to the Linux environment.
2. The users have supplied either libraries or object files that support all functions in their applications. There cannot be any unresolved externals, or externals that will result in the Linux libraries being pulled in.
3. The image is located to execute at a known physical location in memory.
4. An entry point is defined for the application.

The makefile contained in Listing 7 shows a typical command line for doing final linkage of an embedded, ROM-based application:

```
ld -Ttext 2000 -Tdata 4000 -e romStart -o romCode \
romCode.o \
-Map romCode.map
```

There are three key options passed to the linker. The option `-Ttext 2000` tells the linker to link the application text segment (at location 0x2000 in this example). This is our defined location from where the code will execute. The option `-Tdata 4000` tells the linker to link the application's data segment (at location 0x4000 in this example). This is our defined location where the data segment will reside in physical memory. The option `-e romStart` tells the linker that the entry point is the symbol (procedure) `romStart()`. This option causes no real effect on the resultant object file, other than keeping the linker from complaining that `main()` does not exist. In the above example, the linker is instructed to assign physical addresses to all symbols in the text and data segments; we have "located" the code. Since we have not instructed the linker to place the bss segment at a specific physical location, the bss segment will follow the data segment in physical memory. If we had omitted the directive for the data segment, such as in the example below, then the data segment would simply follow the text segment, and again, the bss segment following the data segment.

```
ld -Ttext 2000 -e romStart -o romCode \
romCode.o \
-Map romCode.map
```

More GNU Tools

There are two GNU tools in the Linux environment that allow us to examine object files: `nm` and `objdump`. The utility `nm` allows examination of the symbols in object files. The utility `objdump` allows us to display various information about the

object file, such as segment headers or disassembled code.

Using nm to Display Symbols

Listing 8 shows the output from invoking the `nm` utility on the final linked object file `romCode` through the following command:

```
nm romCode
```

Since the object file was “located” to reside in defined physical locations in memory, all of the symbols displayed with this command have a physical memory address associated with them.

For example, `romStart()` is located at `0x2000`. Since `romStart()` is the first text segment symbol in the final, located link, it occurs at the location we defined as the base of the text segment. The symbol (procedure) `myPrintf()` follows the procedure `romStart()` in the original source file, and thus is located further down in memory at physical location `0x202c`.

For the data segment, source code symbols are visible as well. The symbol `romIntData` is located at the base of the data segment. It was the first symbol for the data segment encountered during linkage, so is assigned the first address of the data segment. Since, during final linkage, an address was not defined for the bss segment, the segment bss is placed at the end of the data segment. The symbol `romIntBss` shows an address that is in the same physical region as the data segment.

Note symbols that were not in the original source code:

```
__bss_start, _edata, _etext, and _end.
```

These are real symbols placed in the final object file by the linker. They can be very useful when moving blocks of memory around, such as copying data from ROM to RAM. Usage of these symbols will be discussed later in this article, so keep them in mind. Also note that physical addresses assigned to symbols are assigned on the order in which the symbols are encountered in the object files during final linkage. If the developer finds, during development, that data is being corrupted through a suspected stray pointer, he can use `nm` to list the symbols, and then the Linux `sort` utility to arrange the output with ascending addresses. Often, such as in the case of array boundary overflow, developers can find the source of your stray pointer by looking at the data before or after other symbols. This is another reason why it is extremely important to be aware of physical memory layout when doing cross-based embedded development.

Using objdump on Object Files

Another extremely useful utility is `objdump`. This utility allows the developer to perform further examination on any object file. It has a large number of options (refer to the manpage), and can perform functions such as displaying the ELF header file information, or disassemble the assembly language contained in object files. The compiler will generate object files that are in ELF format, and the linker is able to read these ELF files to produce incremental or final links. Regardless of whether an incremental or final link is performed (in our example above, a final link was performed that linked and located our object file to a specific set of physical memory addresses), the resulting output file is in ELF format. At some point during development, it will be necessary to turn this final ELF file into a binary or hex-record type of file for loading on our target. The `objdump` utility can be used for this, and that process will be covered later in the article. For now, let's examine what the following command does:

```
objdump -headers romCode
```

