How to achieve fastest system startup sequences with your embedded system

By Kei Thomsen, MicroSys

This article discusses how to achieve fast startup times, especially in view of different operating systems like a general purpose OS or RTOS. Additionally an overview is given on boot media and their effects on startup behavior, together with practical measurements.

■ To achieve quick boot timings with complex embedded systems, sound knowledge about the internals, the architecture of the operating system and their interaction with the hardware are required. Smaller real-time operating systems are a perfect means to meet such requirements. They are specifically designed to control precisely the timing behavior of an embedded system and they are clearly arranged and configurable. Additionally smaller engineering teams are not overwhelmed by complexity and they can keep control of their development process more easily.

By the configurability of a RTOS, the functionality of the boot sequence can be optimized according to the given hardware design. Time-consuming load processes of hundreds of drivers, libraries or system code which might not even have been used are avoided. On the other hand, compared to more complex and feature-rich general purpose OSs for embedded systems like Linux or Windows variants, a more limited feature set has to be taken into account. Other aspects influencing system startup timings are hardware initializing functions and the boot media and method the operating systems are loaded from. Fast boot is another term to express a quick start of a system from power-on or reset to show at least a system prompt on a screen with loaded operating system, or have a window opened for the first user interaction. The timings to get a system up and running vary strongly by nature of the system, application and target market. A Windows desktop user might be used to waiting a couple of minutes before he can use his system, whereas in many deeply embedded environments the device is required to boot up in fractions of a second. A first conclusion is that for some applications a minutes-long startup sequence can be tolerated, while in others it is a must to restart the system in an instant. If time-critical functions need to be considered related to the startup behavior, the complete system design from component to system software level (including operating system) has to be the right fit for that. Secondly the media the boot software is loaded from has to be selected thoroughly. Flash, SD & CF Cards, rotating media or a network connection offer by nature different timings and have a strong influence on the cost structure of a system as well. Especially for deeply embedded systems, longevity of parts supply and maintainability might have an influence as well. As a short summary, fast boot has many aspects and a precise definition of it is not that easy. To be able to have a more defined basis for comparisons we will discuss the results of experiments we did on an i.MX 53 ARM based platform supporting different typical boot environments and operating systems. Generally we differentiate two different types of operating system. On one hand the standard OSs like Linux and Windows and on other hand real-time operating systems. Linux and Embedded Linux-Systems, representing standard OSs, are generally complex, have an extensive kernel, including many extensions, and the full functionality is merely seen by experts. Booting a Linux system means first loading and initializing the kernel and drivers, and then starting a large number of system services and additional programs. Here again it needs a lot of knowledge to understand the function and usage of the services and programs. Optimizations to achieve faster boot and startup times are not as simple as they should to be. Another aspect is optimizing a system after the functionality has been defined and implemented, but the startup time still requires to be tuned. The question here is how to assure the warranted system characteristics and properties by optimizing the system?

To face this problem right at the beginning, a very practical method is to plan and implement a test system based on minimal OS functionality. Avoid all the nice-to-have features and develop the optimal functions step-by-step. The required know-how and the additional development resources are to be considered accordingly. If it becomes obvious under the project work that new kernel

	SD/µSD	CF-Card	NOR Flash	NAND Flash
Connection	SD Controller	IDE Controller	CPU Bus	NAND Controller
Bus width (bits)	4	8	8 / 16 / 32 / 64	8
Speed (MB/sec.)	>20	5 – 8 (>20 DMA)	10 - 80	5 - 8
Size (typical)	1 - 32 GB	32 MB - 32 GB	2 - 64 MB	¼ – 4 GB
Price/MB	Extremely low cost	Very low cost	Expensive	Low cost
OS Startup (Sec.)	1,5	2,4	0,3	2,4
Pro	Simple to replace	Simple to replace	Extremely fast, direct execution	Direct soldered chips with high capacity
Contra	Mostly only consumer quality (MLC)	Mostly only consumer quality (MLC)	Less capacity, file system is not common	Defects by writing, difficult to replace (MLC & SLC)
Driver effort	Medium	Medium to high	Only for writing	High
Recommendation	Periodic replacement	Periodic replacement	No file system	Main use as "read only" device
Frustration factor	Medium	Medium	Low	High

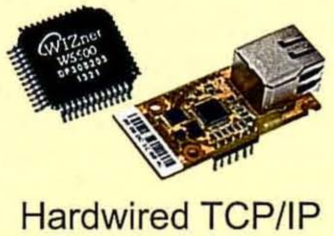
Table 1. Comparison of different storage media influencing system startup times

features or system services are required, then they can be added and activated later. Alternative approaches might be to add small libraries providing similar functions like a full system service solution. Why carry a full OpenSSL if only a MD5 is needed, which can be provided by small library. The system start of a Linux system depends on the applied Linux distribution, e.g. LTIB, Debian or ELDK. For optimizations of the startup sequence, special knowledge of the methods used by the selected distribution is needed. There are many startup methods available, for example, SysV-init, BSD-init, busybox, launchd, runit, systemd, upstart, just to name a few.

