BUILD APPLICATIONS ON LINUX RTOS
(EMBBEDDED SYSTEM)

Rakesh Trivedi, Darshil Modi, Nehal Parmar
Vlsi & Embedded Systems Design,
Gujarat Technological University,
Ahmedabad, Gujarat, India
trivedi_rakesh4u@yahoo.co.in, darshil_modi8@yahoo.co.in, nehal213@yahoo.in

Abstract **- Whether you are planning a move to embedded Linux or are just considering the investment needed to convert existing applications to run on embedded Linux, this paper will help you understand the transition process, assess the challenges and risks, and appreciate the benefits realized from such a move. This paper addresses how to map legacy architectures onto Linux, options for migrated application execution, API and IPC translation, enhanced reliability.**

Keywords- Linux RTOS Architecture, Process and Thread Creation in RTLinux, Building Benefits on Linux RTOS

*****

I. INTRODUCTION

Embedded Linux is rapidly encroaching upon application spaces once considered the exclusive domain of embedded kernels like VxWorks, pSOS, and in-house platforms. Industry analyst show embedded Linux and open source garnering up to one third of 32 and 64 bit designs, more than twice the share of any other embedded OS. So, whether you are planning a move to embedded Linux or are just considering the investment needed to convert your existing application to run on embedded Linux, this paper will help you understand the transition process, assess the challenges and risks involved, and appreciate the benefits realized from such a move.

1.1 Migration Execution Architectures

While Linux increasingly takes the place of traditional RTOSs, executives, and kernels, the architecture of the Linux operating system is very different from legacy OS architectures. Moreover, there exists more than one means to host legacy RTOS-based applications on a POSIX-type OS like Linux. The following section lays out three approaches to migration, from conservative means that preserve legacy attributes and architecture to more extensive revamping of code and application structure.

Emulation, Virtualization, and Native

This section compares and contrasts the three most relevant migration and re-hosting paradigms for legacy software under Linux:

1. RTOS API emulation over Linux
2. Run-time partitioning with virtualization
3. Full native Linux application port

1.2 RTOS Emulation over Linux

For legacy applications to execute on Linux, some mechanism must exist to service RTOS system calls and other APIs. Many RTOS entry points and stand-alone compiler library routines have exact analogs in Linux and the glib run-time library, but not all do. Frequently new code must intervene to emulate missing functionality. And even when analogous APIs do exist, they may present parameters that differ in type and number.

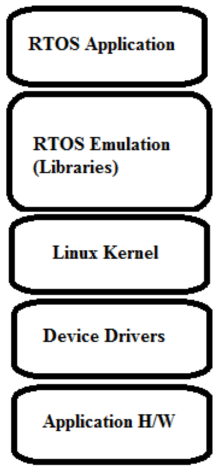

Figure-1 RTOS Emulation over Linux

A classic RTOS can implement literally hundreds of system calls and library APIs. For example, VxWorks documentation describes over one thousand unique functions and subroutines. Real-world applications typically use only a few dozen RTOS-unique APIs and call functions from standard C/C++ libraries for the rest of their (inter)operation.

To emulate these interfaces for purposes of migration, developers only need a core subset of RTOS calls.

Many OEMs choose to build and maintain emulation lightweight libraries themselves; others look to more comprehensive commercial offerings from vendors such as MapuSoft. There also exists an open source project called v2lin that emulates several dozen commonly used VxWorks APIs.

1.3 Partitioned Run-time with Virtualization

Virtualization involves the hosting of one operating system running as an application “over” another virtual platform, where a piece of system software (running on “bare metal”) hosts the execution of one or more “guest” operating systems instances. In enterprise computing, Linux-based virtualization technology is a mainstream feature of the data center, but it also has many applications on the desktop and in embedded systems.

Data center virtualization enables server consolidation, load-balancing, creating secure “sandbox” environments, and legacy code migration. Enterprise-type virtualization projects and products include the Xen Hypervisor, VMware and others. Enterprise virtualization implements execution partitions for each guest OS instance and the different technologies enhance performance, scalability, manageability and security.

