

Linux on the EPC9302 ARM development board

Platform specifications

The EPC9302 board we use comes from Olimex Ltd. It contains a 200MHz EP9302 microcontroller, 16MB of Flash ROM and 32MB of Synchronous Dynamic RAM (SDRAM).

Boot process

On reset, the ARM processor checks the state of the following pins: BOOT1, BOOT0, EECLK, EEDAT, ASD0, CSN7 and CSN6 (See EP9302 User's guide, Table 5-2). Depending upon the states of these bits, the processor will boot in a number of different ways. In the case of the ARM board we are using, these bits are configured such that the processor searches particular addresses in flash memory for the ASCII sequence "CRUS". Once it finds this sequence (0xe0001000 on our board) it begins executing code just after it. In this case, the code in question is the Redboot boot loader.

Redboot

Redboot is an acronym for "Red Hat Embedded Debug and Bootstrap". Full documentation for redboot may be found here:

<http://ecos.sourceware.org/docs-latest/redboot/getting-started-with-redboot.html>

In summary, Redboot allows system developers download operating system images (and file systems) into flash memory or SDRAM and then boot these images. It also allows the system developers to write startup boot scripts which can load a particular OS image from flash (or other locations), configure network settings, configure debugging support and pass command line/start-up options to the operating system.

In our case, the operating system that is loaded by default is Linux version 2.6.21.

Root disks

Linux was written for computer systems that had disk drives and as a result, a root file system is deeply ingrained in the kernel code. The kernel will fail to boot if it cannot locate a root file system of some kind. At first glance, this does not seem to sit well with embedded systems that have no physical disk drives at all. This difficulty is overcome by using a memory based pseudo disk. This 'fake' disk can be based in a region of RAM (a 'ramdisk') or ROM or a combination of both (the ROM based portions such a disk are not writeable).

Booting Linux

If left uninterrupted, the Redboot start-up script will load a compressed kernel image (zImage) and a compressed root file system image (ramdisk.gz) from flash memory to RAM. It then transfers control to the start of the kernel image which decompresses itself (there is non-compressed section of code at the start of zImage) and boots or starts up. During the boot process, the kernel locates the compressed

root disk image and mounts it. At the end of the boot process, the kernel runs a program called `init` in the `/sbin` directory. This program is the first process run on the system. All other processes are spawned in a tree like fashion from this.

References:

Embedded Linux Primer by Christopher Hallinan

Building Embedded Linux Systems by Karim Yaghmour

Busy box

Busy box is a clever program that emulates a large number of Linux utility programs such as: `ls`, `cd`, `cat`, `more`, `less`, `rm`, etc. In short it emulates most of the utility programs that are commonly found within the `/bin` directory of linux systems. It also emulates other privileged programs including `init`. Why do this? The main reason is to save space. Busybox is a single executable which can be symbolically linked to using links called '`init`', '`cd`' and so on. When a script or user 'runs' one of these links, busybox is called instead. The busybox program is able to determine the name used to invoke it and can then act accordingly. Thus, if a user 'runs' the '`more`' command which in fact is redirected to busybox, then busybox can discover that the original command issued was '`more`' and emulate the expected behaviour. Busybox can significantly reduce the size of the root file-system and is widely used in embedded products. See www.busybox.net for more information.

In the case of our board, virtually all of the executable programs on the root file system (including `init`) are linked to busybox.