Install the GNU ARM toolchain under Linux

Embedded development tools for a popular processor

Skill Level: Introductory

William B. Zimmerly
Freelance writer and knowledge engineer

20 May 2009

Many tools are available for programming various versions of ARM cores, but one particularly popular set is the GNU ARM toolchain. Learn more about embedded development using the ARM core, as well as how to install the GNU tools and begin using them.

If you are interested in embedded systems development on a widely used microprocessor, the Advanced RISC Machines (ARM) core fits the bill. This article provides a starting point for understanding the software side of embedded systems development by describing one set of commonly used tools: the GNU ARM toolchain.

The ARM family

Paramount among the concerns of embedded systems developers is how to get the most processing power from the least amount of electricity. The ARM processor family has one of the better designs for balancing processor power and battery power consumption.

The ARM core has gained technology advances through several versions over the last 20 years. Recent system on a chip (SoC) processors running on mobile phones such as the T-Mobile G1 Android combine dual-core (ARM9 and ARM11) processors to improve the performance of multimedia applications on low-powered platforms.

The more recent ARM cores support two operational states: *ARM state,* in which the

core executes 32-bit, word-aligned instructions, and *THUMB state,* which executes 16-bit, halfword-aligned instructions. ARM mode provides the maximum power and addressing range capability of the processor, whereas THUMB mode allows portions of a program to be written in a very tight, memory-conserving way that keeps memory costs low. Switching between the modes is a trivial exercise, and for many algorithms, the size of the code required can be reduced significantly.

ARM processors improve performance by taking advantage of a *modified Harvard architecture.* In this architecture, the processor employs separate data and instruction caches, but they "feed" off of the same bus to access external memory. Furthermore, the instructions are put into a five-stage "pipeline" so that parallel processing occurs on the five most recent instructions loaded into the pipeline. In other words, each of the five separate actions (fetch, decode, ALU, memory, write) involving instructions occur in parallel. So long as the pipeline is flowing steadily, your code enjoys the speed advantages of parallelism, but the moment a branch occurs to code outside the pipeline, the whole pipeline must be reset, incurring a performance penalty. The moral is to be careful in designing your code so that you use branching only minimally.

A unique feature that the ARM architecture provides—forcing programmers to think in new and unique ways—is that every instruction can be optionally executed based on the current state of the system flags. This feature can eliminate the need for branching in some algorithms and thus keep the advantages of a pipeline-based instruction- and data-caching mechanism running at best performance (as branching can force the caches to be unnecessarily cleared).

The GNU ARM toolchain

Most ARM systems programming occurs on non-ARM workstations using tools that cross-compile, targeting the ARM platform. The GNU ARM toolchain is such a programming environment, allowing you to use your favorite workstation environments and tools for designing, developing, and even testing on ARM simulators.

Hosted by CodeSourcery (see Resources for a link), the GNU toolchain described in this article—also known as *Sourcery G++ Lite*—is available for download and use at no cost. All the tools but the GNU C Library are licensed under the standard GNU Public License version 3 (GPL3). The GNU C Library is licensed under the GPL version 2.1. Included in the GNU toolchain are the binary utilities (`binutils`), the GNU Compiler Collection (GCC), the GNU Remote Debugger (GDB), GNU `make`, and the GNU core utilities.

Also included in the Sourcery G++ Lite package is extensive documentation of the GNU toolchain tools, including the GNU Coding Standards document—good reading all around! Under the documentation for the GNU assembler, *as,* you will find a lot of

ARM-specific information: opcodes, syntax, directives, and so on.

Downloading and installing the GNU toolchain

To download the GNU toolchain, visit the CodeSourcery download site (see Resources) and choose the IA32 GNU/Linux TAR file:

```
arm-2008q3-72-arm-none-linux-gnueabi-i686-pc-linux-gnu.tar.bz2
```

Versions of the GNU toolchain are available for all the major client operating systems, but this article focuses on installing and using the Lite version of the toolchain under Linux®.