Embedded virtualization entails partitioning of CPU, memory and other resources to host an RTOS and one or more guest OSs (usually Linux), to run higher-level application software. Virtualization supports migration by allowing an RTOS-based application and the RTOS itself to run intact in a new design, while Linux executes in its own partition. This arrangement (see Figure 1.) is useful when legacy code not only has dependencies on RTOS APIs but on particular performance characteristics, for example real-time performance or RTOS-specific implementations of protocol stacks.

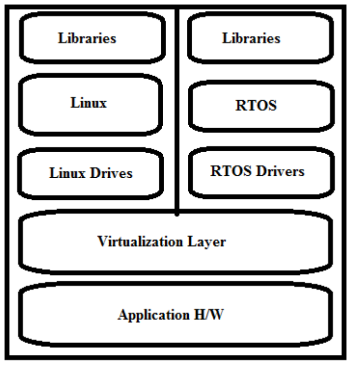

Figure-2 Partitioned Run time with Virtualization

Embedded virtualization as such represents a short and solid bridge from legacy RTOS code to new Linux-based designs, but that bridge exacts a toll OEMs will continue to pay legacy RTOS run-time royalties and will also need to negotiate a commercial license from the virtual machine supplier.

A wide range of options exist for virtualization, including the mainstream KVM (Kernel-based Virtualization Manager) and Xen. Embedded-specific paravirtualization solutions are available from companies like Virtual Logix.

1.4 Mapping Legacy Constructs onto Linux

The above architecture descriptions readily suggest a very straightforward architecture for porting RTOS code to Linux: the entirety of RTOS application code (minus kernel and libraries) migrates into a single Linux process; RTOS tasks translate to Linux threads; RTOS physical memory spaces, (i.e., entire system memory complements), map into Linux virtual address spaces – a multi-board or multiple processor architecture (like a VME rack) migrates into a multi-process Linux application as in Figure 4 below.

1.5 Process and Thread Creation

Whether you use RTOS emulation for Wind River VxWorks and pSOS, or perform your port unaided, you will ultimately have to make decisions regarding whether to implement RTOS tasks as processes or as threads. While at its heart, the Linux kernel treats both processes and threads as co-equal for scheduling purposes, there are different APIs for creating and managing each type of entity, and performance and resource costs (and benefits) associated with each.

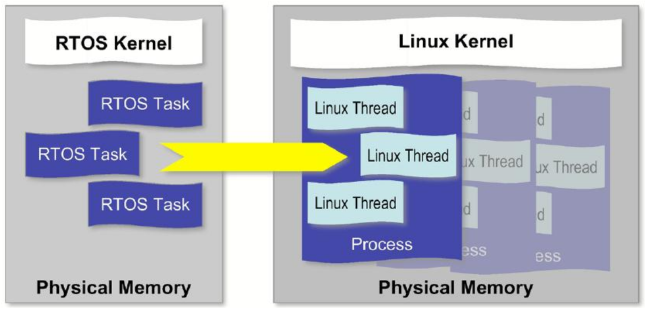

Figure-3 Mapping RTOS tasks onto Linux Threads

The kernel mechanism for creating processes is the fork () system call. Linux process creation is not intentionally a more cumbersome operation – Linux processes are heavier because they offer greater benefits of protection and reliability.

Forking New Processes

RTOS task and thread creation in both RTOSes and Linux essentially identify existing program functions as new schedulable entities (as in VxWorks task creation). By contrast, the Linux system call/API fork () causes the currently executing file to split, amoeba-like, into identical copies, a parent and a child. The parent and child initially only differ in their PID (Process ID), so the first thing programs do after a fork is to ponder, existentially, who am I? This deliberation is accomplished most often with a switch statement in C. The return value of fork () for the parent will be the child's PID, whereas the child will see the return as 0. Thus, the parent can "watch over" the child and each "knows" its identity. Example of fork function below:

```
    #include <sys/types.h>
    #include <unistd.h>
    #include <errno.h>
    pid_t new_PID;
    new_PID = fork();
    switch (new_PID) {
    case 0 : /* child code runs here */
    printf("I am the child -- my PID is ??\n");
    break;
    case -1 : /* oops - something went wrong */
    exit( errno );
    break;
    default : /* parent code runs here */
    printf("I am the parent -- my child's PID is %d\n",
    new_PID);
    break;
} /* switch */
```

Thread Creation

Thread creation with the function clone() system call or the function pthread_create() API is altogether a simpler affair, since all threads within a process share the same address space, file descriptors, etc.

