# AN IMPLEMENTATION OF THE μC/OS-II KERNEL AND AN ANALYSIS OF ITS SUITABILITY AS A REAL TIME OPERATING SYSTEM FOR EMBEDDED APPLICATIONS

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,

FACULTY OF SCIENCE

AT THE UNIVERSITY OF CAPE TOWN

IN FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

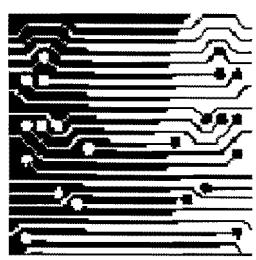
By

Gabriël de Vos

September 2003

Supervised by

Prof. Ken McGregor



# ACKNOWLEDGEMENTS

I would like to thank my wife, Suzaan, and our two daughters, Liza and Nerine, for their love and support. Without their support this dissertation would not have been possible.

# INDEX

# LIST OF TABLES

# LIST OF FIGURES

# Chapter 1 Introduction

Most desktop operating systems exercise a large degree of control over the programs and resources being used by the computer. The relationship of the operating system and the application software in a typical embedded system is much simpler in some ways and much more complex in others. There are specific requirements that grow out of the dedicated environment that simply are not present in the multipurpose world of the desktop or larger computers. Most of the operating systems that are specifically targeted at the embedded world are characterised as Real-Time Operating Systems (RTOS). A real-time system can be described as a system that is deterministic and executes its tasks in a predictable way. Desktop operating systems are not suitable for embedded applications mainly because of the huge requirement on memory (RAM and ROM).

An embedded application is not bound to a specific RTOS and can use any available RTOS or can even be implemented without using a RTOS. Many commercial and open source RTOS's are available today. Every embedded system will differ in the type of micro processor/microcontroller, size and type of memory (RAM and ROM) as well as external peripherals. Normally the RTOS has to be ported to a specific processor and memory arrangement before it can be used.



## 1.1 Goal

The goal of this thesis is to port an existing RTOS to an embedded microcontroller in order to evaluate its quality and effectiveness as a RTOS for embedded applications.

µC/OS-II, is a Real-Time Kernel developed by Jean Labrosse [1], which was chosen because of its small footprint, scalability and portable design. Because of these features, it can almost be used for every embedded application. Jean also made the source available for educational purposes. We decided to port the µC/OS-II Real Time Kernel to one of the ARM7 variants of microcontrollers. Microcontrollers that are based on the ARM7TDMI core are very powerful 32 bit microcontrollers and are available at a highly competitive price ($5 - $6). The Atmel AT91 ARM Thumb Microcontroller (AT91 ARM7TDMI), which utilises the ARM7TDMI Core, was chosen. We used the AT91EB55 evaluation board which has an AT91M55800A ARM7TDMI microcontroller, 2MB flash and 256KB SRAM.

## 1.2 Thesis Structure

Chapter 2 provides an overview of some of the concepts of Real-time Systems in general. These concepts serve as background to the rest of the thesis.

Chapter 3 describes the μC/OS-II Kernel structure in order to understand the inner workings of the kernel and its services. A good understanding of how the kernel interfaces to processor is required to port μC/OS-II. In order to do a proper evaluation of the kernel services provided by μC/OS-II, we discuss in detail how these are implemented in μC/OS-II.

Chapter 4 provides an overview of the ARM7 programmer's model. Specifications from Advanced RISC Machines (ARM) and Atmel describing the ARM7TDMI Core are summarised. The information from this chapter is very important when porting μC/OS-II to the Atmel AT91 ARM7 microcontroller.

Most of μC/OS-II's code is written in C and is portable between different processors. However, some of the code is processor-specific and needs to be ported to the specific processor. Chapter 5 discusses in detail how the processor-specific part of μC/OS-II Real Time Kernel is ported to the Atmel AT91 ARM7 microcontroller.

Dedicated Systems Experts started a project which they called the "The real-time operating system evaluation project". In this project they evaluate an RTOS to a generic set of requirements that are described in "Evaluation Report Definition" [15] and in "What makes a good RTS" [14]. An evaluation of μC/OS-II against this set of metrics is provided in Chapter 6.

Chapter 7 provides an overview of the research done during the dissertation. A final conclusion is made on the suitability of μC/OS-II as a RTOS for embedded applications.

# Chapter 2 Real-time Operating Systems

## 2.1 Overview

This chapter provides a broad overview of what a real-time system, as described by Dedicated Systems Experts [14], consists. It also describes the characteristics of a real-time operating system as summarised by D. Stepner et al [16]. Finally we take a closer look at real-time operating systems for smaller embedded applications as described by Jean Labrosse [1]. The figures used in this chapter are taken from the book, "MicroC/OS-II The Real-Time Kernel" by Jean Labrosse [1].

## 2.2 Real-time operating systems in general

### 2.2.1 What is a real-time system?

On the question of "What is a real-time system", Dedicated Systems Experts [14] quoted the following definitions:

1. "Real-time computing is computing where system correctness depends not only on the correctness of the logical result of the computation but also on the result delivery time."
2. DIN44300: "The real-time operating mode is the operating mode of a computer system in which the programs for the processing of data arriving from the outside are always ready, so that their results will be available within predetermined periods of time. The arrival times of the data may be randomly distributed or may already be determined depending on the different applications."
3. Koymans, Kuiper, Zijlstra – 1988: "A Real-Time System is an interactive system that maintains an ongoing relationship with an asynchronous environment, i.e. an environment that progresses irrespective of the real-time system, in an uncooperative manner."
4. Real-time (software) (IEEE 610.12 - 1990): "Pertaining a system or mode of operation in which computation is performed during the actual time that an external process occurs, in order that the computation results may be used to control, monitor, or respond in a timely manner to the external process."
5. Dedicated System Experts: "A real-time system responds in a (timely) predictable way to unpredictable external stimuli arrivals."

### 2.2.2 Hard and Soft Real-time

To build a predictable system, all its components (hardware & software) should enable this requirement to be fulfilled. In a real-time (RT) system, each individual deadline should be met. There are various types of real-time systems:

- Hard real-time: an activity must be completed always by a specified deadline (which may be a particular time or time interval, or at the arrival of some event), usually in tens of microseconds to few milliseconds. Some examples include the processing of a video stream, the firing of spark plugs in an automobile engine, or the processing of echoes in a Doppler radar, missing one of these deadlines will have a catastrophic result for the system.
- Soft real-time: it is not hard real-time, but some sort of timeliness is implied. That is, missing the deadline will not compromise the system's integrity, but will have a harmful effect. Examples of this type of system are point of sale (POS) systems in retail stores, ATMs and other credit card machines, and PDAs. When a POS system can not read the bar code because the item was scanned too quickly, the system simply indicates an error, and the item will be scanned again for identification. Deadlines may be missed and can be recovered from and the resulted reduction in system quality is acceptable.
- Non real-time - no deadlines have to be met.

Further confusing the notion of "hard" and "soft" real-time is increased processor speeds. When the processor speed increases, interrupts are processed more quickly. More importantly, the interrupt window in which interrupts are disabled keeps shrinking and this will improve the timeliness of response. This means that a soft real-time performance may improve just as a function of processor speed. But countering this trend is the increasing complexity of the applications, requiring more processing to be done at interrupts, and the blurring of the hardware-software interface.



### 2.2.3 RTOS Characteristics

All embedded operating systems (hard or soft real-time) have four characteristics in common that differentiate them from desktop or mainframe OS's.

**Bounded Interrupt Servicing**

There is a maximum allowable time that the system can be diverted to process an interrupt. The interrupt service routine must do the absolute minimum processing and terminate. Similarly, an RTOS must minimize the window during which interrupts are disabled.

**Priority Based Scheduling**

In a real-time system, all tasks are assigned a level of priority. This priority may be based on any number of criteria (including run time). This implies that tasks do not execute just because they are "ready," but rather because they are the highest priority task that is ready.

**Pre-Emptive Tasks**

All tasks and routines must be constructed in such a way that they can be pre-empted by some higher priority task or routine becoming ready.

**Scalability**

The OS services provided is not monolithic. Rather, they are provided as a set of modules or libraries. The services needed for an application are included in the build by simply setting flags at the time of the application build; or, in the case of libraries, by having the linker pull in the services used by the application; or by using conditional compilation to scale the OS.



Besides these four characteristics, there are other differences between real-time and desktop OS's that have more to do with the needs of the end application, the needs of the embedded developer, and the restrictions placed on the application by the resources available. The most obvious is the RAM requirement. Considering the volumes and tight end user pricing of most embedded systems, RAM is a very precious commodity. The OS must use this memory efficiently while preventing fragmentation, recovering RAM when tasks are terminated, requiring the minimum amount of RAM when tasks are created, and providing for efficient stack and heap structures.

Probably just as important are the scheduling algorithms, since these are at the heart of system performance. There are wide varieties of algorithms that have been developed and, depending on the end application, the developer would want to choose the one satisfying the response requirements while being the stingiest on resources. Some of the algorithms developed are listed in **Table 2-1** below:

| **Scheduling Algorithm** | **Description** |
|---|---|
| Heuristic | At any time, the task with the earliest deadline will be executed. This algorithm is efficient, but it may not find a feasible schedule even if one exists. |
| Cyclic | A cyclic executive is typically based on one or several major cycles that describe the order in which minor cycles are executed. One segment of the |

| | |
|---|---|
| | program is executed per minor cycle, and typically all minor cycles are the same length. The major cycle is executed repeatedly. If a segment is shorter than the minor cycle, the processor will idle. |
| Round Robin | Each task is assigned a fixed amount of processor time, and when that time is up, the next task executes. |
| Simple Priority | The user assigns a priority at the time of thread creation. The next thread to execute is based on the priority of the "ready" thread. In a large system, it can be difficult for the user to decide the priorities of each thread. |
| Rate Monotonic | The tasks of the program are assigned priorities in descending order according to the length of the period. The task with the shortest period has the highest priority, and the task with the longest period has the lowest priority. In its simplest form, it does not provide support for sporadic events. Modifications have been proposed, such as polling, priority exchange algorithm, and deferrable server algorithm. |
| Deadline Monotonic Scheduling | This is close to the Rate Monotonic but accommodates sporadic tasks. |
| Priority Inheritance Protocol | Neither of the rate monotonic scheduling algorithms assures protection from priority inversion block. The priority inheritance protocol ensures that priority inversion is controlled and the blocking time for a task is bounded. |

**Table 2-1: Scheduling algorithms**

Not as obvious, but just as important, are mechanisms that have been created to synchronize and communicate between tasks. While also found in non-RTOSs, these mechanisms take on a critical role in embedded systems due to the requirements on response and the scarcity of resources. The well known synchronization mechanisms are semaphores, mutexes, and condition variables, with message queues and mailboxes being among the more common task communication devices. But just having these mechanisms is not sufficient. These mechanisms must be designed in such a way as to take a bounded amount of time for worst case situations. For example, if a set of tasks have to wait for a semaphore in a wait queue, the wait queue should not be singly linked, since removing a task from the list will require traversing the entire list of waiting tasks. Some more efficient algorithm, with bounded worst case performance, must be used. Also the intimacy of the RTOS with the hardware imposes unique requirements. Embedded developers need flexibility and ease of use in managing IO devices along with minimal overhead from the Operating System. For example, a serial device driver may need to provide a synchronous or asynchronous driver interface and process ASCII or binary

characters in buffered or non-buffered mode. Also the RTOS must provide timer services that allow a thread to wait for a message or semaphore for a fixed timeout value or the ability to sleep for a specified time. They should also allow for multiple timers to be set by the user threads to make state transitions or notify the application of any system failure.

### 2.2.4 Popular Commercial RTOSs

A review of the popular RTOSs is beyond the scope of this dissertation, but there have been several compendiums provided to do just that. For example, the RTOS Buyer's Guide [16] characterizes more than 70 products to assist in choosing an RTOS that best suits your needs. They have been classified into three large groups:

| | |
|---|---|
| Non-commercial RTOSs: | They are mainly targeted towards small embedded systems. They are most likely free, but the support may be poor or non-existent and the list of supported devices may be short. |
| Real-time extensions to NT: | These have a different philosophy, with two OSs running on the same platform. |
| Commercial RTOSs: | These are divided into two: small embedded applications and large complex real-time applications. |

µC/OS-II is a commercial RTOS which can be applied to small or large embedded systems. It is been classified as one of the smaller RTOS's, because it does not provide a network stack (TCP/IP) or any memory protection models. µC/OS-II can be extended by adding an embedded file system (µC/FS), a graphical user interface (µC/GUI) or any other third party developed protocol stacks. µC/OS-II can also be compared to systems which are implemented without using an RTOS. These systems are normally referred to as Foreground/Background Systems.



## 2.3 Real-time operating systems for smaller embedded systems

### 2.3.1 Foreground/Background Systems

Products that do not use a RTOS make use of the foreground/background structure for their program execution. This is normally used in small systems of low complexity. The design is shown in **Figure 2-1** and a code example is shown in **Figure 2-2**. An application consists of an infinite loop that calls modules (i.e. functions) to perform the desired operations (background). Interrupt service routines (ISRs) handle asynchronous events (foreground). Foreground is also called interrupt level; background is called task level. ISRs are normally caused by timer and I/O interrupts.

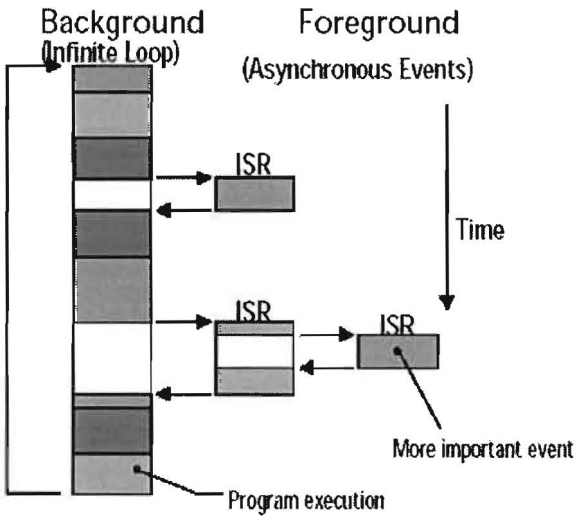



**Figure 2-1: Foreground/background systems**

```
void main (void) /* Background */
{
      Initialization;
      FOREVER
      {
            Read analog inputs;
            Read discrete inputs;
            Perform monitoring functions;
            Perform control functions;
            Update analog outputs;
            Update discrete outputs;
            Scan keyboard;
            Handle user interface;
            Update display;
            Handle communication requests;
            Other...
      }
}

ISR (void) /* Foreground */
{
      Handle asynchronous event;
}
```

**Figure 2-2: Foreground/background example**

**Advantages of Foreground/Background Systems**

Memory requirements only depends on the application and is therefore suitable for low cost Embedded Applications, where the cost of memory parts (ROM and RAM) are very significant. These small applications can easily be implemented by using a single stack area for sub-routine nesting, local variables and ISR nesting. Interrupt latency can be kept to a minimum, because there is no other overhead as required in the case of a RTOS. As no royalties require to be paid with a foreground/background implementation this contributes to a lower cost product. Most high-volume microcontroller-based applications (e.g. microwave ovens, telephones, toys, and so on) are designed

as foreground/background systems. In small microcontroller-based applications like these, it might be better (from a power consumption point of view) to halt and perform all the processing in ISRs. In this case the processor is normally in sleep mode (low power consumption) and is only wakened when an interrupt occurs. The processor is put back into sleep mode when the ISR is completed.

**Disadvantages of Foreground/Background Systems**

Critical operations must be performed by the ISR's to ensure that they are dealt with in a timely fashion. Because of this, ISRs have a tendency to take longer that they should. Also, information for a background module that an ISR makes available is not processed until the background routine gets its turn to execute, which is called task-level response. The worst case task-level response time depends on how long the background loop takes to execute. Because the execution time of typical code is not constant (affected by if, for, while, etc.), the time for successive passes through a portion of the loop is nondeterministic. Furthermore, if a code change is made, the timing of the loop is affected. All the tasks in the loop have the same priority. Services like time delays and timeouts, message passing and resource management have to be implemented within the application. Maintenance on a system like this is very difficult and adding additional features can make the code even more difficult to maintain.

### 2.3.2 Kernels

A real-time kernel is software that manages the time of a microprocessor or microcontroller. It is the part of a multitasking system responsible for management of tasks and communications between tasks. An application is broken down into multiple tasks each handling one aspect of the application as illustrated in **Figure 2-3**. The fundamental service provided by the kernel is scheduling and switching the central processing unit (CPU) between these tasks. It ensures that the most important task runs first. The use of a real-time kernel generally simplifies the design of systems by allowing the application to be divided into multiple tasks that the kernel manages.

