Using QEMU for Embedded Systems Development, Part 1

By Manoj Kumar on June 1, 2011 in Coding, Developers · 16 Comments

Last month, we covered the basic use of QEMU. Now let's dig deeper into its abilities, looking at the embedded domain.

Techies who work in the embedded domain must be familiar with the ARM (Advanced RISC Machine) architecture. In the modern era, our lives have been taken over by mobile devices like phones, PDAs, MP3 players and GPS devices that use this architecture. ARM has cemented its place in the embedded devices market because of its low cost, lower power requirements, less heat dissipation and good performance.

Purchasing ARM development hardware can be an expensive proposition. Thankfully, the QEMU developers have added the functionality of emulating the ARM processor to QEMU. You can use QEMU for two purposes in this arena — to run an ARM program, and to boot and run the ARM kernel.

In the first case, you can run and test ARM programs without installing ARM OS or its kernel. This feature is very helpful and time-saving. In the second case, you can try to boot the Linux kernel for ARM, and test it.

Compiling QEMU for ARM

In the last article, we compiled QEMU for x86. This time let's compile it for ARM. Download the QEMU source, if you don't have it already. Extract the tarball, change to the extracted directory, configure and build it as follows:

```
$ tar -zxvf qemu-0.14.0.tar.gz
$ cd qemu-0.14.0
$ ./configure –target-list=arm-softmmu
$ make
$ su
# make install
```

You will find two output binaries, `qemu-arm` and `qemu-system-arm`, in the source code directory. The first is used to execute ARM binary files, and the second to boot the ARM OS.

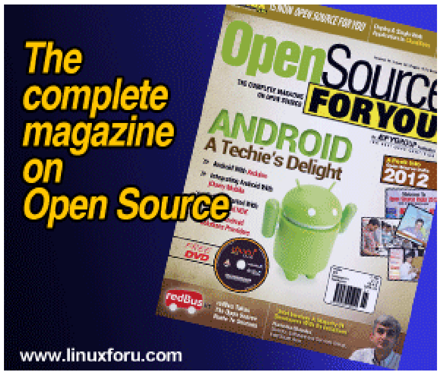
Obtaining an ARM tool-chain

Let's develop a small test program. Just as you need the x86 tool-chain to develop programs for Intel, you need the ARM tool-chain for ARM program development. You can download it from here.

Extract the archive's contents, and view a list of the available binaries:

```
$ tar -jxvf arm-2010.09-50-arm-none-linux-gnueabi-i686-pc-linux-gnu.tar.bz2
$ cd arm-2010.09/bin/
$ ls
-rwxr-xr-x 1 root root 569820 Nov 7 22:23 arm-none-linux-gnueabi-addr2line
-rwxr-xr-x 2 root root 593236 Nov 7 22:23 arm-none-linux-gnueabi-ar
-rwxr-xr-x 2 root root 1046336 Nov 7 22:23 arm-none-linux-gnueabi-as
-rwxr-xr-x 2 root root 225860 Nov 7 22:23 arm-none-linux-gnueabi-c++
-rwxr-xr-x 1 root root 572028 Nov 7 22:23 arm-none-linux-gnueabi-c++filt
-rwxr-xr-x 1 root root 224196 Nov 7 22:23 arm-none-linux-gnueabi-cpp
```

Search for: Search



Get Connected
RSS Feed Twitter

Submit

```
-rwxr-xr-x 1 root root 18612 Nov 7 22:23 arm-none-linux-gnueabi-elfedit
-rwxr-xr-x 2 root root 225860 Nov 7 22:23 arm-none-linux-gnueabi-g++
-rwxr-xr-x 2 root root 222948 Nov 7 22:23 arm-none-linux-gnueabi-gcc
```

Cross-compiling and running the test program for ARM

Now use the `arm-none-linux-gnueabi-gcc` tool to compile a test C program. Before proceeding, you should add the ARM tool-chain to your PATH:

```
# PATH=/(Your-path)/arm-2010.09/bin:$PATH
```

Create a small test program, `test.c`, with the basic "Hello world":

```
#include<stdio.h>
int main(){
    printf("Welcome to Open World\n");
}
```

Use the ARM compiler to compile this program:

```
# arm-none-linux-gnueabi-gcc test.c -o test
```

Once the file is compiled successfully, check the properties of the output file, showing that the output executable is built for ARM:

```
# file test
test: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically linked (uses shared libs), for (
```

Run the test program:

```
#qemu-arm -L /your-path/arm-2010.09/arm-none-linux-gnueabi/libc ./test
Welcome to Open World
```

While executing the program, you must link it to the ARM library. The option `-L` is used for this purpose.

Building the Linux kernel for ARM

So, you are now done with the ARM tool-chain and `qemu-arm`. The next step is to build the Linux kernel for ARM. The mainstream Linux kernel already contains supporting files and code for ARM; you need not patch it, as you used to do some years ago.