Creating new threads proceeds as follows:
1. Lay out new stack in current user process space.
2. Create scheduler entry.
3. Assign new ID (TID).
4. Schedule new thread or wait per semantics of pthreads interface.

Context Switch Implications

Switching among threads and processes involves different amounts of effort and context saving. The fastest context switch is of course among threads running in a single process-based virtual address space. Switching between threads across process boundaries involves TLB (Translation Look-aside Buffer) spills, reloading of page translation table entries, and potential saves/restores of additional context such as FPU, MMX, Altivec, and ARM co-processor registers.

1.6 The Porting Process

The process for porting from a proprietary RTOS to embedded Linux is really no different from moving any application across host platforms, although the dependencies are more involved. Let's start with a discussion of the basic steps required and subsequently address key dependencies such as APIs and IPCs.

Basic Steps:

Whatever the particulars of your legacy code base, you and your team will likely follow these elementary steps:

1. Set up a Linux-based cross development environment including cross development tools (e.g., MontaVista Linux Professional Edition with DevRocket).
2. Copy RTOS application source tree to development environment.
3. Modify build scripts and IDE configurations to link emulation libraries (if any).
4. Modify/alias pathnames and/or modify source files to reference substitute header files (original RTOS header files can introduce conflicts with native Linux headers).
5. Add #includes for Linux header files to your application sources themselves (usually stdio.h,

stdlib.h, string.h, unistd.h, and errno.h) or via emulation headers (if any).

6. Attempt to make/build and examine results.
7. FIRST resolve symbolic issues for implemented APIs (e.g., simple naming and type-safe linkage issues).
8. Address unimplemented APIs and data structures.
9. Repeat steps 5-8 as needed
10. Tune performance, as needed, using tools
11. Selectively recode and re-architect to leverage native Linux constructs.

1.7 IPCs and Synchronization

Every operating system, whether general-purpose or embedded, supports inter-task communication and synchronization in a slightly different way. The good news is that the most common set of IPC (inter-process communication) mechanisms found in RTOS repertoires have ready analogues in embedded Linux;. Indeed, Linux is extremely rich in this area. The bad news is that RTOS-to-Linux mapping is seldom completely one-to-one and that even when apparent IPC equivalents exist, their scope may be focused on communications among processes rather than among lighter-weight threads most analogous to RTOS tasks, with subtly differing semantics.

Mechanisms

Here is a brief discussion of the mapping of RTOS IPCs and synchronization mechanisms onto their Linux equivalents as implemented for C language programming. The focus is on mechanisms supplied through core Linux system calls and common libraries. The world of Linux and open source is vast, however, and many additional options exist as patches and add-on libraries from other sources.

Semaphores and Mutexes

Linux offers two levels of compatibility with RTOS synchronization and mutual exclusion mechanisms. In general the SVR4 semaphores are the most robust and most used, especially for synchronization among Linux processes. In contrast, the POSIX.1c interfaces offer mutex/condition variable-based synchronization and mutual exclusion.

