Embedded Java with GCJ

Gene Sally

Abstract

You don't always need a Java Virtual Machine to run Java in an embedded system.

This article discusses how to use GCJ, part of the GCC compiler suite, in an embedded Linux project. Like all tools, GCJ has benefits, namely the ability to code in a high-level language like Java, and its share of drawbacks as well. The notion of getting GCJ running on an embedded target may be daunting at first, but you'll see that doing so requires less effort than you may think.

After reading the article, you should be inspired to try this out on a target to see whether GCJ can fit into your next project. The Java language has all sorts of nifty features, like automatic garbage collection, an extensive, robust run-time library and expressive object-oriented constructs that help you quickly develop reliable code.

Why Use GCJ in the First Place?

The native code compiler for Java does what is says: compiles your Java source down to the machine code for the target. This means you won't have to get a JVM (Java Virtual Machine) on your target. When you run the program's code, it won't start a VM, it will simply load and run like any other program. This doesn't necessarily mean your code will run faster. Sometimes you get better performance numbers for byte code running on a hot-spot VM versus GCJ-compiled code.

One advantage of using GCJ is that you save space by not needing the JVM. You may save money in royalties as well. Furthermore, using GCJ lets you deliver a solution using all open-source software, and that's usually a good thing.

Pitfalls

The first thing embedded engineers reach for when creating a root filesystem for a target is trusty uClibc, a compact implementation of the glibc library. For those new to using Linux on an embedded target, the standard C library can be a bit on the large side when working with targets that may have only 8MB (for example) for a root filesystem. To conserve space on an embedded system's root filesystem, engineers will switch from the standard C library to something smaller, like uClibc. GCJ requires unicode support, which is not supported by uClibc, so glibc is a requirement.

The standard library for GCJ weighs in at 16MB, so even if you could conserve space by switching to a smaller standard C library, it wouldn't make that much difference overall. The standard GCJ library can be trimmed by removing support for executing Java byte code, but the loss of that feature would reduce the overall value of GCJ.

The Host and Target Configuration

Because this article is about using GCJ in an embedded environment, it shows you how to build a cross compiler and a simple root filesystem for the target system. For those new to embedded development, a cross compiler builds code that runs on a processor different from the machine where the compilation occurred. The machine that runs the compiler is called the host and the one where the code runs is called the target.

In this article, the target system is a PPC 745/755-based system running at 350MHz. This particular board comes wrapped in a translucent case with a monitor and hard drive and is otherwise known as an iMac. Okay, this is hardly a prime example of an embedded system, but it does present some of the same challenges you'll encounter with a true embedded system. The things you learn here should apply well to embedded systems using other processors.

The host system is a run-of-the-mill IBM ThinkPad notebook running a Pentium III processor. Yellow Dog Linux is already running on the host system, but with a little sleight of hand, we'll make it use the root filesystem created in the article for testing.

Getting GCJ Ready

First, we need a cross compiler that runs on our Pentium machine that creates code for a PowerPC 750-based processor. Building a cross compiler for a target system can be a very tedious process; a working compiler is more than GCC, it also contains some extra tools (affectionately named binutils) and the standard libraries for the language.

To get a cross compiler up and running quickly, try using the crosstool package, compliments of Dan Kegel. Crosstool does all of the hard work necessary to get a cross compiler built: it fetches the source and patches, applies the patches, configures the packages and kicks off the build. After obtaining and unpacking crosstool, here are the steps for building your GCJ cross compiler:

```
$ export TARBALLS_DIR=~/crosstool-download
$ export RESULT_TOP=/opt/crosstool
$ export GCC_LANGUAGES="c,c++,Java"
$ eval `cat powerpc-750.dat gcc-4.0.1-glibc-2.2.2.dat' sh.all --notest
```

While waiting for the compilation to finish, let's take a look at what we just did to start our crosstool build. TARBALLS_DIR is the location where crosstool downloads its files. Crosstool fetches all of the files it needs for a build by default. RESULT_TOP is the installation directory of the cross compiler. Lastly, GCC_LANGUAGES controls which language front ends will be enabled for the compiler. GCC supports many different language front ends and each front end adds a considerable amount of time to the compilation process, so only the necessary ones were selected for this toolchain build.