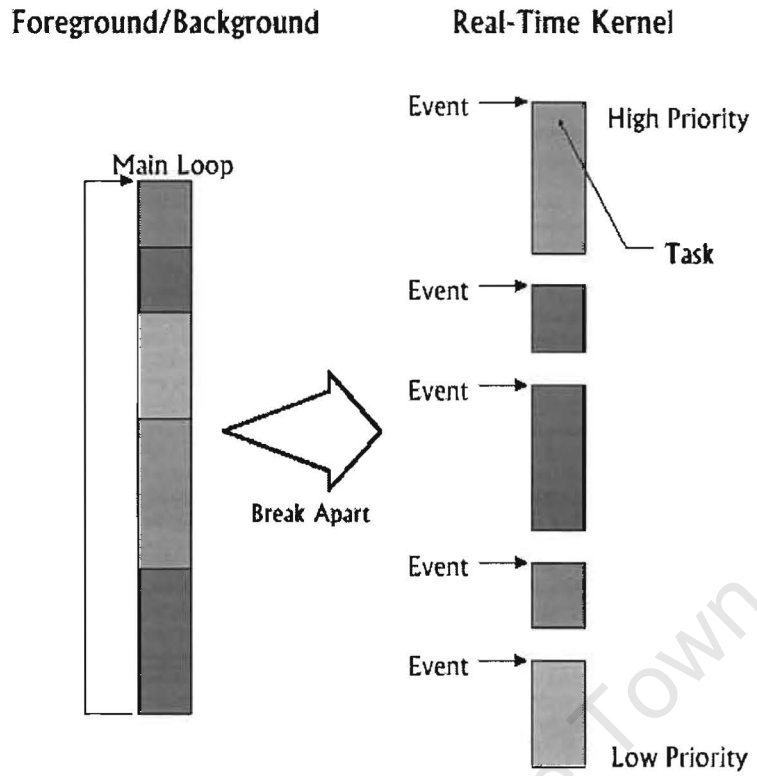



**Figure 2-3: Splitting application into tasks**

A kernel also provides valuable services like time delays, semaphore management, inter-task communication and synchronization. These services require execution time and will therefore add overhead to a system. The amount of overhead depends on how often these services are invoked. Because a kernel is software that gets added to your application, it requires ROM (code space) and additional RAM (data space) for the kernel data structures, and each task require its own stack space, which eats up RAM quickly. Single-chip microcontrollers are generally not able to run a real-time kernel because they have very little RAM. A kernel allows better use of CPU by providing indispensable services, such as semaphore management, mailboxes, queues, and time delays.

The scheduler is the part of the kernel responsible for determining which task runs next. Most real-time kernels are priority based. Each task is assigned a priority based on its importance. The priority for each task is application specific. In a priority-based kernel, control of the CPU is always given to the highest priority task ready to run. When the highest priority task gets the CPU, however, is determined by the type of kernel used. Two types of priority-based kernels exist: non-preemptive and preemptive.

**Non-Preemptive Kernels**

Non-preemptive kernels require that each task does something to explicitly give up control of the CPU. To maintain the illusion of concurrency, this process must be done frequently. Non-preemptive scheduling is also called cooperative multitasking; tasks cooperate with each other to share the CPU.

Asynchronous events are still handled by ISRs. An ISR can make a higher priority task ready to run, but the ISR always return to the interrupted task. The new higher priority task gains control of the CPU only when the current task gives up the CPU.

One of the advantages of a non-preemptive kernel is that interrupt latency is typically low. At the task level, non-preemptive kernels can also use non-reentrant functions. Non-reentrant functions can be used by each task without fear of corruption by another task. This is because each task can run to completion before it relinquishes the CPU. However, non-reentrant functions should not be allowed to give up control of the CPU.

Task-level response using a non-preemptive kernel can be much lower than with foreground/background systems because task-level response is now given by the time of the longest task.

Another advantage of non-preemptive kernels is the lesser need to guard shared data through the use of semaphores. Each task owns the CPU and you don't have to fear that a task will be pre-empted. This rule is not absolute, and, in some instances, semaphores should still be used. Shared I/O devices can still require the use of mutual exclusion semaphores; for example a task might still need exclusive access to a printer.

The execution profile of a non-preemptive kernel is shown in **Figure 2-4**.

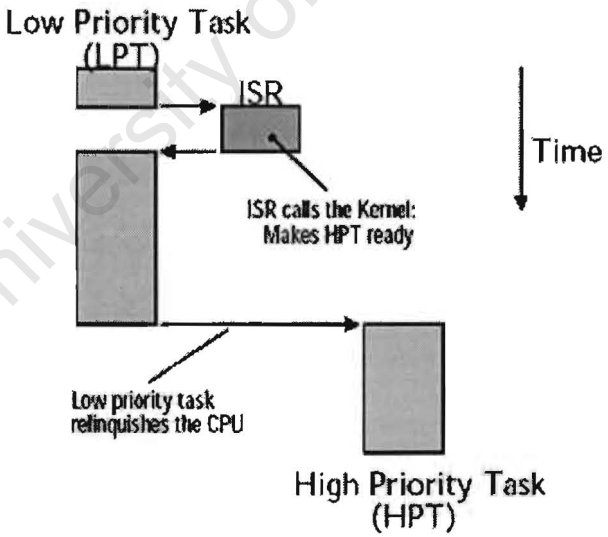



**Figure 2-4: Non-Preemptive Kernel**

The most important drawback of a non-preemptive kernel is responsiveness. A higher priority task that has been made ready to run might have to wait a long time to run because the current task must give up the CPU when it is ready to do so. As with background execution in foreground/background systems, task-level response time in a non-preemptive kernel is nondeterministic. It is not know when

the highest priority task will get control of the CPU. It is up to the application to relinquish control of the CPU.

To summarise, a non-preemptive kernel allows each task to run until it voluntarily gives up control of the CPU. An interrupt preempts a task. Upon completion of the ISR, the ISR returns to the interrupted task. Task-level response is much better than with foreground/background system, but is still nondeterministic. Very few commercial kernels are non-preemptive.

**Preemptive Kernels**

A preemptive kernel is used when system responsiveness is important; therefore most commercial kernels are preemptive. The highest priority task ready to run is always given control of the CPU. When a task makes a higher priority task ready to run, the current task is pre-empted (suspended), and the higher priority task is immediately given control of the CPU. If an ISR makes a higher priority task ready, when the ISR completes, the interrupted task is suspended and the new higher priority task is resumed.

The execution profile of a pre-empted kernel is shown in **Figure 2-5**.

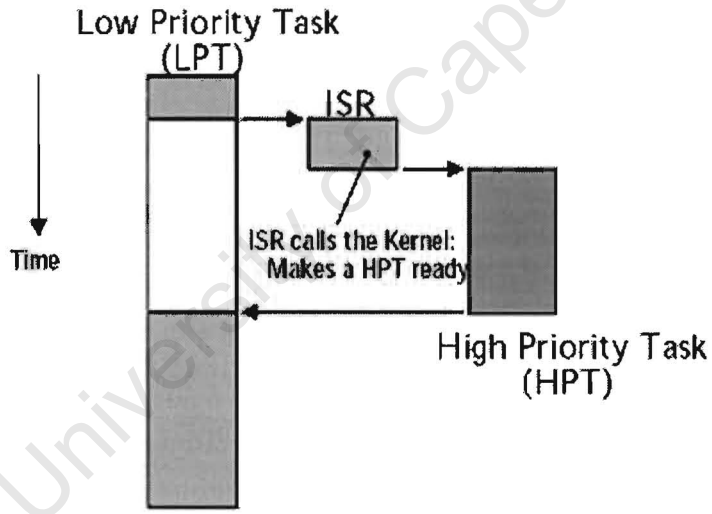



**Figure 2-5: Preemptive Kernel**

With a preemptive kernel, execution of the highest priority task is deterministic; you can determine when it will get control of the CPU. Task-level response time is thus minimised by using a preemptive kernel.

Application code using a preemptive kernel should not use non-reentrant functions unless exclusive access to these functions is ensured through the use of mutual exclusion semaphores, because both a low and a high priority task can use a common function. Corruption of data can occur if the highest priority task preempts a lower priority task that is using the function.

To summarise, a preemptive kernel always executes the highest priority task that is ready to run. An interrupt preempts a task. Upon completion of an ISR, the kernel resumes execution of the highest priority task ready to run. Task-level response is optimum and deterministic. The μC/OS-II Kernel is preemptive.

# Chapter 3 µC/OS-II Kernel Services

## 3.1 Overview

This chapter provides a description of the kernel services as implemented in µC/OS-II [1]. Each service is discussed in detail to provide the necessary information needed to evaluate it in Chapter 6. We also provide background information required to port the kernel to the AT91 ARM7TDMI microcontroller as discussed in Chapter 5.

## 3.2 Tasks

A task, also called a thread, is a simple program that thinks it has the CPU all to itself. The design process for a real-time application involves splitting the work to be done into tasks responsible for a portion of the problem. µC/OS-II requires that each task is assigned its own priority, its own set of CPU registers, and its own stack area. Each task in µC/OS-II is an infinite loop that can be in any of five states: dormant, ready, running, waiting, or ISR (interrupted). A graphical representation of these states is shown in **Figure 3-1**. The dormant state corresponds to a task that resides in memory but has not been made available to the multitasking kernel. A task is ready when it can execute but its priority is less than the currently running task. A task is running when it has control of the CPU. A task is waiting when it requires the occurrence of an event. Finally a task is in the ISR state when an interrupt has occurred and the CPU is in the process of servicing the interrupt.

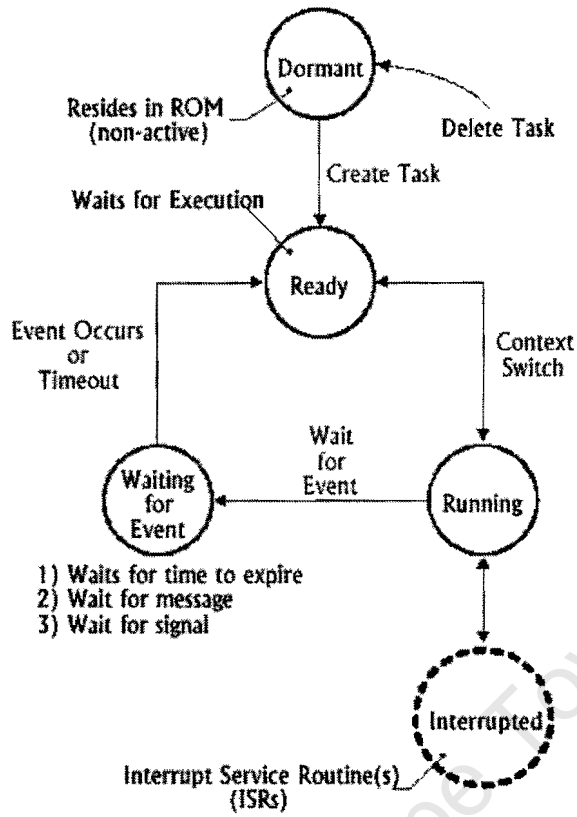



**Figure 3-1: Task States**

When a µC/OS-II decides to run a different task, it saves the current task's context (CPU registers) in the current task's context storage area – its stack. After this operation is performed, the new task's context is restored from its storage area and then the CPU resumes execution of the new task's code. This process is called context switch or a task switch. Context switching adds overhead to the application. The more registers a CPU has, the higher the overhead. The time required to perform a context switch is determined by how many registers have to be saved and restored by the CPU.

### 3.2.1 Task Control Blocks

A task is created by calling the `OSTaskCreate()` service provided by the µC/OS-II. When a task is created, it is assigned a task control block (TCB). The TCB is a data structure that is used by µC/OS-II to maintain the state of a task when it is pre-empted. When the task regains control of the CPU, the task control block allows the task to resume execution exactly where it left off. A TCB contains the task's priority, its state, a pointer to its Top_of_stack (TOS) and other task related data. All TCB's are placed in a Task Control Block Table. When µC/OS-II is initialised, all TCB's in the table are linked in a singly linked list of free TCB's. µC/OS-II can handle up to 64 tasks.

### 3.2.2 Ready List

Each task is assigned a unique priority level between 0 and 64. Each task that is ready to run is placed in a ready list. The scheduler uses the ready list to determine which priority (and thus which task) needs to run next.

### 3.2.3 Task Scheduling

µC/OS-II always executes the highest priority task ready to run. The determination of which task has the highest priority, thus which task will be next to run, is determined by the scheduler. Scheduling will takes place

- when a task decides to wait for time to expire,
- when a task sends a message or a signal to another task,
- when an Interrupt Service Routine (ISR) send a message or a signal to a task and
- at the end of all nested ISR's.

Task-level scheduling is performed by `OS_Sched()`. Interrupt Service Routine (ISR)-level scheduling is handled by another function [`OSIntExit()`].µC/OS-II task-scheduling time is constant irrespective of the number of tasks created in an application. Context switch will takes place if a more important task has been made ready-to-run or it will return to the caller or the interrupted task otherwise.

A context switch consists of saving the processor registers on the stack being suspended and restoring the registers of the higher priority task from its stack. In µC/OS-II, the stack frame for a ready task always looks as if an interrupt has just occurred and all processor registers were saved onto it. In other words, all that µC/OS-II has to do to run a ready task is restore all processor registers from the task's stack and execute a return from interrupt. To switch context, you implement `OS_TASK_SW()` so that you simulate an interrupt.

All of the code in `OS_Sched()` is considered a critical section and is therefore executed with interrupts disabled. Interrupts are disabled to prevent ISR's from setting the ready list during the process of finding the highest priority task to run.

## 3.3 Interrupts under µC/OS-II

Interrupts are always more important than tasks and therefore interrupts are always recognised with µC/OS-II. The only exceptions are when interrupts are disabled by the kernel or the application level code.

An ISR tells the kernel that an ISR is being processed by calling the `OSIntEnter()` kernel function which will increment the ISR nesting counter. While servicing the interrupt, another task can be signalled by a post message for instance. An ISR also tells the kernel when processing an ISR is done by call the `OSIntExit()` kernel function, which decrements the ISR nesting counter. When the nesting counter reaches 0, all the nested interrupts are complete, and µC/OS-II needs to determine whether a higher priority task has been awakened by the ISR (or another nested ISR). If a higher priority task is ready to run, µC/OS-II returns to the higher priority task rather than to the interrupted task. If the interrupted task is still the most important task to run, `OSIntExit()` returns to the ISR. At that point the saved registers are restored and a return from interrupt instruction is executed. The pseudocode for an ISR is shown in **Figure 3-2** and is further illustrated in **Figure 3-3**. ISR's are normally written in assembly language because it is not possible to access CPU register directly from C. ISR's are processor dependent and form part of the processor specific code.

```
ISR:
(4)   Save all CPU registers;
(5)   Call OSIntEnter();
      If first interrupt level - save current SP in current TCB
      Clear interrupting device;
      Re-enable interrupts (optional);
(6)   Execute user code to service interrupt;
(7)   Call OSIntExit();
(8)   Restore all CPU registers;
(9)   Execute a return from interrupt instruction;
```

**Figure 3-2: ISR Pseudo code**

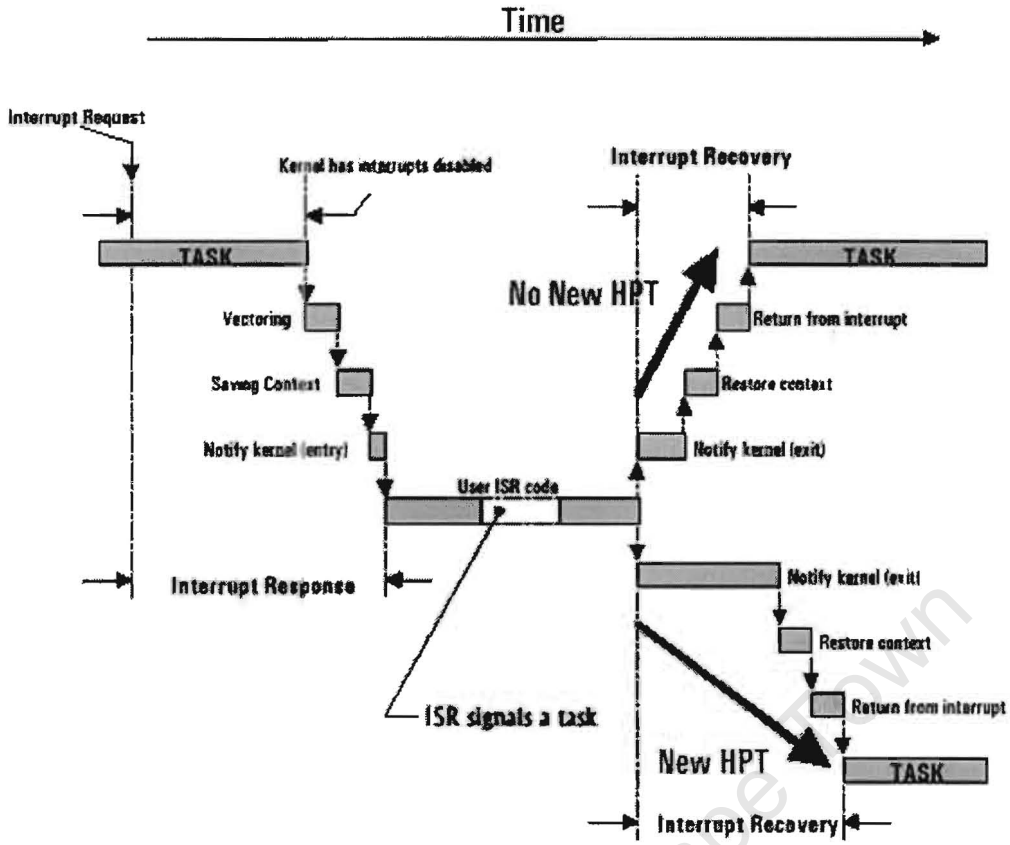



**Figure 3-3: Servicing an interrupt**

OSIntExit() takes care of switching context on ISR-level by calling `OSIntCtxSw()`. `OSIntCtxSw()` is different to `OS_TASK_SW()` in that the ISR has already saved the CPU registers onto the interrupted task and thus should not be saved again.

