AC 2007-1261: EMBEDDED SOFTWARE DESIGN METHODOLOGY TO HELP STUDENTS SUCCEED IN THE REAL WORLD

Keith Curtis, Microchip Technology Inc.
Embedded Software Design Methodology to Help Students Succeed in the Real World

Introduction: A Tool for Entering the Workforce with Experience

In the good old days, new engineers could look forward to a long and rewarding career, working for a well-established engineering firm. They would typically spend their first year of employment “learning the ropes” from older, more-experienced engineers. During this apprenticeship, they would pick up the tips, tricks and engineering shortcuts necessary to accomplish miracles in engineering productivity. Then, in an engineering rite of passage, they would graduate to handling their own projects and become a full partner in the engineering brotherhood. In time, their experience would transform them into the older, more-experienced engineers that had mentored them, and they would in turn pass their wisdom on to the next generation of new college graduates.

Unfortunately, in today’s embedded microcontroller job market, this ancient and beneficent brotherhood of engineering is long gone. Engineers seldom spend more than 3-5 years with a single company, so employers are reluctant to invest a year’s salary in mentoring. The older, more-experienced engineers that once would have been mentors are now competitors in an increasingly tough job market. On top of all this, new engineers face competition from offshore design centers and foreign engineers immigrating to the U.S.

If junior engineers want to compete in this fast-paced, competitive job market, they have to be productive on their very first day. To be productive, they have to be able to create complex, solid code quickly. Experienced engineers accomplish this using their personal collection of tips, tricks, and shortcuts that they have picked up over the course of their career. New engineers do not have this luxury. What they need instead is a design methodology that will take the place of the tips, tricks and shortcuts. Thus presents the basic point of this paper—to teach a design methodology that will allow new engineers to create complex, solid code quickly.

How can junior engineers quickly develop complex, solid code? Let us start by defining the specific goals:

1. The methodology must produce code that is capable of multitasking. Today’s electronics do not perform just one task, so engineers must be able to produce code that can accomplish more than one thing at a time. Additionally, being able to replace hardware with software is always a prime consideration in cutting production costs. In order to do this, the code must be able to multitask and execute the soft peripherals with other main software functions.

2. The methodology must produce code that is capable of real-time control. The world operates in real time, and if the code has to deal with the real world, then it must produce controls with predictable, repeatable timing.
3. The methodology must produce extensible code. Changes and additions, both at the time of the design and later, in the form of upgrades and bug fixes, must be easy to implement. To accomplish this, the design must be well documented and modular, with a clear, top-down design. Without these capabilities, the affects of code changes cannot be predicted or even limited to one module or group of modules. This leaves the engineer with only the option of a full rewrite in response to changes.

4. The methodology must produce reusable code. This falls under the shortcut heading—the fastest way to produce solid-code is to reuse existing solid code. Here, the well-documented and modular nature of the methodology should keep in mind the goal of reuse.

5. The methodology must be able to handle complexity. A top-down, modular approach starts by breaking the main large and complex functions into smaller, less-complex blocks, then break each of those blocks into smaller, less-complex blocks, and so on until the design is just a group of smaller manageable pieces. To distill these goals down to a simpler expression, what is needed is a top-down, modular design methodology that is capable of producing multitasking code with predictable, repeatable timing and an emphasis on documentation. To accomplish this goal, simple software state machine constructs can be combined with data structures, timers, logic and a structured design flow to create a simple way for engineers to quickly create complex, solid code.

Before diving into the methodology, we need to start with a few basic concepts related to multitasking. The next section of this paper will cover these concepts by examining how real-time operating systems (RTOSs) handle multitasking. Once we have these concepts in hand, we will move on to the methodology itself, starting with the top (system level), and working our way down through the component level, to the actual implementation.