Download latest version of Linux from kernel.org (v2.6.37 as of this writing), and extract the tarball, enter the extracted directory, and configure the kernel for ARM:

```
# tar -jxvf linux-2.6.37.tar.bz2
# cd linux-2.6.37
# make menuconfig ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
```

Here, specify the architecture as ARM, and invoke the ARM tool-chain to build the kernel. In the configuration window, navigate to "Kernel Features", and enable "Use the ARM EABI to compile the kernel". (EABI is Embedded Application Binary Interface.) Without this option, the kernel won't be able to load your test program.

Modified kernel for u-boot

In subsequent articles, we will be doing lots of testing on u-boot — and for that, we need a modified kernel. The kernel `zImage` files are not compatible with u-boot, so let's use `uImage` instead, which is a kernel image with the header modified to support u-boot. Compile the kernel, while electing to build a `uImage` for u-boot. Once again, specify the architecture and use the ARM tool-chain:

```
# make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- uImage -s
  Generating include/generated/mach-types.h
arch/arm/mm/alignment.c: In function 'do_alignment':
arch/arm/mm/alignment.c:720:21: warning: 'offset.un' may be used uninitialized in this function
.
.
.
  Kernel: arch/arm/boot/Image is ready
  Kernel: arch/arm/boot/zImage is ready
Image Name:   Linux-2.6.37
Created:      Thu May  5 16:59:28 2011
Image Type:   ARM Linux Kernel Image (uncompressed)
Data Size:    1575492 Bytes = 1538.57 kB = 1.50 MB
Load Address: 00008000
Entry Point:  00008000
  Image arch/arm/boot/uImage is ready
```

After the compilation step, the `uImage` is ready. Check the file's properties:

```
# file arch/arm/boot /uImage
uImage: u-boot legacy uImage, Linux-2.6.37, Linux/ARM, OS Kernel Image (Not compressed), 1575
```

Popular Comments Tag cloud

August 13, 2013 · 46 Comments · Diksha P Gupta
'India has immense under-utilised talent in the cloud security space'

June 20, 2013 · 5 Comments · sophie-samuel
New and amazing features of Linux

June 20, 2013 · 3 Comments · Priyanka Sarkar
What it Takes to be an Open Source Expert

August 24, 2013 · 3 Comments · Priyanka Sarkar
Secure Your Career with 'Ethical Hacking'!


September 21, 2013 · 1 Comments · Mihalis-T
Getting Started with WireShark

Now test this image on QEMU; the result is shown in Figure 1:

```
# qemu-system-arm -M versatilepb -m 128M -kernel /home/manoj/Downloads/linux-2.6.37/arch/a
```

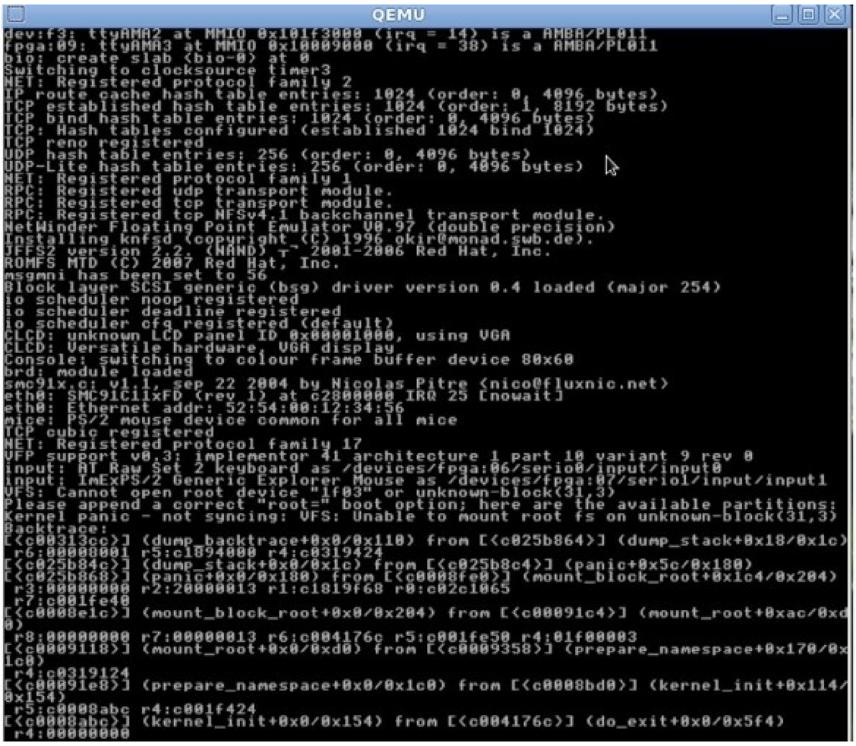


Figure 1: ARM kernel inside QEMU

The kernel will crash at the point where it searches for a root filesystem, which you didn't specify in the above command.

