RTLinux on Memory Constraint Systems

David Selvakumar and Chester Rebeiro
Center For Development Of Advanced Computing
No.1, Old Madras Road, Byapanahalli, Bangalore 560038, INDIA
{david,rebeiro}@cdacindia.com

Chester Rebeiro
Center For Development Of Advanced Computing
No.1, Old Madras Road, Byapanahalli, Bangalore 560038, INDIA
rebeiro@cdacindia.com

Abstract

Real Time Linux is rapidly finding application in several embedded systems because it is scalable, customizable and has a good development environment. However, problems arise on systems with low memory, and on systems that require to be booted fast. Linux is not optimized to boot fast, and requires at least 4MBytes of RAM.

This paper presents an embedded application: a Remote Terminal Unit (RTU), used in a Supervisory Control and Data Acquisition System (SCADA) to monitor and control critical process control plants. The RTU is based on Intels' 80486 microprocessor and has 2 MBytes of SRAM and 2MBytes of Flash memory. Besides this, the application requires the system to be booted in under one second.

FSMLabs' [11] RTLinux was tuned to run on this system. This paper discusses two aspects of the tuning. First, the embedding of RTLinux on this low memory system. Second, the modifications made to reduce the boot time of the system.

1 Introduction

The design of RTLinux is based on the dual kernel principle, in which a small Real-Time kernel (RTKernel) coexists with a general purpose operating system such as Linux [1]. The RTKernel provides a stable, predictable low-latency environment for real time tasks, while all non critical applications are run under the Linux kernel. The RTKernel relies on the Linux kernel for booting, system initialization, memory management, user interface etc. After the system has been booted by the Linux kernel, the RTLinux modules are inserted into the kernel. Running RTLinux on a system, will first require Linux to be run on the system.

The Linux kernel is available under the General Public License. The development of Linux is targeted to the workstation and server market, and not to custom made embedded devices. However, Linux is scalable, well documented and customizable [2] and can be scaled down for an embedded application.

Many embedded system devices are characterized by the limited amount of memory available and the absence of a disk based secondary storage memory. Generally, a small flash memory is used as secondary storage. These systems do not require the wide range of powerful features supported by Linux. Linux can be stripped to the bare minimum, to make it suitable for the embedded target.

Several mission critical systems, such as in military or process control, require the application to begin execution as fast as possible. These systems have to be booted quickly. Linux is not optimized for a fast boot, and takes 5 to 6 seconds to boot. This time is unacceptable for such critical systems and has to be reduced.

Several efforts have been made to scale Linux for an embedded application. MiniRTL [3] is a 2.2 kernel based real time Linux that fits into a floppy disk. uClinux [10] is an embeddable version of Linux for systems without a memory management unit. There have been efforts to reduce the boot time of Linux. Quickboot from FSMLabs [11] has reduced the boot time to under 500 milliseconds on a PowerPC405GP embedded processor board.

This paper presents our efforts in tuning RTLinux (and Linux) to suit our embedded system application: a Remote Terminal Unit (RTU). FSM-Labs' RTLinuxpro 1.2 and 2.4.16 Linux kernel with the RTLinux patch was used for the purpose.

The paper is structured as follows: section 2 describes the RTU hardware and the system architecture. Section 3 presents techniques to design a minimum Linux system. Section 4 discusses techniques to reduce the boot time of the system. Section 5 has the conclusion.

2 System Architecture

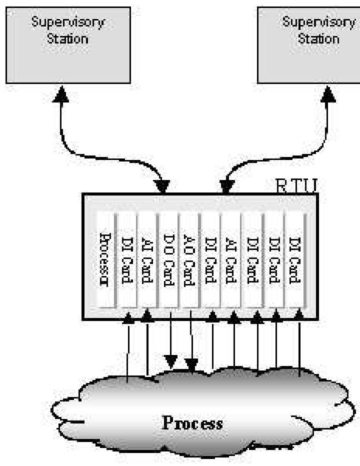

FIGURE 1: *SCADA Architecture*

Industrial process plants such as power and steel plants,use Supervisory Control And Data Acquisition (SCADA) to monitor and control their processes. RTUs form the bottom layer of the SCADA architecture *(Fig: 1)*. The RTUs monitor and control the process through four Input/Output (I/O) cards: Analog Input, Digital Input, Analog Output and Digital Output. They maintain a consistent database of the process parameters.

Clients to the RTU database are Supervisory Stations, which process, analyze and display RTU data providing an overall view of plant status. Communication between the RTU and the Supervisory Station follow the IEC-870-5 (101) [9] serial protocol standard to ensure data integrity. A maximum of four Supervisory Stations can communicate with an RTU.

