Adaptability, Extensibility, and Flexibility in Real-Time Operating Systems

Pramote Kuacharoen, Tankut Akgul, Vincent J. Mooney, and Vijay K. Madisetti
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332
U.S.A.
{pramote, tankut, mooney, vkm}@ece.gatech.edu

Abstract

In this paper, we present a mechanism for runtime updating of all kernel modules of a highly modular dynamic real-time operating system. Our approach can help solve the lack of adaptability, extensibility, and flexibility of existing real-time operating systems. The dynamic real-time operating system will efficiently support a wide range of applications since any kernel module can be dynamically loaded at runtime to exactly suit the applications without necessitating a reboot of the system.

1. Introduction

Existing Real-Time Operating Systems (RTOSs) provide fixed interfaces and implementations of services and resources. The RTOS services are provided as a set of modules or libraries. Often, the RTOS and an application are compiled together. The services needed for the application are selected by simply setting flags at the time of the application build [1]. As a result, this standard RTOS compilation methodology prevents updating the application or the RTOS at runtime. However, future embedded systems are likely to include a multitude of applications, some subset of which changes every few months (or possibly every few weeks in some cases).

Even if a static RTOS were to include all possible services, the RTOS would likely become inefficient and insufficient for emerging applications, especially when real-time considerations are included. A static RTOS cannot efficiently support a wide range of applications with different service demands. For example, a priority-based scheduler is suitable for applications where the deadline of the highest-priority task is the most important. However, this scheduler does not work well in scheduling network packets such as video-on-demand and streamed audio, where quality of service is required. In order to support these applications, the RTOS must be rewritten, compiled, and loaded to the device. Fortunately, today's embedded systems tend to support applications of one type, e.g., voice or multimedia. However, tomorrow's embedded devices will surely be much more diverse in the application set to be supported.

In this paper, a totally flexible, easily extensible, and highly modular Dynamic Real-Time Operating System (DRTOS) is introduced. Each kernel module can be changed or integrated to the kernel at runtime. Kernel services can be specific to each application. The DRTOS is adaptable, including all parts of the kernel, even the part responsible for dynamic installation of new kernel modules! Therefore, the DRTOS can provide efficient and sufficient services and satisfy any application's requirements since any kernel module can be dynamically loaded to exactly suit the application.

2. Background and Motivation

With the rapid growth of the embedded industry, applications such as cellular phones and personal digital assistants require more and more functionality in order to sell in high volumes. Third generation cellular phones will also be upgradeable over the air to allow more internal features to be added without the need for physically going into a store. To best support new features efficiently, all RTOS kernel modules must be updateable at runtime.

An operating system, which enables system services to be defined in an application-specific fashion, offers finer-grained control over a machine's resources to an application through runtime adaptation of the system to application requirements [2]. The services are efficient in both functionalities and performance for the application. These services come with applications and are loaded when the applications need them. This requires a modular design of the RTOS. To provide modularity and

performance, operating system kernels should have only minimal embedded functionality [3].

Our primary motivation is to allow all kernel components to be updated at runtime on a heterogeneous multiprocessor architecture, which was not addressed in the previous work [2,3,4]. The approach presented here can ease RTOS management by omitting a reboot of the system each time a kernel update occurs. All kernel components are modular and can be dynamically loaded and unloaded as needed. For example, a fixed priority-based scheduler can be updated to an earliest deadline scheduler, written by the application developer, at runtime when the application's requirements change. If a bug fix is necessary in one of the DRTOS modules or a newer version module is ready to be integrated to the system, this can be accomplished without recompilation of the whole system. For future work, we will address security issues such as keeping the kernel space free from malicious users.

3. Methodology

To support adaptability and flexibility in real-time operating systems, the kernel must be simple to enable easy modification and allow applications to implement system modules and services. In this system, dynamic runtime loading and linking are necessary. However, the methodology used in general-purpose real-time operating systems does not fully support such dynamic loading and linking of all kernel modules. A methodology to support adaptability and flexibility in real-time operating systems is presented in the following sections, starting with dynamic instantiation of kernel modules, dynamic update of modules, and kernel-level API support.

3.1. Kernel Modules

Runtime updating of a real-time operating system component requires each kernel component to be designed as a module. Each module has its own text and data sections. Therefore, switching of one module with another consists of changing both the text and the data sections in the kernel space. This requires dynamic linking of new text and data into the kernel space. The appropriate modules for supporting the application are integrated into the RTOS and unneeded ones can be removed from the RTOS at runtime.