The next task is to develop a dummy filesystem for your testing. It's very simple — develop a small test C program `hello.c`, and use it to build a small dummy filesystem:

```c
#include<stdio.h>
int main(){
   while(1){
      printf("Hello Open World\n");
      getchar();
   }
```

The endless loop (while(1)) will print a message when the user presses a key. Compile this program for ARM, but compile it statically; as you are trying to create a small dummy filesystem, you will not use any library in it. In GCC, the `-static` option does this for you:

```
# arm-none-linux-gnueabi-gcc hello.c -static -o hello
```

Use the output file to create a root filesystem. The command `cpio` is used for this purpose. Execute the following command:

```
# echo hello | cpio -o --format=newc > rootfs
1269 blocks
```

Check the output file:

```
# file rootfs
rootfs: ASCII cpio archive (SVR4 with no CRC)
```

You now have a dummy filesystem ready for testing with this command:

```
# qemu-system-arm -M versatilepb -m 128M -kernel /home/manoj/Downloads/
linux-2.6.37/arch/arm/boot/uImage -initrd rootfs -append "root=/dev/ram rdinit=/hello"
```

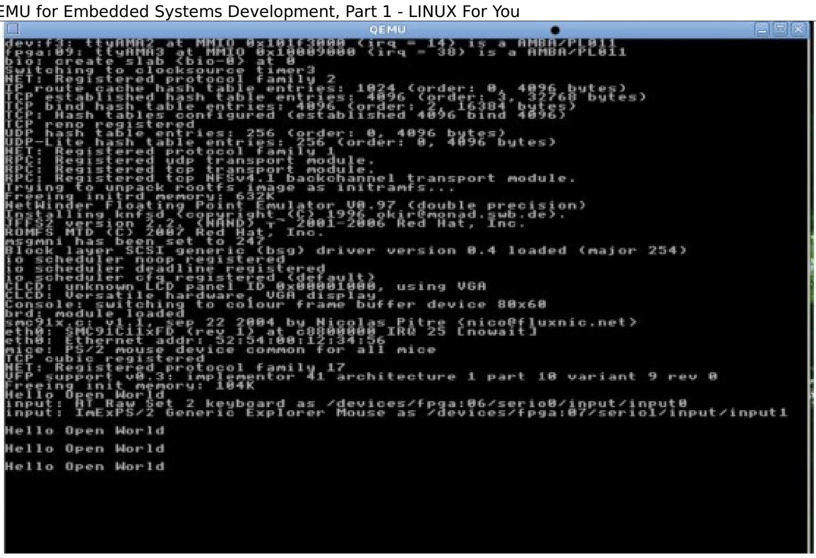


Figure 2: ARM kernel with a dummy filesystem

When the kernel boots, it mounts rootfs as its filesystem, and starts the hello program as init. So now you are able to run ARM programs, and boot the ARM kernel inside QEMU.

The next step would be to use u-boot on QEMU. An array of testing is ahead of us, which we will cover in a forthcoming article.

Related Posts:

- Using QEMU for Embedded Systems Development, Part 3
- Using QEMU for Embedded Systems Development, Part 2
- The Quick Guide to QEMU Setup
- Kernel Development & Debugging Using the Eclipse IDE
- Playing with User-mode Linux

Tags: ARM architecture, ARM library, arm processor, arm program, Cross compiler, development hardware, embedded systems, LFY June 2011, Linux kernel, machine architecture, QEMU, risc machine, SYSV, test program, tool chain, u-boot

Article written by:

Manoj Kumar

The author is a freelance developer and trainer. He leads a team in Linux kernel programming, Linux administration, cluster computing, embedded systems and QT/GTK programming on Linux. View and participate in the latest discussions on his Yahoo Group.

Connect with him: Website

Previous Post
Exploring Software: openSUSE Tumbleweed Rolling Distribution Goes Mainstream

Next Post
Best Practices in Network Security Monitoring

AROUND THE WEB

Citizens Over 50 May Qualify to Get $20,500 this Year Moneynews

How to Use the VIN Number for Value of a Used Car eHow

Lose Belly Fat With 6 Stand-Up Exercises Stack

Video: Watch Episodes 1 & 2 of the New Comedy Hit 'In ... Comedy Central

ALSO ON **LINUX FOR YOU**

Getting Started with WireSl 1 comment

The Semester Project-IV Fil Systems: Formatting a Pen

Cyber Attacks Explained: Th Botnet Army 1 comment

Learn the Art of Linux Troubleshooting 2 comments

16 comments

Leave a message...

Newest Community Share

Vinay Kumar • 5 months ago
Solution for vnc server running on 127.0.0.1:15900 and got hangs...

open vncviewer from command and type the address that is shown on the s
127.0.0.1:15900)
Reply Share ›

Kiran Koneri • 6 months ago
Hi Abhishek,

Followed the above steps as mentioned in this article.