The last line, for those lacking their bash script-foo license, dumps the two dat files on the command line and executes the all.sh script with the parameter --notest. To make building a cross compiler easy, crosstool includes configuration files with the correct environment variables set for the target processor and the gcc/glibc combination. In this case, crosstool builds a gcc 4.0.1 with glibc 2.2.2 targeting a PPC 750 processor. Crosstool includes scripts for all major processor architectures and glib/gcc combinations.

When the build finishes, the cross compiler will be installed at $RESULT_TOP/gcc-4.0.1-glibc-2.2.2/powerpc-750-linux-gnu/bin. Add this to your path to make invoking the cross compiler easier.

Getting and Unpacking Crosstool

Crosstool is the creation of Dan Kegel. You can find out everything you want to know about crosstool by visiting http://kegel.com/crosstool. The page has a great quick start guide as well as complete documentation. This article used version 0.38 available at http://kegel.com/crosstool/crosstool-0.38.tar.gz.

On the crosstool home page, check out the buildlogs link (http://kegel.com/crosstool/crosstool-0.38/buildlogs) to see what combinations of glibc/gcc successfully build for your target architecture.

Configuring the Root Filesystem

The first thing to compile with your newly minted cross compiler is the root filesystem. The root filesystem, in this case, is compliments of BusyBox. For the uninitiated, BusyBox is a single binary that encapsulates mini versions of the most popular UNIX utilities in a surpassingly small executable. Built for people that count bytes, BusyBox has hundreds of knobs to turn to create a filesystem with the utilities you need within your desired space constraints. For the purpose of this article, we change the BusyBox configuration so that it cross compiles, leaving size optimization as an exercise for the reader.

BusyBox is a mainstay of the embedded Linux world and is maintained by Erik Anderson. One way to get BusyBox is to download it at http://www.busybox.net/downloads/busybox-1.01.tar.bz2.

To create a BusyBox root filesystem, you need to invoke **`make menuconfig`** in the directory where BusyBox was untarred. The menuconfig program works just like the 2.4/2.6 menuconfig kernel configuration interface. Here's what you'll need to do to build the root filesystem.

First, select the build options. Check the Do you want to build BusyBox with a Cross Compiler? box. Fill in the prefix of the cross compiler in the input control that appears when you click this option, in this case, powerpc-750-linux-gnu-. The BusyBox build scripts concatenate the necessary tool name during compilation (gcc, ld and so on). Make sure that the compiler is on your $PATH.

Next, run make and install:

```
make
make install
```

BusyBox puts the newly minted root filesystem at ./_install. You'll notice that BusyBox compiles in much less time than GCC.

Populating the Root Filesystem with Libraries

Almost there! The root filesystem BusyBox creates does not contain any libraries. GCJ programs require some libraries and so does BusyBox, shown in Table 1.

Table 1. Table 1. Libraries Required by GCJ and BusyBox

Library File	Function
ld.so.1	Dynamically linked file loader. Invoked when the program is run, loads required libraries and performs dynamic linking.
libdl.so.2	Helper functions for manipulating dynamic libraries.

Library File	Function
libgcc_s.so.1	Defines interfaces for handling exceptions.
libgcj.so.6	The GCJ run-time library. Contains implementations of the standard Java library.
libm.so.6	Library of math functions.
libpthread.so.0	POSIX threads library.

