USING MICROCONTROLLER KERNELS IN AN OPERATING SYSTEMS COURSE

Bruce P. Tis, Ph.D.
Computer Science Department
Boston University
Metropolitan College
755 Commonwealth Ave.
Boston, Ma 02215
bruce@bu.edu

ABSTRACT
Historically, introductory level operating systems courses have emphasized concepts and theory. However during the past 10 years there has been an increased interest in addressing design and implementation issues. Courses with this new emphasis generally require students to do projects ranging in complexity from using operating system calls to writing an entire operating system. Often these projects are too time consuming and detract from teaching the basic concepts. Also, some audiences do not need in-depth, highly technical exposure to a large operating system. I have found the use of microcontroller kernels exposes students to significant operating systems implementation detail, without requiring an inordinate amount of class or student time. This paper summarizes the significant features of microcontroller kernels, illustrates implementation details that reinforce major concepts, and outlines the assignments used in a introductory level operating systems course.

INTRODUCTION
The ACM SIGCSE literature and conferences are rich with examples of projects suitable for operating system courses. Wagner and Ressler [1] recently discussed the value of programming projects using system calls, accessing system structures and implementing classical synchronization problems. Berk [2] simplified the interface to SunOs threads to facilitate students exploration of concurrency issues through various programming projects. Holiday [3] focused on system calls and interrupt vectors. Ramakrishman [4] documented projects involving use of system calls, implementation of a UNIX command, and actual modification of UNIX source code. Lane [5] has students implement a small multiprogramming executive called MPX and Reek [6] teaches a second level course which requires students to implement an entire operating system. There are entire texts based on operating system implementation, such as
Tanenbaum [7] and Comer [8]. There are also supporting texts on the internals of specific operating systems such as Leffler [9] on 4.3BSD UNIX and M. Beck [10] on Linux.

Exploring system calls, operating system simulation, and investigating the operating system database are manageable activities and do not require a huge investment of time, however the student is not exposed to an actual operating system implementation. Reading source code and modifying an operating system such as BSD UNIX, Linux or MINIX can be very time consuming and may not be suitable for all audiences. Writing an entire operating system is even more time intensive. I have an alternative that allows students to gain exposure to operating system internals, but does not require a large investment of time. This approach is based on the use of microcontroller kernels (microcontroller software developers use the word kernel in lieu of operating system). Microcontroller kernels are relatively small operating systems, less than 5,000 lines of source code, that are used on single chip computers, microcontrollers, in embedded applications. While microcontroller-based applications are quite different than those which run under the typical time sharing, workstation, or server operating systems, the basic functionality required of all operating system is the same. Microcontroller kernels exhibit the properties of a microkernel, which in itself are worth teaching as discussed by Camp and Oberhauser [11]. Courses that are oriented to real time or embedded operating system development, as might be taught to engineering students, benefit even more from the use of microcontroller kernels. Due to the size of most microcontroller kernels, students can read and understand every line of code, write applications that use the kernel and even modify the kernel. In this paper I discuss the usefulness of microcontroller kernels in traditional operating system courses. More specifically I will discuss - microcontroller hardware, evolution of microcontroller kernels, uC/OS The Real Time Kernel, uC/OS implementation details, and assignments based on uC/OS.

MICROCONTROLLER HARDWARE

Microcontrollers are embedded control processors programmed to manage specific tasks. Typically these microcontrollers are embedded in a host system such that their presence and operation are invisible to the user of the system. Embedded control applications include:

- office automation products such as copiers, laser printers, mass storage systems, modems and fax machines
- consumer electronics such as VCRs, microwave ovens, video and audio electronics and air conditioners
- avionics systems
- industrial automation systems
• automotive systems

Microcontrollers are single chip computers which include the CPU, ROM to contain the kernel and application software, RAM for dynamic storage, timers, I/O ports for sensing input data and controlling output devices, serial ports and sometimes A/D converters. Table 1 summarizes key characteristics of various popular microcontrollers.