When I try to test uImage on Qemu using this command and resonse as follo
qemu-system-arm -M versatilepb -m 128M -kernel /home/kiran/linux-3.8.2/arch/arm/boot/uImage

VNC server running on `127.0.0.1:5900'
qemu: hardware error: pl011_read: Bad offset d60

CPU #0:
R00=00000000 R01=00000000 R02=00000000 R03=00000000
R04=00000000 R05=00000000 R06=101f3d60 R07=00000000
R08=00000000 R09=00000000 R10=00000000 R11=00000000
R12=00000000 R13=00000000 R14=00000000 R15=00000004
PSR=400001db -Z-- A und32
Aborted (core dumped)

Gone through many blogs but could not able to succeed.
Thanks in advance,
1 Reply Share ›

Bin Wang • 10 months ago
Hi, I'm wonder what is "-M versatilepb" stand for? I don't find it on qemu's hel
message. Thanks!
Reply Share ›

fila • 10 months ago
./configure –target-list=arm-softmmu

when executing make command i got the following error.
Makefile:24: no file name for `-include'
make[1]: *** No rule to make target `vl.o', needed by `all'. Stop.
make: *** [subdir-libhw32] Error 2

How i can resolve this issue?Please help me.
Reply Share ›

ashok • a year ago
/configure --target-list=arm-linux-user when i execute this command it is shc
Error: zlib check failed
Make sure to have the zlib libs and headers installed.
how i can solve this issue please give me sugg
Reply Share ›

Abhishek Dwivedi > ashok • a year ago
#apt-get install zlib1g-dev
will resolve your error.
1 Reply Share ›

Abhishek Dwivedi • a year ago
If you are miss to get qemu-arm installed. Just add arm-linux-user in qemu co
and compile,
$./configure --target-list=arm-linux-user

$make
$ make install
3 Reply Share ›

Rupesh KP • a year ago

When I try to run the executable it gives me command not found error:
qemu-arm -L /home/netuser/el/el_test/arm-2010q1/arm-none-linux-gnueabi/li

If 'qemu-arm' is not a typo you can use command-not-found to lookup the pa
that contains it, like this:
cnf qemu-arm

How to resolve this issue?

△ | ▽ Reply Share ›

Abhishek Dwivedi > Rupesh KP • a year ago

check above post

2 △ | ▽ Reply Share ›

Daredevil Vivek • a year ago

qemu-arm not installed can any one help me

△ | ▽ Reply Share ›

vivek • a year ago

can any one help me adding the PATH.
i get this error arm-none-linux-gnueabi-gcc: command not found

△ | ▽ Reply Share ›

jebin • 2 years ago

hi...i'm new to embedded linux field. i am trying to emulate arm develpmnt bo
Qemu and boot linux on to it. i compiled the kernel and got the uImage.but w
this image on qemu it shows--
qemu: fatal: Trying to execute code outside RAM or ROM at 0x50008000
R00=00000000 R01=00000183 R02=00000100 R03=00000000 R04=0
R05=00000000 R06=00000000 R07=00000000 R08=00000000 R09=0
R10=00000000 R11=00000000 R12=00000000 R13=00000000 R14=0
R15=50008000 PSR=400001d3 -Z-- A svc32 Aborted.
i used this command for testing> qemu-system-arm -M versatilepb -m 128M
/root/ls6410/kernel/s3c-linux-2.6.28.6-Real6410/arch/arm/boot/uImage.
can you suggest an method to fix this?
thanks,

△ | ▽ Reply Share ›

vamsidhar • 2 years ago

i follow the same steps but qemu-arm not generated to execute binary files

△ | ▽ Reply Share ›

Javier Fileiv > vamsidhar • 5 months ago

check the post above! ;)

△ | ▽ Reply Share ›

linux > Javier Fileiv • 2 months ago

i used this command for testing> qemu-system-arm -M versatil
128M -kernel my_path/arch/arm/boot/uImage.
i get a blank screen with no prints...plz suggest how to overcor

△ | ▽ Reply Share ›

JimHarris > linux • a month ago

I used the command line below and I got output.
qemu-system-arm -M versatilepb -kernel /home/jharris/linu
2.6.39/arch/arm/boot/uImage -initrd rootfs -append "root
rdinit=/hello console=ttyAMA0" -nographic

△ | ▽ Reply Share ›

Reviews How-Tos Coding Interviews Features Overview Blogs

Search

For You & Me
Developers

Popular tags

Linux, ubuntu, Java, MySQL, Google, python, Android, Fedora, PHP, C, html, web applications, India, Microsoft, unix, Windows, Red Hat, Oracle, Security, Apache, xml, LFY April 2012, FOSS, GNOME, http, JavaScript, LFY June 2011, open source, RAM, operating systems

Sysadmins
Open Gurus
CXOs
Columns

All published articles are released under Creative Commons Attribution-NonCommercial 3.0 Unported License, unless otherwise noted.
LINUX For You is powered by WordPress, which gladly sits on top of a CentOS-based LEMP stack.

Write For Us | Submit Tips | Subscribe to Print Edition | Contact Us

Search

LINUX For You

HOME REVIEWS ↓ HOW-TOS ↓ CODING INTERVIEWS FEATURES OVERVIEW ↓ BLOGS SERIES ↓ IT ADMIN

Using QEMU for Embedded Systems Development, Part 2

By Manoj Kumar on July 1, 2011 in Coding, Developers · 3 Comments

In the previous articles, we learnt how to use QEMU for a generic Linux OS installation, for networking using OpenVPN and TAP/TUN, for cross-compilation of the Linux kernel for ARM, to boot the kernel from QEMU, and how to build a small filesystem and then mount it on the vanilla kernel. Now we will step out further.