## 3.4 Clock Tick

µC/OS-II requires a periodic time source to keep track of time delays and timeouts. A clock timeout should occur between 10 and 100 times per second. The faster the tick rate, the more overhead µC/OS-II imposes on the system. The actual frequency of the clock tick depends on the desired tick resolution of the application. A dedicated hardware timer is normally used to generate the required tick.

The µC/OS-II clock tick is serviced by calling `OSTimeTick()` from the tick ISR. This way the kernel keeps track of all the task timers and timeouts. The tick ISR follows the normal ISR rules as described in the previous section. The pseudocode for the tick ISR is shown in **Figure 3-4**. The tick ISR is always needed by µC/OS-II and like all µC/OS-II ISR's, is written in assembly language.

```
ISR:
(4)   Save all CPU registers;
(5)   Call OSIntEnter();
      If first interrupt level - save current SP in current TCB
(6)   Call OSTickISR();
      Clear interrupting device;
      Re-enable interrupts (optional);
(7)   Call OSIntExit();
(8)   Restore all CPU registers;
(9)   Execute a return from interrupt instruction;
```

**Figure 3-4: Tick ISR Pseudo code**

### 3.5 Time Management

µC/OS-II provides a service that allows a task to delay itself for a specified number of clock ticks. Calling this service causes a context switch and forces µC/OS-II to execute the next highest priority task that is ready to run. The task that is calling this service is made ready to run as soon as the time expires or if another task cancels the delay.

### 3.6 Semaphore Management

Semaphores are used to control access to a shared resource, signal the occurrence of an event and allow two tasks to synchronise their activities. A semaphore is a key that is acquired in order to continue executing. If the semaphore is already in use, the requesting task is suspended until the semaphore is released by its current owner. Two types of semaphores are provided as service by µC/OS-II: binary semaphores and counting semaphores. µC/OS-II supplies Create, Wait and Signal operations on its semaphores.

A task acquires the semaphore by performing a Wait [`OSSemPend()`] operation. If the semaphore is available, the semaphore value is decremented and the task continues execution. If the semaphore is not available, the task is blocked and is placed in a waiting list. A timeout can be specified with the Wait [`OSSemPend()`] operation; if the semaphore is not available within the specified amount of time, the requesting task is made ready to run and the timeout is indicated to the caller.

A task releases a semaphore by performing a Signal [`OSSemPost()`] operation. If no task is waiting for the semaphore, the semaphore value is simple incremented. If any task is waiting for the semaphore however, one of the tasks is made ready to run and the key is given to one of the tasks waiting for it. µC/OS-II will always make the highest priority task ready to run. If the readied task has a higher priority that the task releasing the semaphore, a context switch occurs and the higher priority task resumes execution. **Figure 3-5** shows a graphical representation of how semaphore management is implemented within µC/OS-II.

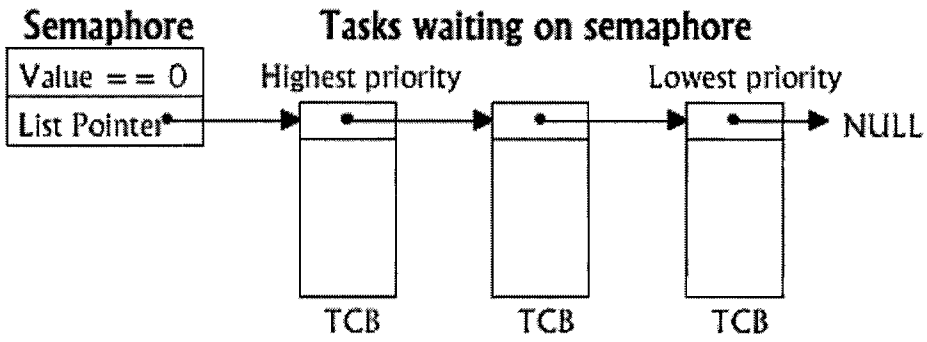



**Figure 3-5: Semaphore Management**

µC/OS-II provides three standard services and another three extended services to access semaphores. These are listed in **Table 3-1**.

| Standard Services | | Available from | Extended Services | Available from |
|---|---|---|---|---|
| Create | `OSSemCreate()` | Task | `OSSemAccept()` | Task, ISR |
| Wait | `OSSemPend()` | Task | `OSSemDel()` | Task |
| Signal | `OSSemPost()` | Task, ISR | `OSSemQuery()` | Task |

**Table 3-1: µC/OS-II semaphore services**

The extended services have to be enabled before they are available. `OSSemAccept()` gets a semaphore without waiting if a semaphore is not available. The return code of the operation indicates if the semaphore was available or not. This is very useful if an ISR needs to get access to a semaphore, but cannot block if it is not available. `OSSemQuery()` obtains the status of the semaphore.

## 3.7 Mutual Exclusion Semaphore

Mutual exclusion semaphores (mutexes) are used to gain exclusive access to resources. Mutexes are binary semaphores that have additional features beyond the normal semaphores mechanism provided by µC/OS-II. A mutex is used to reduce the priority inversion problem. A priority inversion occurs when a low priority task owns a resource needed by a high priority task. In order to reduce priority inversion, the kernel can increase the priority of the lower priority to the priority of the higher task until the lower priority task is done with the resource. In order to implement mutexes, a real-time kernel has to support multiple tasks at the same priority. µC/OS-II does not allow multiple tasks at the same priority and therefore increases the priority of the lower priority to a priority higher than the highest priority. This priority is reserved as the priority inheritance priority (PIP) when µC/OS-II is started and initialised.

A task acquires the mutex by performing a Wait [`OSMutexPend()`] operation. If the mutex is available, the mutex owner is updated and the task continues execution. If the mutex is not available, the task is blocked and is placed in a waiting list. A timeout can be specified with the Wait [`OSMutexPend()`] operation; if the mutex is not available within the specified amount of time, the requesting task is made ready to run and the timeout is indicated to the caller.

A task releases a mutex by performing a Signal [`OSMutexPost()`] operation. If no task is waiting for the mutex, the mutex's owner is kept clear. If any task is waiting for the mutex however, one of the tasks is made ready to run and the mutex is now owned by one of the tasks waiting for it. µC/OS-II will always make the highest priority task ready to run. If the readied task has a higher priority that the task releasing the mutex, a context switch occurs and the higher priority task resumes execution. **Figure 3-6** shows a graphical representation of how a mutex is implemented within µC/OS-II.

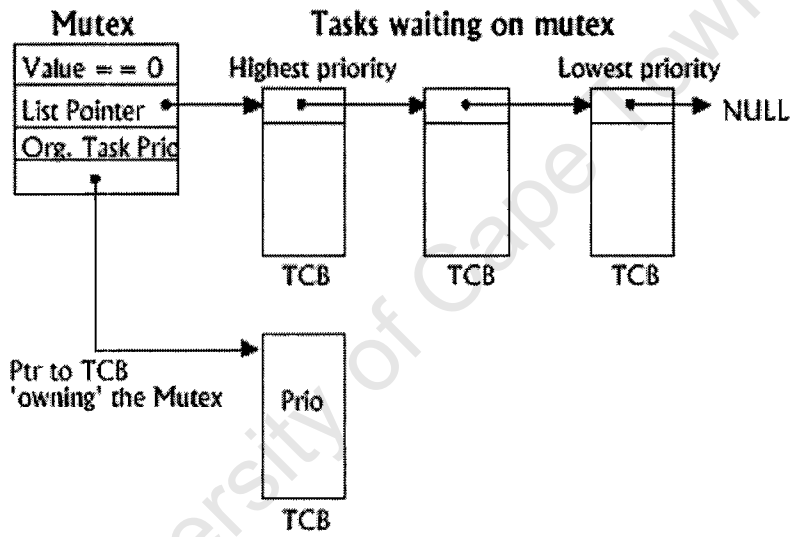



**Figure 3-6: Mutual Exclusive Semaphore Management**

µC/OS-II provides three standard services and another three extended services to access mutexes. These are listed in **Table 3-2**.

| Standard Services | | Available from | Extended Services | Available from |
|---|---|---|---|---|
| Create | `OSMutexCreate()` | Task | `OSMutexAccept()` | Task, ISR |
| Wait | `OSMutexPend()` | Task | `OSMutexDel()` | Task |
| Signa; | `OSMutexPost()` | Task, ISR | `OSMutexQuery()` | Task |

**Table 3-2: µC/OS-II mutex services**

The extended services have to be enabled before they are available. `OSMutexAccept()` gets a mutex without waiting if the mutex is not available. The return code of the operation indicates if the

mutex was available or not. This is very useful if an ISR needs to get access to a mutex, but cannot block if it is not available. `OSMutexQuery()` obtains the status of the mutex.

### 3.8 Message Mailbox Management

Messages can be sent to a task through kernel services. A message mailbox is a µC/OS-II object that allows a task or ISR to send a variable, with the size of a pointer, to another task. The pointer is typical initialised to point to some application specific data structure containing a message. µC/OS-II supplies Create, Wait and Post operations on its mailboxes.

A task or an ISR can deposit a message into a mailbox using the Post service [`OSMBoxPost()` / `OSMBoxPostOpt()`] provided by µC/OS-II. Similarly, one or more tasks can receive messages through the Wait service [`OSMBoxPend()`] provided by µC/OS-II. Both the sender and receiving tasks agree on what the pointer (message) is actually pointing to.

A waiting list is associated with each mailbox in case more than one task wants to receive messages through the mailbox. A task desiring a message from an empty mailbox is suspended and placed on the waiting list until a message is received. µC/OS-II allows the task waiting for a message to specify a timeout. If a message is not received before the timeout expires, the requesting task is made ready to run and an error code indicating that a timeout has occurred is returned to it. When a message is deposited into the mailbox, the highest priority task waiting for the message is given the message.

Message mailboxes can also simulate binary semaphores. A message in the mailbox indicates that the resource is available and an empty mailbox indicates that the resource is already in use by another task.

µC/OS-II provides three standard services and another three extended services to access mailboxes. These are listed in **Table 3-3**.

| **Standard Services** | | **Available from** | **Extended Services** | **Available from** |
|---|---|---|---|---|
| Create | `OSMBoxCreate()` | Task | `OSMBoxAccept()` | Task, ISR |
| Wait | `OSMBoxPend()` | Task | `OSMBoxDel()` | Task |
| Post | `OSMBoxPost()`<br>`OSMBoxPostOpt()` | Task, ISR | `OSMBoxQuery()` | Task |

**Table 3-3: µC/OS-II message mailbox services**

The extended services have to be enabled before they are available. `OSMBoxAccept()` obtain a message without putting a task to sleep if the mailbox is empty. The return code of the operation indicates if a message was available or not. This is very useful if an ISR needs to get access to a mailbox, but cannot block if it is not available. `OSMBoxQuery()` obtains the status of the mailbox.

### 3.9 Message Queue Management

A message queue is used to send one or more messages to a task. It is basically an array of mailboxes. A message queue is a µC/OS-II object that allows a task or ISR to send pointer-sized variables to another task. Each pointer is initialised to some application specific data structure containing a message.

A task or an ISR can deposit a message into a queue using the Post service [`OSQPost()` / `OSQPostOpt()` / `OSQPostFront()`] provided by µC/OS-II. Similarly, one or more tasks can receive messages through the Wait [`OSQPend()`] service provided by µC/OS-II. Both the sender and receiving tasks agree on what the pointer (message) is actually pointing to. The first message inserted in the queue is the first message extracted from the queue (FIFO). µC/OS-II allows a task to get messages Last-In-First-Out (LIFO) as well as depositing high priority messages in the front of the queue (out of bound).

As with a mailbox, a waiting list is associated with each message queue, in case more than one task is to receive messages through the queue. The waiting list has a fixed size and is implemented as a ring buffer. A task desiring a message from an empty queue is suspended and placed on the waiting list until a message is received. µC/OS-II allows the task waiting for a message to specify a timeout. If a message is not received before the timeout expires, the requesting task is made ready to run and an error code indicating that a timeout has occurred is returned to it. When a message is deposited into the queue, the highest priority task waiting for the message is given the message. **Figure 3-7** shows a graphical representation of how message queues are implemented within µC/OS-II.

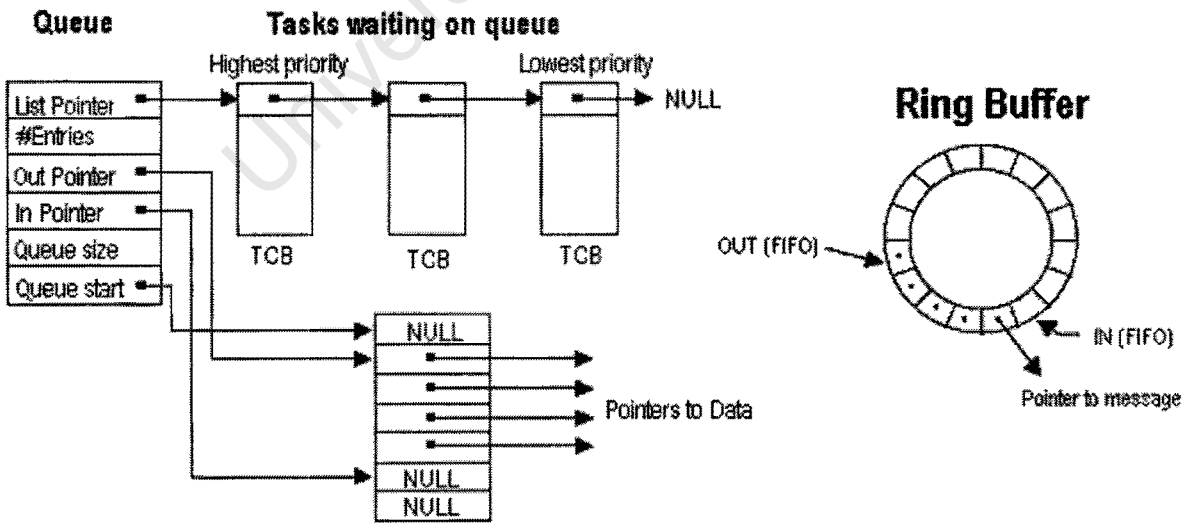



**Figure 3-7: Message Queue Management**

µC/OS-II provides nine services to access message queues. These are listed in **Table 3-4**.

| Services | | Available from | Services | Available from |
|---|---|---|---|---|
| Create | `OSQCreate()` | Task | `OSQAccept()` | Task, ISR |
| Wait | `OSQPend()` | Task | `OSQDel()` | Task |
| Post | `OSQPost()`<br>`OSQPostFront()`<br>`OSQPostOpt()` | Task, ISR | `OSQQuery()` | Task |
| | | | `OSQFlush()` | Task, ISR |

**Table 3-4: µC/OS-II message queue services**

`OSQAccept()` obtain a message without putting a task to sleep if the queue is empty. The return code of the operation indicates if a message was available or not. This is very useful if an ISR needs to get access to a queue, but cannot block if it is not available. `OSQQuery()` obtains the status of the queue.

### 3.10 Event Flag Management

Event flags are used when a task needs to synchronise with the occurrence of multiple events. The task can be synchronised when any of the events occur, disjunctive synchronisation, or when all the events have occurred, conjunctive synchronisation. This is illustrated in **Figure 3-8**. Common events can be used to signal multiple tasks. Events are grouped and each group can contain 8, 16 or 32 events. Tasks and ISRs can set or clear any event in a group. A task is resumed when all the events it requires are satisfied. The evaluation of which task will be resumed is performed after any event is set.

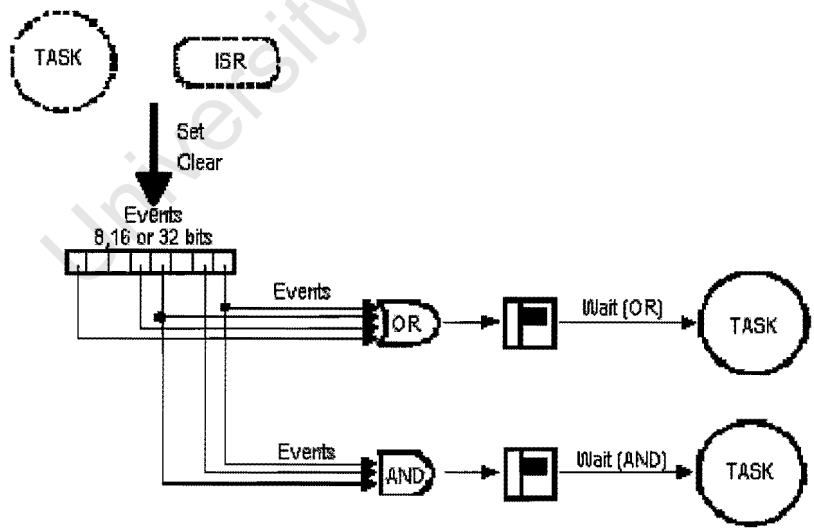



**Figure 3-8: Disjunctive and conjunctive synchronisation**

µC/OS-II offer services to Set event flags, Clear event flag and Wait for event flags (disjunctively and conjunctively).

| Standard Services | | Available from | Extended Services | Available from |
|---|---|---|---|---|
| Create | `OSFlagCreate()` | Task | `OSFlagAccept()` | Task, ISR |
| Wait | `OSFlagPend()` | Task | `OSFlagDel()` | Task |
| Set | `OSFlagPost()` | Task, ISR | `OSFlagQuery()` | Task |

**Table 3-5: µC/OS-II event flag services**

`OSFlagAccept()` obtains the status of the events without putting a task to sleep if the event group is not true. The return code of the operation indicates if a message was available or not. This is very useful if an ISR needs to get access to event flags, but cannot block if it is not available. `OSFlagQuery()` obtains the status of the event group.

