Reducing boot time in Linux devices

Class WE-2.2, Embedded Live Conference, UK. 2010

Chris Simmonds (chris@2net.co.uk)

October 2010

Abstract

Boot time is an issue in many embedded systems. In this paper I analyse the steps from power-on to a working application and show how to minimise the time taken at each stage. I will demonstrate some of the tools available to measure boot time, such as bootchart and printk timestamps, and I will describe some of the techniques you can use to improve the start-up time of most Linux systems, from better device probing to optimised boot scripts.

1. Why wait?

It is an annoyance to many people that the gadgets they use every day - mobile phone, printer, digital radio - take a significant length of time from power-on to being operational. In many cases the response of the manufacturers and users is to use a “standby mode” in which the device is working at reduced power waiting to be “turned on” and resume full operation “immediately”. Standby mode is valuable: my mobile phone would have a much shorter battery life without it, but in other cases it is a bit of a kludge which continues to consume power and drain the battery. I know of a case of a set-top box where the standby mode consumed slightly more power than operational mode because the only difference between the two was an LED with “standby” written next to it.

One reason for the start-up time, and the subject of this paper, is that the use of complex operating systems such as Linux inside those devices means that we have to wait for the operating system to boot. However, there are a set of techniques that you can use to reduce this time. In this case as in many others there is a scale of effort versus reward and in very broad terms I would say it goes like this

Boot time	**Effort**
40 s	Zero: this is what you might get from an unoptimised O/S
15 s	Little: some optimisation of the boot scripts
7 s	Moderate: optimisation of kernel
<= 2 s	Much: radical re-engineering of the boot process

So, 7 t o15 seconds should be possible in most cases. As an example device I am going to use a camera demo implemented on a fairly modern development platform, and measure the time from power on to image capture.

2. The device

Hardware

In choosing a platform I wanted something that is representative of current hardware, with a recent ARM core. The system I chose is a Digi ConnectCore Wi-i.MX51 starter kit, which is based on a Freescale i.MX515 processor (ARM Cortex A-8), clocked at 800 MHz, with 512 MiB SDRAM and 512 MiB NAND flash. It has a 800x480 LCD touch screen.

Software

I wanted software to be as representative also, so I have chosen the Ångström distribution which is built using the OpenEmbedded framework. The complete run-down is

- Toolchain and root file system: Ångström armv7a (gcc-4.3.3, glibc-2.9)

- Boot-loader: U-Boot 2009.08
- Kernel: 2.6.31
- Camera application: Qt embedded 4.6.2

The kernel is loaded from the NAND flash and the root file system takes up the remainder of the flash, 470 MiB, and uses the JFFS2 format.

3. Initial measurements

First I need to quantify the problem. I could use a stop-watch. But, that is not very accurate and rather tedious after the first few runs. A better solution is to instrument the code or to monitor the boot from outside. I will describe both techniques, starting with external monitoring.

grabserial

There is a very useful program called *grabserial*, written by Tim Bird and available from http://elinux.org/Grabserial. It is a Python script that adds a time stamp to each string received from a serial port. The target I am using has a serial port for the console and general debugging, so I will use that. Most embedded devices have such a thing.

Usage:

-d <serial device>

-b <baudrate>

-m <match pattern that will reset time stamps>

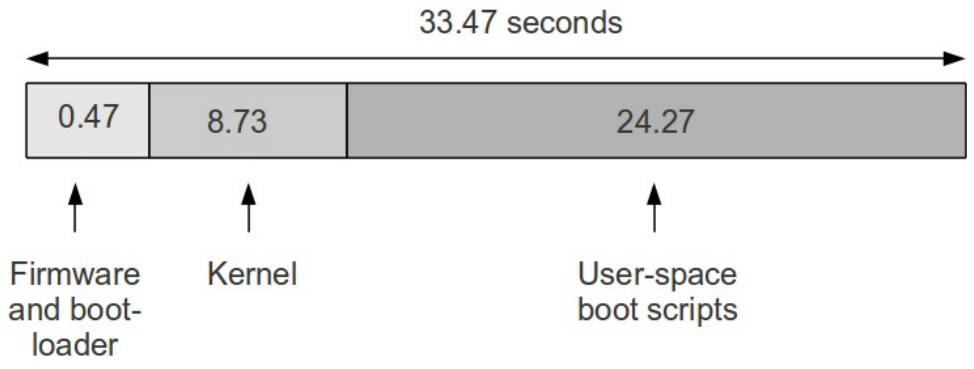
-t

For example:

```
$ grabserial -t -d /dev/ttyUSB0 -b 38400 -m "Starting kernel*"

[    7.463323] Starting kernel ...
[    0.009875]
[    0.010001] Uncompressing
Linux...............................................
...........................................................................
.....
....... done, booting the kernel.
[    1.099339] Linux version 2.6.31 (chris@chris-laptop) (gcc version 4.3.3
(GCC)
 ) #1 PREEMPT Wed Sep 29 17:03:06 BST 2010
[    1.126183] CPU: ARMv7 Processor [412fc085] revision 5 (ARMv7),
cr=10c53c7f
[    1.142990] CPU: VIPT nonaliasing data cache, VIPT nonaliasing
instruction cache
[    1.159425] Machine: Digi ConnectCore Wi-MX51 on a JSK Board
```

Analysing the output, we have:



Clearly the place to start is with the user-space.

4. Optimising user-space

Here is a list of the tasks taking a significant length of time (> 220 ms):

Task	Time (seconds)
Starting Avahi mDNS/DNS-SD Daemon	20.26
Starting Bluetooth subsystem	1.19
Starting udevd	0.52
Remounting root file system	0.22

This illustrates what can happen if you take an off-the-shelf root file system without looking at what it does. The first two services, Avahi and Bluetooth are not needed at all in this application, so I can just edit the boot script and leave them out. But, supposing that I did need them, not unreasonable in a camera application; then I would modify the boot sequence to start them in the background *after* the main application is up and running.

udev

Starting udev is taking 0.52 seconds in this case, which puts it down the scale of things to look at. Nevertheless, I have seen cases where udev takes multiple seconds so start, so it is worth while understanding what it is and how useful it is.

The udev service is responsible for populating the /dev directory with device nodes and responding to run-time events including

- adding/removing hardware
- loading/unloading modules
- creating/deleting device nodes

It consists of a daemon process, udevd, a set of rules in /lib/udev/rules.d and a set of helper scripts in /lib/udev. Thus it is quite complex and quite slow.

Do you really need udev? If your device has a static collection of peripherals, then the answer is no. If you have, say a USB port, into which you will allow the user to plug a memory stick then you can hard code that quite easily.

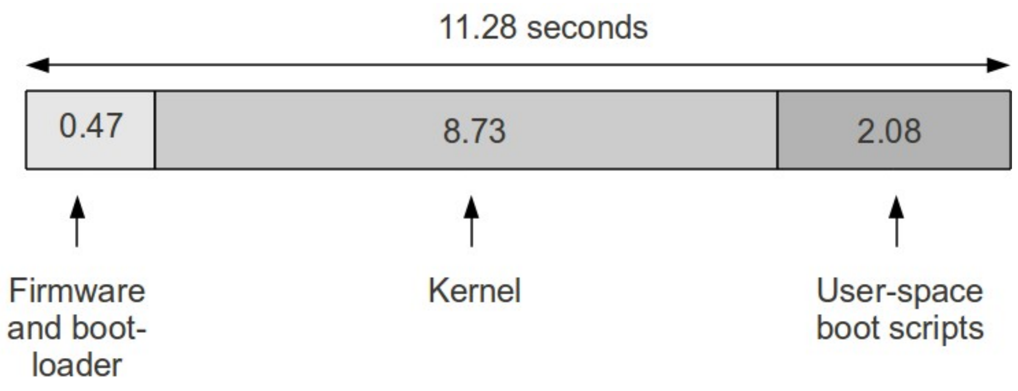
Doing without udev

The hard way is to make a list of all the device nodes you need and to create each one using mknod. The easy way is to boot using udev and use tar to create an archive of /dev and then to extract that into the master copy of the root file system on the development system. Then disable the script that runs udev, making sure that /dev is no longer a ram-based tmpfs and then re-flash the root partition with the new files.

Boot time with optimised user-space

To summarise the savings so far:

• Disable Avahi	20.26 s
• Disable Blutooth	1.19 s
• Replaced udev with static nodes	0.57 s
• Disable rootfs check	0.22 s
Total saving	**22.24 s**



That has made a considerable improvement to the user space time. Next I will look at the kernel boot time.

