Improving the solution

Vincent Sanders
Daniel Silverstone

Revision History

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

Table of Contents

This article considers the confitions and planning for a projects conclusion and considers alternative processes for constructing systems.

1. Introduction

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

We discuss some issues about the end of a project and then considers alternative tools and methods for building systems.

The whole series or articles 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. If it's not broken, don't fix it

Although the targeting of the ARM system in the previous article produced some issues, the system did work and was usable. In a real project the issues raised might simply not be worth addressing. If the solution arrived at meets all the requirements from the project brief it is self-evident no further development is required.

The other side of the argument is that the project brief may not be complete. During the development of a solution factors which were not originally foreseen often arise, the project planning must be flexible enough to integrate these updates without compromising the project.

This point is made explicitly as experience shows embedded system projects suffer from either over or under compensation. Projects like these fail because fundamental issues arise which the project brief cannot reflect or because the schedule has to be repeatedly extended to accommodate a ephemeral specification. A workable compromise must be struck.

An especially common reason the brief may be changed is the question of continued development and support. Experience shows that once a system reaches a state where it becomes usable and may have fulfilled its original brief the potential to extend and improve it causes feature creep. If a project is to succeed this tendency must be controlled.

The best way to handle the “feature creep” issue is to plan for it. This statement may seem obvious but it is often an omitted and these articles are partly an attempt to reduce common mistakes. Planning might be as simple as a requirement in the brief that development is conducted in a way which will allow updates in the future, this means a project can be completed after having met its original requirements and then a new project created to extend the feature set.

Embedded developers seem stubbornly myopic about the issue of ongoing support. There seems to be a view that at the end of a project everything will be archived and no further development will be required. For any project complex enough to employ a Linux kernel and userspace this is rarely true. Given the large NRE of any embedded system it is almost always more cost effective to refine an existing product, perhaps updating it with new technologies as they become available than to start afresh. A successful, well planned, project enables the one that follows it to succeed more easily.

3. Is there another way?

So far in this series we have used the approach of taking pre-compiled executables and libraries from a host system and constructing a suitably arranged file system image. This has run into the issue that the build and target system have differing requirements which has undesirable effects on development.

Another issue which has not yet been considered is that the resulting systems tend to be larger than necessary. This is because the executables and libraries are built for a generic operating system which must provide the full user-space API. If we were to build only the libraries with options specifically required for our system the dependencies would be fewer and the result smaller.

Finally, the host-based approach requires the target system be capable of running a complete operating system either on the real hardware or under emulation. This may not be practical if the target system is heavily customised.

The solution to these issues, which is employed by many projects, is to use cross-compilation of the entire system from source. Cross-compilation is a technique where a host PC runs a compiler which generates output executables for a different architecture. For our ARM web kiosk example, this compiler would execute on the x86 machine and generate ARM executables.

Building an entire operating system from source is a daunting task even for a seasoned professional familiar with the process. To indicate the scale of the task we will return to our very first example and outline the process of building a simple busybox environment.

To build busybox you require a C library and a compiler. The compiler must be built against the selected C library. The most common choices of C library are GLibc which is feature complete and as a result very large, or uClibc which has less functionality but is much smaller. Building the cross compiler with the appropriate C library requires extensive configuration and may take several hours to compile. Once you have a working cross compiler you may configure busybox; there are over a thousand configuration items in busybox, of course you don't need to set all of them but deciding what needs to be enabled is a challenging task in itself. Assuming the busybox compile has been successful the binary and C library must be deployed in a file system as in our previous examples.

The above description is of course a gross simplification of the process as each of the selected components may have version interactions and need additional patches applying to produce a workable solution. Perhaps the reader will begin to appreciate why these articles started with the most straightforward approach and waited to the fifth before introducing such a intimidating concept.

All is not lost! Because of the complexity of building these systems from source, developers responded by automating the process. These automated process bear a superficial resemblance to the **mkbusyfs.sh** script we have used previously but perform a huge number of operations.

The most common OS builders in use at this time are buildroot [http://buildroot.uclibc.org/] and Openembedded [http://wiki.openembedded.net/]. Neither are particularly easy to use but do reduce a nearly impossible task to something more practical.