Listing 8 Object file symbols

```
0000400c A __bss_start
0000400c A _edata
00004014 A _end
00002031 A _etext
00002000 t gcc2_compiled.
0000202c T myPrintf
0000400c b romCharBss
00004004 d romCharData
00004010 B romIntBss
00004000 D romIntData
00002000 T romStart
00004008 D textString
```

Listing 9

Header information from objdump

```
romCode:      file format elf32-i386
Sections:
Idx Name          Size      VMA           LMA           File off  Algn
 0 .text          00000031  00002000  00002000  00001000  2**2
                CONTENTS, ALLOC, LOAD, READONLY, CODE
 1 .data          0000000c  00004000  00004000  00002000  2**2
                CONTENTS, ALLOC, LOAD, DATA
 2 .rodata        0000000a  00002031  00002031  00001031  2**0
                CONTENTS, ALLOC, LOAD, READONLY, DATA
 3 .bss           00000008  0000400c  0000400c  0000200c  2**2
                ALLOC
 4 .comment       0000003d  00000000  00000000  0000200c  2**0
                CONTENTS, READONLY
 5 .note          00000014  0000003d  0000003d  00002049  2**0
                CONTENTS, READONLY
```

Listing 9 shows the output from this operation. Each entry in the Name column is a segment in the final link. The information is similar to what `nm` provides, but more global in nature. For example, in Listing 9, the base address is found for each segment, its size, and most importantly, its offset in the ELF file. As noted above, at some point the ELF file will need to be converted to some sort of absolute binary or hex file for loading on the target. Developers can either write a tool that understands ELF file formats, or can use `objdump` to tell us where the data is located in the file and work forward from this.

Applying Understandings

For developers, schedules are always tight, and it's very important to understand how to leverage existing tools to help the application develop. To this point, a significant number of important concepts and tools have been introduced. Now it's time to apply these concepts and take advantage of the tools.

Listing 10

Binary extraction shell script

```
#!/bin/sh
#
# getText - script to extract the text segment from
# an ELF file.
#
# Get the size of the text segment from the ELF file
#
SIZE='objdump romCode -headers | sed -n '/?text/p' |\
    awk '{print $3}''
#
# Get the offset of the text segment in the ELF file
#
OFFSET='objdump romCode -headers | sed -n '/?text/p' |\
    awk '{print $6}''
#
# Extract the text segment from the ELF file
#
./dumpBin -s romCode -d romCode_text.bin -o $OFFSET -n $SIZE
```

Building a Binary

A hurdle in cross developing is building a raw binary image suitable for installation in ROM, or execution out of RAM. Without an ELF loader, dealing with the final linked and located image can be difficult. ROM programming tools, for the most part, don't understand file formats such as ELF — they like binaries or hex files. Developers may write their own custom target loaders, and will want to work with a file format they understand, instead of ELF. To generate binary or hex images, it's essential that we extract the raw binary data from the source ELF file. To do this, the developer requires either a fluent understanding of ELF file format, or more simply, an understanding of where the data is

located in the ELF file, and how long it is. The utility `objdump` provides us with this information. When doing GNU-based development in the past, I wrote a tool called `dumpBin` that is used in the following examples. This tool can be downloaded from the my Web site in source-code format, and has the following options:

```
usage: dumpBin -s src_file -d dst_file -o offset (hex)
        -n number bytes (hex)
```

As shown, it allows for both a source and destination file to be specified, the offset within the source file to start with, and the number of bytes to dump. To pull all of the text segment data from the source, final-linked, and located ELF file, the following is done:

```
dumpBin -s romCode -d romCode_text.bin -o 1000 -n 31
```

Reviewing Listing 9, the output from `objdump` shows that the text segment is at offset `0x1000` in the ELF file, and that the size of the text segment is `0x31` bytes in length. The tool `dumpBin` simply reads the source ELF file, seeks to offset `0x1000`, and writes `0x31` bytes from the source file to destination file `romCode_text.bin`. Without having to understand the ELF file format, the binary data for the text segment is extracted and placed in a binary file for further manipulation.