RTOS Concepts

Whether a system uses an RTOS or multitasking software, four basic functions are needed:

1. Context Switching (switching between tasks)
2. Intertask Communication
3. Priority Handling
4. Timing (the system “Tick”)

Each function has an important role in the system. As such, a short explanation of each is necessary before discussion of the state-machine design methodology.
Context Switching

Context switching is the process of multiplexing several tasks through the CPU in such a way that the tasks are transparent to each other. This involves storing and retrieving the context of each task (the program counter, stack pointer, accumulators and status register) at the stop and start of each task’s allotted time. Repeated context switching between multiple tasks creates a system that appears to execute all of the tasks simultaneously, resulting in a multitasking system. The two main types of RTOSs -- preemptive and cooperative -- handle context switching in much the same way. The differences lie in what triggers the switch and the impact that the timing of the switch has on the system.

In a preemptive RTOS, context switching is accomplished using the microcontroller’s interrupt hardware in much the same way as an Interrupt Service Routine (ISR). The difference is that, prior to the return from interrupt, the preemptive RTOS retrieves the context of the next task, instead of the task that was interrupted. The interrupt is repeated over and over, giving each task a segment of time in the CPU. Due to the asynchronous nature of the interrupt, a preemptive RTOS has additional overhead in the context switcher because it must not only store the normal register set, but also any other variable that could be used by the task. Asynchronous timing is an additional reason a more complicated protocol is needed when handling multi-byte variables that are shared by more than one task.

The cooperative RTOS accomplishes context switching in much the same way. The difference is that, instead of an Interrupt, the cooperative RTOS uses a subroutine CALL. Regular GOSUB calls are scattered throughout the various tasks. When a task executes a CALL to the RTOS, the context switcher is the subroutine called. Prior to executing a RETURN from the subroutine, the context switcher swaps out the context of the current tasks for the next task, returning to the new task. This significantly reduces the context switcher’s storage overhead and simplifies variable handling, because the designer now controls the timing of the context switch. However, this leaves designers with the added burden of placing the RTOS GOSUBs at regular intervals.

Whether the RTOS is preemptive or cooperative, the concept of context switching is the same. The current task’s context is saved at the end of its tick, and the context of the next state is retrieved prior to the start of the next task. This allows each task to execute seemingly without interruption, even though it is regularly swapped in and out of the CPU.

Intertask Communication

When a system has multiple tasks to operate, the next step is to provide communication between the tasks to share information and coordinate activity. This is collectively referred to as intertask communication. Special communication protocols are required to manage this intertask communication, due to the asynchronous nature of the tasks and the different rates at which they process information. Because this communication can have the effect of synchronizing two tasks, the protocols must also handle situations where tasks are so completely out of synchronization that they lock up waiting for each other.
**Priority Handling**

The next concept common to multitasking systems is a means for dynamically changing the order or priority of the tasks that are granted execution time. In a simple system, all of the tasks may have the same priority, and the order in which they are executed does not change. However, in higher-performance systems, it may be necessary to delay the execution of lower-importance tasks in favor of one or more other tasks that perform functions that are more important. For instance, a task that monitors water pressure in a building may normally have a lower priority than the task that smoothes the ride of the elevators. However, when a fire breaks out, water pressure is suddenly more important than smooth elevator travel, requiring a change in system priorities. This is a case of priority handling—giving one task more preference than another, based upon the current state of the system.

**Timing**

Finally, multitasking systems need a timing Tick. The Tick is a construct that sets the system’s timing and regulates the context switcher, so that each task is allotted a consistent block of execution time. In a preemptive RTOS system, the Tick is typically a hardware timer interrupt. When the timer times out, the interrupt service routine resets it and the context switcher swaps the tasks.

In a cooperative RTOS, the Tick is less formal, relying on the placement of GOSUB calls to the RTOS to release the processor at regular intervals. In this case, the Tick is the average time between calls to each task’s RTOS. In either system -- hardware or software, preemptive or cooperative -- the Tick is always the basic unit of continuous execution time for a given task.