Totally different approaches are used in small and clearly defined real-time operating systems like Microware OS-9, QNX, VxWorks and others. Here, a special kernel is used exactly for the chosen CPU. The operating system functions for components like the MMU, cache, interrupt and exception handling etc. are precisely adapted and are open for optimizations. The hardware functions can be tuned with software drivers exactly to match the required timings. Only the pre-planned and desired function is integrated into the operating system to avoid all time-consuming, unnecessary and redundant program code. Such a clearly defined system is much smaller and starts faster by factors than a full blown all inclusive system. Thus, real-time operating systems have, due to their slimness, configurability and clearness, the capability to provide the fastest startup times compared to standard operating systems. Embedded RTOSs offer the beauty that one need not to be a dedicated expert to understand the functionality of each single element in the operating system, its internal assignments and dependencies. A RTOS like e.g. OS-9 is based on a clear architecture with a modular process and thread structure that is fully understandable by a developer and accordingly manageable. For example, modules like kernel, pcf, epicirq or sppro1000e are exactly defined in their function and do exist apart from each other. That means they can be loaded and added on demand to support system optimizations. The naming convention helps to understand the function of program modules and thus adds to the clarity of the system architecture. This results in a system structure whereby only required program functions are selected and in use. This approach is straight forward, comprehensive and much easier to validate and test. In contrast, the reverse process applied for standard operating systems, to remove functions to optimize, possibly with a trial and error method, implies many risks, especially due to unknown or damaged system dependencies. If we are talking about real or deeply embedded systems, then the boot medium

WIZnet.eu

... Internet for everything



Hardwired TCP/IP Chips & Modules

Simple

Hardwired TCP/IP + MAC + 10/100 PHY
A single chip Internet I/F for any 8 to 32bit MCU.

Stable

No OS and no Interrups needed. Free driver and application source code available.

Secure

Stack in Gate-Logic and separated buffer.
Linux drivers available.

electronica **A6.579**

meet us in Munich
11th - 14th November

WIZnet.eu

... Internet for everything

WIZnet Europe GmbH
Edisonstrasse 15
68623 Lampertheim
+49-6206-94623-0
info@wiznet.eu
http://www.wiznet.eu

is generally not a hard disk. Operating systems are loaded out of non-rotating memory media like NOR- or NAND-flashes and SD/µSD/CF cards. The boot medium strongly influences the timing of the system startup sequence. What are the differences between the types and resulting performance effects? The NOR-flash is a fast memory, linearly mapped into the address space. Depending on the hardware design it is 8/16/32/64 bits wide attached to the CPU with fixed defined access speed. It is the fastest type of all. NOR-flash types offer the beauty that the boot-monitor (e.g. U-Boot) is running directly out of it. An initial load or copy process into RAM is not required. Most RTOSs for example, are able to run directly out of flash, which adds additional system safety. The RAM is free for data, whereby the OS and application still resides in the flash, hence these important code elements cannot be damaged by a program error.

NOR-flashes however offer compared to NAND-flashes far less memory capacity per chip space and are much more expensive. Because of that they lose attraction in cost-sensitive designs, but nevertheless they are still used in environments requiring fastest startup times. The typical data rate is 10-80 MByte/sec and allows fastest access to boot and program code. Yet NAND-flashes are block-oriented devices and therefore not linearly seen in the address space.