<table>
<thead>
<tr>
<th></th>
<th>Intel MCS 51</th>
<th>Moto 68hc05</th>
<th>Moto 68hc11</th>
<th>Intel MCS 96</th>
<th>Moto 68333</th>
</tr>
</thead>
<tbody>
<tr>
<td>No of bits</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>ROM</td>
<td>4-8k</td>
<td>4k</td>
<td>24k</td>
<td>8-18k</td>
<td>16mb ext</td>
</tr>
<tr>
<td>RAM</td>
<td>128</td>
<td>176</td>
<td>256-1024</td>
<td>256-512</td>
<td>2k 16mb ext</td>
</tr>
<tr>
<td>I/O ports</td>
<td>32</td>
<td>10-24</td>
<td>32</td>
<td>32-56</td>
<td>47</td>
</tr>
<tr>
<td># Timers</td>
<td>2-3</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>Serial Port</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>A/D</td>
<td>none</td>
<td>1</td>
<td>1</td>
<td>4-8</td>
<td>4-8</td>
</tr>
</tbody>
</table>

Table 1 - Microcontroller Characteristics

THE EVOLUTION OF THE MICROCONTROLLER KERNEL

Wood and Barrett [12] recount the evolution of microcontroller kernels. The precursor to an actual microcontroller kernel was the use of the technique known as a "super loop" to control the microcontroller. The super loop approach involves writing code in the form of one big loop of code which is executed repeatedly. In this loop all application functions, including timing, are performed. An attempt was made to write a synchronous program to handle an application that may need to handle events that are asynchronous in nature, without the use of a interrupt system. This approach is only appropriate for very simple applications, typically those that do not have real-time constraints. Most applications are not serial in nature and do not lend themselves to being solved by the super loop approach.

The next step in the evolution of microcontroller kernels involved the use of the hardware interrupt system and interrupt handlers in the kernel. This form of kernel was strictly an asynchronous event handler which gave rise to multitasking. In order to provide real-time response to certain tasks, priority based schedulers were then developed. Applications were organized as a series of tasks, each with the appropriate
priority, and the kernel was responsible for responding to interrupts, scheduling tasks, pre-empting tasks, and context switching between tasks. Most kernels provide synchronization services via semaphores or event flags, and intertask communication via mailboxes or queued message delivery services. Microcontroller kernels also typically provide timer based services which allow a task to suspend itself for a predetermined period of time, or schedule a time based interrupt for another task. Memory management is also provided, usually to manage a heap of storage for dynamic storage allocation. Seldom is any form of virtual memory or dynamic relocation of tasks provided.

There are dozens of commercially sold kernels for microcontrollers. They vary in what target processors they support, as well as what functions they provide. There are two kernels in the public domain that are quite suitable for educational use. MCX-11, developed by A.T. Barrett, is a real-time kernel for the Motorola 68hc11 microcontroller. This kernel provides[13,14]: multitasking, pre-emptive task scheduling by priority (fixed priority per task), intertask communication and synchronization via semaphores, messages and queues, support for timed operations, fast context switch, small RAM and ROM requirements, and 15 executive service request functions. Task characteristics are specified at assembly time and cannot be changed dynamically. The kernel code requires approximately 1250 bytes of ROM and is written entirely in assembly language. I have used this kernel in operating system courses. However, since it is written in 68hc11 assembly language, there is a learning curve for mastering the assembly language before the kernel source code can be understood. The second kernel uC/OS, written almost entirely in C, is much easier for students to understand.

**uC/OS - THE REAL TIME KERNEL**

uC/OS is a small, but powerful real time kernel written by Jean J. Labrosse, and first described in two articles [15,16] published in Embedded Systems Programming. Mr Labrosse also published a book titled "uC/OS The Real Time Kernel" [17] which fully documents the kernel. The software is available from the publisher and on the Internet via ftp from ftp://ftp.cygnsus.com/pub/embedded/ucos/. The book fully explains the internals of the kernel, as well as real time programming concepts. It is an excellent supplementary text for students.

uC/OS is a portable, preemptive, real time multitasking kernel for x86, 68K, 68HC11, CPU32, Z-80, MCS251, XA, H8/300H, ARM and 80196 target processors. Applications, as well as uC/OS itself, can be developed on Windows NT, MS-DOS or Windows 95.

uC/OS is comprised of 5 source code files totaling 1759 lines of source code. The bulk of the operating system is in uCOS.h and uCOS.c (1498 lines), which are target processor independent. The target processor dependent code, for the x86 architecture, is in 2 C language files ix86s.h and ix86s_c.c (total of 111 lines of code) and one assembly language module, is86s_a.asm (150 lines of code). The C language code was originally developed using a Borland C compiler under MS-DOS.
uC/OS has a preemptive priority driven scheduler which supports up to 63 user definable tasks, each with a unique priority. uC/OS provides 22 system services including services for mailboxes, semaphores, task management, scheduling/multitasking control, message queues, timer management, interrupt service routine entry/exit and time management. All tasks are memory resident and the kernel does not handle any I/O. System calls are made by simple C function calls. Application tasks are C functions compiled with uC/OS.