First of all, I would like to explain the need for a bootloader. The bootloader is code that is used to load the kernel into RAM, and then specify which partition will be mounted as the root filesystem. The bootloader resides in the MBR (Master Boot Record). In general-purpose computing machines, an important component is the BIOS (Basic Input Output System). The BIOS contains the low-level drivers for devices like the keyboard, mouse, display, etc. It initiates the bootloader, which then loads the kernel. Linux users are very familiar with boot-loaders like GRUB (Grand Unified Boot-Loader) and LILO (Linux Loader).

Micro-controller programmers are very familiar with the term “Bare-Metal Programming”. It means that there is nothing between your program and the processor — the code you write runs directly on the processor. It becomes the programmer’s responsibility to check each and every possible condition that can corrupt the system.

Now, let us build a small program for the ARM Versatile Platform Baseboard, which will run on the QEMU emulator, and then print a message on the serial console. Downloaded the tool-chain for ARM EABI from here. As described in the previous article, add this tool-chain in your PATH.

By default, QEMU redirects the serial console output to the terminal, when it is initialised with the `nographic` option:

```
$  qemu-system-arm --help | grep nographic
-nographic      disable graphical output and redirect serial I/Os to console. When using -nographic
```

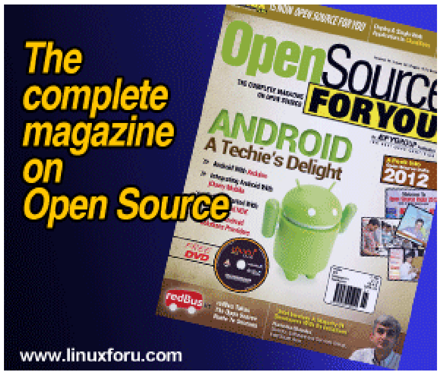
We can make good use of this feature; let’s write some data to the serial port, and it can be a good working example.

Before going further, we must make sure which processor the GNU EABI tool-chain supports, and which processor QEMU can emulate. There should be a similar processor supported by both the tool-chain and the emulator. Let’s check first in QEMU. In the earlier articles, we compiled the QEMU source code, so use that source code to get the list of the supported ARM processors:

```
$ cd  (your-path)/qemu/qemu-0.14.0/hw
$ grep  "arm" versatilepb.c
#include "arm-misc.h"
static struct arm_boot_info versatile_binfo;
cpu_model = "arm926";
```

It’s very clear that the “arm926” is supported by QEMU. Let’s check its availability in the GNU

Search for: Search



Get Connected

RSS Feed Twitter

Inside the market

ARM tool-chain:

```
$ cd (your-path)/CodeSourcery/Sourcery_G++_Lite/share/doc/arm-arm-none-eabi/info
$ cat gcc.info | grep arm | head -n 20
    .
    .
`strongarm1110', `arm8', `arm810', `arm9', `arm9e', `arm920',
`arm920t', `arm922t', `arm946e-s', `arm966e-s', `arm968e-s',
`arm926ej-s', `arm940t', `arm9tdmi', `arm10tdmi', `arm1020t',
`arm1026ej-s', `arm10e', `arm1020e', `arm1022e', `arm1136j-s',
```

Great!! The ARM926EJ-S processor is supported by the GNU ARM tool-chain. Now, let's write some data to the serial port of this processor. As we are not using any header file that describes the address of UART0, we must find it manually, from the file `(your-path)/qemu/qemu-0.14.0/hw/versatilepb.c`:

```
/* 0x101f0000 Smart card 0.  */
/*  0x101f1000 UART0.  */
/*  0x101f2000 UART1.  */
/*  0x101f3000 UART2.  */
```

Open source code is so powerful, it gives you each and every detail. UART0 is present at address `0x101f1000`. For testing purposes, we can write data directly to this address, and check output on the terminal.

Our first test program is a bare-metal program running directly on the processor, without the help of a bootloader. We have to create three important files. First of all, let us develop a small application program (`init.c`):

```
volatile unsigned char * const UART0_PTR = (unsigned char *)0x0101f1000;
void display(const char *string){
    while(*string != '\0'){
        *UART0_PTR = *string;
        string++;
    }
}

int my_init(){
    display("Hello Open World\n");
}
```

Let's run through this code snippet.

First, we declared a volatile variable pointer, and assigned the address of the serial port (UART0). The function `my_init()`, is the main routine. It merely calls the function `display()`, which writes a string to the UART0.

Engineers familiar with base-level micro-controller programming will find this very easy. If you are not experienced in embedded systems programming, then you can stick to the basics of digital electronics. The microprocessor is an integrated chip, with input/output lines, different ports, etc. The ARM926EJ-S has four serial ports (information obtained from its data-sheet); and they have their data lines (the address). When the processor is programmed to write data to one of the serial ports, it writes data to these lines. That's what this program does.