For a modular and reusable design, an object-oriented kernel provides the application programmer a customizable operating system [5]. However, to reduce the overhead of object-oriented programming, we implement the kernel modules in C. As illustrated in Figure 1, a module consists of text and data sections. In the data section, there are module variables and APIs. The module variables and the APIs are equivalent to class fields and methods in object-oriented programming.

Since each module can be installed anywhere in the kernel space, absolute address references to the module's variables are not feasible. The executable code must be position independent. In order to implement Position Independent Code (PIC) without modifying the compiler, the module global variables are aggregated into one structure. As a result, the module has only one global variable for its data. Therefore, runtime linking can be done with a few instructions. The base address of the global variables is located in the module global offset table, which is installed during module initialization. The module also has APIs that allow other modules or applications to access its internal data or services. The APIs are also put into a structure. The address of the API structure is linked to the kernel and can be accessed by other modules. Thus, updating a module does not affect other modules.

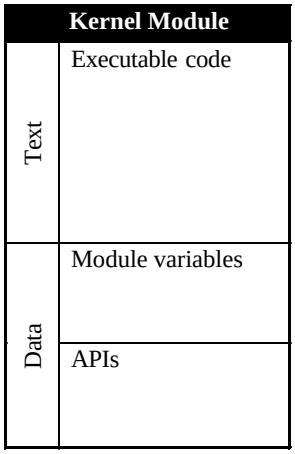

Figure 1. Module structure.

3.2. Module Updating

The application or the kernel can request a module to be updated. For module updating, the following steps occur. The new module to be loaded is read into memory. The module links itself to the kernel data and initializes its necessary data. The old module is unlinked and can be deleted from the kernel space.

When the module is loaded into the kernel space, the initializeModule() function is invoked with a pointer to the kernel data. In this function, the location of the module data is written to the global offset table, and the API

structure is initialized and installed in the kernel data section. After initializing the module, other modules and applications have access to its APIs.

Figure 2 represents a system after installing the task manager module in the RTOS. The task manager module installs its APIs in the core module and stores the pointer to the system APIs in its system data field. The task manager's APIs are available to other modules or applications.

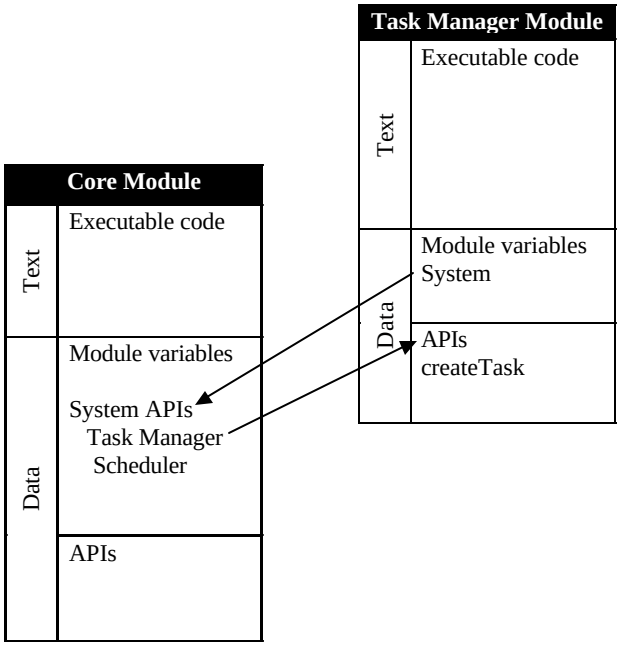

Figure 2. Installing a module to the kernel.

Example 1 [Dynamic Updating of the Scheduler] In this example, updating from one scheduler to another is illustrated. The update event may occur when a new scheduler can manage tasks more efficiently than the current scheduler. If the new module is not already in the memory, the first step is to allocate memory space and load the module to this memory location. This process can be done in the background. For this example, we assume that the system is currently using a priority-based scheduler to manage the tasks, and a request for switching to round robin scheduler is made where the round robin scheduler module has already been loaded into memory. The module loader invokes the initModule() API of the round robin scheduler to initialize variables. The round robin initModule() API is responsible for creating a ready queue and updating the scheduler entry in the system APIs as shown in Figure 3. Finally, initModule() initializes the timer with the round robin quantum and enables the timer to decrement. After the initialization, the scheduler entry in system APIs is changed to the round robin scheduler API, and the system is ready to operate using the round robin scheduler. Other modules, which use the scheduler APIs, do not have to updated. The priority-based scheduler can now be deleted from the system. ?