An interesting characteristic of uC/OS is its determinism. It is possible, given the target processor’s instruction timing, to calculate worst case response times for any event or system call. No kernel functions are dependent on the number of tasks, or the number of tasks in a given state etc. All queues are implemented to make their management - insertions, searches and removals - independent of the number of items in the queue. Critical sections are handled by the use of two macros, OS_Enter_Critical() and OS.Exit_Critical(), whose definition is target processor dependent. Typically the microcontroller interrupt system is turned off and on before and after a critical section.

Tasks can be in one of six different states - dormant, ready, running, delayed, pending or interrupted. Each task has a task control block containing a pointer to the top of the task’s stack, status, priority, delay in case the task is waiting, and various bit masks to facilitate queue manipulation. There is a table, OSTCBPrio[], indexed priority number, which points to all the TCBs.

**uC/OS IMPLEMENTATION DETAILS**

One reason for reading operating system code is to relate major concepts to implementation. To illustrate this, I’ll describe an important database, and mechanism to manipulate this database, which are fundamental to all operating systems and implemented in uC/OS in a clever way.

Operating systems are highly dependent on the use of queues to manage tasks in various states. Queue manipulation mechanisms are needed to add and remove tasks, as well as to search for tasks that meet some specific criteria. Students rarely get exposure to actual implementation details, and while they can relate what they have learned in a data structures course, there is no substitute for seeing an actual implementation.

The uC/OS implementation of queues is rather unique and clever. More importantly, it illustrates how to implement a deterministic operating system. There are numerous queues in uC/OS. The ready queue is an important example. The ready queue contains a set of tasks in order by priority. Mechanisms are needed to add and remove a task and to determine the highest priority task in the queue. The traditional approach to implementing a priority queue is with a linked list, where the entries in the list are task control blocks. Pointers are used to implement the links, and routines are written to perform the basic operations. Such an approach produces non-deterministic performance that is a function of the number of tasks in the queue. This variation in
performance is not acceptable in a real time operating system. uC/OS avoids the use of pointers and linked lists by representing the queue by two bit maps. All operations involve bit map manipulation and are facilitated by storing additional data in the task control block and two additional tables. All algorithms are strictly deterministic.

A queue, in this specific case the ready queue, is represented by an 8 byte table (see figure 1) with bits numbered 0 thru 63, where each bit corresponds to a single task with a priority equal to the bit number. If the bit is set (1) the task is in the queue and if cleared (0) the task is not in the queue.

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
<td>18</td>
<td>17</td>
<td>16</td>
</tr>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
</tr>
<tr>
<td>39</td>
<td>38</td>
<td>37</td>
<td>36</td>
<td>35</td>
<td>34</td>
<td>33</td>
<td>32</td>
</tr>
<tr>
<td>47</td>
<td>46</td>
<td>45</td>
<td>44</td>
<td>43</td>
<td>42</td>
<td>41</td>
<td>40</td>
</tr>
<tr>
<td>55</td>
<td>54</td>
<td>53</td>
<td>52</td>
<td>51</td>
<td>50</td>
<td>49</td>
<td>48</td>
</tr>
<tr>
<td>63</td>
<td>62</td>
<td>61</td>
<td>60</td>
<td>59</td>
<td>58</td>
<td>57</td>
<td>56</td>
</tr>
</tbody>
</table>

Figure 1 - RdyTbl

These 64 bits are viewed as 8 groups of 8 bits each. Group 0 corresponds to bits 0-7, group 1 to bits 8-15 etc. An additional bit map, figure 2, called RdyGrp, consists of 1 byte where each bit corresponds to one group in the RdyTbl. Bit number 1 in RdyGrp corresponds to group 1, or tasks 8-15. If a bit in RdyGrp is set then there is at least one bit set in the corresponding group in the RdyTbl. If a bit is not set then there are no bits set in the corresponding group in RdyTbl.

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

Figure 2 - RdyGrp

| 0 | 0 | y | y | y | x | x | x |

Figure 3 - Task Priority
For instance if RdyGrp contains 0x15 there is at least one task in group 4 (tasks 32-39), group 2 (tasks 16-23) and group 0 (tasks 0-7).