In any multitasking system, the following units and the Tick will all be present in some form:

1. Context Switcher
2. Intertask Communications
3. Priority Control

These are multitasking’s overhead and must be addressed at each level of the design, in order for the system to operate properly.
Overview of State Machine-Based Multitasking

To explain the idea of state machine-based multitasking, it is best to start with a good understanding of software state machines. A software state machine is a software construct that constrains multiple blocks of software that are accessed via an execution pointer called the “state variable.” Each time the state machine is called, the block associated with the state variable’s current value is executed. If the state machine is called repeatedly and the state variable is incremented after each call, the blocks will be executed one after another in a linear fashion. Holding the value of the state variable constant and repeatedly calling the state machine will execute the same block repeatedly, creating a LOOP. Assigning values to the state variable within a block of software changes the flow of the state machine, creating the equivalent of a GOTO or BRANCH.

State machine-based multitasking is based on the idea of creating a group of state machines—one for each task—and then calling them, one after another within an infinite loop. Each pass through the loop executes one state in each state machine, creating multitasking. This is similar to the cooperative RTOS. The difference is that, in a multiple state-machine system, the only things that need to be stored are the state variables. In fact, designing a state machine-based multitasking system is simply the process of replacing calls to the RTOS with the state-machine framework.

Design Methodology

The design methodology is broken down into three levels:

1. System-Level Design
2. Component-Level Design

The levels are organized here in a top-down design format, with each step adding detail to the previous level of the design. Included in each section of this paper are questions to be answered in the design and the general documentation suggested. As an addendum to the implementation and testing section, a fourth section covering optimization (tips and tricks) is included. This section contains general suggestions for reducing the size and increasing the speed of the design.

System-Level Design

System-level design covers the overall design of the system, including grouping system functions into tasks, specifying the type and layout of intertask communication, analyzing the timing needs of the system and documenting each task’s priority.

Design decisions at the system level should be based upon specific project goals. If a specification document for the system is not available, it is strongly recommended that a document, even an informal one, be generated to clearly outline the objectives of the design before beginning.
Component-Level Design

Component-level design takes decisions made at the system level and converts them into templates for the major system blocks. This includes system design of the state machines, variable and protocol definitions, timer routine design and the selection of a priority-control algorithm. Component-level design does not reach the level of actual coding, but it does generate the outlines for the specific code components to be written.

Implementation and Testing

The implementation and testing section covers the project level of actually writing software. Included in this section are suggestions on the project layout, development flow and testing, common methods for implementing state decoding, and a discussion of some common problems with multitasking and state machine-based design—specifically, state lock.

Optimization

This addendum deals with tips and tricks for writing smaller and faster code—specifically, auditing a compiler, writing faster conditional statements, tricks to speed up address decoding and ways to take advantage of the microcontroller’s architecture.

System-Level Design

System-level design involves four areas:
1. Task Definition
2. Intertask Communication Layout
3. System Timing Analysis
4. Documenting Task Priorities

Task definition sets the requirements for the state machine design at the component level. Intertask communication layout defines the framework of variables and the associated protocols that handle communication between state machines. System timing analysis determines each task’s timing needs and those of the overall system. The final area – documenting task priorities – involves analysis of the system’s priority-handling requirements. These sections are important, as they determine the requirements that the design must follow. Designers are encouraged to take the time to consider all possibilities before settling on a specific design.

Task Definition

The first step is to make a list of all functions that the software will be required to perform. Include all communication functions, command decoders, time bases, peripheral input/outputs (I/Os) and any other functions that the software will perform during its lifetime. When listing communication and I/O functions, break transmissions and receive functions out separately. Do not forget to include initialization, error-handling and error-recovery functions. Figure 1 shows an example task list for a garden-variety alarm clock. A general description of the clock is included as appendix A.
When this list is compiled, group the functions into tasks. Often, the first impulse is to give every function its own task. While this may seem to simplify and make the design modular, assigning each function its own tasks means that related tasks will have to communicate through a more complex protocol, due to their asynchronous nature. Placing related functions into a single task simplifies communication, which reduces overhead and potential problems. Even if the functions do not normally communicate, a common task for mutually exclusive functions can save program memory by reusing common elements.