The next step is to develop the startup code for the processor. When a processor is powered on, it jumps to a specified location, reads code from that location, and executes it. Even in the case of a reset (like on a desktop machine), the processor jumps to a predefined location. Here's the startup code, `startup.s`:

```
.global _Start
_Start:
LDR sp, = sp_top
BL my_init
B .
```

In the first line, `_Start` is declared as global. The next line is the beginning of `_Start`'s code. We set the address of the stack to `sp_top`. (The instruction LDR will move the data value of `sp_top` in the stack pointer (`sp`). The instruction BL will instruct the processor to jump to `my_init` (previously defined in `init.c`). Then the processor will step into an infinite loop with the instruction `B .`, which is like a `while(1)` or `for(;;)` loop. If we don't do this, our system will crash. The basics of embedded systems programming is that our code should run into an infinite loop.

Now, the final task is to write a linker script for these two files (`linker.ld`):

```
ENTRY(_Start)
SECTIONS
{
. = 0x10000;
startup : { startup.o(.text)}
.data : {*(.data)}
.bss : {*(.bss)}
. = . + 0x500;
sp_top = .;
}
```

Popular Comments Tag cloud

August 13, 2013 · 46 Comments · Diksha P Gupta
'India has immense under-utilised talent in the cloud security space'

June 20, 2013 · 5 Comments · sophie-samuel
New and amazing features of Linux

June 20, 2013 · 3 Comments · Priyanka Sarkar
What it Takes to be an Open Source Expert

August 24, 2013 · 3 Comments · Priyanka Sarkar
Secure Your Career with 'Ethical Hacking'!

September 21, 2013 · 1 Comments · Mihalis-T
Getting Started with WireShark

The first line tells the linker that the entry point is `_Start` (defined in `startup.s`). As this is a basic program, we can ignore the *Interrupts* section. The QEMU emulator, when executed with the `-kernel` option, starts execution from the address `0x10000`, so we must place our code at this address. That's what we have done in Line 4. The section "SECTIONS", defines the different sections of a program.

In this, `startup.o` forms the text (code) part. Then comes the subsequent data and the bss part. The final step is to define the address of the stack pointer. The stack usually grows downward, so it's better to give it a safe address. We have a very small code snippet, and can place the stack at `0x500` ahead of the current position. The variable `sp_top` will store the address for the stack.

We are now done with the coding part. Let's compile and link these files. Assemble the `startup.s` file with:

```
$ arm-none-eabi-as  -mcpu=arm926ej-s startup.s -o startup.o
```

Compile `init.c`:

```
$ arm-none-eabi-gcc -c -mcpu=arm926ej-s init.c -o init.o
```

Link the object files into an ELF file:

```
$ arm-none-eabi-ld -T linker.ld init.o startup.o -o output.elf
```

Finally, create a binary file from the ELF file:

```
$ arm-none-eabi-objcopy -O binary output.elf output.bin
```

The above instructions are easy to understand. All the tools used are part of the ARM tool-chain. Check their help/man pages for details.

After all these steps, finally we will run our program on the QEMU emulator:

```
$ qemu-system-arm -M versatilepb -nographic -kernel output.bin
```

The above command has been explained in previous articles (1, 2), so we won't go into the details. The binary file is executed on QEMU and will write the message "Hello Open World" to UART0 of the ARM926EJ-S, which QEMU redirects as output in the terminal.

Acknowledgement

This article is inspired by the following blog post: "Hello world for bare metal ARM using QEMU".

Related Posts:

- Using QEMU for Embedded Systems Development, Part 3
- Using QEMU for Embedded Systems Development, Part 1
- The Quick Guide to QEMU Setup
- Kernel Development & Debugging Using the Eclipse IDE
- Building Image Processing Embedded Systems using Python, Part 3

Tags: ARM926EJ-S processor, bare-metal program, Bare-Metal Programming, binary file, bootloader, digital electronics, ELF, embedded systems, embedded systems programming, integrated chip, LFY July 2011, Linux kernel, MBR, openvpn, QEMU, root filesystem, serial ports

Article written by:

Manoj Kumar

The author is a freelance developer and trainer. He leads a team in Linux kernel programming, Linux administration, cluster computing, embedded systems and QT/GTK programming on Linux. View and participate in the latest discussions on his Yahoo Group.

Connect with him: Website

Previous Post
The Nagios Setup Explained

Next Post
WordPress Multi-site Servers on Production Machines

Find us on Facebook

Open Source For You
Like

259,728 people like Open Source For You.

Facebook social plugin

Reviews How-Tos Coding Interviews Features Overview Blogs

Search

Popular tags

Linux, ubuntu, Java, MySQL, Google, python, Android, Fedora, PHP, C, html, web applications, India, Microsoft, unix, Windows, Red Hat, Oracle, Security, Apache, xml, LFY April 2012, FOSS, GNOME, http, JavaScript, LFY June 2011, open source, RAM, operating systems

For You & Me
Developers
Sysadmins
Open Gurus
CXOs
Columns

All published articles are released under Creative Commons Attribution-NonCommercial 3.0 Unported License, unless otherwise noted.
LINUX For You is powered by WordPress, which gladly sits on top of a CentOS-based LEMP stack.

Using QEMU for Embedded Systems Development, Part 3

By Manoj Kumar on August 1, 2011 in Coding, Developers · 2 Comments

This is the last article of this series on QEMU. In the previous article, we worked on bare-metal programming, and discussed the need for a bootloader. Most GNU/Linux distros use GRUB as their boot-loader (earlier, LILO was the choice). In this article, we will test the famous U-Boot (Universal BootLoader).

