Simple Embedded Linux System

Vincent Sanders
Daniel Silverstone

Revision History

Revision 1.00	5th March 2009 Initial Release.	VRS, DJAS

Table of Contents

This article describes how to construct a simple Linux-based embedded system.

1. Introduction

Constructing an embedded system with Linux is often seen as a complex undertaking. This article is the first in a series which will show the fundamental aspects of constructing such systems and enable the reader to apply this knowledge to their specific situation.

This article covers the construction of the most basic system possible which will provide a command shell on the console.

This article, and indeed the whole series, assumes a basic understanding of a Linux-based operating system. While discussing concepts and general approaches these concepts are demonstrated with extensive practical examples. All the practical examples are based upon a Debian- or Ubuntu-based distribution.

2. What is an embedded system?

The term “Embedded System” has been applied to such a large number of systems that its meaning has become somewhat ill-defined. The term has been applied to everything from 4 bit micro controller systems to huge industrial control systems.

The context in which we are using the term here is to refer to systems where the user is limited to a specific range of interaction with a limited number of applications (typically one). Thus from the whole spectrum of applications which a general purpose computer can run a very narrow selection is made by the creator of the embedded system software.

It should be realized that the limits of interaction with a system may involve hardware as well as software. For example, if a system is limited to a keypad with only the digits 0 to 9, user interaction will more constrained than if the user had access to a full 102 key keyboard.

In addition to the limiting of user interaction, there may also be limits on the system resources available. Such limits are typically imposed by a systems cost, size or environment. However wherever possible these limits should be arrived at with as much knowledge of the system requirements as possible. Many projects fail unnecessarily because an arbitrary limit has been set which makes a workable solution unachievable. An example of this would be the selection of a system's main memory size before the application's memory requirements have been determined.

3. What do you want to achieve?

> A project must have a clearly defined goal.

This may be viewed as a statement of the obvious but it bears repeating as for some unfortunately inexplicable reason, embedded systems seem to suffer from poorly-defined goals.

An “embedded” project, like any other, should have a clear statement of what must be achieved to be declared a success. The project brief must contain *all the requirements* and, in addition, a list of “desirable properties”. It is essential that the two should not be confused; e.g. if the product *must* fit inside a 100mm by 80mm enclosure that is a *requirement*. However, a statement that the lowest cost *should* be achieved is a *desirable item* whereas a *fixed* upper cost would be a *requirement*.

If information necessary to formulate a requirement is not known then it should be kept as a “desirable item” couched in terms of that unknown information. It may be possible that once that information is determined, a requirement can be added.

It is, again, self-evident that any project plan must be flexible enough to cope with changes to requirements, but it must be appreciated that such changes may have a huge impact on the whole project and, indeed, may invalidate central decisions which have already been made.

General IT project management is outside the scope of this article. Fortunately there exist many good references on this topic.

Requirements which might be added to a project brief based on the assumptions of this article are:

The system software will be based upon a Linux kernel.
The system software will use standard Unix-like tools and layout.

The implications of these statements mean the chosen hardware should have a Linux kernel port available and must have sufficient resources to run the chosen programs.

Another important consideration is what kind of OS the project warrants. For example, if you have a project *requirement* of in-field updates then you may want to use a full OS with package management such as Debian GNU/Linux or Fedora. Such a requirement would, however, imply a need for a non-flash-based storage medium such as a hard disc for storing the OS, as these kinds of systems are typically very large (even in minimal installations) and not designed with the constraints of flash-based storage in mind. However, given that additional wrinkle, using an extant operating system can reduce software development costs significantly.

4. Anatomy of a Linux-based system

Much has been written on how Linux-based systems are put together; however a brief review is in order, to ensure that basic concepts are understood.

To be strictly correct the term “Linux” refers only to the kernel. Various arguments have been made as to whether the kernel constitutes an Operating System (OS) in its entirety, or that the term should refer to the whole assemblage of software that makes up the system. We use the latter interpretation here.