The buildroot tool is a collection of Makefiles and script which is controlled by a configuration file. Buildroot is generally used for smaller systems with a limited interface, it is actively maintained and supports several architectures. The configuration file is generated using the kconfig system (the system the Linux kernel uses) which allows for easy manipulation of settings. Once configured the system may be built with a single make command, the initial build may take several hours as the entire cross compiler tool chain must be built. Subsequent builds should be incremental and only rebuild those components that have been modified.

The Openembedded system uses the bitbake tool to generate binary packages, the resulting system is generated by installing these packages into a target directory. The systems Openembedded builds tend to be much larger (e.g. PDAs and netbook devices) have a graphical interface and have rich user interaction. The initial build may take many hours but subsequent builds only rebuild and install the packages required so are much faster.

Other tools exist for building system; in fact there are a substantial number. The two highlighted above have been selected as examples purely because of their popularity and general flexibility. Many of the other build tools available are domain specific i.e.

they target a single device or area. One such tool is OpenWrt [http://openwrt.org/] which builds systems specifically for networking applications.

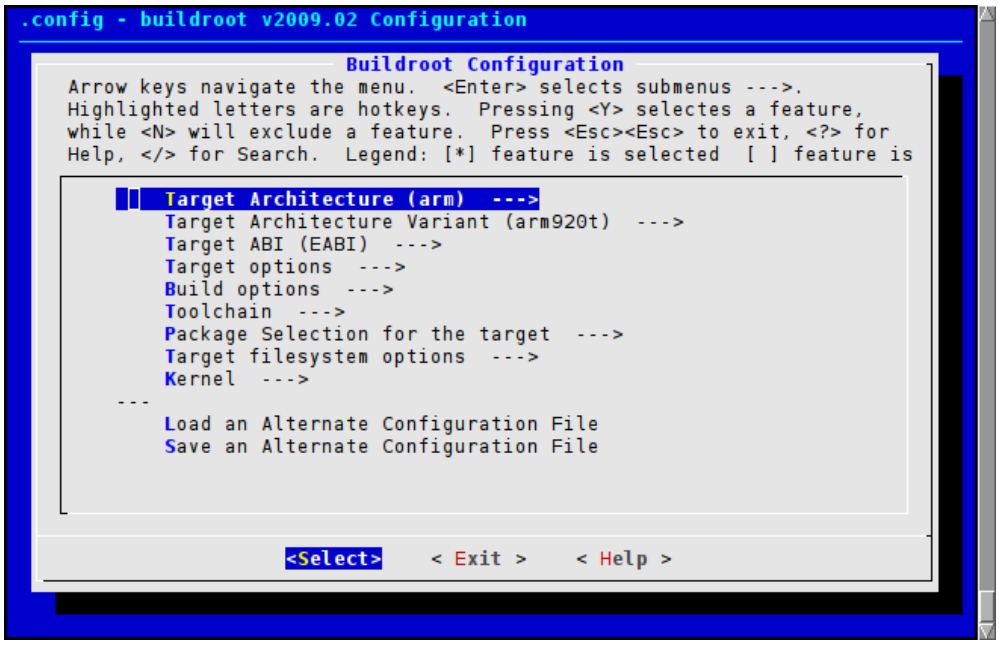
It may be desirable to use one of these more targeted tools depending on a projects requirements. The selection should not be made without a good deal of research as the tool will have a major influence on the outcome of a project.

4. A simple system with buildroot

To demonstrate the buildroot system we shall use it to build a simple busybox system for our ARM system.