Example 2 [Dynamic Updating of the Loader] When the system updates the loader module, it calls the update API of the current loader module. The process for the module is similar to updating other modules. However, after the initModule() function of the new loader module is invoked, the new module replaces the old one. The initModule() function cannot return back to the old loader since the old loader is already unlinked. The return address from the initModule() function must be adjusted to the location which calls the update API of the old loader module. This is done by clearing the stack to ignore the call from the old loader to the initModule() function of the new loader. ?

Module API Identifier	Name	Module API Address
Task	Task v. 1	0x00200000
Scheduler	Priority-based	0x00201000
Synchronizer	Synch v. 1	0x00202000
Time	Time v. 1	0x00203000
…	…	…

Module API Identifier	Name	Module API Address
Task	Task v. 1	0x00200000
Scheduler	Round Robin	0x0020C000
Synchronizer	Synch v. 1	0x00202000
Time	Time v. 1	0x00203000
…	…	…

Figure 3. Scheduler module updating.

3.3. API invocations

When data or services of other kernel modules are needed, the appropriate API must be invoked. However, the module must obtain the location of the system APIs from its data section and the location of the target APIs. The system APIs are located in the data section of the core module. It contains locations of all the modules' APIs. Each module updates its location in the system APIs and stores the location of the system APIs in its data section. A kernel module can access others' APIs by referring to the system APIs. Accessing the new kernel module's data or services is done the same way as prior to the kernel module updating.

For example, if the task manager module creates a task and invokes the scheduler API to schedule a task, it implements the following steps:

```
System *pSys =
(System *) pTaskData->pSystem;
SchedulerMethod *scheduler =
(SchedulerMethod *) pSys->scheduler;
scheduler->schedule(pNewTask,
TASK_READY);
```

Figure 4 shows our implementation for invoking an API. Since the task manager has a pointer to the system APIs, it can invoke standard scheduler APIs such as schedule() in constant time providing predictability. If the scheduler module is updated, it does not affect how the task manager invokes the scheduler APIs. This provides adaptability and flexibility in the RTOS by use of pointers.

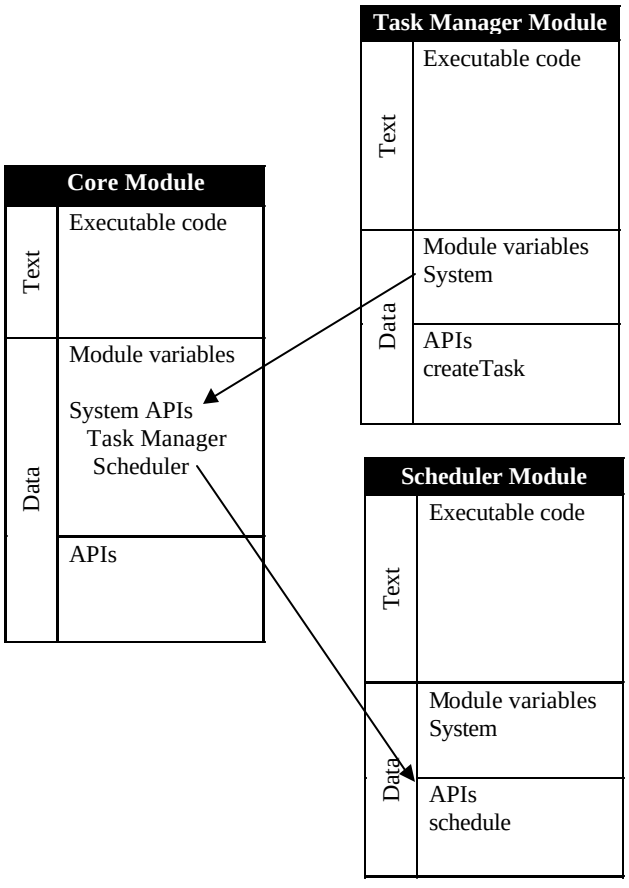

Figure 4. Invoking an API.

