Web Server Embedded Linux System

Vincent Sanders
Daniel Silverstone

Revision History

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

Table of Contents

1. Introduction 1
2. Automated, reproducible, reliable building 1
3. Scripting builds and common tasks 2
4. Creating a web server's file system 2
5. What's next? 4
6. About the authors 4

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

1. Introduction

This is the second article in a series demonstrating the fundamental aspects of constructing embedded systems.

In this article, we cover the construction of a simple web server with 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. Automated, reproducible, reliable building

One of the common pitfalls in building embedded systems is the tendency towards too much manual involvement in the build process. There seems to to be a misconception that because embedded systems are built, deployed and rarely updated that building them by hand saves time that would otherwise be spent automating the process.

Making the build process automatic and repeatable should be viewed as a critical part of the project. This enables the software engineers developing a product to have as short an edit, build and test cycle as possible. The desirability of a short development process should be self-evident in that an engineer who can perform only one or two tests a day can only hope to debug and fix a small number of issues where one who can perform a hundred tests can find and fix a far larger number of issues.

Another common mistake is not keeping the whole project in a Revision Control System (RCS). The benefits of revision control on any project have become increasingly evident and a wide selection of extremely powerful systems exist. The Subversion (**svn**) system is extremely popular for centralised RCS where Bazaar (**bzr**) and GIT have become common for distributed RCS. Regardless of the model and tools chosen revision control should never be omitted from a project.

For larger projects a centralised “build manager” is often desirable. This is a piece of software which builds the current project from the revision control system on a regular basis. Some projects rebuild on every commit, this may not be practical where a build takes an extended period of time. In such cases a system which rebuilds as often as it can, perhaps including numerous commits, should be employed. The results of these builds should be made available, and the developers informed as soon as possible of failures. This ensures the project is always in a state where it might be branched ready for formal testing and release.

A good article discussing these ideas further is Daily Builds Are Your Friend [http://www.joelonsoftware.com/articles/fog0000000023.html] by Joel Spolsky. Although this article refers to application development specifically, its analysis is valid on the larger project scale. Several other articles on best-practice for building software projects exist.

3. Scripting builds and common tasks

In the previous article we constructed an initramfs cpio-based system using the binaries of the host system. The steps were performed manually, and were we to continue with that approach any increase in complexity would rapidly make it impractical.

To solve this issue we turn to the Unix system's scripting tools. A shell script to perform the build actions makes the build easily repeatable and saves continually re-typing a lot of commands.

The `mkbusyfs.sh` [http://www.simtec.co.uk/products/SWLINUX/files/mkbusyfs/mkbusyfs.sh] script is an automation of the steps performed in the previous article.

In addition to the basic setup the script adds functions to copy executables and their library dependencies, configure a DHCP client and copy kernel modules from the host to the target. These functions are straightforward and self-contained and their operation should be obvious.

The script is written to be reusable for a number of projects by including a second “application” script. This enables us to reuse the base functionality in future articles. The scripts are provided under a BSD-style licence and may be taken and modified as desired.

To implement the simple system illustrated in the previous article, the `simple.sh` [http://www.simtec.co.uk/products/SWLINUX/files/mkbusyfs/simple.sh] configuration script can be used.

To keep things neat these scripts should be placed in a directory (these examples assume that it will be called `mkbusyfs`) alongside where the output should be generated.

The simple system would be generated using the command **./mkbusyfs.sh simple** and output would be placed in `simple.gz` in the parent directory.

```
$ pwd
/home/dev/mkbusyfs
$ ./mkbusyfs.sh simple
Building simple
simple specific
Creating CPIO ../simple.gz
$ ls ..
mkbusyfs simple simple.gz
$
```

4. Creating a web server's file system

The first thing we must consider is which web server to install, this is of course dependant on our project requirements.

One choice might be the Apache Web Server, however this would be impractical for all but the largest embedded system. Apache's executables, library dependencies and other system dependencies are relatively large. A more suitable alternative would be thttpd which is a few hundred kilobytes and has very few dependencies.

First we need to create a **mkbusyfs** configuration script.

```
#!/bin/sh
# web server application specific mkbusyfs shell fragment

USE_DHCPC=y
EXTRA_LIBS="ld-linux.so.2 libnss_dns.so.2"
#KERNEL_VER=2.6.26-1-686
KERNEL_MODULES="kernel/drivers/net/ne2k-pci.ko \
                kernel/drivers/net/8390.ko \
                kernel/drivers/net/e1000e/e1000e.ko"

application_specific()
{
    DESTDIR=$1
}
```

This configuration enables the DHCP client (which will require an **init** script [http://www.simtec.co.uk/products/SWLINUX/files/mkbusyfs/udhcp_default]) and lists the kernel modules to be copied into the output. These are required drivers for network cards.

The project could be built and tested at this point if desired. It should initialise a system and acquire an IP address using DHCP.

Next the thttpd binary needs to be acquired. Instead of simply copying this from the host file system we can acquire the deb package from the package mirror, unpack it and extract the items we require without needing to install the package on the host. This does require the `devscripts` package to be installed but means no superuser privileges are required to build the system.

```
#!/bin/sh
# web server application specific mkbusyfs shell fragment

USE_DHCPC=y
EXTRA_LIBS="ld-linux.so.2 libnss_dns.so.2"
KERNEL_VER=2.6.26-1-686
KERNEL_MODULES="kernel/drivers/net/ne2k-pci.ko \
                kernel/drivers/net/8390.ko \
                kernel/drivers/net/e1000e/e1000e.ko"

application_specific()
{
    DESTDIR=$1

    CURDIR=$(pwd)

    mkdir -p /tmp/thttpd/thttpd
    cd /tmp/thttpd
    dget thttpd
    dpkg -x thttpd*.deb thttpd
    cd ${CURDIR}

    add_program /tmp/thttpd/thttpd/usr/sbin/thttpd /usr/sbin/thttpd

    rm -rf /tmp/thttpd
}
```

If we were going to add a second package it might be worth extracting this package acquisition and unpacking into a function. Such judgements are of course arbitrary, but if something is done more than once it is often sensible to create a helper function as then, if a change must be made, all the affected uses will be updated.

The next step is to add the configuration to start the web server and add some basic content. The complete script [http://www.simtec.co.uk/products/SWLINUX/files/mkbusyfs/webserver.sh] may be downloaded.

The only slight difference here is that an additional library and a `/etc/passwd` file was required so the web server could change to execute as the www-data user.

When the output is generated it may be tested using QEMU again. The command line to run QEMU is slightly different as it needs to enable a NIC and redirect the emulated systems port 80 to the host's port 8080. This allows the web server to be accessed from the host using a web browser and the URL http://localhost:8080/.

```
$ qemu -kernel ./vmlinuz-2.6.26-1-686 -initrd webserver.gz \
  -append "root=/dev/ram" -net nic -net user \
  -redir tcp:8080:10.0.2.15:80 /dev/zero
```

The pre-built Kernel [http://www.simtec.co.uk/products/SWLINUX/files/WebServerEmbeddedSystem/vmlinuz-2.6.26-1-686] and generated output [http://www.simtec.co.uk/products/SWLINUX/files/WebServerEmbeddedSystem/webserver.gz] for an x86 system are provided.

5. What's next?

This second step demonstrates the ideas of automation and repeatability and shows how the basic environment constructed in the previous article can be expanded to produce a system capable of interacting with a user.

The next step is to use the concepts presented here and expand them by introducing a more complex application built from source and discussing some limitations of real hardware and issues that arise from it.

6. 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.