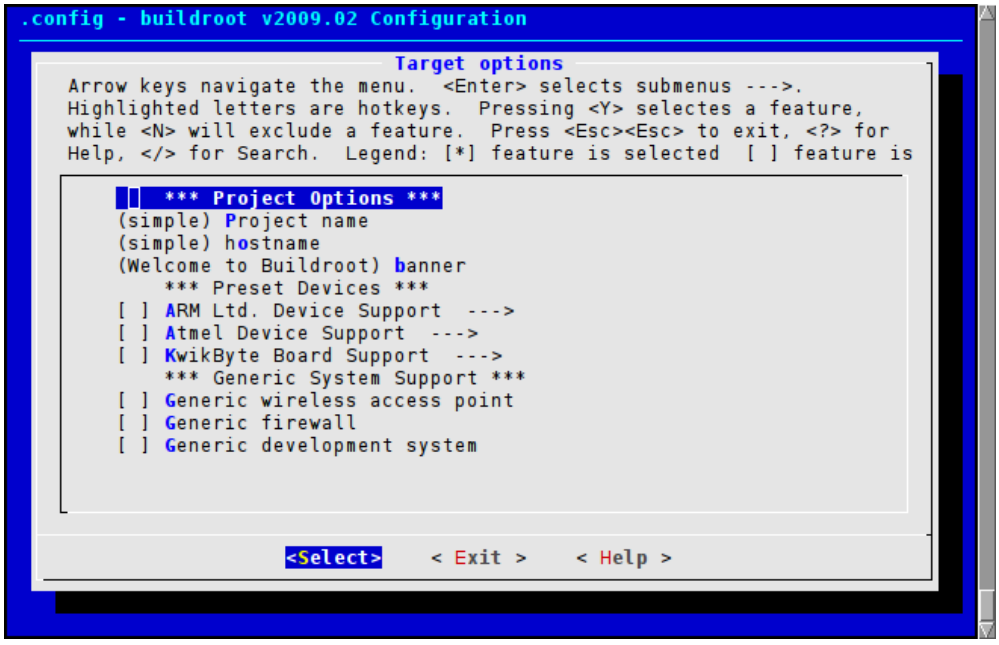
First the buildroot tool must be obtained. The stable 2009.02 [http://buildroot.uclibc.org/downloads/buildroot-2009.02.tar.bz2] release was downloaded and unpacked. An configuration was generated using **make menuconfig**, the ARM target was configured along with a CPU type of 920t (the type of the S3C2440 SoC) and an EABI build.

Figure 1. Configuring buldroot



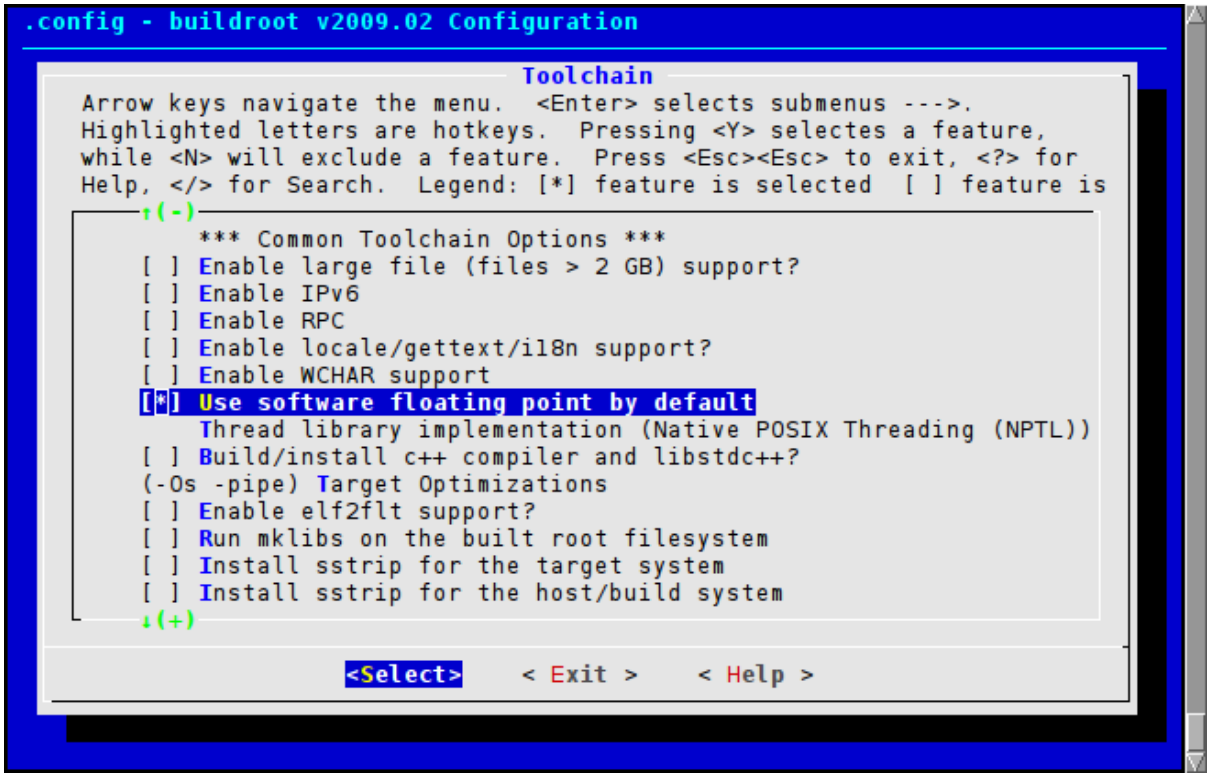
The target options were changed to make the project name “simple”.