The RTU contains one processor board along with a maximum of 14 I/O slots. The processor board is based on Intel 80486 microprocessor at 66MHz having a 2 MByte flash memory and 2 MByte SRAM. The communication between processor and the memory devices is with a 32 bit bus at 33MHz. There are four standard UARTs present for communication. There are three Programmable Interval Timers, one used by the scheduler, the other used by the watch dog, and the third used by the application.

Shutdown of the RTU for a long duration could hamper the monitored process. Therefore a reset from the watch dog should cause a quick reboot of the system.

3 Reducing Memory Requirements

Conventional Linux kernel, system utilities and associated libraries require large system RAM [3] . Many embedded applications are constraint by the amount of memory available on board. Running Linux on such systems will require customizations and optimizations as discussed below.

3.1 Reducing Linux kernel size

The first step to reducing the size of the Linux kernel is to remove all unrequired subsystems. Only the required modules must be compiled. The modules can be selected by the kernel configuration utility *make xconfig* or *make menuconfig*.

The kernel size can be further reduced by fine grained modifications to the source code. For example, threads in the kernel which are not used need not be started. Size of global arrays can be reduced to what is required for the system. The number of devices (tty, mtd for example) can be restricted to the required.

Functionalities within the kernel that are not used by the application can be eliminated. For example, removing the socket system call support from the kernel reduces the size by around 30 KBytes.

Such fine tuning of the kernel is not easy. Care has to be taken to find all dependencies before modifying the code. Even though a lot of effort was spent on fine tuning the kernel source, the amount of reduction in kernel size was marginal. Therefore, such tweaking of the kernel is not advised.

3.2 Efficient utilization of memory

On 32 bit Intel Architecture (IA32) systems, the kernel leaves the first 1 MBytes memory unutilized [4]. This is to support BIOS based systems that use this memory area to store configuration data, the Video BIOS, etc.

The kernel boot starts at location 0x100000. To boot Linux on the RTU, we need to make use of the entire limited memory (2 MBytes) available. The kernel has to be tuned to utilize the memory below 1 MBytes. The kernel start location must be changed from 0x100000 to a lower location (0x20000 for example) in memory.

- The kernels' setup code *(arch/i386/boot/setup.S)* moves the processor from real mode to protected mode. The setup code is hardcoded to jump to 0x100000, the kernel start location. This has to be changed to a more suitable location (0x20000 in our example).
- The page table initialization is done in *(arch/i386/kernel/head.S)*. The provisional page tables are contained in the tables *pg0* and *pg1*. *pg0* and *pg1* are present at offsets 0x102000 and 0x103000 in physical memory respectively. These tables map the linear address 0x0 through 0x7FFFFF and the linear address 0xC0000000 through 0xC07FFFFF to physical address 0x0 through 0x007FFFFF. The page directory is present in *swapper_pg_dir*. It contains information about the location of the page tables. If we change the kernel start location to 0x20000, the entries in the *swapper_pg_dir* should also be changed appropriately.
- The linker in the kernel makefile produces a non relocatable kernel image starting at the virtual address 0xC0100000. The linker descriptor script *(arch/i386/vmlinux.lds)* has to be modified with the new start location of the Linux kernel (0xC0020000).

3.3 Selecting Filesystems

The Virtual File System (VFS) of Linux allows several types of file systems to coexist on the system. For Flash Memory, the most popular storage device for embedded systems, Linux supports the following file systems.

- **cramfs :** is a compressed read only file system. It takes minimum storage space on the flash and allows random page access by compressing each page separately.
- **jffs :** is a Journalling Flash File System [7] developed by Axis Communications [16]. It is a log-structured file system, with the nodes containing data and metadata stored sequentially and progressing linearly through the storage space available. It allows read-write access.
- **jffs2 :** is the second version of jffs, which supports compression, automatic leveling and a hard power-down safe filesystem.

For the RTU we require a file system that has a small image, does not bloat the kernel size and boots quickly. We made a comparison *(Table 1)* of these criteria for the three file systems with a 2 MByte flash (AMD 29F040 interleaved by 4).

Description	**cramfs**	**jffs**	**jffs2**
Kernel size increased by	20KB	31KB	81KB
File system image size	152KB	349KB	170KB
Boot time (seconds)	0.530	9.11	3.07

TABLE 1: *Flash system comparison*

Cramfs offers the best file system image size and has the least code size. (The kernel size in the Table 1 corresponds to the increase in size of the uncompressed kernel) It also has the fastest boot time. The boot times for jffs and jffs2 are high because they scan the entire 2 MByte flash during initialization. For file systems that require only read access, cramfs is the obvious choice. However, for systems requiring both read and write flash access, a combination of cramfs and jffs2 will yield best results [6]. This requires the partitioning of the flash. All directories requiring read only permissions, such as /lib, /, /usr etc. can be placed in the cramfs partition. All directories requiring read/write access can be placed in the jffs2 partition. The jffs2 partition should be made as small as possible to reduce boot times. The compromise one has to make here is the increased size of kernel image. The flash partitioning would require support from the MTD (Memory Technology Devices) driver. This adds 8 KBytes. Besides this, the kernel has to support two file systems, resulting in an increase of 109 KBytes in the kernel size.