The general steps when any modern computer is turned on or reset is:

- The CPU (or designated boot CPU on multi-core/processor systems) initialises its internal hardware state, loads microcode etc.
- The CPU commences execution of the initial boot code e.g. the BIOS on x86 or the boot-loader on ARM.
- The boot code loads and executes the kernel. However, it is worth noting that x86 systems generally use the BIOS to load an intermediate loader such as GRUB or syslinux which then fetches and starts the kernel.
- The kernel configures the hardware and executes the **init** process.
- The **init** process executes other processes to get all the required software running.

The kernel's role in the system is to provide a generic interface to programs and arbitrate access to resources. Each program running on the system is called a process. Each operates as if it were the only process running. The kernel completely insulates a program from the implementation details of physical memory layout, peripheral access, networking etc.

The first process executed is special in that it is not expected to exit and is expected to perform some basic housekeeping tasks to keep a system running. Except in very specific circumstances the **init** process is provided by a program named `/sbin/init` which behaves correctly. The **init** process typically starts a shell script at boot to execute additional programs.

Some projects have chosen to run their primary application as the **init** process, while this is possible it is not recommended as a such a program is exceptionally difficult to debug and control. A programming bug in the application halts the system and provides no way to debug the issue.

One feature of almost all Unix-like systems is the shell, an interactive command parser. Most common shells have the Bourne shell syntax.

5. A simple beginning

We shall now consider creating a minimal system. The approach taken here requires no additional hardware beyond the host PC and the absolute minimum of additional software.

As already mentioned, these examples assume a Debian or Ubuntu host system. To use the QEMU emulator for testing the host system *must* be supported by QEMU as a target. An example where this might not be the case is where the target system is x86-64, which QEMU does not support.

To ease construction of the examples we will use the kernel's initramfs support. An initramfs is a gzip-compressed cpio archive of a file system which is unpacked into a RAM disc at kernel initialisation. A slight difference to normal system start-up is that while the first process executed must still be called `init`, it must be in the root of the file system. We will use the `/init` script to create some symbolic links and device nodes before executing the more-typical `/sbin/init` program.

This example system will use a program called busybox, which provides a large number of utilities in a single executable including a shell and an **init** process. Busybox is used extensively to build embedded systems of many types.

The `busybox-static` package is required to obtain pre-built copy of the **busybox** binary and the `qemu` package is required to test the constructed images. These may be obtained by executing:

```
$ sudo apt-get install busybox-static qemu
```

As mentioned, our initramfs-based approach requires a small `/init` script. This configures some basic device nodes and directories, mounts the special `/sys` and `/proc` file systems and starts the processing of hotplug events using **mdev**.

```
#!/bin/sh

# Create all the busybox symbolic links
/bin/busybox --install -s

# Create base directories
[ -d /dev ] || mkdir -m 0755 /dev
[ -d /root ] || mkdir --mode=0700 /root
[ -d /sys ] || mkdir /sys
[ -d /proc ] || mkdir /proc
[ -d /tmp ] || mkdir /tmp
mkdir -p /var/lock

# Mount essential filesystems
mount -t sysfs none /sys -onodev,noexec,nosuid
mount -t proc none /proc -onodev,noexec,nosuid

# Create essential filesystem nodes
mknod /dev/zero c 1 5
```

```
mknod /dev/null c 1 3

mknod /dev/tty c 5 0
mknod /dev/console c 5 1
mknod /dev/ptmx c 5 2

mknod /dev/tty0 c 4 0
mknod /dev/tty1 c 4 1

echo "/sbin/mdev" > /proc/sys/kernel/hotplug

echo "Creating devices"
/sbin/mdev -s

exec /sbin/init
```