## 3.11 Memory Management

An application can allocate and free dynamic memory using standard `malloc()` and `free()` functions from ANSI C. However, using these functions in an embedded real-time system is dangerous because the memory will eventually become fragmented. µC/OS-II provides an alternative to `malloc()` and `free()` by allowing an application to obtain fixed-sized memory blocks from a partition made of a contiguous memory area. All memory blocks are the same size and the partition contains an integral number of blocks. More than one memory partition can exist, so an application can obtain memory blocks of different sizes. However, a specific memory block must be returned to the partition from which it came. This type of memory management, as implemented by µC/OS-II is not subjected to fragmentation and suitable for use in embedded real-time systems. **Figure 3-9** shows a graphical representation of how fixed-block size memory partitions are implemented within µC/OS-II.

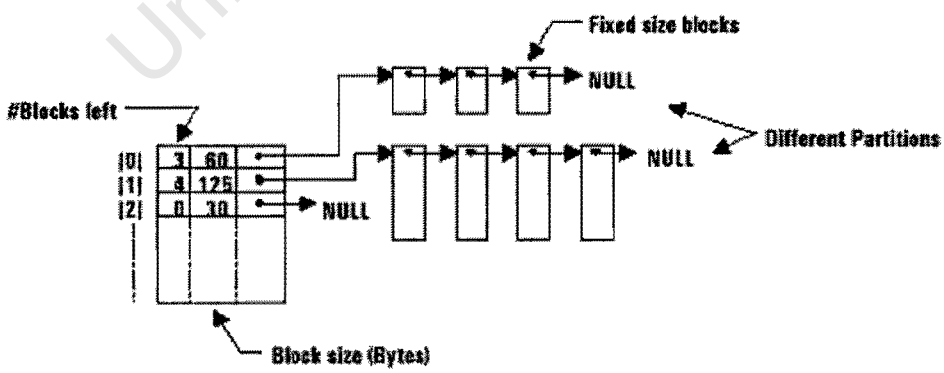



**Figure 3-9: Memory Management**

µC/OS-II provides four functions to access the memory management service. These are listed in **Table 3-6**.

| Function | Description |
|---|---|
| `OSMemCreate()` | Creating an partition |
| `OSMemGet()` | Obtaining a memory block |
| `OSMemPut()` | Returning a memory block |
| `OSMemQuery()` | Obtaining the status of a memory partition |

**Table 3-6: µC/OS-II Memory Management Service**

# Chapter 4 ARM7 Programmer's Model

## 4.1 Overview

This chapter provides a summary of the specifications of the ARM7TDMI Core as described in the various Advanced RISC Machines (ARM) published documents [4], [5] and [6]. Information as published by Atmel [8] is also covered. The information from this chapter is very important when porting µC/OS-II to the Atmel AT91 ARM7 microcontroller.

## 4.2 Introduction

The ARM7TDMI is a member of the ARM family of general purpose 32-bit microprocessors, which offer high performance for very low power consumption and price. The ARM architecture is based on Reduced Instruction Set Computer (RISC) principles. This simplicity results in a high instruction throughput and impressive real-time interrupt response from a small and cost-effective chip.

### 4.2.1 ARM7TDMI Architecture

The ARM7TDMI processor employs a unique architectural strategy known as THUMB, which makes it ideally suited to high-volume applications with memory restrictions, or applications where code density is an issue.

### 4.2.2 The THUMB Concept

The key idea behind THUMB is that of a super-reduced instruction set. Essentially, the ARM7TDMI processor has two instruction sets:

- the standard 32-bit ARM set
- a 16-bit THUMB set

The THUMB set's 16-bit instruction length allows it to approach twice the density of standard ARM code while retaining most of the ARM's performance advantage over a traditional 16-bit processor using 16-bit registers. This is possible because THUMB code operates on the same 32-bit register set as ARM code. THUMB code is able to provide up to 65% of the code size of ARM, and 160% of the performance of an equivalent ARM processor connected to a 16-bit memory system.

## 4.3 Processor Operating States

From the programmer's point of view, the ARM7TDMI can be in one of two states:

*ARM state* which executes 32-bit, word-aligned ARM instructions.

*THUMB state* which operates with 16-bit, halfword-aligned THUMB instructions. In this state, the PC uses bit 1 to select between alternate halfwords.

Transition between these two states does not affect the processor mode or the contents of the registers.

## 4.4 Operating Modes

ARM7TDMI supports seven modes of operation:

User (usr): The normal ARM program execution state

FIQ (fiq): Designed to support a data transfer or channel process

IRQ (irq): Used for general-purpose interrupt handling

Supervisor (svc): Protected mode for the operating system

Abort mode (abt): Entered after a data or instruction prefetch abort

System (sys): A privileged user mode for the operating system

Undefined (und): Entered when an undefined instruction is executed

Mode changes may be made under software control, or may be brought about by external interrupts or exception processing. Most application programs will execute in User mode. The non-user modes - known as privileged modes - are entered in order to service interrupts or exceptions, or to access protected resources.

## 4.5 Registers

ARM7TDMI has a total of 37 registers - 31 general-purpose 32-bit registers and six status registers - but these cannot all be seen at once. The processor state and operating mode dictate which registers are available to the programmer.

### 4.5.1 The ARM state register set

In ARM state, 16 general registers and one or two status registers are visible at any one time. In privileged (non-User) modes, mode-specific banked registers are switched in. **Figure 4-1** shows which registers are available in each mode: the banked registers are marked with a shaded triangle. The ARM state register set contains 16 directly accessible registers: R0 to R15. All of these except R15 are general-purpose, and may be used to hold either data or address values. In addition to these, there is a seventeenth register used to store status information

Register 14 is used as the subroutine link register. This receives a copy of R15 when a Branch and Link (BL) instruction is executed. At all other times it may be treated as a general-

purpose register. The corresponding banked registers R14_svc, R14_irq, R14_fiq, R14_abt and R14_und are similarly used to hold the return values of R15 when interrupts and exceptions arise, or when Branch and Link instructions are executed within interrupt or exception routines.

Register 15 holds the Program Counter (PC). In ARM state, bits [1:0] of R15 are zero and bits [31:2] contain the PC. In THUMB state, bit [0] is zero and bits [31:1] contain the PC.

Register 16 is the CPSR (Current Program Status Register). This contains condition code flags and the current mode bits.

FIQ mode has seven banked registers mapped to R8-14 (R8_fiq-R14_fiq). In ARM state, many FIQ handlers do not need to save any registers. User, IRQ, Supervisor, Abort and Undefined each have two banked registers mapped to R13 and R14, allowing each of these modes to have a private stack pointer and link registers.

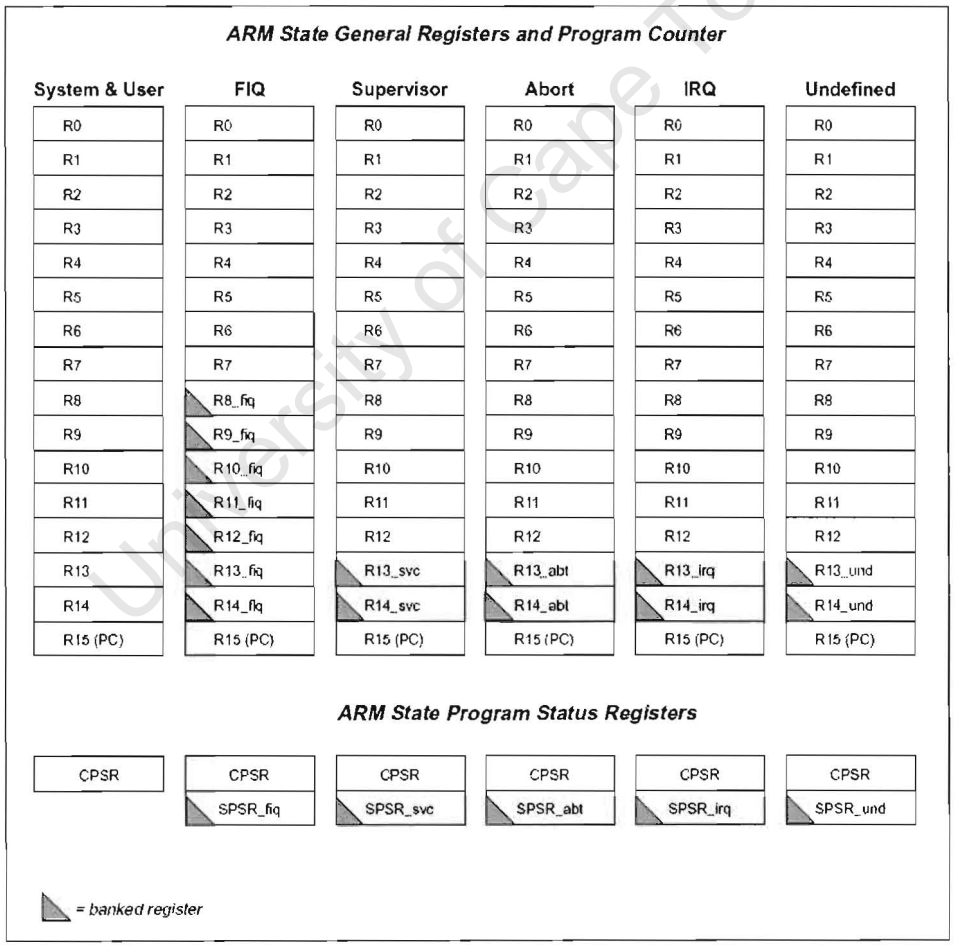



**Figure 4-1: Register organization in ARM state**

## 4.6 The Program Status Registers

The ARM7TDMI contains a Current Program Status Register (CPSR), plus five Saved Program Status Registers (SPSRs) for use by exception handlers. These registers hold information about the most recently performed ALU operation, control the enabling and disabling of interrupts and set the processor operating mode.

The arrangement of bits is shown in **Figure 4-2**.

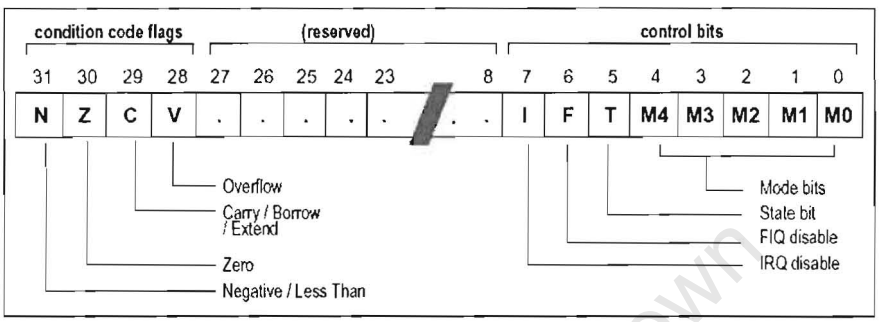



**Figure 4-2: Program status register format**

### 4.6.1 The condition code flags

The N, Z, C and V bits are the condition code flags. These may be changed as a result of arithmetic and logical operations, and may be tested to determine whether an instruction should be executed.

### 4.6.2 The control bits

The bottom 8 bits of a PSR (incorporating I, F, T and M[4:0]) are known collectively as the control bits. These will change when an exception arises. If the processor is operating in a privileged mode, they can also be manipulated by software.

The T bit — This reflects the operating state. When this bit is set, the processor is executing in THUMB state, otherwise it is executing in ARM state. This is reflected on the **TBIT** external signal. Note that the software must never change the state of the **TBIT** in the CPSR. If this happens, the processor will enter an unpredictable state.

Interrupt disable bits — The I and F bits are the interrupt disable bits. When set, these disable the IRQ and FIQ interrupts respectively.

| M[4:0] | Mode | Visible THUMB state registers | Visible ARM state registers |
|---|---|---|---|
| 10000 | User | R7..R0,<br>LR, SP<br>PC, CPSR | R14..R0,<br>PC, CPSR |
| 10001 | FIQ | R7..R0,<br>LR_fiq, SP_fiq<br>PC, CPSR, SPSR_fiq | R7..R0,<br>R14_fiq..R8_fiq,<br>PC, CPSR, SPSR_fiq |
| 10010 | IRQ | R7..R0,<br>LR_irq, SP_irq<br>PC, CPSR, SPSR_irq | R12..R0,<br>R14_irq..R13_irq,<br>PC, CPSR, SPSR_irq |
| 10011 | Supervisor | R7..R0,<br>LR_svc, SP_svc,<br>PC, CPSR, SPSR_svc | R12..R0,<br>R14_svc..R13_svc,<br>PC, CPSR, SPSR_svc |
| 10111 | Abort | R7..R0,<br>LR_abt, SP_abt,<br>PC, CPSR, SPSR_abt | R12..R0,<br>R14_abt..R13_abt,<br>PC, CPSR, SPSR_abt |
| 11011 | Undefined | R7..R0<br>LR_und, SP_und,<br>PC, CPSR, SPSR_und | R12..R0,<br>R14_und..R13_und,<br>PC, CPSR |
| 11111 | System | R7..R0,<br>LR, SP<br>PC, CPSR | R14..R0,<br>PC, CPSR |

**Table 4-1: PSR mode bit values**

The mode bits

The M4, M3, M2, M1 and M0 bits (M[4:0]) are the modebits. These determine the processor's operating mode, as shown in **Table 4-1**. Not all combinations of the mode bits define a valid processor mode. Only those explicitly described shall be used. The user should be aware that if any illegal value is programmed into the mode bits, M[4:0], then the processor will enter an unrecoverable state. If this occurs, reset should be applied.

Reserved bits

The remaining bits in the PSRs are reserved. When changing a PSR's flag or control bits, you must ensure that these unused bits are not altered. Also, your program should not rely on them containing specific values, since in future processors they may read as one or zero.



## 4.7 Exceptions

Exceptions arise whenever the normal flow of a program has to be halted temporarily, for example to service an interrupt from a peripheral. Before an exception can be handled, the current processor state must be preserved so that the original program can resume when the handler routine has finished.

### 4.7.1 Action on entering an exception

When handling an exception, the ARM7TDMI:

1. Preserves the address of the next instruction in the appropriate Link Register. If the exception has been entered from ARM state, then the address of the next instruction is copied into the Link Register (that is, current PC + 4 or PC + 8 depending on the exception). If the exception

has been entered from THUMB state, then the value written into the Link Register is the current PC offset by a value such that the program resumes from the correct place on return from the exception. This means that the exception handler need not determine which state the exception was entered from. For example, in the case of a Software Interrupt (SWI), MOVS PC, R14_svc will always return to the next instruction regardless of whether the SWI was executed in ARM or THUMB state.

2. Copies the CPSR into the appropriate SPSR
3. Forces the CPSR mode bits to a value which depends on the exception
4. Forces the PC to fetch the next instruction from the relevant exception vector

It may also set the interrupt disable flags to prevent otherwise unmanageable nestings of exceptions.

If the processor is in THUMB state when an exception occurs, it will automatically switch into ARM state when the PC is loaded with the exception vector address.

### 4.7.2 Action on leaving an exception

On completion, the exception handler:

1. Moves the Link Register, minus an offset where appropriate, to the PC. (The offset will vary depending on the type of exception.)
2. Copies the SPSR back to the CPSR
3. Clears the interrupt disable flags, if they were set on entry

An explicit switch back to THUMB state is never needed, since restoring the CPSR from the SPSR automatically sets the T bit to the value it held immediately prior to the exception.

### 4.7.3 FIQ

The FIQ (Fast Interrupt Request) exception is designed to support a data transfer or channel process, and in ARM state has sufficient private registers to remove the need for register saving (thus minimizing the overhead of context switching).

FIQ is externally generated by taking the FIQ-pin input LOW. Irrespective of whether the exception was entered from ARM or Thumb state, a FIQ handler should leave the interrupt by executing SUBS PC,R14_fiq,#4

FIQ may be disabled by setting the CPSR's F flag (but note that this is not possible from User mode). If the F flag is clear, ARM7TDMI checks for a LOW level on the output of the FIQ synchronizer at the end of each instruction.

### 4.7.4 IRQ

IRQ has a lower priority than FIQ and is masked out when a FIQ sequence is entered. It may be disabled at any time by setting the I bit in the CPSR, though this can only be done from a privileged (non-User) mode. Irrespective of whether the exception was entered from ARM or Thumb state, an IRQ handler should return from the interrupt by executing SUBS PC,R14_irq,#4

### 4.7.5 Software interrupt

The software interrupt instruction (SWI) is used for entering Supervisor mode, usually to request a particular supervisor function. A SWI handler should return by executing the following irrespective of the state (ARM or Thumb):

MOV PC, R14_svc

This restores the PC and CPSR, and returns to the instruction following the SWI.