Figure 2. Selecting target options



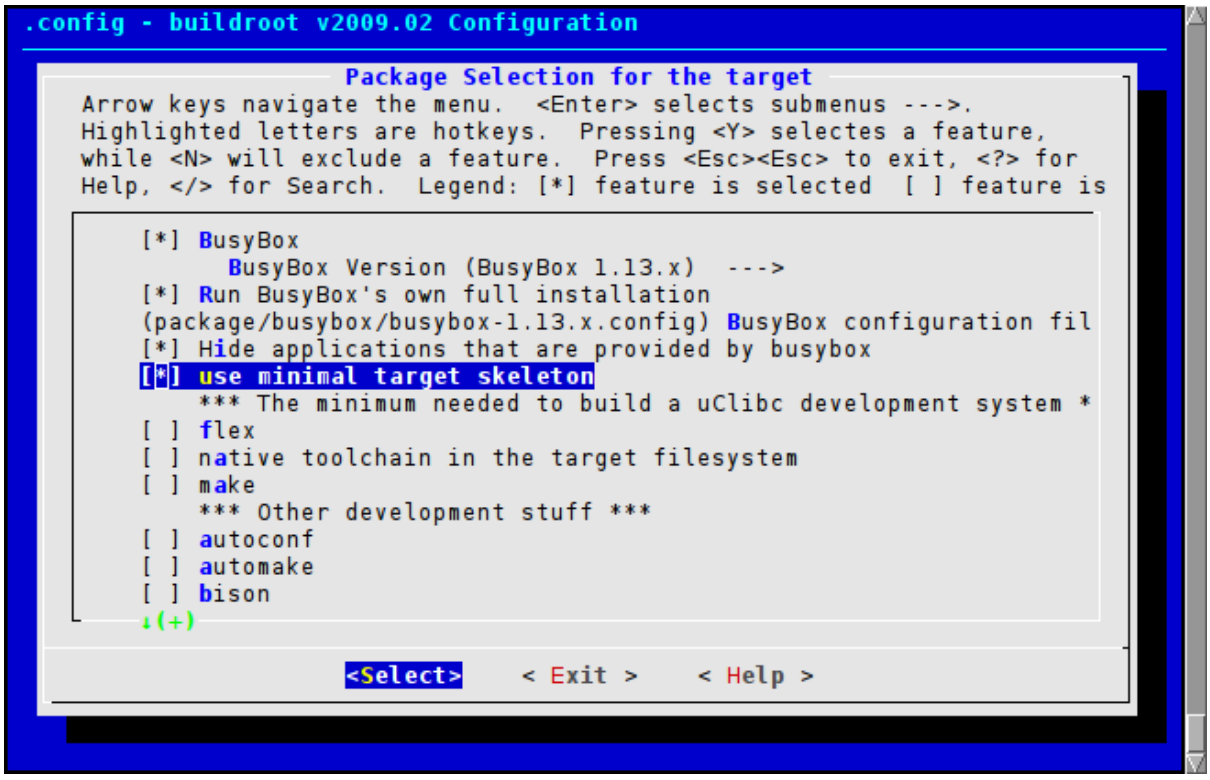
The softfloat option was selected in the tool chain configuration.