5. Measuring Kernel boot time

The grabserial program would work fine. However there is a better option, PRINTK_TIME, which adds timestamps to the boot messages and which can be used together with CONFIG_BOOT_TRACER to create a graphical view of where the time goes.

PRINTK_TIME

Run the kernel configuration program and enable in “Kernel Hacking”->”Show timing information on printks”. Rebuild and reboot and you will see messages like this:

```
[    0.000000] Linux version 2.6.31 (chris@chris-laptop) (gcc version 4.3.3
(GCC
) ) #2 PREEMPT Wed Sep 29 17:59:01 BST 2010
[    0.000000] CPU: ARMv7 Processor [412fc085] revision 5 (ARMv7),
cr=10c53c7f
[    0.000000] CPU: VIPT nonaliasing data cache, VIPT nonaliasing
instruction ca
che
[    0.000000] Machine: Digi ConnectCore Wi-MX51 on a JSK Board

...
```

Bootchart

Note: this requires kernel version 2.6.28 or later.

If you also enable “Kernel Hacking”-> “Tracers”-> “Trace boot initcalls” the boot messages include information about the length of time it takes to call each function. While you are changing the configuration, you should increase the kernel log buffer size to at lease 16 (64 KiB).

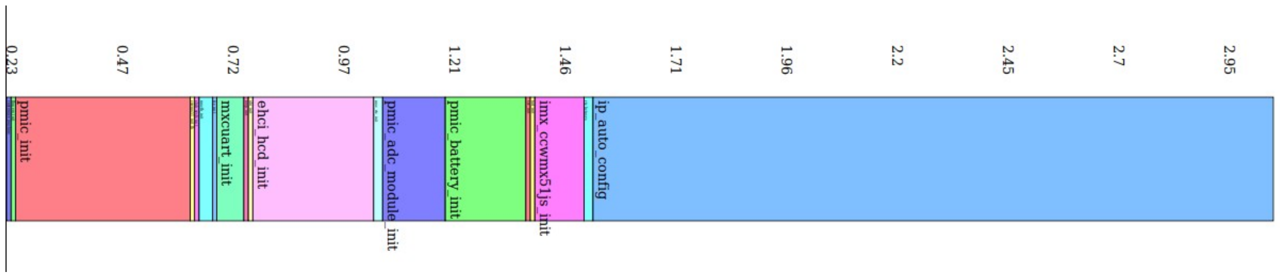
To analyse the messages, log on to the device and dump the kernel messages to a file:

```
dmesg -s 65536 > /boot.log
```

Copy the file to your development machine and run it through a script that is part of the kernel source:

```
cat boot.log | perl linux-2.6.31/scripts/bootgraph.pl > bootgraph.svg
```

Then view the resulting file using a web browser (Firefox for example) or graphics program. You will see something like this.



The three longest sections are

- ip_auto_config: 1.48 s
- pmic_init: 0.38 s
- ehci_hcd_init: 0.26 s

I will discuss these later on.

6. Kernel optimisation

Quiet boot

The kernel produces a lot of messages as it boots up. In my case they are being sent to a serial port running at 38400 bps and so are taking up quite a lot of time. I can suppress almost all of them by adding “quiet” to the kernel command-line. The messages are still available: they are kept in the kernel log buffer which I can display at any time using the dmesg command. Since I have enabled PRINTK_TIME, I still have the timing information for further analysis even though they are missing from the grabserial log.

Saving using quiet boot: 3.07 seconds

Choosing a better file system

One section of the kernel log stands out:

```
[   10.340000] JFFS2 doesn't use OOB.
[   12.960000] VFS: Mounted root (jffs2 filesystem) on device 31:3.
```

It is taking 2.62 seconds to mount a 470 MiB jffs2 file system which is 8% full. In fact jffs2 is well known for taking a long time to mount, sometimes 10's of seconds, so maybe it is not the best choice for the root file system. Here are some other options

- Make the root partition smaller, since I am only using 8% of it. Smaller jffs2 partitions mount faster
- Use UBIFS
- Use squashfs for the read-only part of the data, which is most of it. Squashfs mounts really fast.