In embedded systems, especially in mobile devices, ARM processor-based devices are leading the market. For ARM, U-Boot is the best choice for a bootloader. The good thing about it is that we can use it for different architectures like PPC, MIPS, x86, etc. So let's get started.

Download and compile U-Boot

U-Boot is released under a GPL licence. Download it from this FTP server, which has every version of U-Boot available. For this article, I got version 1.2.0 (`u-boot-1.2.0.tar.bz2`). Extract the downloaded tar ball and enter the source code directory:

```
# tar -jxvf u-boot-1.2.0.tar.bz2
# cd u-boot-1.2.0
```

To begin, we must configure U-Boot for a particular board. We will use the same ARM Versatile Platform Baseboard (`versatilepb`) we used in the previous article, so let's run:

```
# make versatilepb_config arch=ARM CROSS_COMPILE=arm-none-eabi-
Configuring for versatile board...
Variant:: PB926EJ-S
```

After configuration is done, compile the source code:

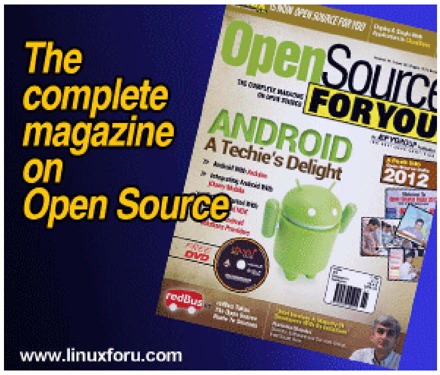
```
# make all arch=ARM CROSS_COMPILE=arm-none-eabi-
for dir in tools examples post post/cpu ; do make -C $dir _depend ; done
make[1]: Entering directory `/root/qemu/u-boot-1.2.0/tools'
ln -s ../common/environment.c environment.c
.
.
G++_Lite/bin/../lib/gcc/arm-none-eabi/4.4.1 -lgcc \
          -Map u-boot.map -o u-boot
arm-none-eabi-objcopy --gap-fill=0xff -O srec u-boot u-boot.srec
arm-none-eabi-objcopy --gap-fill=0xff -O binary u-boot u-boot.bin
```

Find the size of the compiled U-Boot binary file (around 72 KB in my experience) with `ls -lh u-boot*` — we will use it later in this article. I assume that you have set up QEMU, networking and the ARM tool chain, as explained in previous articles in this series (1, 2, 3). If not, then I suggest you read the last three articles.

Boot U-Boot in QEMU

Now we can boot the U-Boot binary in QEMU, which is simple. Instead of specifying the Linux kernel as the file to boot in QEMU, use the U-Boot binary:

Search for: Search



Get Connected

RSS Feed Twitter

```
# qemu-system-arm -M versatilepb -nographic -kernel u-boot.bin
```

Run some commands in U-Boot, to check if it is working:

```
Versatile # printenv
bootargs=root=/dev/nfs mem=128M ip=dhcp netdev=25,0,0xf1010000,0xf1010010,eth0
bootdelay=2
baudrate=38400
bootfile="/tftpboot/uImage"
stdin=serial
stdout=serial
stderr=serial
verify=n
Environment size: 184/65532 bytes
```

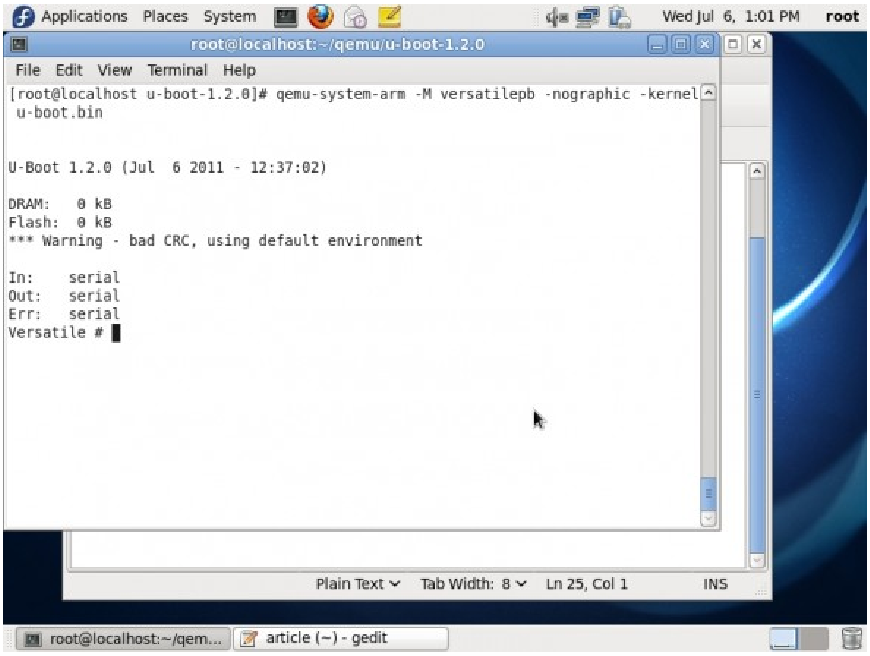


Figure 1: U-Boot

The next step is to boot a small program from U-Boot. In the previous article, we wrote a small bare-metal program — so let us use that.