Figure 3. Tool chain configuration



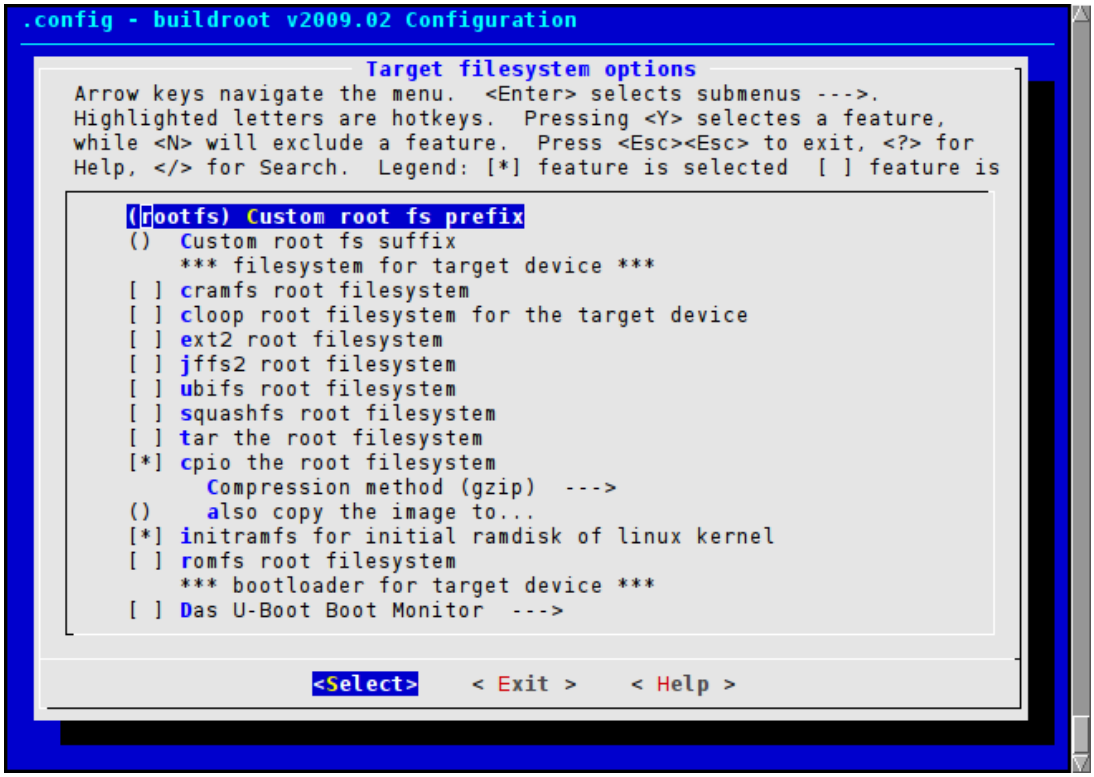
The minimal target skeleton option was selected. this causes the target system to use the mdev system to generate device nodes. we have used mdev in the previous examples.

Figure 4. Package Selection for the target configuration.



Finally the target file system options were selected.

Figure 5. Target file system configuration



All the other options were left at their default values. Once configured **make O=/buildroot/build** was used to start the compilation process. The O option causes the build to be performed outside the source tree, this is desirable for repeatability as it does not alter the source.

Fifty minutes later the build completed and a compressed cpio archive was produced as before. The file `/buildroot/build/binaries/simple/rootfs.arm.cpio.gz` was copied to the TFTP server and started from the boot-loader. the image was loaded and kernel execution commenced. The system failed to start with a somewhat unhelpful message.

```
Kernel panic - not syncing: Attempted to kill init
```

This is both unfortunate (our example did not work first time) and illuminating (we get to explain how to debug the issue). In a previous article it was mentioned that the first process the kernel starts, the init process, has a special constraints within the system different to those of other user processes. One of these constraints is the init process may *never* exit, if it does the kernel will panic and you will receive the above message.

As our system only actually contains the busybox program which is providing all the commands for our system (including init) we have a fairly simple problem to solve, why is busybox exiting? Experience shows the easiest way to test such a binary is to run it on an already operational system.