The second option is the most attractive because I don't have to re-engineer any other part of the system, just replace one file system with another. The first option would work so long as I don't want to store too many photos, or maybe I would store them somewhere else. The last option has a small complication because squashfs does not handle NAND flash directly. I would need to create a UBI volume first and then mount squashfs on top of that.

UBIFS requires a UBI volume as well, so now there are two items in the log to look at.

```
[    2.240000] UBI: attaching mtd3 to ubi0
...
[    3.630000]
```

So, the UBI attach takes 1.39 seconds. Then I have to mount the UBIFS file system:

```
[    5.670000] UBIFS: mounted UBI device 0, volume 0, name "ubi_rfs"
...
[    5.780000] VFS: Mounted root (ubifs filesystem) on device 0:12.
```

The UBIFS mount it fast: only 0.11 s, making a total time of 1.50 s.

Saving: 1 second

ip_auto_config

This code allows you to configure an interface with a network addresses during the kernel boot by adding “ip=...” to the command line. From the bootchart shown earlier we can see that it takes 1.48 seconds, but the network is configured again during the user-space start up. The only time ip_auto_config is really needed is during development when you have a root file system mounted via NFS. In production system I can leave it out.

Saving: 1.48 seconds

pmic_init and ehci_hcd_init

Also in the bootchart we saw that these two functions take a significant amount of time. The first is setting up the power management controller, which is essential to the operation of the chip. The second it initialising the EHCI USB host controller, which we also need. So nothing can be done to reduce these times.

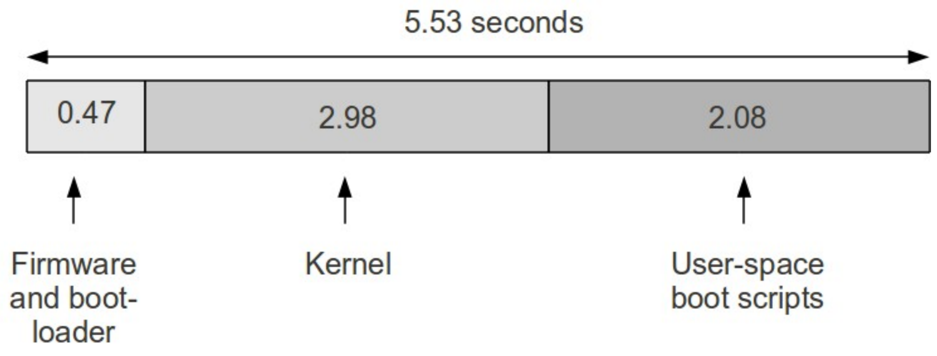
Loops-per-jiffy

Linux calibrates a delay loop by sitting in a busy wait for a number of timer ticks (“jiffies”) and then storing the result. But, the calculation will be the same each time, and can be set with the lpj command line:

```
console=ttymxc1,38400 quiet lpj=3997696
```

Saving: 0.23 seconds

After kernel optimisation



Now the boot time is of the order of 5 seconds, a 6 fold reduction on the initial configuration.

7. Other strategies

I could go on to reduce boot time to less than 5 seconds (but I won't). Here is a brief list of techniques I could try in my quest for "instant-on"

- Reduce the size of the kernel by pruning unwanted driver code
- Compile some drivers as modules and load them after the main application has started
- Optimise the boot-loader, for example turning on the processor caches early, or using DMA to copy kernel code from flash
- Add a quiet option to the boot-loader to reduce the time spent sending text out of the serial port
- Consider using an uncompressed kernel image: if the processor is slow it can take a significant time just to uncompress.

8. Summary

Device boot times do not have to be multiple 10's of seconds. I have given a range of options to get it down to, in my case, about 5 seconds. The biggest gain was eliminating unnecessary services from the boot scripts and the second biggest was to add the "quiet" option to the kernel command line. Next, choosing the right file system (UBIFS in my case) is important.

There is a downside to all of this. The more you step away from a standard distribution, such as Angstrom, the more maintenance you will have to do yourself and so the higher the skill level you need. Here are some useful links to pages where you can find more information on this subject.

Inner Penguin blog at

http://www.embedded-linux.co.uk

2net web site

http://www.2net.co.uk

Embedded Linux Wiki

http://elinux.org/Boot_Time