Because the ARM design has progressed through different versions throughout its history, different C libraries for three of the most common versions of the processor design—ARM v4T, ARM v5T, and ARM 7—are included in the Lite package.

Next, use the `bunzip2` command to extract the file into your home directory.

Listing 1. Extracting the downloaded GNU toolchain

```
$ bunzip2 arm-2008q3-72-arm-none-linux-gnueabi-i686-pc-linux-gnu.tar.bz2
$ tar -xvf arm-2008q3-72-arm-none-linux-gnueabi-i686-pc-linux-gnu.tar
  .
  .
  .
  (Files listed while being extracted from the archive.)
  .
  .
  .
$
```

Now, modify your PATH environment variable to access the bin directory of the toolchain, and the tools are ready to use.

Configuring Linux to use the GNU toolchain

On the toolchain download page, you'll also find several useful PDF files—as well as a detailed Getting Started guide—that document the included tools. The instructions in this article are a condensed version that should get your toolchain up and running quickly.

If you do a lot more ARM programming than Intel® processor programming, an alternative method is to add symbolic links to your /usr/local/bin directory that reference the tools in the toolchain bin directory. In this way, you can use shortcuts like `as` to run the `arm-none-linux-gnueabi-as` command interactively. Listing 2 shows examples of how to set up these symbolic links.

Listing 2. Setting up symbolic links to ARM tools

```
# cd /usr/local/bin
# which arm-none-linux-gnueabi-as
/home/bzimmerly/Sourcery_G++_Lite/bin/arm-none-linux-gnueabi-as
# ln -s /home/bzimmerly/Sourcery_G++_Lite/bin/arm-none-linux-gnueabi-as as
# ls -l as
lrwxrwxrwx 1 bzimmerly bzimmerly 76 2009-03-13 02:48 as -> /home/bzimmerly
/Sourcery_G++_Lite/bin/arm-none-linux-gnueabi-as
# ./as --version
GNU assembler (Sourcery G++ Lite 2008q1-126) 2.18.50.20080215
Copyright 2007 Free Software Foundation, Inc.
This program is free software; you may redistribute it under the terms of
the GNU General Public License version 3 or later.
This program has absolutely no warranty.
This assembler was configured for a target of `arm-none-linux-gnueabi'.
#
```

Use the `which` command to locate the full path name of the command to make it easier to copy and paste it into the `ln` command. Then, use the `ln -s` command to create the symbolic link. After that, verify that the link was successfully created by listing it, then running it. By creating a simple symbolic link to all the tools in the GNU toolchain's bin directory, you won't have to type out the long names. Certainly, `as` is easier to type than `arm-none-linux-gnueabi-as` every time you want to run the assembler!

Writing a program for the ARM architecture

Many guides are available for writing ARM programs in the ever-popular C programming language, but few are written that address pure assembler. I'm going to break tradition for the purpose of this article and write an example program in pure assembler—something usually considered the "black art" of programming. It is a simple "Hello World" type of program targeted to a unique version of ARM Linux.

The example program described in this article is designed to run on the T-Mobile G1 mobile phone running Android Linux. It is generically written so that it should run on other ARM-based Linux platforms, as well (making only standard system calls). However, the minimal version of the Linux kernel that you'll need to use is version 2.6.16, which allows you to use the new Embedded Application Binary Interface (EABI) kernel system calls.

Note: To read a good article on programming the ARM for bare metal instead of under Linux, see the link to the Embedded.com article in Resources.

Using your favorite editor, create a script called *build* using the code in Listing 3. This script runs the GNU ARM assembler, followed by the linker. The goal in creating this program was to make it into a very tiny executable, so I stripped it of all debugging information with the linker switch `--strip-all`. After creating the script, make it executable by issuing the `chmod +x build` command.

Listing 3. Build the ARM Linux Hello World application

```
#!/bin/bash
echo Building the ARM Linux Hello World...
arm-none-linux-gnueabi-as -alh -o hw.o hw.S > hw.lst
arm-none-linux-gnueabi-ld --strip-all -o hw hw.o
```

After this, create the source module, called *hw.S,* and put the code in Listing 4 into it.