3.4 Shrinking user space programs

A desktop or server Unix system supports several utility programs and uses powerful libraries. Such powerful libraries and range of utility programs are not required for the embedded applications.

3.4.1 System Utilities

A subset of the utilities present on a standard Unix system can be used to build the file system for an embedded system. Utilities selected should depend on the embedded application. For example: networking utilities such as telnet or ping need not be added to a system which does not support networking. Utilities such as df, du, cat etc. would be useful on systems such as the RTU, which handle a lot of data.

Several packages are available that support the common Unix utilities and are tuned for an embedded system. They generally are stripped versions of the original utilities, and may not contain all options of the original utility.Some of the available tools are listed below.

- **Busybox :** [13] combines tiny versions of many common Unix utilities into a single executable. Using a single executable has the advantage of having only one parser and a common set of base functions. The drawback is that at execution time the entire application (containing several sub programs) has to be loaded to RAM.
- **Embedded Utilities :** [14] provide statically linked versions of several Unix utilities. They use dietlibc[12] to provide utilities optimized for size and speed.

3.4.2 Libraries

The standard glibc library is a large and powerful C library. The applications run on an embedded system generally do not require such powerful library support. Several alternate libraries are present with reduced functionality and size. These are more suited for the embedded application. uClibc [15] and dietlibc are examples of such libraries. The *Table 2* shows the reduction in the executable size for a simple 'Hello World' application on a desktop with different libraries.

Description	**static**	**shared**
glibc(v2.2.4)	1.4MB	13KB
uClibc	12KB	2KB
dietlibc	2KB	-

TABLE 2: *Performance of C Libraries (for a 'Hello World' program)*

4 Reducing Boot Time

With the minimum configuration, Linux takes around six seconds to boot the RTU and start the application. Insertion of the RTLinux modules into the system using the 'insmod' utility takes an additional four seconds. This boot up time is too long for our application. Luckily there are ways to reduce the boot up time for the system. These are discussed below.

4.1 Using an uncompressed Linux kernel

Normally Linux kernel is built as a compressed image (bzImage or zImage). This image is uncompressed during booting. The uncompression procedure consumes a lot of time. A faster method would be to store an uncompressed Linux kernel in flash. This would bypass the requirement to uncompress the kernel while booting.

The uncompressed kernel is copied to RAM during the initial boot phase. The size of the uncompressed image is around three times that of the compressed image. This results in three times the amount of memory required in flash to store the kernel, and three times the amount of copying from flash to RAM during bootup.

Another method of booting the kernel is execute-in-place (XIP)[5], where an uncompressed kernel is stored in flash and executed from flash. Only the kernel data segments need to be copied to RAM. This eliminates the uncompression as well as the copying. The only overhead of XIP is the slower access time of flash memory, resulting in slower execution of programs.

4.2 Reduce RTLinux initialization time

The basic functionality of RTLinux is initialized in a system by inserting five modules into the kernel: rtl.o, rtl_time.o, rtl_posixio.o, rtl_fifo.o and rtl_sched.o. The Busybox 'insmod' program, was compiled with dietlibc to provide a small, fast application to insert the rtl modules. It however takes around four seconds *(Table :3)* to insert all five modules.

One way to reduce this time is to link all the five modules together to form a single rtl module (rtl_master.o). Inserting this module with the same 'insmod' utility reduces the initialization time to 1.84 seconds.

The initialization time can be further reduced linking the rtl_master.o module along with the kernel during the kernel compilation, and completely eliminating the 'insmod' program. This reduces the RTLinux initialization to around 30 milliseconds. Linking the rtl modules along with the kernel can be easily done (though tedious) by modifying the makefiles of Linux kernel and RTLinux. All the RTLinux modules must be compiled without the -*DMODULE* compilation flag. The *__init* attribute should be added to all *create_module* functions in the rtl modules to ensure that it gets invoked during the module initialization.

When a module is inserted into the kernel using 'insmod', all non static functions in the module have entries in the kernel symbol table. However, when the module is linked with the kernel, only functions which are exported with the macro *EXPORT_SYMBOL* get added to the kernel symbol table. All rtl API functions have to be exported this way, and the files should be compiled with the -*DEXPORT_SYMTAB* flag.