Listing 10 shows a primitive shell script that automates the entire process of extracting the text segment from the final linked and located ELF file. Although primitive, this example shows how scripting, using both native and simply developed user's tools, can be implemented in a form suitable for use in a makefile. The output file `romCode_text.bin` is a raw binary file that contains the opcodes for the entire text segment. The developer can, if needed, write a simple tool to turn this into a file suitable for a ROM programmer or his own internal custom loader. The concept can easily be expanded upon. For example, if the developer defines and implements a custom loader, a known file format can be installed at the beginning of the output file that shows the type of data contained, its size, and its load location. This allows the developer to implement a simple loader

Listing 11
Application text segment in C

```
char romCode_text[] = {
0x55, 0x89, 0xE5, 0x83, 0xEC, 0x04, 0xA1, 0x00, 0x40,
0x00, 0x00, 0x0F, 0xAF, 0x05, 0x10, 0x40, 0x00, 0x00,
0x89, 0x45, 0xFC, 0x8B, 0x45, 0xFC, 0x50, 0xA1, 0x08,
0x40, 0x00, 0x00, 0x50, 0xE8, 0x08, 0x00, 0x00, 0x00,
0x83, 0xC4, 0x08, 0xC9, 0xC3, 0x8D, 0x76, 0x00, 0x55,
0x89, 0xE5, 0xC9, 0xC3, };
int romCode_text_size = sizeof( romCode_text );
```

Target Environment Configuration

In most embedded applications, there are two memory block configurations to deal with:

1. Boot code that executes when the processor is coming out of reset — this image is a combination of assembly language startup code and C-level target configuration code.
2. Application code, which resides in ROM, but is copied to RAM and placed into execution. Boot code configures the processor and target hardware, and possibly does some power on self-tests. It is then responsible for loading the actual application code from ROM into RAM and passing execution to it. In this type of configuration, two absolute binaries must be built and physically located: the boot code and the application.

A ROM Loader

The ROM loader knows about the environment, and contains a copy of the application code in binary form. Using the examples in this article, you can locate images and build binaries from

Listing 12
Application copy to RAM

```
#define TEXT_RAM_LOCATION 0x1000

extern char romCode_text[];
extern int romCode_text_size;

void copyText() {
int i;
char *srcPtr, *dstPtr;

srcPtr = romCode_text;
dstPtr = (char *)TEXT_RAM_LOCATION;

for( i = 0; i < romCode_text_size; i++ )
*dstPtr++ = *srcPtr++;
}
```

these images. For this example, assume that the developer is able to construct a binary image that will execute from ROM. This binary image will be a “first-stage” executable, and after doing hardware initialization, will need to copy the actual application from ROM into RAM for execution. With this assumption, we have two final images that get linked and located for ROM: the boot loader and the application.

The application needs to be linked and turned into a binary that is suitable for the ROM loader to deal with. A good way to do this is to expand on the concept of extracting the binaries from the ELF files. In this, it’s important to take the concept of extracting the binary information from the ELF file a little bit further — let’s take the application and turn it into a C data structure on which the boot loader can operate. A tool can be easily built to take raw binaries and turn them into C code that can be compiled into the boot-code image. Another author-generated tool, called `bin2c`, takes raw binaries and converts them to a C data structure. This source code for this tool may also be downloaded from my Web site. Example output from this tool is shown in Listing 11.

This array, called `romCode_text`, is an array that can be compiled into the boot loader. It contains the entire text segment from the example `romCode.c`. A practical example of a C code fragment for a boot loader is shown in Listing 12. This example copies the text segment from the binary array into RAM. Control can be passed to this segment after the copy operation.

The important concepts in this example are that we construct two binaries, linked at two different locations — a boot ROM binary that configures the hardware, and an application binary that is compiled into the boot ROM. The boot ROM, after doing initialization, copies the application binary into RAM and then can pass control to it.