3.4. Kernel Structure

In this section, the DRTOS modules are described briefly. The DRTOS is written mostly in ANSI C. The kernel also includes assembly code specifically for MPC750 processors. The names of the modules, their descriptions, and sizes in terms of the number of C lines are shown in Table 1.

Table 1. DRTOS modules

Module	Description	Size (# of C lines)
Core	Initialization of the kernel	528
Task	Task creation, deletion, suspension, query	468
Time	Setting/getting system time, task delay	164
Scheduler (priority)	Fixed priority scheduler	273
Scheduler (round-robin)	Round robin scheduler	200
TOTAL		1633

4. Experiments and Results

We simulated an implementation of the DRTOS on a System-on-a-Chip (SoC) using Seamless CVE [7]. Furthermore, we implemented the DRTOS on an off-the-shelf hardware using an MBX860 board with a PowerPC 860 processor. The experimental setups and results are explained in the following two sections.

4.1. Simulation results for an SoC

It is likely that the next generation handheld devices will support multiple applications such as wireless communication and a voice user interface (VUI) running on a multiprocessor SoC. In this environment, static compilation of the applications and the RTOS may become infeasible since the applications frequently change. The adaptability, extensibility, and flexibility of the RTOS are desirable.

In this experiment, we simulated such a handheld device using Seamless CVE. Initially, the handheld device

is running a VUI application. Necessary RTOS modules are loaded to support the application. When the user switches to a wireless communication application, the RTOS must install necessary modules for orthogonal frequency division multiplexing (OFDM) transmitter and switch to the new application. The block diagram of the OFDM transmitter is illustrated in Figure 5 [6]. The application is partitioned among four processors according to its functions. The first processor handles channel initialization, train pulse generation, symbol generation, data generation and mapping, and bit-reversal for the Inverse Discrete Fourier Transform (IDFT). The next processor computes the IDFT. Another processor is responsible for normalizing the IDFT. The last processor is responsible for normalization and insertion of a guard signal. The synchronization, communication, and I/O kernel modules are loaded on each processor. Only necessary services are installed on each processor. When they are installed, the application is ready to run and the application switching is complete.

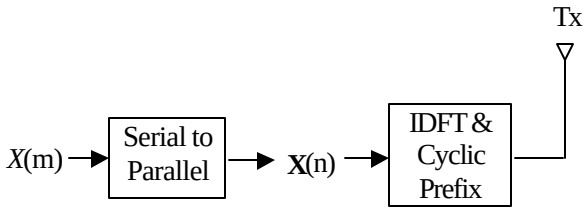

Figure 5. Block diagram of the OFDM transmitter.

The hardware setup is shown in Figure 6. The hardware includes four MPC750 processors, a memory, a bus arbiter, an interrupt controller, and a storage device. The shared bus, shared memory architecture is used as the platform for our simulation. A Kernel Management Process (KMP) runs on one of the processors. When an application switching occurs, the KMP installs necessary kernel modules for the application. After the installation, the KMP sends interrupt signals to notify each processor to execute the new application. Communication and synchronization are implemented using shared memory.

In this experiment, the DRTOS enables the device to support a wide range of applications by allowing the kernel modules to be dynamically loaded to exactly suit the applications. When a kernel module is changed, there is no need to reboot the whole system. This provides runtime extensibility of the kernel.

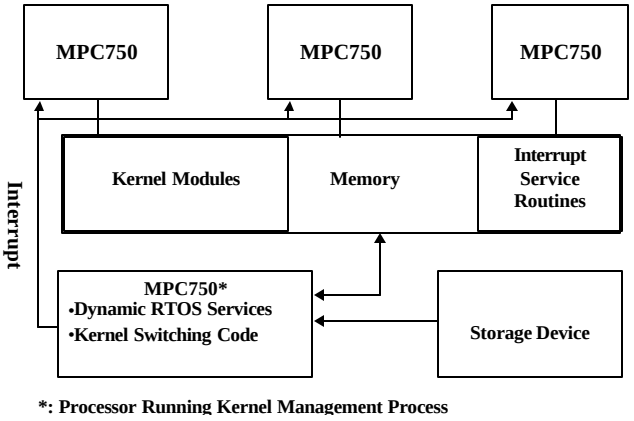

Figure 6. Experiment setup for the SoC simulation.