4.3 Disabling kernel printk

The RTU uses a serial console as the default console. All printk boot messages are directed to the serial console. The serial console is a slow device; 1.74 seconds is consumed in just printing the debug messages to the console. The Linux kernel has a command line option 'quiet', which when enabled disables the printk messages. The messages are still buffered, and can be viewed by the 'dmesg' command.

	Description	**Boot time (seconds)**	**Reduced by (seconds)**
1	Linux kernel (LK) + RTLinux insertion	10.139 5.98(LK)	-
2	Linux kernel + rtl_master insertion	8.3 5.98(LK)	1.839
3	Linux kernel with linked rtl_master	6.009	2.291
4	kernel from 3 with quiet enabled	4.269	1.74
5	kernel from 4 without get_cmos_time()	0.779	3.49
6	kernel from 5 without delay loop calibration	0.570	0.209

TABLE 3: *Bootup timing*

4.4 Disabling Real Time Clock (RTC) synchronization

The *get_cmos_time()* function is called during the kernel booting to read the current time from the RTC. The kernel loops until the start of the next second. This happens when the Update-In-Progress flag toggles from 1 to 0. This is required to synchronize the Linux time with the RTC. For system having an RTC this may take as long as one second. The kernel loops 2000000 if the Update-In-Progress flag does not toggle. On the RTU, this takes 3.49 seconds. For systems without an RTC, or systems which do not require to synchronize with the current time, the call to this function can be disabled.

4.5 Disable delay loop calibration

The function *calibrate_delay()* is called at kernel startup to calculate the value for the variable *loops_per_jiffy*. This value is used by the kernel for executing short delays. The calibration of delay takes around 200 milliseconds on the RTU. This time can be reduced by hardcoding the *loops_per_jiffy* variable. For the RTU it is hardcoded to 19328 (3.86 BogoMIPS).

5 Conclusion

Linux has a good development environment, and supports a large number of hardware platforms and devices. It is scalable, easily customizable and has good documentation. RTLinux provides a deterministic platform for hard real time applications. These attributes make Linux with RTLinux a suitable platform for embedded applications.

In this paper, we demonstrated the scalability of Linux. Although the OS is designed targeting large workstations and servers, we were able to scale it down to a deeply embedded system with minimum hardware resources and having critical memory constraints.

The size of the Linux kernel image is around 510 KBytes. This is still huge, however reducing the size further is difficult without modification of the kernel source. There are however several possibilities for reducing the boot time of the system further. Having an execute-in-place kernel will reduce the boot time. Having a partial-execute-in-place kernel, where only part of the kernel is copied to RAM (the part which is used most frequently), should reduce the boot time and provide a better runtime performance. A threaded initialization of drivers can further reduce the boot time.

Acknowledgments: We express our sincere gratitude to Mr. N. Mohan Ram, Center Head, CDAC, Bangalore and Dr. N. Sarat Chandra Babu, Center Head , CDAC, Hyderabad for their support and cooperation during the execution of the project.

References

[1] V. Yodaiken, M. Barabanov, 1996, *Real-Time Linux.*

[2] A. Przywara, R. Kusch, D. Naunin, *Real-Time Operating Systems on Small Embedded Devices for Industrial Control and Communication.*

[3] N. Mc Guire, 2000, *MiniRTL - Hard Real Time Linux for Embedded Systems*, 2ND REAL TIME LINUX WORKSHOP.

[4] A. Rubini, 1997, *Booting the kernel*, LINUX JOURNAL (KERNEL KORNER, JUNE 1997).

[5] T.R. Bird, 2004, *Methods to Improve Bootup Time in Linux*, PROCEEDINGS OF THE LINUX SYMPOSIUM, pg79-88.

[6] C. Brake, J. Sutherland, 2001, *Flash Filesystems for Embedded Linux Systems*, ELJONLINE, JULY 2001.

[7] D. Woodhouse, 2001, *JFFS : The Journalling Flash File System.*

[8] D.P. Boviet, M. Cesati, 2003, *Understanding the Linux Kernel, Second Edition* , O'Reilly & Associates Inc, ISBN:0-596-00213-0.

[9] IEC 870-5, IEC Technical Committee.

[10] uClinux, *(http://www.uclinux.org).*

[11] FSMLabs RTLinux, *(http://www.fsmlabs.com).*

[12] Dietlibc *(http://www.fefe.de/dietlibc)*

[13] Busybox *(http://www.busybox.net)*

[14] Embedded Utilities *(http://www.fefe.de/embutils)*

[15] uClibc *(http://www.uclibc.org)*

[16] Axis Communication JFFS *(http://developer.axis.com/software/jffs)*