Accesses to these memories have to be organized by dedicated drivers. Due to the organization and the 8-bit interface they are slower than the classical NOR-flashes. NAND-flash components are much cheaper compared to NOR-flash memory and have their main market in consumer electronic devices. Thus the NAND technology is very attractive for embedded designs, but especially in industrial rugged designs exposes the application to issues like long term availability or product lifecycle management.Similar considerations apply to SPI-NOR-flashes, which are NOR flashes connected via a serial protocol interface (SPI). Due to its non-linear organization, the program code must first be transferred from NAND-flash or SPI-NOR-flash to RAM before the U-Boot or the OS can be executed. A typical transfer rate here is 10-14MByte/sec. Similar to NAND are CF cards. CF cards are typically read by the boot monitor in PIO mode, at typically 5MB/sec. SD/µSD cards are connected with a 4-bit wide structure and offer with >20MB/sec transfer rate the best performance of the block-oriented medium. As an example, we have measured the startup times from reset to completely running an application for different environments using the RTOS Microware OS-9 loading a control application: NOR-flash, 0.3sec; SD-Card, 1.5sec; NAND-flash, 2.4sec.

Another strong influence on the boot timing is the use of a compressed or uncompressed OS image. The decompression time must be compared with the load time for a larger, uncompressed image and properly analyzed for an optimization. Another decision criteria is keeping the kernel small, because the OS drivers (with DMA and IRQ handling) are faster than the Booter (simple polling drivers), so that the components are loaded faster. Or using the variant "as we are already loading, load everything in one chunk". As the boot devices are too different in read performance, there is no generic statement possible, which way to go to get the most promising approach.

As concluded earlier, the initial system design has a strong influence on the boot performance, and if there are options on different boot media available, individual tests are required to find the optimum. Several tests showed up to a 25% time difference, e.g. for a Linux System a speed-up from 24 to 17 seconds, by using an uncompressed image (-3 seconds) loaded from SD Card (-4 seconds) instead of compressed image on NAND flash. As already described, the different boot media have strong effects on the timing of a system start. Other annoying thieves of time are the initialization sequences of Ethernet and USB interfaces. For initialization Ethernet communication starts an auto negotiation on the physical layer (PHY). Typically this takes between 1 and 3 seconds, if a cable is connected. Without a cable connected, it waits for a timeout, e.g. 5 seconds before it proceeds. If this happens within the kernel startup process, where no process scheduling is possible and nothing else can run at the same time, then it delays the system start until the action is finished. To achieve better boot results, it is by far better to start the network at a later point, so that it can run with normal scheduling in the background. To initialize the USB communication the USB bus must be scanned for the connected topology with all the hubs and devices. This might take up 10 seconds with nearly no CPU usage. It makes sense to start the USB stack as early as possible, so that it can run in the background during other devices and services are starting

Most embedded systems (at least during the development) have a serial line for the system start message and the shell prompt. As the serial line is attached to a PC by a terminal program, a first and very simple tool to analyze the startup timings is the terminal program itself. Most of the terminal programs (like TeraTerm) can use timestamp logging. Each received line gets a timestamp on arrival. Here we see the duration between the messages. Most times the message itself tells a lot about where it comes from. Example:

[Wed Sep 05 13:53:07.491 2012] NAND read: device 0 offset 0x200000, size 0x400000

[Wed Sep 05 13:53:08.171 2012] 4194304 bytes read: OK

It explains that reading 4MB from the NAND flash takes about 0.7 seconds. To get a meaningful time stamping, the baud rate should be as high as possible (>=115200Baud). Hint: As USB serial adapters are buffering the data internal and sending them as bulk blocks to the system, it can happen that multiple lines are getting the same time stamp. It is of advantage to use the internal serial line of the PC (if it is still available), to achieve a better time resolution, as the internal serial line directly reads the data without any delay like the USB has due to the buffering. ■

Product News

■ AdaCore: conference on reliable, safe and secure software

AdaCore announced that, along with partner Altran, it will be a major sponsor of the inaugural High Integrity Software Conference taking place in Bristol, UK on October 23rd 2014. HIS 2014 is a brand new UK conference for sharing information about key challenges and recent developments in high integrity software engineering.

News ID 1980

■ ETAS debuts at InnoTrans

ETAS will be making its first-ever appearance at InnoTrans, the International Trade Fair for Transport Technology, where the company will present innovative solutions for developing, testing, and integrating real-time-capable software designed for rail vehicles. ETAS RTA Solutions represent professional, made-to-order consulting and software engineering for real-time applications.

News ID 1905

■ N.A.T.: firmware v2.15 for NAT-MCH and NATview v2.13 now available

N.A.T. make version 2.15 of the firmware for the NAT-MCH-Family of products as well as the improved and extended version 2.13 of the JAVA based GUI NATview available to customers. The NAT-MCH family of products consists of: NAT-MCH, NAT-MCH-PHYS and NAT-MCH-PHYS80.

News ID 1892