Listing 4. ARM Linux Hello World

```
@filename: hw.S

.text
.align 2
.global _start

@ ssize_t sys_write(unsigned int fd, const char * buf,
size_t count)
@         r7                 r0      r1                r2

_start:
    adr     r1, msg         @ Address
    mov     r0, #1          @ STDOUT
    mov     r2, #16         @ Length
    mov     r7, #4          @ sys_write
    svc     0x00000000

@ int sys_exit(int status)
@     r7        r0

    mov     r0, #0          @ Return code
    mov     r7, #1          @ sys_exit
    svc     0x00000000

.align 2

msg:
    .asciz "Hello Android!\n\n"
```

In GNU assembler parlance, the "at" sign (@) is used to designate line comments. The assembler ignores everything after the @ up to the end of the line.

The program uses two standard Linux system calls: `sys_write` and `sys_exit`. Above the assembler code in each of these calls is the C language equivalent of it in commented form. This makes it easier to see the how the ARM registers properly map into the calling parameters that the system calls use. The rule to remember is, parameters go left to right, `r0` to `r6`. Register `r7` is special in that this is where you place the system call number being used.

The `svc 0x00000000` instruction tells the ARM processor to call the "supervisor," which in this case is the Linux kernel.

Testing the ARM program

The toolchain provides the ever-popular GDB for debugging low-level programs. When the program is targeted for a single-board computer with a JTAG or ICE unit attached, you can use the Sourcery G++ Lite debugger (gdb) to debug the ARM code remotely.

If you wish to test the code as I did—on the Android Linux system running on a mobile phone—you need to attach the phone to the workstation using the USB cable that came with it, then use the Android software development kit's (SDK's) `adb push` command to transfer the program to the phone. Once on the phone, in a directory that can contain executable code (/data/local/bin), make the program executable by issuing the `chmod 555 hw` command. (The `chmod` command on Android doesn't use `+x`, so `555` is necessary, instead.)

Finally, use the `adb shell` command to connect to the phone, use `cd` to change to the correct directory, and run it with `./hw`. If all goes according to plan, the program should respond as it did on my phone, by greeting you with "Hello Android!"

Conclusion

If this small taste of ARM assembly programming interests you, feel free to learn more about this processor design from the links provided in Resources. The best resource for in-depth study of the ARM core is the Bible of ARM development, the *ARM ARM,* which is short for *ARM Architecture Reference Manual.*

For the career-minded, consider this: There are millions upon millions of mobile phones in the world today, with more being created each year. The state of the art has gotten to the point that we are actually able to carry around in our pockets a dual-core processor machine with multiple gigabytes of storage, fully networked into the Internet for instant information and entertainment. There is a crying demand among mobile phone vendors for talented programmers. With ARM as prevalent as it is, there's plenty of work to be done—and it's fun work, too! As always, feel free to have fun with the tools and enjoy yourself. Programming is part art, part science, and can be one of the most enjoyable careers in the business of technology.

Resources

Learn

- Visit the ARM Web site.
- Visit the GNU ARM Web site.
- From Embedded.com, check out the series on "Building Bare-Metal ARM Systems with GNU."
- Read about Qualcomm's dual-core SoC.
- In the developerWorks Linux zone, find more resources for Linux developers, and scan our most popular articles and tutorials.
- See all Linux tips and Linux tutorials on developerWorks.
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- Get the Sourcery G++ Lite package from the CodeSourcery download site.
- Download and read ARM ARM, the Bible of ARM programming.
- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Get involved in the My developerWorks community; with your personal profile and custom home page, you can tailor developerWorks to your interests and interact with other developerWorks users.

About the author

William B. Zimmerly
Bill Zimmerly is a knowledge engineer, a low-level systems programmer with expertise in various versions of UNIX and Microsoft Windows, and a free thinker who worships at the altar of Logic. Creating new technologies and writing about them are Bill's passions. He resides in rural Hillsboro, Missouri, where the air is fresh, the views are inspiring, and good wineries are all around.