To construct the cpio archive, the following commands should be executed in a shell. Note however that *INITSCRIPT* should be replaced with the location of the above script.

```
$ mkdir simple
$ cd simple
$ mkdir -p bin sbin usr/bin usr/sbin
$ cp /bin/busybox bin/busybox
$ ln -s busybox bin/sh
$ cp INITSCRIPT init
$ chmod a+x init
$ find . | cpio --quiet -o -H newc | gzip >../simple.gz
$ cd ..
```

To test the constructed image use a command like:

```
$ qemu -kernel /boot/vmlinuz-2.6.26-1-686 -initrd simple.gz \
             -append "root=/dev/ram" /dev/zero
```

This should present a QEMU window where the OS you just constructed boots and displays the message “Please press Enter to activate this console”. Press enter and you should be presented with an interactive shell from which you can experiment with the commands busybox provides. This environment is executing entirely from a RAM disc and is completely volatile. As such, any changes you make will not persist when the emulator is stopped.

6. Booting a real system

Starting the image under emulation proves the image ought to work on a real system, but there is no substitute for testing on real hardware. The `syslinux` package allows us to construct bootable systems for standard PCs on DOS formatted storage.

A suitable medium should be chosen to boot from, e.g. a DOS formatted floppy disc or a DOS-formatted USB stick. The DOS partition of the USB stick must be marked bootable. Some USB sticks might need repartitioning and reformatting with the Linux tools in order to work correctly.

The **syslinux** program should be run on the device `/dev/fd0` for a floppy disk or something similar to `/dev/sdx1` for a USB stick. Care must be taken as selecting the wrong device name might overwrite your host systems hard drive.

The target device should then be mounted and the kernel and the `simple.gz` file copied on.

The syslinux loader can be configured using a file called `syslinux.cfg` which would look something like:

```
default simple
timeout 100
```

```
prompt 1

label simple
  kernel vmlinuz
  append initrd=simple root=/dev/ram
```

The complete command sequence to perform these actions, substituting file locations as appropriate, is:

```
$ sudo syslinux -s /dev/sdd1
$ sudo mount -t vfat -o shortname=mixed  /dev/sdd1 /mnt/
$ cd /mnt
$ sudo cp /boot/vmlinuz-2.6.26-1-686 VMLINUZ
$ sudo cp simple.gz SIMPLE
$ sudo cp syslinux.cfg SYSLINUX.CFG
$ cd /mnt
$ sudo umount /mnt
```

The device may now be removed and booted on an appropriate PC. The PC should boot the image and present a prompt exactly the same way the emulator did.

7. What's next?

This first step, while simple, provides a complete Operating System and demonstrates that constructing an embedded system can be a straightforward process.

The next step is to expand this simple example to encompass a specific application which will be covered in the next article.

8. About the authors

Vincent Sanders	Vincent is the senior software engineer at Simtec Electronics and has extensive experience in the computer industry. He has worked on projects from large fault tolerant systems through accounting software to right down to programmable logic systems. He is an active developer for numerous open source projects including the Linux kernel and is also a Debian developer.
Daniel Silverstone	Daniel is a software engineer at Simtec Electronics and has experience in architecting robust systems. He develops software for a large number of Open source projects, contributes to the Linux kernel and is both an Ubuntu and Debian developer.
Simtec Electronics [http://www.simtec.co.uk]	Simtec is a full solutions provider with a proven track record of helping clients with all aspects of a project, from initial concept and design through to manufacturing finished product. With 20 years in the industry, and producing ARM CPU modules since 1992, Simtec's wide experience in embedded systems and the Linux kernel provide a strong base on which to build custom hardware and software solutions, from the smallest of USB devices to the largest complex Linux systems. Simtec's custom-off-the-shelf design service, utilising a range of pre-designed modules of various functions, allows for rapid design and prototype turnaround, reducing time-to-market. Simtec also provide a full software development consultancy with an extensive range of products from boot loaders to full Linux based operating system environments and a range of development boards showcasing Simtec's modular designs.