### 4.7.6 Interrupt Management: Auto-vectoring and Prioritization

**Background**

The Atmel AT91 (microcontroller used for this thesis) is based on the ARM7TDM microcontroller core. It features the Advanced Interrupt Controller (AIC), an 8-level priority, individually maskable, vectored interrupt controller.

This microcontroller core implements two physically independent sources of interrupt:

• FIQ - Fast Interrupt

• IRQ - Normal Interrupt

Each of these interrupts has a corresponding vector, at addresses 0x00000018 for the IRQ and 0x0000001C for the FIQ. The AIC is connected to the NFIQ (Fast Interrupt Request) and the NIRQ (Standard Interrupt Request) inputs of the ARM7TDMI processor. The processor's NFIQ line can only be asserted by the external fast interrupt request input: FIQ (multiplexed with the PIO P12). Therefore, when an FIQ occurs, it is not necessary to de-multiplex the handler according to the cause of the interrupt (it is assumed that there is no multiplexing added by the external hardware). The FIQ management code can be reached either directly from the vector (0x0000001C), or by using the Fast Interrupt Vector Register (AIC_FVR) as described in the datasheet of the AT91 products.

The NIRQ line can be asserted by the interrupts generated by the on-chip peripherals and the external interrupt request lines: IRQ0 to IRQ2. Therefore it is necessary to manage a prioritization when

several interrupt sources are asserted at once and to de-multiplex the handler according to the source of the interrupt.

**Auto-Vectoring**

This feature consists of a set of registers which provide the address of the handler to execute according to the source of an interrupt.

Each interrupt source is associated with a Source Vector Register (AIC_SVR1 - AIC_SVR31) which contains the address of the function corresponding to the active interrupt. When the Interrupt Vector Register (AIC_IVR) is read, it automatically returns the contents of the source vector register corresponding to the active interrupt with the highest priority. Note that AIC_IVR is located at address 0xFFFFF100.

During the boot sequence and before enabling the interrupts, the software must:

1. Initialize the source vector registers for each interrupt
2. Initialize the IRQ vector at address 0x00000018 with the following code:

   ldr pc,[pc,#-0xF20]

When an interrupt occurs, the core performs the following (see 4.7.1 ):

R14_irq = address of next instruction to be executed + 4

SPSR_irq = CPSR

CPSR[5:0] = 0b010010 Interrupt mode

CPSR[6] = unchanged Fast interrupt status is unchanged

CPSR[7] = 1 Normal interrupts disabled

PC = 0x00000018

When the instruction at the address 0x00000018 is executed, the effective address is:

0x00000020 - 0x0F20 = 0xFFFFF100

(0x00000020 is the value of the PC when the instruction at address 0x18 is executed)

This causes the core to load the PC with the value read in AIC_IVR which returns the value of AIC_SVR corresponding to the active interrupt. This has the effect of directly jumping to the correct interrupt service routine.

When the AIC_IVR is read, the AIC does the following:

- deasserts the NIRQ line on the core

- determines which pending interrupt has the highest priority
- pushes the level of this interrupt in its internal hardware stack
- clears the interrupt if it is configured to be edge triggered

The interrupt level is popped when the End of Interrupt (EOI) is indicated to the AIC by a write in AIC_EOICR.

**Prioritization**

The NIRQ line is controlled by an 8-level priority encoder. Each source has a programmable priority level of 7 to 0. Level 7 is the highest priority and level 0 the lowest.

When the AIC receives more than one unmasked interrupt at a time, the interrupt with the highest priority is serviced first. The interrupt management of the interrupt with the lower priority level is therefore delayed.

The AIC manages the prioritization by using an internal stack on which the current interrupt level is automatically pushed when AIC_IVR is read, and popped when AIC_EOICR is written (any value). Between these two events, the software can manage the state and the mode of the core in order to re-enable the IRQ line and to allow an interrupt with a higher priority.

When an interrupt is managed by the core, R14_irq and SPSR_irq are automatically overwritten without being saved: it is mandatory to save these registers before re-enabling the IRQ line and to restore them before exiting the interrupt management routine. Moreover, if the interrupt treatment performs function calls (Branch with Link), R14_irq is used. In this case, IRQ can not be re-enabled while the core is in IRQ mode. It is mandatory to first change the mode of the core. In order to keep all exceptions available, the SYSTEM mode must be used. Therefore, the stack used during the interrupt execution is the same as that used out of the interrupt. This must be taken into account in the sizing of the SYSTEM/USER stack.

This is performed as follows:

1. Save R14_irq and SPSR_irq in the IRQ stack (current)
2. Set the mode bits in CPSR with the SYSTEM value (0b11111)
3. Re-enable IRQ by clearing bit I in CPSR
4. Execute the actions related to the interrupt
5. Disable IRQ by clearing bit I in CPSR
6. Set the mode bits in CPSR with the USER value (0b10000)

7. Restore R14_irq and SPSR_irq from the IRQ stack

This sequence is automatically preceded by a read of AIC_IVR (see above) and must be followed by a write in AIC_EOICR before exiting from the interrupt.

### 4.8 Interrupt Latencies

The worst case latency for FIQ, assuming that it is enabled, consists of the longest time the request can take to pass through the synchronizer ( $T_{syncmax}$ if asynchronous), plus the time for the longest instruction to complete ( $T_{ldm}$, the longest instruction is an LDM which loads all the registers including the PC), plus the time for the data abort entry ( $T_{exc}$), plus the time for FIQ entry ( $T_{fiq}$). At the end of this time ARM7TDMI will be executing the instruction at 0x1C. $T_{syncmax}$ is 3 processor cycles, $T_{ldm}$ is 20 cycles, $T_{exc}$ is 3 cycles, and $T_{fiq}$ is 2 cycles. The total time is therefore 28 processor cycles. This is just over 1.4 microseconds in a system which uses a continuous 20 MHz processor clock. The maximum IRQ latency calculation is similar, but must allow for the fact that FIQ has higher priority and could delay entry into the IRQ handling routine for an arbitrary length of time. The minimum latency for FIQ or IRQ consists of the shortest time the request can take through the synchronizer ( $T_{syncmin}$) plus $T_{fiq}$. This is 4 processor cycles.

### 4.9 Reset

When the RESET signal goes LOW, ARM7TDMI abandons the executing instruction and then continues to fetch instructions from incrementing word addresses.

When RESET goes HIGH again, ARM7TDMI:

1. Overwrites R14_svc and SPSR_svc by copying the current values of the PC and CPSR into them. The value of the saved PC and SPSR is not defined.
2. Forces M[4:0] to 10011 (Supervisor mode), sets the I and F bits in the CPSR, and clears the CPSR's T bit.
3. Forces the PC to fetch the next instruction from address 0x00.
4. Execution resumes in ARM state.

# Chapter 5 µC/OS-II Port to the AT91 ARM7TDMI

This chapter describes in detail what has been done to port µC/OS-II to the Atmel AT91 ARM7 microcontroller. Most of µC/OS-II's code is written in C and is portable between different processors. However, some of the code is processor-specific and needs to be ported to the specific processor. Because the architecture of µC/OS-II was designed to be highly portable, the processor-specific code is grouped in a couple of source files. µC/OS-II will require the manipulation of processor registers and therefore some of the processor-specific code will be in assembly language.

Jean Labrosse [2] suggests that the µC/OS-II kernel code should run ARM mode, with the application code running in either ARM or Thumb mode. However, all tasks will be created in ARM mode but they can call Thumb mode functions.

## 5.1 Porting requirements

**Figure 5-1** shows the µC/OS-II architecture and its relationship with the hardware. We will be concentrating on the processor-specific code in this chapter.



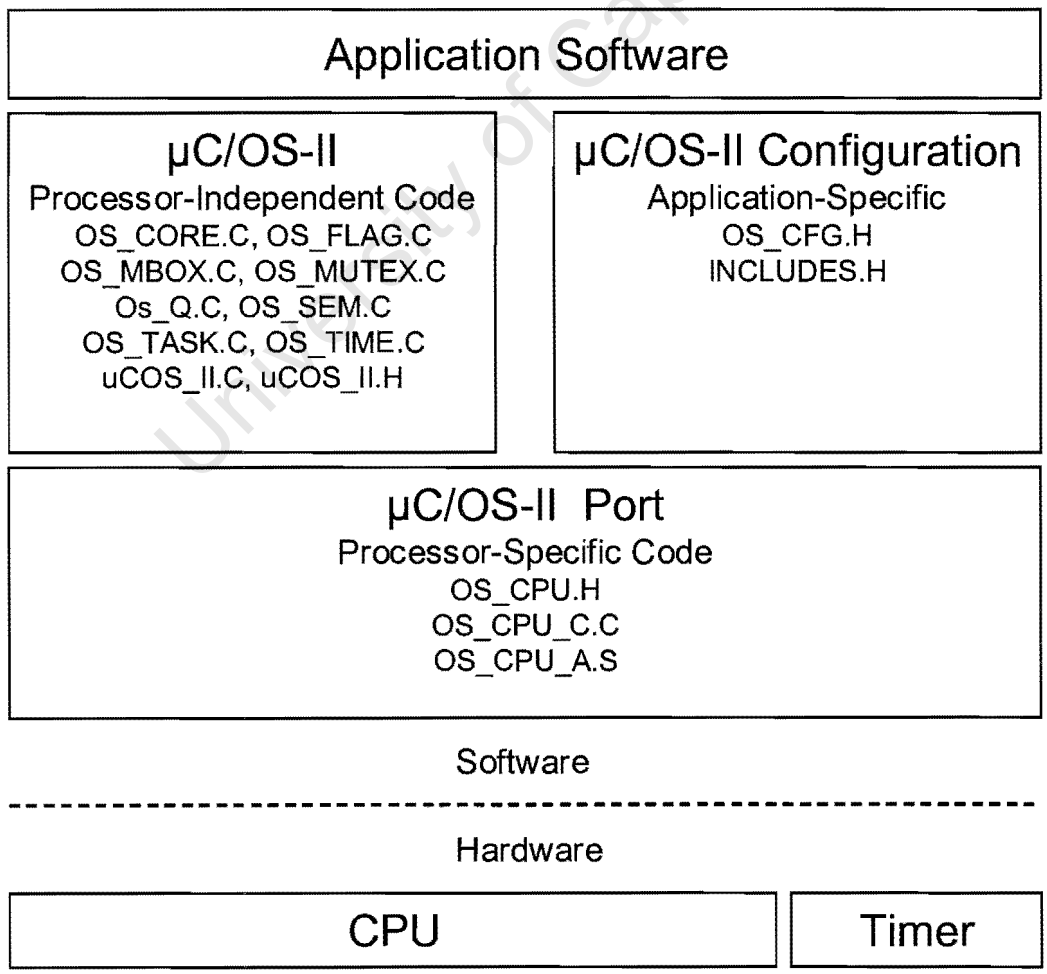



**Figure 5-1: µC/OS-II hardware/software architecture**

The source that is specific to AT91 ARM7TDMI microcontroller will be grouped in three files; OS_CPU.H, OS_CPU_C.C and OS_CPU_A.S.

## 5.2 Source File Description

**OS_CPU.H** contains processor- and implementation-specific `#define` constants, macros and typedefs.

**OS_CPU_C.C** contains processor- and implementation-specific source in C.

**OS_CPU_A.S** contains processor- and implementation-specific source in assembler.

### 5.2.1 OS_CPU.H

**Compiler-Specific Data Types**

µC/OS-II uses its own data types to ensure portability between different microprocessors and compilers. The data type definitions have been updated with the corresponding AT91 ARM7TDMI compiler definitions as describe in **Figure 5-2**. The AT91 ARM7TDMI Compiler defines a `short` as 16 bits and an `int` as 32 bits.

```
typedef unsigned char  BOOLEAN;
typedef unsigned char  INT8U;  /* Unsigned  8 bit quantity*/
typedef signed   char  INT8S;  /* Signed    8 bit quantity*/
typedef unsigned short INT16U; /* Unsigned 16 bit quantity*/
typedef signed   short INT16S; /* Signed   16 bit quantity*/
typedef unsigned int   INT32U; /* Unsigned 32 bit quantity*/
typedef signed   int   INT32S; /* Signed   32 bit quantity*/
typedef float          FP32;   /* Single precision floating point*/
typedef double         FP64;   /* Double precision floating point*/

typedef unsigned int   OS_STK; /* Each stack entry is 32-bit wide*/
typedef unsigned int   OS_CPU_SR; /* Define size of CPU status
register (PSR = 32 bits) */
```

**Figure 5-2: OS_CPU.H, Data Types**

**Critical Section**

µC/OS-II needs to disable interrupts in order to access critical sections of code and to reenable interrupts when done. This ability allows µC/OS-II to protect critical code from being entered simultaneously from either multiple tasks or ISRs. The methods used to disable and enable interrupts differ from processor to processor and from compiler to compiler. To hide the implementation method, µC/OS-II defines two macros to disable and enable interrupts: `OS_ENTER_CRITICAL()` and `OS_EXIT_CRITICAL()`.

`OS_ENTER_CRITICAL()`: A function, `OS_CPU_SaveSR()`, was written that saves the Program Status Register (PSR) of the CPU in a variable before the interrupts are disable. This ensures that the status of the interrupts (enabled or disabled) is preserved. See 4.6 The Program Status Registers.

`OS_EXIT_CRITICAL()`: A function, `OS_CPU_RestoreSR()`, was written that restores the Program Status Register (PSR) of the CPU from the variable used in `OS_ENTER_CRITICAL()`.

These functions are both in assembler language and are declared in OS_CPU_A.S. These functions will be discussed in detail in 5.2.3 OS_CPU_A.S.

**Stack Growth**

The stacks on the ARM grow from high memory to low memory and thus, `OS_STK_GROWTH` is set to 1 to indicate this to µC/OS-II.

**Task level context switches**

`OS_TASK_SW()` is a macro that is invoked when µC/OS-II switches from a low priority to the highest priority task. `OS_TASK_SW()` is always called from task-level code. Another mechanism `OSIntExit()`, is used to perform context switch when an ISR makes a higher priority task ready for execution. A context switch consists of saving the processor registers on the stack of the task being suspended and restoring the registers of the highest priority task from its stack.

Because context switching is processor specific, `OS_TASK_SW()` needs to execute an assembly language function, in this case, `OSCtxSw()` which is declared in OS_CPU_A.S. This function will be discussed in detail in 5.2.3 OS_CPU_A.S.

### 5.2.2 OS_CPU_C.C

µC/OS-II requires the implementation of the following C functions:

```
OSInitHookBegin()
OSInitHookEnd()
OSTaskCreateHook()
OSTaskDelHook()
OSTaskIdleHook()
OSTaskStatHook()
OSTaskStkInit()
OSTaskSwHook()
OSTCBInitHook()
OSTimeTickHook()
```

µC/OS-II only requires `OSTaskStkInit()` but, `OSInitHookBegin()` is also implemented to initialise a variable required for the port. The others need to be declared but do not need to contain code. These functions are hooks supplied by µC/OS-II that can be used by the application. For example `OSTaskSwHook()` will be called every time a context switch happens.

### OSTaskStkInit()

OSTaskStkInit() is called when a task is created to initialise the stack frame of a task so that the stack looks as if an interrupt has just occurred and all the processor registers have been pushed onto the stack. The code listing for OSTaskStkInit() is shown in **Figure 5-3**.

```
OS_STK *OSTaskStkInit (void (*task)(void *pd), void *pdata, OS_STK *ptos, INT16U opt)
{
    OS_STK *stk;
    opt      = opt;                    /* 'opt' is not used, prevent warning*/
    stk      = ptos                    /* Load stack pointer */
    *(stk)   = (OS_STK)task;           /* Entry Point */
    *(--stk) = (INT32U)0x14141414L;    /* LR */
    *(--stk) = (INT32U)0x12121212L;    /* R12 */
    *(--stk) = (INT32U)0x11111111L;    /* R11 */
    *(--stk) = (INT32U)0x10101010L;    /* R10 */
    *(--stk) = (INT32U)0x09090909L;    /* R9  */
    *(--stk) = (INT32U)0x08080808L;    /* R8  */
    *(--stk) = (INT32U)0x07070707L;    /* R7  */
    *(--stk) = (INT32U)0x06060606L;    /* R6  */
    *(--stk) = (INT32U)0x05050505L;    /* R5  */
    *(--stk) = (INT32U)0x04040404L;    /* R4  */
    *(--stk) = (INT32U)0x03030303L;    /* R3  */
    *(--stk) = (INT32U)0x02020202L;    /* R2  */
    *(--stk) = (INT32U)0x01010101L;    /* R1  */
    *(--stk) = (INT32U)pdata;          /* R0 : argument */
    *(--stk) = (INT32U)ARM_SVC_MODE;   /* CPSR (Enable both IRQ and FIQ) */
    *(--stk) = (INT32U)ARM_SVC_MODE;   /* SPSR */

    return (stk);
}
```

**Figure 5-3: OS_CPU_C.C, OSTaskStkInit()**

Each task assumes that they run in supervisory (SVC) mode (the CPSR and SPSR are initialized to ARM_SVC_MODE). This doesn't mean that a task must run ARM mode code. The task body can call Thumb mode code. It is typical for ARM compilers to pass the first argument of a function into the R0 register. The task receives an optional argument 'pdata'. That's why 'pdata' is passed in R0 when the task is created. Figure **5-4** shows how the stack frame is initialized for each task when it is created.