Mapping a task into the RdyTbl is quite easy. The priority, or task number, is a 6 bit number, figure 3, between 0-63. The left most (of the 6 bits) 3 bits corresponds to the group, usually denoted by the letter y, and the rightmost three bits correspond to the bit within the group, usually denoted by x. Therefore task number 55 (decimal) or 37 (hex) would be 00110111 binary and correspond to group 6 and bit 7 within the group, which corresponds to the entry for task 55 in figure 1.

Operations on the queue include inserting an entry, removing an entry, and searching for the highest priority entry. Inserting and removing an entry involves setting or clearing one bit in the RdyTbl, and possibly a second bit in the RdyGrp table. Given a task’s priority, mapping either the group number, y, or the number within the group, x, is done via another table, called the OSMapTbl. This table, figure 4, is eight bytes long, is indexed by either the group number or member within the group to extract and entry with a 1 in the appropriate bit position.

<table>
<thead>
<tr>
<th>Index</th>
<th>OSMapTbl</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000001</td>
</tr>
<tr>
<td>1</td>
<td>00000010</td>
</tr>
<tr>
<td>2</td>
<td>00000100</td>
</tr>
<tr>
<td>3</td>
<td>00001000</td>
</tr>
<tr>
<td>4</td>
<td>00010000</td>
</tr>
<tr>
<td>5</td>
<td>00100000</td>
</tr>
<tr>
<td>6</td>
<td>01000000</td>
</tr>
<tr>
<td>7</td>
<td>10000000</td>
</tr>
</tbody>
</table>

Figure 4 - OSMapTbl

This entry can then be used to set or clear the corresponding bit in RdyTbl and RdyGrp.

Assuming that p contains the task’s priority, the following code will insert a task into the ready queue:

\[
\text{RdyGrp} \quad | = \quad \text{OSMapTbl} \left[ p \triangleright 3 \right] ;
\]

\[
\text{RdyTbl} \left[ p \triangleright 3 \right] \quad | = \quad \text{OSMapTbl} \left[ p \ & \ 0x07 \right] ;
\]

and the following code will remove a task with priority p from the ready queue:
if ( ( RdyTbl[ p >> 3] & = -OSMapTbl[ p & 0x07] ) == 0 )
    RdyGrp & = -OSMapTbl[ p >> 3];

Finding the highest priority task in a queue is potentially more involved. The RdyTbl bit map has to be scanned starting with the highest priority position until the first 1 bit is found. This could be optimized by scanning RdyGrp to find the first group containing a task, and then the group entry in RdyTbl. However, even this scanning of two bytes is not acceptable. An additional table called UnMapTbl (figure 5), with 256 entries, is used to map either a single RdyTbl or RdyGrp byte, which can take on any one of 256 values, to a single integer value signifying the position of the first bit, scanning from right to left, that is set.

*******************************************************************************
PRIORITY RESOLUTION TABLE
*******************************************************************************
* Note: Index into table is bit pattern to resolve highest priority
* Indexed value corresponds to highest priority bit position (i.e. 0..7)
*******************************************************************************

UBYTE const OSUnMapTbl[] = {
    0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0
};

Figure 5 OSUnMapTbl

For example, if RdyGrp contained 0x15, meaning that the bits for groups 4, 2 and 0 were set, indexing into OSUnMapTbl by 0x15 would yield a value of 0. 0 signifies that the group with the highest priority with an entry in the queue is group 0. To determine which task in the group is the one with the highest priority OSUnMapTbl is indexed by the RdyGrp[0] entry. Assume that RdyGrp[0] has a value of 0xB6 (1011 0110) meaning that tasks 1, 2, 4, 5, and 7 are in the queue. Indexing OSUnMapTbl by RdyGrp[0], or 0xB6, yields the integer value of 1, the bit position of the highest priority task within the group. Now that we have the group number, call it y, and the position within the group, call it x, calculating the priority number is a simple matter. The following code illustrates this process.
y = OSUnMapTbl [ RdyGrp ];
x = OSUnMapTbl [ RdyTbl [ y ] ];
p = [ y << 3 ] + x;

A priority queue is represented by a bit map. Queue operations for inserting and removing a member from the queue and determining highest priority member in the queue are table driven bit manipulation operations. These operations are deterministic, which is important in a real time environment. Such a concrete example is invaluable in relating theory to practice. Students are generally surprised to see a queue implemented in such a fashion.