4.2. Implementation results for MBX860 board

We tested the DRTOS on an MBX860 evaluation board. As shown in Figure 7, the board is connected to a host machine through a JTAG interface. To validate our DRTOS concept, we set up the experiment to switch between two schedulers. The application consists of five tasks. Three tasks do some basic computation on a fixed set of input data and display the resulting output on three corresponding graphs on the host machine. The fourth task requests the kernel to update the scheduler in the DRTOS. Finally, the last task is the idle task and is scheduled when no other task is ready.

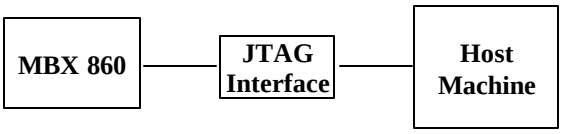

Figure 7. Experimental setup for board implementation.

The first scheduler loaded is the priority-based scheduler. The switcher routine allocates memory space for the priority-based scheduler and returns a pointer to the beginning address of the allocated space. Then the binary image of the priority-based scheduler module is loaded into this memory location. After the scheduler is initialized, the five tasks mentioned above are created and multitasking is started.

All of the tasks work in loops. All of them, besides the idle task, are suspended for a certain amount of time after finishing their job and before going into the next iteration of their loop. The three graphics-displaying tasks have higher priorities than Task 4, the scheduler-changing task.

We can see from the simulation that when the highest priority task finishes its calculation and is suspended, the next highest priority task takes control of the CPU and starts to display its graphical data. We can also observe that a higher priority task, when one becomes ready again, preempts the lower priority graphics-displaying task. At some point in time, when all of the high priority graphics-displaying tasks are suspended, Task 4 is scheduled on the CPU. Task 4, when it takes control of the CPU the first time, calls the switcher routine to allocate memory space for the round-robin scheduler, load the round-robin scheduler into this memory location, and switches from the priority scheduler to the newly downloaded and dynamically linked round-robin scheduler. Afterwards, Task 4 is suspended until it is needed to change the scheduler again.

After the round-robin scheduler is initialized and starts running, graphics-displaying tasks take turns during their computation. At this time we can observe that the three graphs are updated concurrently depending on the time slice of the round-robin scheduler.

5. Conclusion

The resource requirements of current and future embedded applications can be met by a dynamic real-time operating system where all kernel services can be updated at runtime depending on the applications. In this paper, we provide a methodology for dynamic loading and linking of RTOS kernel modules. This approach enables real-time operating systems to support a wide range of applications and improves the real-time operating systems' adaptability, extensibility, and flexibility.

We validated our DRTOS concepts by both implementing a subset of the DRTOS for the PowerPC architecture (specifically, the MPC860) and by simulating the DRTOS on an SoC simulation environment. We can update any kernel module at runtime without any significant side effects.

For future work, we will address security issues such as keeping kernel space free from malicious users and task protection.

6. Acknowledgements

This research is funded by the State of Georgia under the Yamacraw initiative [8] and by NSF under INT-9973120, CCR-9984808 and CCR-0082164.

We also would like to acknowledge software donations from Mentor Graphics and Synopsys Inc. [9] as well as hardware donations from Sun Microsystems and Intel Corporation.

7. References

[1] D. Stepner, N. Rajan, and D. Hui, "Embedded application design using a real-time OS," in *Proceedings of the Thirty-sixth Design Automation Conference*, 1999, pp. 151-156.

[2] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers, "Extensibility, safety, and perfermance in the SPIN operating system," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, 1995, pp. 267-284.

[3] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole Jr., "The operating system kernel as a secure programmable machine," *Operating Systems Review*, Jan. 1995, pp. 78-82.

[4] V. Yodaiken, "The RTLinux manifesto," in *Proceedings of the Fifth Linux Expo*, 1999.

[5] F. Colaco and F. Cardoso, "Flying Object: a modular real-time object operating system," in *Proceedings of the eleventh IEEE NPSS on Real Time*, 1999, pp. 543-546.

[6] K. F. Lee and D. B. Williams, "A space-frequency transmitter diversity technique of OFDM systems," in *Proceedings IEEE GLOBECOM*, 2000, vol. 3, pp. 1473-1477.

[7] Seamless CVE, http://www.mentor.com/seamless

[8] Yamacraw, http://www.yamacraw.org

[9] Synopsys Inc., http://www.synopsys.com