Summary

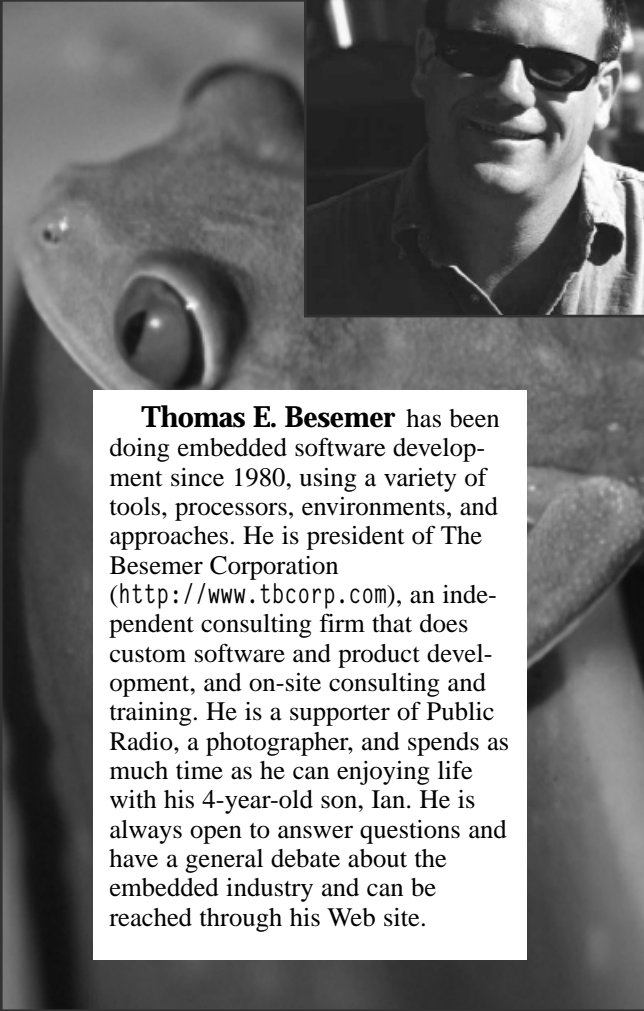
The examples in this article focus on memory segments and what they are about. These examples have split “text” from “data” and in reality, users will probably deal with a homogeneous image that contains text, data, and bss for their applications, while having a separate boot loader. An image that deals with a single linked application can easily expand upon these concepts — all segments can be extracted from the ELF files and turned into binaries, and those binaries turned into an initialized C data structure such as shown in Listing 12. The developer’s loader configures the hardware, copies the application to RAM, and then passes control to it. The important concepts to remember are the following:

1. The data segment is writable, so it must exist in RAM, but it must be initialized by copying out of a known location such as a C-based compiled-in array. One example is given on how to do this; the usage of symbols such as `_edata` and `_etext` can also be used to derive locations.
2. The segment bss must be initialized to zero before the application runs, whether the “application” is the boot loader or the functional application. A boot loader that zeros all of RAM during startup ensures that bss is in good shape before the application runs.
3. No runtime libraries will exist to support the simplest of functions, such as `printf()`. The user must provide all libraries, and the source code for `glibc` is an excellent starting point.
4. With all of Linux and the GNU tools available in source code format, readers have a tremendous advantage over typical commercial environments, because they have reference code to study and possibly work with. And it’s not as overwhelming as it seems — having an understanding gives you a good head start. Working with Linux distributions is also important since there is a significant amount of source code to start with when building libraries.

Odds and Ends

When using Linux and the GNU tools for cross development, there are a few other things to keep in mind:

1. This article deals with ELF file formats. The GNU tools can be built to generate other file formats.
2. This article targets iAPXx86 for the target. Again, the GNU tools can be built to support a variety of target processor architectures while still allowing the host to be standard PC-type equipment.
3. Linux can be configured to run on the target, avoiding the need to construct a home-grown operating system. Several vendors, such as Monte Vista (<http://www.mvista.com>), are providing support for this.
4. Developers can use the POSIX thread capability to build simulators that allow their target code to be tested and debugged on the Linux host. Using Linux and GNU for development is a new way of thinking and opens the doors to extremely economical development and target environments. The reader is encouraged to use the above examples and run the same types of tests and experiments. A day spent doing this will pay off time and again when debugging in a development laboratory environment.



Thomas E. Besemer has been doing embedded software development since 1980, using a variety of tools, processors, environments, and approaches. He is president of The Besemer Corporation (<http://www.tbcorp.com>), an independent consulting firm that does custom software and product development, and on-site consulting and training. He is a supporter of Public Radio, a photographer, and spends as much time as he can enjoying life with his 4-year-old son, Ian. He is always open to answer questions and have a general debate about the embedded industry and can be reached through his Web site.