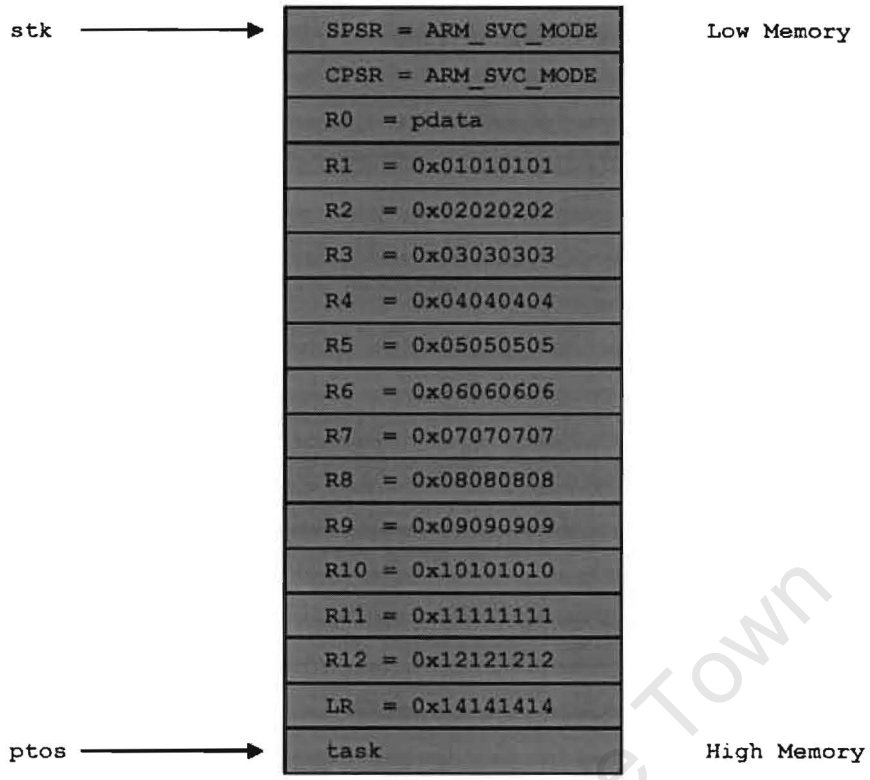



**Figure 5-4: The Stack Frame for each Task for ARM port.**

When the task is created, the final value of `stk` is placed in the `OS_TCB` of that task.

**`OSInitHookBegin()`**

```
void  OSInitHookBegin (void)
{
     OSIntCtxSwFlag = 0;
}
```

**Figure 5-5: OS_CPU_C.C, `OSInitHookBegin()`**

`OSInitHookBegin()` is called by `OSInit()` to initialize port specific variables such as `OSIntCtxSwFlag` which is used to indicate that a context switch needs to be performed at the end of all nested ISRs.

**Additional Functions**

The ARM port contains an additional three functions from the 'standard' µC/OS-II functions:

```
OSIntCtxSw()
OS_Time_Tick_Handler()
OS_CPU_FIQ_Handler()
```

**`OSIntCtxSw()`**

This function is normally written in assembly language. This function is called by `OSIntExit()` at the end of all nested ISRs and indicates that the ISR must perform a context switch. In other words, an ISR has made a more important task ready to run and thus, the CPU cannot return to the interrupted

task but instead, to a higher priority task. For this port, all this function does is set the flag `OSIntCtxSwFlag` to `TRUE` and when `OSIntExit()` returns to the ISR, the ISR examines this flag and does the context switch from the ISR. This will be described in greater detail later.

**`OS_Time_Tick_Handler()`**

This function is called by `OSTickISR` (see OS_CPU_A.S), the ISR that services the `Timer IRQ` interrupt. This function is called to handle the ISR from C instead of assembly language. `OSTickISR()` can call either ARM code or Thumb code. This function only contains a call to `OSTimeTick()` which notifies µC/OS-II that a tick interrupt occurred..

**`OS_CPU_FIQ_Handler()`**

This function is called by `OS_CPU_FIQ_ISR` (see OS_CPU_A.S), the ISR that services the `FIQ` interrupt. This function is called to handle the ISR from C instead of assembly language. `OS_CPU_FIQ_Handler()` can call either ARM code or Thumb code.

### 5.2.3 OS_CPU_A.S

A µC/OS-II port requires the implementation four assembly language functions. These functions are needed because it is not possible to save/restore the AT91 ARM7TDMI registers from C functions. The four functions are:

```
OSStartHighRdy()
OSCtxSw()
OSIntCtxSw()
OSTickISR()
```

**OSStartHighRdy()**

```
OSStartHighRdy

    BL      OSTaskSwHook      ; Call user defined task switch hook

    LDR     R4,=OSRunning     ; OSRunning = TRUE
    MOV     R5,#1
    STRB    R5,[R4]

    LDR     R4,=OSTCBHighRdy  ; Get highest priority task TCB address
    LDR     R4,[R4]           ; get stack pointer
    LDR     SP,[R4]           ; switch to the new stack

    LDMFD   SP!,{R4}            ; pop new task's spsr
    MSR     SPSR_cxsf,R4
    LDMFD   SP!,{R4}            ; pop new task's CPSR
    MSR     CPSR_cxsf,R4
    LDMFD   SP!,{R0-R12,LR,PC}  ; pop new task's r0-r12,lr & pc
```

**Figure 5-6: OS_CPU_A.S, `OSStartHighRdy()`**

`OSStartHighRdy()` is called after the kernel has started up. This starts the highest priority task that was created before the kernel was started.

Before starting the highest priority task, `OSTaskSwHook()` is called in case a hook call has been declared. There is no code in `OSTaskSwHook()` so this function would return immediately.

The µC/OS-II flag `OSRunning` is set to TRUE indicating that µC/OS-II will be running once the first task is started.

We then get the pointer to the task's top-of-stack (was stored by `OSTaskCreate()` or `OSTaskCreateExt()`). See figure **Figure 5-4** (`stk` is stored in the `OS_TCB` of the created task).

The SPSR and the CPSR of the task is popped off the stack. When the task was created, the CPSR and SPSR registers on the stack frame were initialized with `ARM_SVC_MODE`. Because `OSStartHighRdy()` already executes in SVC mode, loading the CPSR with the same value will not change the mode of the processor.

The LDMFD instruction stands for "**LoaD** **M**ultiple **F**ull **D**ecendant" and means that multiple registers are popped off the stack with a single instruction. Because the PC is the last element popped off the stack, the CPU immediately jumps to that address when it's loaded. In other words, the beginning of the task code will be run as soon as the PC is loaded.

**OSCtxSw()**

The code to perform a 'task level' context switch is shown below in **Figure 5-7**. `OSCtxSw()` is called when a higher priority task is made ready to run by another task or, when the current task can

no longer execute. It calls one the Kernel services like a time delay or it is waiting on a semaphore that is not available, etc.

```
OSCtxSw
      STMFD   SP!,{LR}              ; push pc (lr should be pushed in
                                    ; place of PC)
      STMFD   SP!,{R0-R12,LR}       ; push lr & register file
      MRS     R4,CPSR
      STMFD   SP!,{R4}              ; push current psr
      MRS     R4,SPSR
      STMFD   SP!,{R4}              ; push current spsr

      LDR     R4,=OSPrioCur         ; OSPrioCur = OSPrioHighRdy
      LDR     R5,=OSPrioHighRdy
      LDRB    R6,[r5]
      STRB    R6,[r4]

      LDR     R4,=OSTCBCur          ; Get current task's OS_TCB address
      LDR     R5,[r4]
      STR     SP,[r5]               ; store sp in preempted tasks's TCB

      BL      OSTaskSwHook          ; call Task Switch Hook

      LDR     R6,=OSTCBHighRdy      ; Get highest priority task's
                                    ; OS_TCB address
      LDR     R6,[R6]
      LDR     SP,[R6]               ; get new task's stack pointer

      STR     R6,[R4]               ; OSTCBCur = OSTCBHighRdy

      LDMFD   SP!,{R4}              ; pop new task's spsr
      MSR     SPSR_cxsf,R4
      LDMFD   SP!,{R4}              ; pop new task's psr
      MSR     CPSR_cxsf,r4
      LDMFD   SP!,{R0-R12,LR,PC}    ; pop new task's r0-r12,lr & pc
```

**Figure 5-7: OS_CPU_A.S, `OSCtxSw()`**

All the tasks run in SVC mode, therefore a task level context switch simply consists of saving the SVC registers of the task to suspend and restoring the SVC registers of the new task (see also **Figure 5-9**). The pseudo code for this is shown in **Figure 5-8**.

```
Save the CPU registers onto the old task's stack;     /* (1) */
OSPrioCur = OSPrioHighRdy;                            /* (2) */
OSTCBCur->OSTCBStkPtr = SP;                           /* (3) */
OSTaskSwHook();                                       /* (4) */
SP = OSTCBHighRdy->OSTCBStkPtr;                       /* (5) */
OSTCBCur = OSTCBHighRdy;                              /* (6) */
Restore the CPU registers from the new task's stack; /* (7) */
```

**Figure 5-8: OS_CPU_A.S, `OSCtxSw()` Pseudo code**

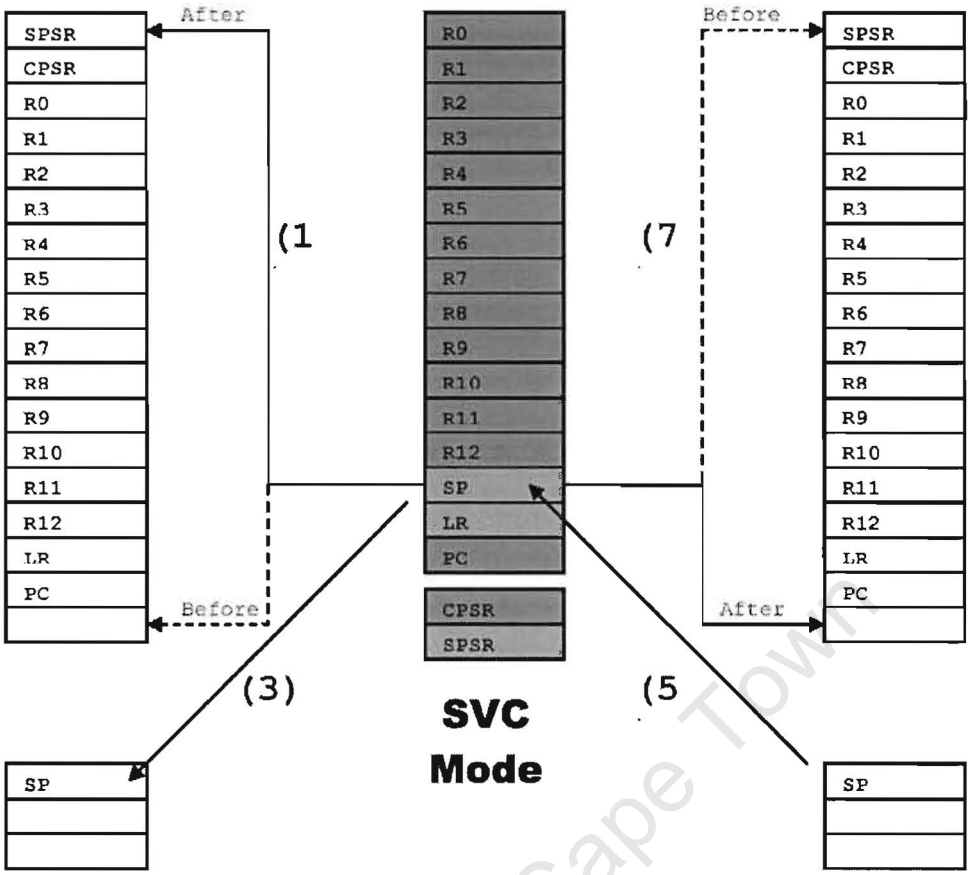



**Figure 5-9: Task Level Context Switch**



**`OS_CPU_SaveSR()` and `OS_CPU_RestoreSR()`**

The code listed in **Figure 5-10** under `OS_CPU_SaveSR()` implements the saving of the CPSR register and then disabling interrupts for `OS_CRITICAL_METHOD`. The code follows the application note published by Atmel ("Disabling Interrupts at Processor Level") [11] for properly disabling interrupts on the ARM. In this implementation, both the FIQ and IRQ interrupts are disabled. When this function returns, R0 contains the state of the CPSR register prior to disabling interrupts.

The code listed in `OS_CPU_RestoreSR()` implements the function to restore the CPSR register for `OS_CRITICAL_METHOD`. When called, it is assumed that R0 contains the desired state of the CPSR register.

```
OS_CPU_SaveSR
        MRS     R0,CPSR     ; Set IRQ and FIQ bits in CPSR to disable
                            ; all interrupts
        ORR     R1,R0,#NO_INT
        MSR     CPSR_c,R1
        MRS     R1,CPSR     ; Confirm that CPSR contains the proper
                            ; interrupt disable flags
        AND     R1,R1,#NO_INT
        CMP     R1,#NO_INT
        BNE     OS_CPU_SaveSR  ; Not properly disabled (try again)
        MOV     PC,LR       ; Disabled, return the original CPSR
                            ; contents in R0

OS_CPU_RestoreSR
        MSR     CPSR_c,R0
        MOV     PC,LR
```

**Figure 5-10: OS_CPU_A.S, `OS_CPU_SaveSR()` and `OS_CPU_RestoreSR()`**

**IRQ Handlers**

Normal ARM7TDMI implementations have only one Interrupt vector for all the devices that can interrupt the processor. But the AT91 ARM&TDMI implementation contains an Advanced Interrupt Controller (AIC) that supplies a vector for each interrupt source. This allows the proper interrupt handler to be called directly instead of having to figure out (through code) what the actual source of the interrupt was. See 4.7.6 Interrupt Management: Auto-vectoring and Prioritization.

**`OSTickISR()` and `OS_IntCtxSw()`**

µC/OS-II needs to be notified when an IRQ is serviced through an ISR. This is done by calling kernel function `OSIntEnter()`. The same is required on conclusion of an ISR. This is done by calling the kernel function `OSIntExit()`. During this call the kernel determine whether a higher priority task has been awakened by the ISR. If a higher priority task is ready to run, the kernel returns to the higher priority task rather than to the interrupted task. This requires a task context switch after an ISR and is done by the `OS_IntCtxSw()` function. As this is a processor specific function, it had to be implemented for the AT91 ARM7TDMI port. The code for `OS_IntCtxSw()` is listed in **Figure 5-12**.

µC/OS-II needs a periodic time source to keep track of time delays and timeouts. A tick should occur between 10 and 100 times per second. This was implemented by configuring one of the Timer Counter Blocks to provide an interrupt every 10ms. The ISR for the timer, `OSTickISR()`, is listed in **Figure 5-11**.

```
OSTickISR

(1)     STMFD   SP!,{R0-R3,R12,LR}

(2)     LDR     R14, =AIC_BASE          ; Write in the IVR to support
                                        ; Protect Mode, no effect in
                                        ; Normal Mode
        STR     R14, [R14, #AIC_IVR]    ; De-assert the NIRQ and clear
                                        ; the source in Protect Mode

(3)     BL      OSIntEnter              ; Indicate beginning of ISR
        BL      OS_Time_Tick_Handler    ; Handle interrupt
        BL      OSIntExit               ; Indicate end of ISR

        LDR     R0,=OSIntCtxSwFlag      ; See if we need to do a
                                        ; context switch
        LDR     R1,[R0]
        CMP     R1,#1
        BEQ     OS_IntCtxSw             ; Yes, Switch to Higher
                                        ; Priority Task

(4)     LDR     R0, =AIC_BASE           ; Mark the End of Interrupt on
                                        ; the AIC
        STR     R0, [R0, #AIC_EOICR]

(5)     LDMFD   SP!,{R0-R3,R12,LR}      ; No, Restore registers of
                                        ; interrupted task's stack
(6)     SUBS    PC,LR,#4                ; Return from IRQ
```

**Figure 5-11: OS_CPU_A.S, `OSTickISR()`**

It is assumed that a task is running (in SVC mode) when an interrupt occurs. The processor's SP (R13) points to a location into the current task's stack.

The processor recognizes the IRQ and branches to `OSTickISR()`. The descriptions below refer to the code listed in **Figure 5-11**.

(1) The ARM Procedure Call Standard (APCS) registers are saved onto the ISR stack.

(2) Write in the Interrupt Vector Register (IVR) to support Protect Mode, no effect in Normal Mode. De-assert the NIRQ and clear the source in Protect Mode.

(3) `OSIntEnter()` increments `OSIntNesting`. `OS_Time_Tick_Handler` will service the IRQ interrupt. The handler needs to determine the source of the interrupt and take appropriate actions. When the handler is done, `OSIntExit()` is called to see if a context switch needs to occur. `OSIntExit()` only sets a global flag called `OSIntCtxSwFlag` to `TRUE` if a context switch is needed, otherwise, the flag is untouched and thus `FALSE`. If `OSIntCtxSwFlag` is `FALSE` then `OSIntExit()` did not find a higher priority task to run and thus, we will be returning to the interrupted task.