A second common impulse is to group all related functions into only two or three tasks. This unnecessarily complicates the design. After all, the purpose is to allow at least some of the functions to operate simultaneously. Putting two unrelated functions into a single task makes the two functions mutually exclusive, which allows the system to execute one function or the other, but not both at once. The secret is to find a balance between these two extremes that meets the design specifications, while minimizing the number of state machines.

To assist in assigning functions to tasks, a short list of criteria is provided below for reference. Please note that this list is by no means all-inclusive. Potential future expansion of the system should be taken into consideration. These rules are not fixed. Designers are encouraged to add to or modify this list as needed for their design.
• Functions that **may** execute simultaneously **should** reside in separate tasks.
• Functions that must execute synchronously to each other, **should** reside in the same task.
• Functions that execute at different rates **must** reside in separate tasks.
• Functions that have asynchronous timing, relative to other functions, **must** reside in separate tasks.
• Functions that are mutually exclusive in their execution **may** reside in the same task.
• Functions that are extensions of another function **should** reside in the same task.
• Functions that operate as subroutines to more than one other function **should** reside in separate tasks.
• Two separate tasks **should not** control a single peripheral or resource (date, variables).
• Functions that are intermittent in operation should exist in separate tasks for priority handling.

To help illustrate this point, Figure 2 shows one possible grouping of functions for the alarm clock example (the functions have been grouped into tasks). The reasoning behind each choice is listed.
TASK LIST:

1. **Time Task**
   1.1. Increment Time at 1 Hz Rate
   1.2. Slow Set Time Increment
   1.3. Fast Set Time Increment
       All operates on a common variable, CurrentTime
       All are mutually exclusive of each other
       All are extensions of the Time keeping function
       None operate as subroutines of other tasks
       All operate with comparable execution rates

2. **Alarm Task**
   2.1. Monitor Time for Alarm Test
   2.2. Slow Set Alarm Increment
   2.3. Fast Set Alarm Increment
   2.4. Snooze (Delay Alarm)
       All operates on a common variable, AlarmTime, AlarmActive, AlarmOn
       All are mutually exclusive of each other
       All are extensions of the Alarm function
       None operate as subroutines of other tasks
       All operate with comparable execution rates

3. **Display Task**
   3.1. Display Time in AM/PM
   3.2. Display Time in Military
   3.3. Display Alarm in AM/PM
   3.4. Display Alarm in Military
   3.5. Flash Time Display at 2 Hz Rate
   3.6. Display Alarm ON/OFF
       All functions read, but don’t modify, external variables
       All functions control the LED display peripheral
       All are extensions of the display function
       None operate as subroutines of other tasks
       All operate with comparable execution rates

4. **Sound Task**
   4.1. Generate 1-2 kHz Tone, Modulated at 2 Hz Rate
       Function is intermittent
       Function is asynchronous to most other tasks
       Function is only control of sound peripheral

5. **Key Task**
   5.1. Key-press scan
       Function is a subroutine to Time, Alarm and Display tasks
       Execution rate is different than most other tasks
       Function operates asynchronously to most other tasks

6. **Command Task**
   6.1. Auto repeat Fast Set Time
   6.2. Auto repeat Fast Set Alarm
   6.3. Auto repeat Slow Set Time
   6.4. Auto repeat Slow Set Alarm
   6.5. Toggle Alarm ON/OFF
   6.6. Toggle AMPM versus Military
       Functions only operate in response to key press task so has different execution
       rate from Key Task
       Functions are a subroutine to Alarm, Time and Display tasks
       Functions operate simultaneously with the Key task function