These libraries match those used by the cross compiler. In this case, the files are stored in the $RESULT_TOP/gcc-4.0.1-glibc-2.2.2/powerpc-750-linux-gnu/powerpc-750-linux-gnu/lib (not a typo!) directory. The easiest way to get them into the root filesystem is simply to copy them:

```
for f in ld.so.1 lib libdl.so.2 libgcc_s.so.1libgcj.so.6
 -->libm.so.6 libpthread.so.0 ; do

cp
$RESULT_TOP/gcc-4.0.1-glibc-2.2.2/powerpc-750-linux-gnu/powerpc-750-linux-gnu/lib/$f
<busybox install directory>/lib

$RESULT_TOP/gcc-4.0.1-glibc-2.2.2/powerpc-750-linux-gnu/bin/power
pc-750-linux-gnu-strip <busybox install directory>/lib/$f

done
```

You also need to create a folder in the root filesystem, /proc, to use as a mountpoint for the proc filesystem. Keen eyes will notice that I'm not preserving the symlinks used to accommodate different versions of the libraries--that's a shortcut typical in embedded systems where library configuration won't change over the life of the device, unlike a desktop system. Running strip greatly reduces the amount of disk space required by the files, almost by 50%.

At this point, the root filesystem can be copied to the target system into the /tmp/bbox directory. To tell the system to use this as the root filesystem, start a terminal as root and execute the chroot command:

```
chroot /tmp/bbox /bin/ash
```

This command remaps the / mountpoint into /tmp/busybox and runs /bin/ash to get a terminal. Did it work? Congratulations! You've created a complete root filesystem for an embedded system from scratch. Pat yourself on the back.

GCJ also needs the proc filesystem mounted. After the chroot, you need to remount the proc filesystem into the current root filesystem by doing the following:

```
mkdir /proc
mount -t proc none /proc
```

Although this root filesystem resides on a standard drive, the root filesystem deployed on a production embedded system wouldn't be much different. The only changes necessary would be creating inittab, so the board will run the right scripts at the start and add a /dev filesystem with the right device files for the target board.

GCJ Development

After building the cross compiler and root filesystem, building your GCJ application will be a bit anticlimactic. We'll start with the traditional hello world:

```
Class hello {
 Static public void main(String argc[]) {
   System.out.println("hello from GCJ");
 }
}
```

Following Java convention, this class resides in the hello.class file. To compile the file, enter:

```
powerpc-750-linux-gnu-gcj hello.class --main=hello -o hello-java
```

What's going on with --main=hello? Any class could define a method with a suitable entry point. The --main=hello option tells the linker to use the main method in the hello class when linking. Leaving off this option results in a link error, "undefined reference to main", which, to the uninitiated, is confusing, because your class contains a main.

Download this file to the target and run it from the chrooted shell. You'll see:

```
# ./java-test
Hello from GCJ
```

At this point, development carries on much like any other Java project, with the exception of invoking the GCJ cross compiler instead of the native javac compiler.

Conserving Space

In this example, the root filesystem weighs in at more than 20MB. Because many embedded systems use Flash memory, which is considerably more expensive on a per-megabyte basis than disk-based storage systems, a minimally sized root filesystem is frequently an important business requirement. One easy way to reduce the size of your root filesystem is to link your application statically. Although this may seem counterintuitive at first, as you'll have an extra copy of libc code in your application, recall that libgcj.so contains the entire Java standard library. Most applications use a fraction of the standard Java library, so using static linking is a great way to winnow out the unused code in the library. Just be sure to strip the resulting binary; otherwise, you'll be shocked at the size due to the amount of debugging information in libgcj.so.

Wrapping Up

From the article, you've seen that creating software for an embedded system using GCJ is something that can be reasonably accomplished using tools already present in the Open Source community. Although there are a few minor nits, configuring the root filesystem doesn't require a heroic effort; you just need to get a few different libraries from what you otherwise would need. For applications requiring a smaller root filesystem, we've seen how you can use static linking of your application to reduce the root filesystem greatly. In short, GCJ is a practical solution for using Java on a resource-constrained embedded system--worthy of consideration for your next project.