(4) The APCS registers are popped from the IRQ stack.

(5) Because the SUBS instruction contains the PC as the destination register, the CPU restores the CPSR from the SPSR_irq register as well as loads the PC with the contents of the LR minus 4 (i.e. pointing to the instruction immediately after the one interrupted.

(6) Mark the End of Interrupt on the AIC.

```
OS_IntCtxSw
(1)     LDR     R0,=OSIntCtxSwFlag      ; OSIntCtxSwFlag = FALSE
        MOV     R1,#0
        STR     R1,[R0]

(2)     LDR     R0, =AIC_BASE           ; Mark the End of Interrupt on
                                        ; the AIC
        STR     R0, [R0, #AIC_EOICR]

(3)     LDMFD   SP!,{R0-R3,R12,LR}      ; Clean up IRQ stack
        STMFD   SP!,{R0-R3}             ; We will use R0-R3 as
temporary registers
(4)     MOV     R1,SP
(5)     ADD     SP,SP,#16
(6)     SUB     R2,LR,#4

(7)     MRS     R3,SPSR                 ; Disable interrupts for when
we go back to SVC mode
(8)     ORR     R0,R3,#NO_INT
        MSR     SPSR_c,R0

(9)     LDR     R0,=.+8                 ; Switch back to SVC mode
                                        ; (Code below, current
                                        ; location + 2 instructions)
(10)    MOVS    PC,R0                   ; Restore PC and CPSR

                                        ; SAVE OLD TASK'S CONTEXT ONTO
                                        ; OLD TASK'S STACK
(11)    STMFD   SP!,{R2}                ; Push task's PC
(12)    STMFD   SP!,{R4-R12,LR}         ; Push task's LR,R12-R4
(13)    MOV     R4,R1                   ; Move R0-R3 from IRQ stack to
                                        ; SVC stack
        MOV     R5,R3
(14)    LDMFD   R4!,{R0-R3}             ; Load R0-R3 from IRQ stack
        STMFD   SP!,{R0-R3}             ; Push R0-R3
(15)    STMFD   SP!,{R5}                ; Push task's CPSR
        MRS     R4,SPSR
        STMFD   SP!,{R4}                ; Push task's SPSR

(16)    LDR     R4,=OSPrioCur           ; OSPrioCur = OSPrioHighRdy
        LDR     R5,=OSPrioHighRdy
        LDRB    R5,[R5]
        STRB    R5,[R4]

        LDR     R4,=OSTCBCur            ; Get current task's OS_TCB
                                        ; address
        LDR     R5,[R4]
        STR     SP,[R5]                 ; store sp in preempted
                                        ; tasks's TCB
```

```
BL      OSTaskSwHook            ; call Task Switch Hook

LDR     R6,=OSTCBHighRdy        ; Get highest priority task's
                                ; OS_TCB address
LDR     R6,[R6]
LDR     SP,[R6]                 ; get new task's stack pointer

STR     R6,[R4]                 ; OSTCBCur = OSTCBHighRdy

LDMFD   SP!,{R4}                ; pop new task's spsr
MSR     SPSR_cxsf,R4
LDMFD   SP!,{R4}                ; pop new task's psr
MSR     CPSR_cxsf,R4

LDMFD   SP!,{R0-R12,LR,PC}      ; pop new task's r0-r12,lr &
                                ; pc
```

**Figure 5-12: OS_CPU_A.S, `OS_IntCtxSw()`**

The descriptions below refer to the code listed in **Figure 5-12**.

(1) `OS_IntCtxSw()` starts by setting the flag `OSIntCtxSwFlag` to FALSE.

(2) Mark the End of Interrupt on the AIC.

(3) The registers that were saved on the IRQ stack at the beginning of the ISR are now restored. However, we save R0-R3 back because they will be used as scratch registers.

(4) The IRQ SP is copied to the R1 register to allow this register to access the IRQ stack when it switches back to SVC mode.

(5) The IRQ SP is adjusted to point at the start of the stack. This is done to empty the IRQ stack since it will switch back to SVC mode and need to clean up the stack. There is no danger in writing over R0-R3 left on the stack frame since further IRQs are disabled.

(6) The return address of the interrupted task is computed from the LR and saved in R2.

(7) The value of the interrupted task's CPSR is copied to R3. It contains the value of the CPSR of the SVC mode prior to servicing the interrupt.

(8) ALL interrupts are disabled for when the return to SVC mode. This is done by altering the SPSR register since the CPSR register is specific to the IRQ mode and interrupts are disabled.

(9) This instruction is 'strange' in that it loads the R0 register with the current contents of the program counter (i.e. THIS instruction plus 8). R0 then contains the address not of the next instruction but, the one after it (i.e. the STMFD instruction described later). The reason this is done will become clear with the next instruction.

(10) This move instruction changes the PC to point to the NEXT instruction AND changes the CPU back to SVC mode. This means that the code is still executing in `OS_IntCtxSw()` but at the instruction following this one but, the mode has changed and thus R13 now points to the interrupted task's stack.

(11) This instruction saves the PC of the task being 'switched out' onto that task's stack.

(12) This instruction saves registers R4-R12 and the LR onto the task's stack.

(13) These instructions are used to move the SP and the SPSR to make room in R0-R3 to save those registers that were saved onto the IRQ stack.

(14) These instructions are used to move registers R0-R3 from the IRQ stack to the task's stack.

(15) The first instruction saves the CPSR of the interrupted task (i.e. it was saved in the SPSR of the IRQ mode) onto the stack. The next two instructions save the SPSR of the task. At this point, the context of the interrupted task is saved onto that task's stack.

(16) Proceed with the context switch which will be performed in SVC mode. The remaining steps are exactly the same as for a task level context switch (in pseudo-code and C):

```
OSTCBCur->OSTCBStkPtr = SP;
OSPrioCur             = OSPrioHighRdy;
OSTaskSwHook();
SP                    = OSTCBHighRdy->OSTCBStkPtr;
OSTCBCur              = OSTCBHighRdy;
Restore the CPU registers from the new task's stack;
```

**Figure 5-13: Task level context switch in pseudo code**

## 5.3 Verifying the port

The port was first tested without any application code to make sure that the basics of the port works. We then add a few simple tasks and the ticker interrupt service routine. If we can prove that the port can do multitasking then we could consider that the port is working because the higher levels of code are processor-independent and can be considered as fully functional.

Verification was done in three steps:

1. Verify `OSTaskStkInit()` and `OSStartHighRd()`
2. Verify `OSCtxSw()`
3. Verify `OSIntCtxSw()` and `OSTickISR()`

### 5.3.1 Verify `OSTaskStkInit()` and `OSStartHighRd()`

The first step is to verify the proper operation of `OSTaskStkInit()` and `OSStartHighRd()`. A simple `main()` function as shown in **Figure 5-14** was used. No application tasks were created. Only the µC/OS-II idle task [OS_TaskIdle()] will be created by the kernel. The code was debugged using the ARM Software Toolkit emulator as a source level debugger. We stepped through `OSStart()` until we got to `OSStartHighRd()` which must start the first task ready to run. As we did not create any other task, than `OS_TaskIdle()`, `OSStartHighRd()` should start this task. We verified that `OSStartHighRd()` populates the CPU registers in the reverse order that they were placed onto the task stack by `OSTaskStkInit()`. We also verified that `OSStartHighRd()` starts the `OS_TaskIdle()` task successfully. We have done this a couple of times and we considered it verified.

```
void main(void)
{
      OSInit();
      OSStart();
{
```

**Figure 5-14: Minimal `main()` for testing `OSTaskStkInit()` and `OSStartHighRd()`**

### 5.3.2 Verify `OSCtxSw()`

In the previous step we verified that that the stack frame of a task is correctly initialised by `OSTaskStkInit()`. For this test we created an application task and force a context switch back to the idle task. The code is shown in **Figure 5-15**.

```
      OS_STK TestTaskStk[100];

      void main(void)
      {
            OSInit();
(1)         OSTaskCreate(TestTask, (void*)0, &TestTaskStk[99], 0);
            OSStart();
      {

(2)   void TestTask(void *pdata)
      {
            pdata = pdata;
            while(1)
            {
(3)               OSTimeDly(1);
            }
      }
```

**Figure 5-15: Code for testing `OSCtxSw()`**

We created a high priority task, `TestTask()`, with the code at `((1)` in **Figure 5-15**). µC/OS-II should start `TestTask()` `((2)` in **Figure 5-15)** as its first task instead of executing the idle task. We verified that this happens. `TestTask()` enters an infinite loop that continuously calls

`OSTimeDly(1)` (`(3)` in **Figure 5-15**). Because we did not enable interrupts nor did we start the clock tick, `OSTimeDly(1)` should never return to `TestTask()`. We verified that `OSTimeDly()` calls `OSSched()` and in turn calls `OSCtxSw()`. We verified that `OSCtxSw()` saved the registers of `TestTask()` and loaded the registers of `OS_TaskIdle()` into the CPU. We also verified that it switched to the idle task's code. This proved that `OSCtxSw()` is working correctly.

### 5.3.3 Verify `OSIntCtxSw()` and `OSTickISR()`

`OSIntCtxSw()` is similar to `OSCtxSw()` but only called from ISR level. For this test we initialised the clock tick and linked its interrupt vector to the clock tick ISR. We also enabled interrupts. The pseudo code is shown in **Figure 5-16** with an explanation below.

```
      OS_STK TestTaskStk[100];

      void main(void)
      {
            OSInit();
(1)         Turn LED OFF;
(2)         Install the clock tick interrupt vector;
(3)         OSTaskCreate(TestTask, (void*)0, &TestTaskStk[99], 0);
            OSStart();
      {

(4)   void TestTask(void *pdata)
      {
            BOOLEAN led_sate

            pdata = pdata;
(5)         Initialise the clock tick interrupt (start timer);
(6)         Enable Interrupts;
            led_state = FALSE;
(7)         Turn LED ON;
            while(1)
            {
(8)             OSTimeDly(1);
(9)             if(led_state == FALSE)
                {
                      led_state = TRUE;
                      Turn LED ON;
                } else {
                      led_state = FALSE;
                      Turn LED OFF;
                }

            }
      }
```

**Figure 5-16**: **Code for testing** `OSIntCtxSw()` **and** `OSTickISR()`

We used an LED for this step and we made sure (`(1)` in **Figure 5-16**) that it is switched off. We install the clock tick interrupt vector (`(2)` in **Figure 5-16**) and created a high priority task (`(3)` in **Figure 5-16**). As verified before `TestTask()` is started first by the kernel. The clock tick is

initialised (`(5)` in **Figure 5-16**) and interrupts enabled (`(6)` in **Figure 5-16**) within `TestTask()`. It is a µC/OS-II requirement to only start the timer within the first task. We turned the Led on (`(7)` in **Figure 5-16**) to indicate that we entered `TestTask()`. The call to `OSTimeDly()(8)` in **Figure 5-16**)should cause a context switch to the idle task using OSCtxSw(). The idle task spins until the tick interrupt is received. The tick interrupt should invoke `OSTickISR()` which in turn calls `OSTimeTick()`. `OSTimeTick()` will decrements `TestTask`'s timeout variable and should make it ready to run when the timeout variable is `0`. When `OSTickISR()` completes and calls `OSIntExit()`, `OSIntExit()` should notice that the more important task, `TestTask()`, is ready to run. The ISR, therefore, does not return to the idle task, but instead performs a context switch back to `TestTask()`, which will toggle the LED (`(9)` in **Figure 5-16**). All this assumes that `OSIntCtxSw()` and `OSTickISR()` are both working correctly. Our LED was flickering and thus verified that `OSIntCtxSw()` and `OSTickISR()` are both working correctly.

## 5.4 Conclusion

We tested and verified all the processor-specific functions by going through the steps as described above. Our port is now ready to be used for embedded applications.

# Chapter 6 Evaluating μC/OS-II as an RTOS

## 6.1 Overview

This chapter evaluates the suitability of μC/OS-II as an RTOS. Dedicated Systems Experts started a project which they called the "The real-time operating system evaluation project". In this project they evaluate an RTOS to a set of requirements that are described in "Evaluation Report Definition" [15] and in "What makes a good RTS" [14]. They divided their evaluation into two approaches. The first approach is a qualitative study called the technical evaluation and the focus is on the system architecture of the operating system. The second part referred to as the practical evaluation is a quantitative approach. In this chapter we will do a full technical evaluation on μC/OS-II. Due to the hardware constraints of the ARM7 evaluation board used, we could not do the practical evaluation as described above.

## 6.2 Technical Evaluation

During this evaluation we studied the architecture of the operating system. The architecture determines, among other things, the task management, the interrupt handling and the memory management.

For an operating system to be classified as an RTOS, it needs to support a standard set of services. These services can be accessed via an applications programming interface (API). An inventory of the API functions gives an appreciation of the richness of the operating system. Dedicated Systems Experts [15] compiled a set of API functions that is needed in an RTOS and we will apply it to the API in μC/OS-II.

### 6.2.1 Installation and configuration

**Installation**

μC/OS-II is supplied in ANSI-C source code and does not come with a set of development tools. μC/OS-II was designed to be highly portable between different processors and development environments. As described in Chapter 5, μC/OS-II consists of processor-dependent and processor-specific code. In order to use μC/OS-II as an RTOS on a specific processor, one needs to port the processor-specific code to the new processor. μC/OS-II has been ported to many processors and Jean Labrosse (author of μC/OS-II) makes some of these available for free. If you are lucky a port exists already and you do not have to go through the motions of porting it yourself. In Chapter 5 we went through the motion of porting it to the AT91 ARM7 microcontroller. As we experienced, porting is

not a trivial exercise and can be a huge stumbling block towards using µC/OS-II as an RTOS. The quality of the port also affects the quality and effectiveness of µC/OS-II. The low level context switches (task and interrupt level) is the heart of any RTOS and in µC/OS-II it forms part of the processor-specific code that needs to be ported.

**Configuration**

µC/OS-II is provided in source form and therefore all the RTOS configurations are done through `#define` constants in one single header file. Changing any configuration parameter requires recompilation of the application and the operating system. Dynamic configuration of the operating system once it is running is not possible, but system objects can be created and deleted during the execution of applications. The resources for these objects are already available as the maximum number of objects is set in the configuration file.

### 6.2.2 RTOS Architecture

µC/OS-II is a small kernel for embedded applications. It is a variant of client-server architecture but does not have a message based communication protocol. It uses a software bus to communicate between the different modules. There is a choice of modules that can be built into the system and these are chosen at compile time.

A graphical user interface (GUI) named µC/GUI and an embedded file system named µC/FS, are optional extras. We did not include these in our evaluations as it was not available free of charge for evaluation purposes. There is a choice of which system components to include and that makes µC/OS-II highly scalable. uC/OS-View, an optional extra, is available to help with debugging and development. It provides a 'view' of the operating system objects while the RTOS is running.

µC/OS-II has a flat memory space. All tasks share the same memory space and also share all objects such as semaphores. This means that task-switching times will be short as there is very little context to change if there is no protection between different tasks. However, any tasks can corrupt memory space being used by others, which will crash the complete system.

Device drivers are processor-specific again and have to be developed for the specific application.

µC/OS-II does not support multiprocessor systems.

**Task Handling Method**

µC/OS-II has no notion of processes; it only has tasks. These are normally called threads all executing in the same process. All system objects are shared between tasks. As all tasks share the same context, task switching times will be quick as there is very little context to change. On the other hand, a clear

division between threads and processes helps define the interfaces between the different activities of the application.

The μC/OS-II kernel defines 64 different priority levels of which the lowest two are used by the kernel. However, Jean Labrosse recommends that the four highest and four lowest priorities tasks be reserved for future use by μC/OS-II. This leaves us with a maximum of 56 priority levels. According to [14], this is acceptable, but it is sometimes desirable to have 128 levels. This is especially so for large real-time applications that are being designed using techniques like RMA (Rate Monotonic Analysis), where every tasks needs to have its own distinct priority level. But μC/OS-II is intended for the smaller embedded applications and therefore each task can still have a different priority. In fact μC/OS-II requires that each task has its own priority and it does not support round-robin scheduling for tasks with the same priority. This limit the number of application tasks that μC/OS-II can manage to 56. This can be a limiting factor for bigger embedded applications.

The scheduling policy is a priority-based, preemptive algorithm. The μC/OS-II kernel ensures that the thread with the highest priority among the ready threads is the one that runs.

μC/OS-II uses priority ceiling protocol as the priority inversion mechanism.

| **Feature** | **Comments** |
|---|---|
| Model | Task (threads) only |
| Priority Levels | 64, 62 usable, 56 recommended |
| Max number of tasks | 62 usable, 56 recommended |
| Scheduling policy | Preemptive priority-based |
| Number of documented states | 5 (Dormant, Waiting, Ready, Running, ISR interrupted) |

**Table 6-1: Task Handling Method**



**Memory Management Method**

As mentioned before μC/OS-II has a flat memory space. μC/OS-II does not support a memory management unit (MMU).