---

**Figure 2:**
One Possible Grouping of Functions for Alarm Clock
Looking at this example, the display tasks grouping is fairly intuitive, and the grouping rules bear this out. However, intuition suggests that the time and alarm increment routines should reside in the command task along with the key-press functions, as they are always synchronous with one another. The rules though, point out that the rate of execution for the Key and Command tasks are different. The rules also point out that the Time and Alarm increment functions operate on variables that they share with the Alarm and Time-Keeping functions. Therefore, intuition is not always a good guideline. Sometimes similarity in function is not enough, and we need to look at the big picture and make decisions from the system’s point of view.

As a mental exercise, consider how the functions would be rearranged if we added the capability to read and set the various variables through a serial port. Will the serial-command decode be added to the Command task, or will it need a separate task? Consider the timing of the serial port and the potential contention between the serial and button interface when setting the Current Time and Alarm Time.

The next step in the design is to map out intertask communications. Therefore, the final step in task definition is to compile a list of the data-input and -output paths for each task. The list does not need to be specific to the type or size of data path—it just needs to show the general requirements for communication, plus the source and destination tasks.

**Intertask Communications**

Before diving into intertask communication layout, a quick overview of the three common types of variables is provided. These are:

1. Broadcast
2. Semaphore
3. Buffer

Broadcast variables have the simplest protocol and are typically used to transfer information between two or more tasks, when the transfer does not require acknowledgement and the timing of the reception is highly variable. The broadcast variable can range in size from bits to multi-byte variables and requires no special handling. However, two liabilities come with this protocol’s simplicity:

1. Communication reliability is not guaranteed. The receiving task may not see every change in the variable, and the sending task will have no indication of the failure.
2. Either partial updates of variables must be prevented, or the receiving tasks must be able to tolerate partially-updated data.

A good example of a broadcast variable is the variable holding the display information for the display task. The display task scans only the selected display data (Alarm or Time) out to the LED display. Its timing is driven by the timing for the display, so its timing relative to changes in the data is asynchronous. There is also no problem if a source variable is only partially updated during the scan, or even if we change the source of data during a scan because the human eye is not fast enough to see the irregularities. Therefore, using a broadcast variable for both Current Time and Alarm Time meets the usage requirements of both variables. One hundred percent reliability is not required, partial updates are tolerable and the receiving task’s timing is asynchronous to updates by the source tasks.
Note that, although the Display task can tolerate partial updates, the tasks driving the variables cannot. Only the driving tasks can be allowed to modify the variables. Fortunately, we grouped those functions into common tasks, so this is not a problem.

Examples of Broadcast variables include:

1. Time     The current time in hours and minutes
2. Alarm    The current alarm time in hours and minutes
3. AMPM/Military The flag indicating 12 or 24 hour display modes
4. AlarmOn  Flag indicating the current enabled/disabled state of the alarm function.

Semaphore protocol variables are used when an action must be synchronized between two or more tasks. This may involve transferring data or merely the coordination of an activity. An example of data transfer is the movement of command flags from the Command tasks to the Time or Alarm tasks. When the Command task determines that a valid key has been pressed, it sets a flag indicating the command. The Alarm or Time tasks then make the appropriate change in Time or Alarm variables, and acknowledge the completion of the command by clearing the command flag. Handshaking on the data transfer ensures that the command is reliably transferred once, and only once, from the Command task.

An example of a data transfer semaphore is the transfer of a character from a serial-port task to a serial-command decoder task in our clock example. When a valid character is received from the serial port, the serial-port task must transmit that character to the serial-command task. It does this by placing the data in a data variable, and then setting a flag variable to indicate that new data is available. The command task then accepts the data and clears the flag, indicating that the transfer is complete. The serial task can then send any subsequent data, secure in the knowledge that it will not overwrite the previous data before the serial-command task receives it.