This basic mechanism is used throughout uC/OS to implement lists of tasks waiting for various events associated with semaphores, mailboxes and message queues.

Other aspects of uC/OS, such as the implementation of semaphores, mailboxes, message queues, interrupt processing and initialization, are equally as interesting and more importantly within the range of a student to thoroughly understand because of the size of the kernel. The reader is referred to the text "uC/OS The Real Time Kernel" [17] for more detail.

ASSIGNMENTS BASED ON uC/OS

My students are required to read the two uC/OS articles that appeared in Embedded Systems Programming [15,16]. They are given a listing of the source code, and I also put the uC/OS text [17] on reserve in the library for them. I spend about three hours in class reviewing various aspects of the code. They are then capable of reading the code in depth and completing the assignments. These assignments include the following areas:

- Kernel database and architecture
- Clock interrupt and basic clock cycle of the kernel
- CPU scheduling in depth
- Application Development
- Task Management
- Interrupt service routines and interface
- Software timer management
- Intertask communication using message queues and mailboxes
- CPU Scheduling

The assignments involve reading and understanding all the source code. They are given various initial conditions and events, and then asked to trace the resulting execution path to determine what code is executed and how the database is changed. Most questions require line numbers of the code or definitions to support their answers. Application development and actual modification of the kernel source code is also encouraged.

**Kernel Database and Architecture** - Students explore the organization of the kernel in general. They learn the function of each module and look at the header files in detail. They also investigate the parameters that are used to configure the kernel at compile
time. All the major data structures are investigated, as is the code for kernel initialization.

**Interrupt service routines** - Students learn the conventions for writing interrupt service routines and how these routines interface to the rest of the kernel. Important routines such as IntEnter (Interrupt service routine entry) and IntExit (interrupt service routine exit) are investigated. Saving and restoring the state of the machine is illustrated. Kernel code is related to the hardware architecture. The interface to CPU scheduling, resulting in context switching and task state changes, are also investigated.

**Clock Interrupt and Basic OS Cycle** - The periodic clock cycle, which every operating system executes, is never studied in depth in a concepts-oriented operating system course. Here, the student is required to trace the cycle from the basic clock interrupt through the timer request processing to CPU scheduling, context switching and dismissal of the interrupt.

**Timer Request Processing** - Most operating systems provide the capability for a process to suspend itself for some period of time. This capability is included within uC/OS as well. Most system calls that initiate some processing, that results in a future event, also have a timeout capability included. Investigating the timer database, how the timers are set, and how they are processed is very important.

**CPU Scheduling** - The CPU scheduler is very simple in uC/OS, as it is a priority driven scheduler. Students explore all conditions under which the scheduler can be called. They also look at the target processor assembly language routines necessary to do context switching. The ability to suspend, and eventually resume, a task is traced.

**Task management** - This area includes task state changes and the events that cause them. Here the relation between system call code and interrupt level code is investigated. System calls such as task creation, task deletion, and changing a task's priority are explored.

**Intertask communication using message queues and mailboxes** - Intertask communications is extremely important in any operating system. uC/OS has two mechanisms for communication - message queues and mailboxes. The code for management of message queues and mailboxes is also traced.

**Application Development** - Studying actual operating system code is extremely beneficial to the student. However, for some students even this can be a bit abstract. Use of operating system services can often help the student relate operating system services, and their implementation, to applications. I have students write small applications that exercise various system calls. These applications are no more than small C functions that are compiled with uC/OS and then run on a PC under uC/OS.

**SUMMARY**

Learning operating system concepts and theory can be enhanced by knowledge of the internals, or actual implementation, of an operating system. Studying the internals
of an operating system such as Minix or Linux, as well as writing your own operating system, can be very time consuming. One type of operating system that is small enough to grasp, in its entirety, in just a few hours is a microcontroller operating system, or kernel. Major functions of a much larger operating systems, such as multiprogramming, task management, CPU scheduling, interrupt processing, synchronization and intertask communication, are implemented in kernels of only a few thousand lines of source code. Students can benefit from reading code that defines and manipulates the operating system's database, as well as interrupt service routines, the periodic clock driven operating system cycle, CPU scheduling, synchronization and task communication. Such small operating systems can also be modified quite easily. I have used uC/OS quite successfully in an introductory operating system course and highly recommend such an approach.

REFERENCES