Fortunately we have the ideal candidate, our NFS-based Debian system we used to build the web kiosk application. This OS was started, the `rootfs.arm.cpio.gz` unpacked into a directory and the **chroot** command used to start a shell inside the target.

```
$ mkdir simple
$ cd simple
$ sudo cpio -i ../rootfs.arm.cpio.gz
$ cd ..
$ sudo chroot simple /bin/sh
SIGILL
$
```

It seems the busybox binary has been built containing instructions which the CPU cannot execute. This can happen on ARM systems when the compiler has built with the wrong architecture target. The CPU in question supports version 4t of the instruction set and

is capable of correct EABI operation, however the compiler appears to have selected the version 5 instruction set instead. This is unexpected as the CPU type was clearly set to 920t within the buildroot configuration implying version 4t.

Investigating the buildroot make files lead us to discover that the CPU type selection only configures the CPU the compiler will optimise for, not the instruction set version. The buildroot configuration script (`.config`) was manualy edited and the variable `BR2_GCC_TARGET_ARCH` set to *`armv4t`*.

```
BR2_ARM_TYPE="ARM920T"
# BR2_ARM_OABI is not set
BR2_ARM_EABI=y
BR2_ARCH="arm"
BR2_ENDIAN="LITTLE"
BR2_GCC_TARGET_TUNE="arm920t"
BR2_GCC_TARGET_ARCH="armv4t"
BR2_GCC_TARGET_ABI="aapcs-linux"

#
# Target options
#
```

The output build directory was cleared (excepting the downloaded source archives) and the build repeated.

The system was once again started from the boot-loader, this time it appeared to start successfully and a login was presented on the video console. Unfortunately none of the kernel modules required for USB HID devices are present in the system and no login was presented on the serial console. Because of this the system as it stands cannot be interacted with.

A serial login can be added simply by editing `/buildroot/build/project_build_arm/simple/root/etc/init-tab` and adding a line for the S3C2440 SoC first serial port.

```
s3c2410_serial0::respawn:/sbin/getty -L s3c2410_serial0 115200 vt100
```

If root should be able to login on the serial port the `/buildroot/build/project_build_arm/simple/root/etc/securetty` file must have s3c2410_serial0 added to it.

The system was rebuilt and the image booted and a successful login made.

```
Welcome to Buildroot
simple login: default
$ ls -l /
drwxrwxr-x    2 root     root            0 Dec 31  1969 bin
drwxr-xr-x    2 root     root            0 Dec 31  1969 config
drwxr-xr-x    3 root     root        13280 Mar  6 19:49 dev
drwxr-xr-x    4 root     root            0 Mar  6 19:49 etc
drwxrwxrwx    3 root     root            0 Dec 31  1969 home
lrwxrwxrwx    1 root     root            9 Dec 31  1969 init -> sbin/init
drwxr-xr-x    2 root     root            0 Dec 31  1969 lib
lrwxrwxrwx    1 root     root           11 Dec 31  1969 linuxrc -> bin/busybox
dr-xr-xr-x   43 root     root            0 Dec 31  1969 proc
drwxr-x---    2 root     root            0 Dec 31  1969 root
drwxrwxr-x    2 root     root            0 Dec 31  1969 sbin
drwxr-xr-x   13 root     root            0 Dec 31  1969 sys
drwxrwxrwt    2 root     root          120 Mar  6 19:49 tmp
drwxr-xr-x    5 root     root            0 Dec 31  1969 usr
drwxr-xr-x    3 root     root            0 Dec 31  1969 var
$
```

As can be seen from this example the buildroot system does reduce the effort of building a system from source but does require a great deal of domain specific knowledge to fix issues when they arise. The illegal instruction issue took several hours to debug and fix, the final solution was relatively simple but the process to find it was involved.

The issue with configuration files being incomplete for our target hardware was simple to fix but we would need some way to ensure this is automated for future builds (by editing the generic target skeleton). This problem did however highlight the benefit of the buildroot system in that the rebuild to include the updated files took seconds.

5. What's next?

This article has covered how the development processes might be refined and introduced a new build strategy. The new strategy also has (differing) issues which have to be considered.

In the next article we examine issues related to moving from simple volatile RAM based systems to deploying on non volatile storage media.

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.