TS-7250 USB Linux Booting

Michael Stegeman

ECE 4220 – Real Time Embedded Computing
Instructor: Dr. DeSouza

Department of Electrical and Computer Engineering
University of Missouri
Columbia, Missouri

December 10, 2009

The TS-7250 development boards are very handy tools when learning real time embedded computing, but setting them up can be a hassle due to inherent limitations. The goal of this project was to make the TS-7250 boot directly from flash, and use a USB drive as the root file system. In the following paragraphs, I will describe the difficulties with this project and the steps taken to complete it.

The TS-7250, shown in Figure 1, is a fully integrated computing device. It consists of an ARM processor, 32 MB SDRAM, 8 MB on-board flash memory, an IDE controller, and serial, USB, and Ethernet ports.

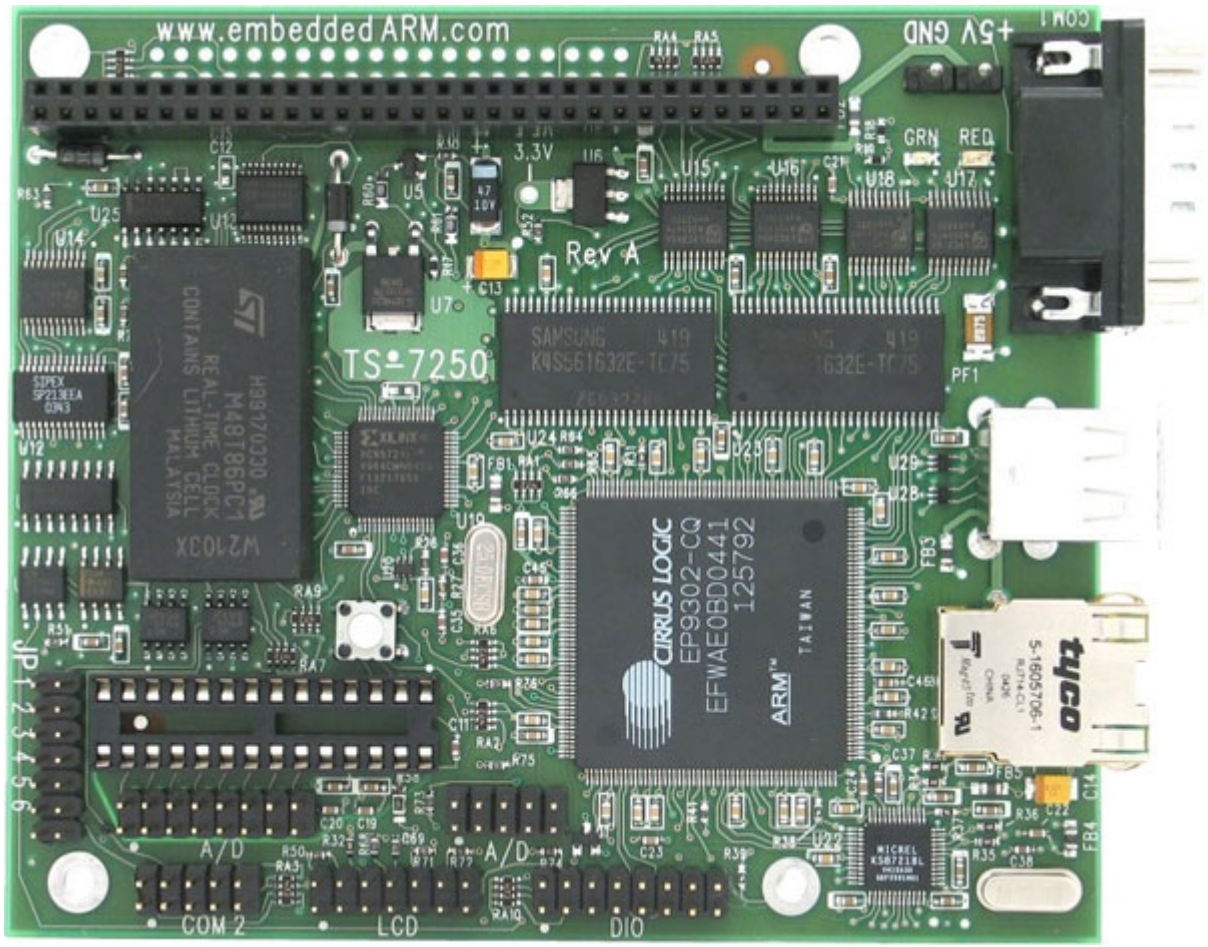

Figure 1 - TS-7250 Development Board

This development board boots similar to a regular desktop computer. When powered on, the board loads the TS-BOOTROM. This is similar to a BIOS (Basic Input/Output System) in

that it performs initial hardware configuration and initializations. This is proprietary code, stored in ROM (Read Only Memory) on the board.

After this stage completes, the board then loads RedBoot. RedBoot is an open source boot loader, built on top of the eCos embedded operating system. RedBoot is similar to any other bootloader, such as GRUB (GNU GRand Unified Bootloader). The RedBoot environment is used to load and execute any binary, although the typical application is to load and execute a Linux or Windows CE kernel. It can also be used to create and delete images stored in the on-board flash memory. The version of RedBoot installed on the TS-7250 can load a binary image directly from flash, or over the network via TFTP (Trivial File Transfer Trotocol) or HTTP (HyperText Transfer Protocol). The default boot script instructs RedBoot to load and execute a pre-installed Linux kernel from the on-board flash, and mount the flash memory as the root file system.

When the Linux kernel is executed, several things take place. First, Linux initializes the hardware devices and loads them into the /dev file system, where all block devices are stored. While this is occurring, the root file system is mounted. With the default Linux kernel installed on the TS-7250, this root file system can either be the on-board flash memory or an NFS (Network File System) share. After these two things happen, a software initialization script is run, and then the user is presented with a login prompt.