μC/OS-II provides a partition based memory management service. Partitions are fixed-size block partitions. Many partitions, which can have different size memory blocks, are supported.

| Feature | Comments |
|---|---|
| MMU | Not supported |
| Paging/Swapping | No/No |
| Virtual memory | No |
| Memory protection models | No protection |

**Table 6-2: Memory Management Method**

**Interrupt Handling Method**

µC/OS-II does not support an Interrupt API. Instead it provides functions that should be called on entry and exit of an ISR. ISRs are processor specific and need to be specifically developed for each application and platform. As the ISRs are not abstracted by the RTOS, it leaves quite a lot in the hands of the developer. On one side the developer will have full control over the processor, but a faulty ISR can compromise the quality and effectiveness of the RTOS.

Many system calls are available from within an interrupt handler. These include functions for releasing semaphores and mutexes, and sending messages as well as those for getting semaphores and mutexes, and reading messages (non-blocking). This allows the interrupt handler to perform many actions. The interrupt handler can also access all the memory of the system. Interrupts can be nested and prioritized (depends on processor). This means that a higher priority interrupt can interrupt a handler and be serviced by its handler.

| Feature | Comments |
|---|---|
| Handling | Nested, prioritized |
| Context | Interrupted task |
| Stack | Interrupt stack |
| Interrupt-to-task communication | Most communication and synchronization objects |
| Minimum RAM | N/A |

**Table 6-3: Interrupt Management Method**

### 6.2.3 API Richness

**POSIX**

µC/OS-II does not support any real-time POSIX compliant calls.

**Task Management**

A good selection of task management function calls is available.

| **Task Management API function calls** | | **Comments** |
|---|---|---|
| Get stack size | √ | Part of TCB which is returned by a OSTaskQuery() call. |
| Set stack size | √ | Parameter is set on creation of task. |
| Get stack address | √ | Part of TCB which is returned by a OSTaskQuery() call. |
| Set stack address | √ | Parameter is set on creation of task. |
| Get task state | √ | Part of TCB which is returned by a OSTaskQuery() call. |
| Set task state | - | |
| Get TCB | √ | Part of TCB which is returned by a OSTaskQuery() call. |
| Set TCB | √ | Parameter is set on creation of task. |
| Get priority | √ | Part of TCB which is returned by a OSTaskQuery() call. |
| Set priority | √ | Parameter is set on creation of task, but can be changed when the kernel is running. |
| Get task ID | √ | Part of TCB which is returned by a OSTaskQuery() call. |
| Task state change handler | √ | Provided by hook functions that are called from the kernel |
| Get current stack pointer | √ | Part of TCB which is returned by a OSTaskQuery() call. |
| Set task CPU usage | - | |
| Set scheduling mechanism | N/A | Kernel only supports one type of mechanism. |
| Lock task in memory | N/A | Tasks are always in memory as there is no page swapping to disk |
| Disable scheduling | √ | |
| **Total** | **13/15** | |
| **Total in percentage** | **87%** | |

**Table 6-4: Task Management API richness**



**Clock and Timer**

µC/OS-II does not support absolute time functions. A 32-bit counter is used to do time related function and is updated with every kernel clock tick. There is a limited choice of time interval function calls.

| Clock API function calls | | Comments |
|---|---|---|
| Get time of day | - | |
| Set time of day | - | |
| Get resolution | - | |
| Set resolution | - | |
| Adjust time | - | |
| Read counter register | - | |
| Automatically adjust time | - | |
| **Total** | **0/8** | |
| **Total in percentage** | **0%** | |

**Table 6-5: Clock API richness**

| Interval API function calls | | Comments |
|---|---|---|
| Timer expires on an absolute date | - | |
| Timer expires on a relative date | √ | |
| Timer expires cyclical | - | |
| Get remaining time | - | |
| Get number of overruns | - | |
| Connect user routine | - | |
| **Total** | **1/6** | |
| **Total in percentage** | **17%** | |

**Table 6-6: Interval Timer API richness**



**Memory Management**

Only fixed block size partitions are available.

| Fixed block size partition | | Comments |
|---|---|---|
| Set partition size | √ | |
| Get partition size | √ | |
| Set memory block size | √ | |
| Get memory block size | √ | |
| Specify partition location | - | |
| Get memory block - blocking | - | |
| Get memory block - non blocking | √ | |
| Get memory block - with timeout | - | |

| Fixed block size partition | | Comments |
|---|---|---|
| Release memory block | √ | |
| Extend partition | - | |
| Get number of free memory blocks | √ | |
| Lock/unlock partition in memory | N/A | Partitions are always in memory as there is no page swapping to disk |
| **Total** | **7/12** | |
| **Total in percentage** | **64%** | |

**Table 6-7: Memory management - Fixed block size API richness**

| Non-fixed block size partition | | Comments |
|---|---|---|
| Set pool size | - | |
| Get pool size | - | |
| Make new pool | - | |
| Get memory block size | - | |
| Specify partition location | - | |
| Get memory block - blocking | - | |
| Get memory block - non blocking | - | |
| Get memory block - with timeout | - | |
| Release memory block | - | |
| Extend pool | - | |
| Extend block | - | |
| Get number of free bytes | - | |
| Lock/unlock pool in memory | N/A | Partitions are always in memory as there is no page swapping to disk |
| Lock/unlock block in memory | N/A | Partitions are always in memory as there is no page swapping to disk |
| **Total** | **0/12** | |
| **Total in percentage** | **0%** | |

**Table 6-8: Memory management – Non-fixed block size API richness**



**Interrupt Management**

There is a limited choice of interrupt handling functions.

| Interrupt handling | | Comments |
|---|---|---|
| Attach interrupt handler | - | |
| Detach interrupt handler | - | |
| Wait for interrupt - blocking | - | |
| Wait for interrupt - with timeout | - | |
| Raise interrupt | - | |
| Disable/Enable hardware interrupts | √ | |
| Mask/Unmask a hardware interrupt | - | |
| Interrupt sharing | - | |
| **Total** | **1/8** | |
| **Total in percentage** | **13%** | |

**Table 6-9: Interrupt handling API richness**

**Synchronization and Exclusion Objects**

Semaphores (binary and counting), event flags and mutexes are supported, but not POSIX signals. Binary and counting semaphores use the same system objects with the max count set to 1 for binary semaphores.

| Counting Semaphore | | Comments |
|---|---|---|
| Get maximum count | - | |
| Set maximum count | √ | |
| Set initial value | √ | |
| Share between processes | N/A | Processes are not supported |
| Wait – blocking | √ | |
| Wait - non blocking | √ | |
| Wait - with timeout | √ | |
| Post | √ | |
| Post – Broadcast | - | |
| Get status (value) | √ | |
| **Total** | **7/9** | |
| **Total in percentage** | **78%** | |

**Table 6-10: Counting semaphore API richness**

| Binary Semaphore | | Comments |
|---|---|---|
| Set initial value | √ | |
| Share between processes | N/A | Processes are not supported |

| **Binary Semaphore** | | **Comments** |
|---|---|---|
| Wait – blocking | √ | |
| Wait - non blocking | √ | |
| Wait - with timeout | √ | |
| Post | √ | |
| Get status (value) | √ | |
| **Total** | **6/6** | |
| **Total in percentage** | **100%** | |

**Table 6-11: Binary semaphore API richness**

| **Mutex** | | **Comments** |
|---|---|---|
| Set initial value | √ | |
| Share between processes | N/A | Processes are not supported |
| Priority inversion avoidance mechanism | √ | |
| Recursive getting | - | |
| Task deletion safety | √ | |
| Wait – blocking | √ | |
| Wait - non blocking | √ | |
| Wait - with timeout | √ | |
| Release | √ | |
| Get status | √ | |
| Get owner's thread ID | √ | |
| Get blocked thread ID | √ | |
| **Total** | **10/11** | |
| **Total in percentage** | **91%** | |

**Table 6-12: Mutex API richness**

| **Event Flags** | | **Comments** |
|---|---|---|
| Set one at a time | √ | |
| Set multiple | √ | |
| Pend on one | √ | |
| Pend on multiple | √ | |
| Pend with OR conditions | √ | |
| Pend with AND conditions | √ | |

| Event Flags | | Comments |
|---|---|---|
| Pend with AND and OR conditions | - | |
| Pend with timeout | √ | |
| **Total** | **7/8** | |
| **Total in percentage** | **88%** | |

**Table 6-13: Event Flags API richness**

| POSIX signals | | Comments |
|---|---|---|
| Install signal handler | - | |
| Detach signal handler | - | |
| Mask/unmask signals | - | |
| Identify sender | - | |
| Set destination ID | - | |
| Set signal ID | - | |
| Get signal ID | - | |
| Signal thread | - | |
| Queued signals | - | |
| **Total** | **0/9** | |
| **Total in percentage** | **0%** | |

**Table 6-14: POSIX signals API richness**

**Communication and Message Passing Objects**

µC/OS-II supports both mailboxes and message queue. Both contain only a pointer to a message structure. Each pointer is typically initialised to point an application specific data structure.

| Message Queue | | Comments |
|---|---|---|
| Set maximum size of message | N/A | All messages are pointers to message blocks. |
| Get maximum size of message | N/A | All messages are pointers to message blocks. |
| Set size of queue | √ | |
| Get size of queue | √ | |
| Get number of messages in queue | √ | |
| Share between processes | N/A | The kernel does not support processes |
| Receive – blocking | √ | |
| Receive – non blocking | √ | |
| Receive – with timeout | √ | |
| Send - with ACK | - | |

| Message Queue | | Comments |
|---|---|---|
| Send - with priority | - | |
| Send – OOB (out of band) | √ | |
| Send - with timeout | - | |
| Send – broadcast | √ | |
| Timestamp | - | |
| Notify | - | |
| Total | **8/13** | |
| Total in percentage | **62%** | |

**Table 6-15: Message queue API richness**

| Mailbox | | Comments |
|---|---|---|
| Set maximum size of message | N/A | All messages are pointers to message blocks. |
| Get maximum size of message | N/A | All messages are pointers to message blocks. |
| Share between processes | N/A | The kernel does not support processes |
| Send - with ACK | - | |
| Send - with timeout | - | |
| Send – broadcast | √ | |
| Receive – blocking | √ | |
| Receive – non blocking | √ | |
| Receive – with timeout | √ | |
| Get status | √ | |
| **Total** | **5/7** | |
| **Total in percentage** | **71%** | |

**Table 6-16: Mailbox API richness**



### 6.2.4 Internet Support

µC/OS-II does not support any networking protocols by default. Many protocols stacks are commercially available from third parties for µC/OS-II. µC/OS-II's support for networking protocols was not evaluated as we could not find any protocols stacks which are free of charge for educational purposes.

### 6.2.5 Tools

As stated earlier, µC/OS-II does not come with its own development environment. It is up to the developer to find the appropriate development environment for the specific processor and then add µC/OS-II source code to his application.

### 6.2.6 Documentation and support

µC/OS-II is well documented by its author in his book "MicroC/OS-II – The Real-Time Kernel" [1]. In addition to his book he also provides many applications notes on the µC/OS-II website [2]. There is also a newsgroup for µC/OS-II [3] where developers can share their knowledge and our experience was that it is reasonably active with quick responses on questions. All our questions to the author were answered in reasonably good time. There is no paid support available for µC/OS-II, other than contracting a consultant that knows the kernel. There are also training courses available from third parties.

### 6.2.7 Development methodology

As µC/OS-II is not bounded to a specific development environment, the development methodology is up to what kind of environment the developer is used to. Normally a separate host and target methodology is used in smaller embedded application development. This is also the methodology that was used in Chapter 5.

### 6.2.8 Conclusion

We did not rate any of the evaluations, except API Richness, as it is a highly subjective rating. One would require a high level of experience with various low and high level RTOS's do those ratings and we do not have that.

The positive and negative points are listed below.

**Positive points**

- Many different targets are supported
- Highly scalable
- Rich API on supported kernel services
- Developer has full control over ISR's context-switches
- Predictable and deterministic

**Negative points**

- Needs to be ported to if a port does not exists
- µC/OS-II context-switch performance is only as good as the port
- Developer has full control over ISR's context-switches

**API Richness Ratings**

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Task Management (87%) | | | | | | | | | 9 | |
| Clock (0%) | | | | | | | | | | |
| Interval Timer (17%) | | 2 | | | | | | | | |
| Memory management – Fixed block size partition (64%) | | | | | | 6 | | | | |
| Memory management – Non-fixed block size pool (0%) | | | | | | | | | | |
| Interrupt Handling (13%) | | 2 | | | | | | | | |
| Counting semaphore (78%) | | | | | | | | 8 | | |
| Binary semaphore (100%) | | | | | | | | | | 10 |
| Mutex (91%) | | | | | | | | | 9 | |
| Event Flags (88%) | | | | | | | | | 9 | |
| POSIX Signals (0%) | | | | | | | | | | |
| Message Queue (62%) | | | | | | 6 | | | | |
| Mailbox (71%) | | | | | | | 7 | | | |

**Table 6-17: API richness summary**

# Chapter 7 Summary

With this dissertation, the following work has been done:

- Study the inner workings of µC/OS-II and the programming model of the ARM7 Core architecture in order to port µC/OS-II to the Atmel AT91 ARM7 microcontroller. µC/OS-II was ported and tested in Chapter 5.
- Research the basic requirements for an RTOS as well as what functionalities are available in commercial RTOS's. With this research as background, the suitability and effectiveness of µC/OS-II as a RTOS for embedded applications was analyzed in Chapter 6.

## 7.1 Porting of µC/OS-II

We have studied real time systems in general and used it to understand the inner workings of the µC/OS-II kernel. We have also studied the ARM7 Core programmer's model in detail to understand how we must implement the necessary context switches from task level and from ISR level. The kernel's requirements for the processor-specific part of the source were studied and implemented as the AT91 ARM7 µC/OS-II port. We have shown how the port was tested in order to qualify it as fully operational.

## 7.2 Evaluating µC/OS-II

A technical evaluation as described by Dedicated System Experts [13] has been done on µC/OS-II. The following aspects were evaluated:

- Installation and configuration
- RTOS Architecture:
    - Task handling method
    - Memory management method
    - Interrupt handling method
- API Richness
- Internet Support
- Documentation and support
- Development methodology

We have seen that µC/OS-II is a good choice for an RTOS in small to medium embedded applications. It is real attractive for the small embedded applications because of it scalability and resulted small footprint. However, it is also suitable for the bigger embedded applications. It does not come standard with support for a File System and an internet protocol stack (TCP/IP), but it can be added when required. These did not form part of the evaluation. µC/OS-II does not provide any memory protection between application tasks as well as the kernel. This means that the embedded application will fail if one of its tasks is corrupting the memory. With the memory not protected, it is more suitable for the smaller embedded application where one can still managed to exercise tight control on memory usage and access.

### 7.3 Future work

This dissertation does not include any practical evaluation of µC/OS-II. Executing a performance analysis would be the natural next step of this work. For this we would have to acquire the necessary hardware. It would be interesting to evaluate the performance of µC/OS-II when it is put under load.

Add a File System (µC/FS) to µC/OS-II and evaluate its performance and suitability for embedded applications.

Add an internet stack (TCP/IP) and evaluate its performance and suitability for embedded applications.



**From Dedicated System Experts [15]:**

*"However it should not be forgotten that a good RTOS is only a building block. Using it in a wrongly designed system may lead to a malfunctioning RT system"*

# Chapter 8 Bibliography


1. Labrosse, Jean J. MicroC/OS-II The Real-Time Kernel, CMP Books, 1999.
2. www.micrium.com, The µC/OS-II website.
3. http://groups.yahoo.com/group/MicriumNewsGroup/, The µC/OS-II newsgroup.
4. www.arm.com, ARM Advanced RISC Machines website.
5. ARM Advanced RISC Machines, ARM7TDMI Data Sheet ARM DDI0029E, August 1995.
6. ARM Advanced RISC Machines, ARM Architecture Reference Manual ARM DDI0100E, June 2000.
7. www.atmel.com, Atmel's website.
8. Atmel, AT91 ARM Thumb Microcontrollers AT91M55800A 1745C-ATARM-12/2, December 2002.
9. Atmel, AT91 ARM Thumb Microcontrollers Application note: AT91 Assembler Code Startup Sequence for C Code Applications Software.
10. Atmel, AT91EB55 Evaluation Board User Guide.
11. Atmel, AT91 ARM Thumb Microcontrollers Application note: Disabling Interrupts at processor level Rev. 1156A–08/98, August 1998.
12. Atmel, AT91 ARM Thumb Microcontrollers Application note: AT91 Lib V2.0 Rev1385A-11/00, November 2000.
13. Dedicated Systems Experts website, www.dedicated-systems.com.
14. Dedicated Systems Experts, What makes a good RTOS, Dedicated Systems Experts website, June 2001.
15. Dedicated Systems Experts, Evaluation report definition, Dedicated Systems Experts website, June 2001.
16. Stepner, D., Nagarajan R., Hui D. Proceedings of the 36th ACM/IEE conference on Design Automation Conference, Embedded Application Design Using a Real-Time OS, June 1999.
17. Kalinsky, David. Context Switch in Embedded Systems Programming Magazine, February 2001.
18. Stewart David, B. Real Time in Embedded Systems Programming Magazine, November 2001.

19. Melkonian, Michael. Get by without an RTOS, Embedded Systems Programming Magazine.

20. Timmerman, Martin. RTOS Market Survey Preliminary Results in Dedicated Systems Magazine, 1Q-99.

21. Barr, Michael. Special Report: Choosing an RTOS, Embedded.com, www.embedded.com, December 2002.