Queues and Mailboxes

Most RTOSs offer lightweight queuing constructs or mailboxes for passing discrete messages and sometimes light payloads among tasks. Linux has a rich repertoire of message-passing capabilities, starting with named pipes and FIFOs, and including SVR4 queues and POSIX.1b mqueues.
Example of Use POSIX mmap() to access memory-mapped peripherals as below:

```
    #include <sys/mman.h>
    #define REG_SIZE 0x4 /* device register size */
    #define REG_OFFSET 0xFA400000 /* physical
    address of device */
    void *mem_ptr; /* de-ref for memory-mapped access
    */
    int fd; /* file descriptor */
    fd=open(“/dev/mem”,O_RDWR); /* open phys
    memory (must be root) */
    mem_ptr = mmap((void *)0x0, REG_SIZE,
    PROT_READ+PROT_WRITE,
    MAP_SHARED, fd, REG_OFFSET);
/* actual call to mmap */
```

Timers and Task Delays

RTOS-based code makes extensive use of both software and hardware timer implementations, as well as task delay APIs (e.g., VxWorks taskDelay()). Linux offers support for large complements of timers and alarms as well as a variety of delay options. RTOS timers translate well into Linux interval timers and alarms (setitimer() and alarm()) that generate signals on timeout.

The key difference between Linux timers and most RTOS timer implementations is that RTOS timer APIs tend to quantify time in terms of RTOS system clock ticks while Linux strives to use “real” time (seconds, microseconds, or nanoseconds).

Delays

RTOS task delay calls translate into a family of sleep APIs: sleep() and nanosleep() are program calls that wait an appropriate number of seconds or nanoseconds, and usleep is a shell utility that pauses for a specified number of microseconds.

Periodic Task Execution

While most RTOSs use timers to implement long-term periodic task execution (once per hour, once per day tasks), Linux offers crone for spawning such activities at the process level.

Clock Resolution

With earlier implementations of Linux, the OS offered the above timer and delay interfaces, but the clock resolution of those mechanisms was extremely coarse. The 2.6 Linux kernel supports high resolution timing as a standard feature, in theory limited only by the granularity of the system clock. You can determine the system clock resolution with the call clock_getres().

Watchdog Timers

RTOSs use watchdogs most commonly to enhance system reliability: programmers pepper their code with watchdog timer resets, so that should a watchdog ever expire (time out), it will be indicative of a critical fault necessitating a reboot.

While Linux offers other means to increase system robustness, applications may still need watchdog-like functionality. Timer signal handlers can easily emulate such terminator behavior, and standard watchdog code also exists for many board-level configurations.

1.8 Building Benefits on Linux RTOS: Improved Reliability

The basic architecture of an RTOS-based application has changed little in the last 20 years, despite huge advances in microprocessors and other aspects of hardware design. RTOS applications are structured as a set of tasks (C functions, typically), statically linked to run-time libraries (including the RTOS kernel itself). These tasks reside and execute in a single physical address space (in RAM or sometimes in ROM) that they share with each other and with global application data, system data, application and kernel stacks, memory-mapped I/O ports, and the RTOS kernel itself.

1.9 Other Issues

This paper focuses on APIs, IPCs, and porting architectures. Other interesting RTOS migration issues not targeted here include:

- Linux real-time performance (including native real-time vs. kernel substitution, and preemption)
- Scheduling priority and policies Scaling Linux for embedded resource footprints
- Booting embedded Linux.
- Embedded Linux spindle-less file systems (RAM, CramFS, JFFS, etc.)

Conclusion

The move is on. Developers are leaving behind first generation RTOSs in search of more reliable and open embedded platforms like Linux.

While the migration from these traditional systems does present a variety of challenges, the benefits far outweigh the investment needed to move to embedded Linux. The risk doesn't arise from leaving behind your familiar environment, tools, and APIs – the real risk lies in standing still while the embedded and pervasive systems development communities move forward, at Internet speed.

By following the steps outlined above, you can successfully build your existing legacy RTOS code to a modern embedded Linux platform.

References

[1] Linux Foundation. Carrier Grade Linux Specifications, versions 3.2 and 4.0, 2007.
[2] Open Group. The Open Group Base Specifications Issue 6 IEEE Std 1003.1, 2004.
[3] Weinberg, William. "*Porting RTOS Device Drivers to Embedded Linux*," Linux Journal n. 126: October 2004.
[4] Weinberg, William. "*Moving from a Proprietary RTOS to Embedded Linux.*" RTC Magazine, April 2002