At the start of this project, the TS-7250 boards in the lab were set up to load a Linux kernel over TFTP and execute that kernel, mounting an NFS share as the root file system. However, the goal is to eliminate the need for TFTP and NFS, instead using the on-board flash and a USB drive. There are several obstacles to overcome in order to accomplish this, though.

The preinstalled Linux kernel on the board has USB support built as a module. The problem with this is that it's not possible to mount a USB drive as the root file system unless USB

support is built into the Linux kernel. Also, the preinstalled kernel is not real-time, which renders it useless for the purposes of this class. The solution to these problems is to compile a custom, real-time Linux kernel with USB support built in.

Upon completing this task, the kernel was loaded onto the board via TFTP (for testing purposes). A USB flash drive was plugged into the board, and the kernel was executed, with the flash drive set as the root file system. However, this proved to be unsuccessful. Since the hardware initializations occur in parallel with mounting the root file system, a new problem was encountered. The USB subsystem was actually being initialized after the kernel tried to mount the root file system. This led to a kernel panic, as the kernel was trying to mount the USB drive before it existed in the /dev file system.

After doing some searching, I developed a new plan of attack. The Linux kernel allows for an initial RAM (Random Access Memory) disk to be used as an initial root file system. Essentially, a disk image is created as a file. This disk image can contain any number of libraries, modules, binaries, and executables. It also contains an initialization script. What happens is that the kernel mounts this RAM disk as the root file system and executes the initialization script contained in it. The purpose of this script is to load any necessary modules, perform any setup routines, and pivot the root file system to the USB drive. For the purpose of this project, the script only needed to mount the USB drive and pivot root. This proved to be unsuccessful as well. The USB subsystem was still being loaded too late, as the initialization script completed running before it was loaded.

This led to a heavy amount of head-scratching. Somehow, I needed to make the kernel sleep before trying to mount the USB drive. I developed three possible solutions. The first solution was to use the 'sleep' program shipped with any Linux distribution to sleep in the RAM disk initialization script. Upon inspection, though, I found that this program was a dynamically

linked executable. What this means is that any libraries it uses would have to be copied to the RAM disk along with the executable. Doing so caused the RAM disk to be extremely large, meaning it could no longer be stored in the flash memory on the TS-7250.

The next possible solution was to compile a statically linked BusyBox binary. BusyBox is a program that has a number of basic Linux utilities built in, one of which is the 'sleep' program. The idea is that the 'sleep' utility built into BusyBox could be used to sleep in the RAM disk initialization script. Since BusyBox can be built as a statically linked executable, it is the only file that would need to be copied into the RAM disk. After setting up a cross-compiler toolchain in order to compile code for the ARM platform, I compiled BusyBox. Whenever I tried to run it from the initialization script, though, I was presented with a number of “obsolete system call” errors, and again, a kernel panic.

This brought me to the final possible solution. Newer Linux kernels have a 'rootdelay' parameter that can be passed to the kernel, which instructs the kernel to sleep for the given number of seconds before trying to mount the root file system, which eliminates the need for a RAM disk. This feature made its way into the 2.6.11 Linux kernel. Later, it was backported to Linux 2.4.37 and above. However, we're using the 2.4.27 kernel, which means that this feature was not present. The only remaining option was to patch our kernel to include the rootdelay feature. After searching around the Internet for awhile, I found instructions on a Linux forum on how to patch a Linux 2.4 kernel to include this feature [1]. I applied the patch and attempted to compile the Linux kernel. When compiling, I was presented with an error for an unknown 'ssleep' function

Again, I turned to the Internet. I found that the 'ssleep' function was not added to the Linux kernel until version 2.4.29. This meant that I had to backport this feature as well. I found instructions on another Internet forum on how to apply this patch [2]. After doing so, I compiled

the kernel successfully. I loaded the new kernel onto the board via HTTP, set a 10 second root-delay, and set the root file system as the USB flash drive. The kernel successfully executed, and I was eventually presented with a login prompt. To ensure that all was working fine, I loaded my code from several previous labs onto the flash drive and executed them on the board. All labs executed correctly. I proceeded to load the kernel into the flash memory, and I wrote a new RedBoot script that loads and executes the new kernel from flash and mounts the USB drive as its root file system.

The goal of this project was to eliminate any dependence on network interaction from the TS-7250 board. To do this, a Linux kernel needed to be loaded from the flash memory, and a USB drive needed to be mounted as the root file system. After several failed attempts at doing this, a solution was finally reached. The Linux kernel had to be patched to include the needed real time modules, built in USB support, and a rootdelay feature that allows for USB root file systems. After patching and compiling the kernel, the system worked correctly and can boot without any need for network attachment.

Appendix A – RedBoot script

The following is the working RedBoot script, to be used after the Linux kernel is loaded into the flash memory. Since no network attachment is needed, the IP addresses that can be set via fconfig in RedBoot are unimportant.

```
fis load vmlinux.bin
exec -c "console=ttyAM0,115200 root=/dev/scsi/host0/bus0/target0/lun0/part1 rootdelay=10 rw"
```

Works Cited

[1] “booting from usb drive,” post #8, Mar. 17, 2008. [Online]. Available: http://www.linuxquestions.org/questions/slackware-14/booting-from-usb-drive-628322/#post3090941. [Accessed: Dec. 6, 2009].

[2] “[patch 2.4] back port msleep(), msleep_interruptible(),” Oct. 31, 2004. [Online]. Available: http://www.opensubscriber.com/message/jgarzik@pobox.com/120502.html. [Accessed: Dec. 6, 2009].