We will create a flash binary image that includes `u-boot.bin` and the bare-metal program in it. The test program from the last article will be used here again with some modification. As the `u-boot.bin` size is around 72 KB, we will move our sample program upward in memory. In the linker script, change the starting address of the program:

```
ENTRY(_Start)
SECTIONS
{
. = 0x100000;
startup : { startup.o(.text)}
.data : {*(.data)}
.bss : {*(.bss)}
. = . + 0x500;
sp_top = .;
}
```

Compile the test program as shown below:

```
# arm-none-eabi-gcc -c -mcpu=arm926ej-s init.c -o init.o
# arm-none-eabi-as -mcpu=arm926ej-s startup.s -o startup.o
# arm-none-eabi-ld -T linker.ld init.o startup.o -o test.elf
# arm-none-eabi-objcopy -O binary test.elf test.bin
```

Now, our test program's binary file and the `u-boot.bin` must be packed in a single file. Let's use the `mkimage` tool for this; locate it in the U-Boot source-code directory.

```
# mkimage -A arm -C none -O linux -T kernel -d test.bin -a 0x00100000 -e 0x00100000 test.uimg
Image Name:
Created:     Wed Jul 6 13:29:54 2011
Image Type:   ARM Linux Kernel Image (uncompressed)
Data Size:    148 Bytes = 0.14 kB = 0.00 MB
Load Address: 0x00100000
Entry Point:  0x00100000
```

Our sample binary file is ready. Let's combine it with `u-boot.bin` to create the final flash image file:

```
#cat u-boot.bin test.uimg > flash.bin
```

Calculate the starting address of the test program in the `flash.bin` file:

Popular Comments Tag cloud
August 13, 2013 · 46 Comments · Diksha P Gupta
'India has immense under-utilised talent in the cloud security space'
June 20, 2013 · 5 Comments · sophie-samuel
New and amazing features of Linux
June 20, 2013 · 3 Comments · Priyanka Sarkar
What it Takes to be an Open Source Expert
August 24, 2013 · 3 Comments · Priyanka Sarkar
Secure Your Career with 'Ethical Hacking'!
September 21, 2013 · 1 Comments · Mihalis-T
Getting Started with WireShark

```
# printf "0x%X" $(expr $(stat -c%s u-boot.bin) + 65536)
0x21C68
```

Boot the flash image in QEMU:

```
# qemu-system-arm -M versatilepb -nographic -kernel flash.bin
```

Now verify the image address in U-Boot:

```
Versatile # iminfo 0x21C68
## Checking Image at 00021c68 ...
Image Name:
Image Type:   ARM Linux Kernel Image (uncompressed)
Data Size:    136 Bytes =  0.1 kB
Load Address: 00100000
Entry Point:  00100000
Verifying Checksum ... OK
```

The image is present at the address `0x21C68`. Boot it by executing the `bootm` command:

```
Versatile # bootm 0x21C68
## Booting image at 00021c68 ...
Image Name:
Image Type:   ARM Linux Kernel Image (uncompressed)
Data Size:    148 Bytes =  0.1 kB
Load Address: 00100000
Entry Point:  00100000
OK

Starting kernel ...

Hello Open World
```

That's all folks!

Acknowledgement

This article is inspired by the following blog post: "U-boot for ARM on QEMU".

Related Posts:

- Using QEMU for Embedded Systems Development, Part 1
- Using QEMU for Embedded Systems Development, Part 2
- The Quick Guide to QEMU Setup
- Install Linux from USB on System without BIOS Support for USB Boot
- Kernel Development & Debugging Using the Eclipse IDE

Tags: ARM, arm processor, bare-metal program, binary file, BIOS, boot arm, bootloader, embedded programming, embedded systems, grub, LFY August 2011, LILO, MIPS, mkimage, mobile devices, QEMU, tool chain, u-boot

Article written by:

Manoj Kumar

The author is a freelance developer and trainer. He leads a team in Linux kernel programming, Linux administration, cluster computing, embedded systems and QT/GTK programming on Linux. View and participate in the latest discussions on his Yahoo Group.

Connect with him: **Website**

Previous Post
The Comprehensive LAMP Guide -- Part 2 (MySQL)

Next Post
Setup Two-Factor Authentication using OpenOTP

Find us on Facebook

Open Source For You

Like

259,728 people like Open Source For You.

Facebook social plugin

Search

Popular tags

Linux, ubuntu, Java, MySQL, Google, python, Android, Fedora, PHP, C, html, web applications, India, Microsoft, unix, Windows, Red Hat, Oracle, Security, Apache, xml, LFY April 2012, FOSS, GNOME, http, JavaScript, LFY June 2011, open source, RAM, operating systems

All published articles are released under Creative Commons Attribution-NonCommercial 3.0 Unported License, unless otherwise noted.
LINUX For You is powered by WordPress, which gladly sits on top of a CentOS-based LEMP stack.