A good example of a coordination Semaphore variable is a Watch Dog Timer-Error handler task for our clock. If the Watch Dog fires, we do not want to simply restart the system—we need to verify the state of the system, reset any corrupted variables and only then release the system. To do this, the error handler sets a group of semaphore flags to alert all system tasks of the problem. The individual tasks then clear their current activity before clearing their semaphore to indicate they are in their idle state. The error-handler task then verifies the system, resets any corrupted variables and sets a broadcast restart flag to tell the system tasks that they can restart.

Semaphores are a simple way of synchronizing events between tasks. However, care must be taken when semaphores crosslink tasks. As seen in Figure 3, cross linking occurs when two tasks have semaphores passing in both directions between them. The problem is that semaphore cross linking can lead to a condition called “state lock.” State lock is when two tasks are so out of synchronization that each task is waiting for the other to acknowledge different semaphores. Later, in the section on implementation, techniques for recovering from state lock will be discussed. For now, any configurations that have the potential for state lock should be examined to determine if a rearrangement of functions, tasks or protocols could alleviate the condition. If the condition cannot be avoided, it should be identified for additional counter measures during the implementation phase of the design.
The type of variable is the **Buffer** variable. These variables are generally reserved for communication between tasks, in which the rate of data handling is significantly different between the two tasks. A good example is the communication between the Serial-Command task and the Serial-Transmit task. The Serial-Transmit task is limited in the rate at which it can process data, due to its external timing requirements. The Serial-Command task, on the other hand, can handle multiple characters very quickly. What is needed is storage space to hold data from the faster task until the slower task can process it. In this example, the Command task does not usually have a problem keeping up with the data received from the serial receive task. It handles the commands one character at a time, as they are received. The problem occurs in the Command task’s response to the commands. The command decoder cannot sit idle, waiting for the Serial-Transmit task to transmit its response one character at a time. If it did, it might miss follow on commands from the Serial-Input task. Using a buffer between it and the serial output task, the Command task can queue-up responses for the next transmit and then move on to other activities.

A buffer variable is typically composed of a block of memory large enough to hold a reasonable amount of data and two data pointers. The block of memory is the queue that holds the data while it is transferred. The pointers provide indices to the last byte retrieved, as well as the last byte stored. In normal operation, the storage pointer is incremented and the new data is then stored via the pointer. If incrementing the pointer moves it past the last location in the block of memory, it is wrapped around to the start of the block, forming a circular queue. Data is retrieved from the buffer in the same way – by incrementing the output pointer, and then using it to retrieve the next piece of data. In this way, the two pointers chase each other around the memory block as data moves through the buffer.
Handshaking control in the protocol is accomplished by comparing the two pointers. If the storage pointer is equal to the retrieval pointer, the buffer is empty. If the pointers are not equal, data is present to be retrieved. If the storage pointer is one less than the retrieval pointer, the buffer is full and no additional data can be added until some data is retrieved.

**Diagramming the Data Flow**

Now that the tasks are defined and we have discussed the various variable types, the system’s communication requirements can be mapped. To make this task simpler and more intuitive, it is recommended that the data paths be drawn graphically in a data-flow diagram to aid in the visualization of the system-data flow. The first step is to draw all tasks as circles on a large piece of paper. Label each circle with its task name and function. Next, draw additional circles to represent any microcontroller ports or peripherals that will drive or be driven by the various tasks. Also, note any handshaking or configuration bits associated with the peripheral.

When all tasks and peripherals are drawn, use the lists of task I/Os from the previous section to draw the data paths between the tasks. These paths are drawn as arrows between the tasks, with the “head” pointing at the destination task and the “tail” connecting to the source task. Label each arrow with a descriptive name and a description of the data being transferred. Some task outputs may drive more than one receiving task. If so, draw a separate arrow for each path, but start the arrows from a common point to denote the common data source. Figure 4 shows the data-flow diagram for our clock example.
Next, decide which protocol is appropriate for each path. Take into account the relative speed of the tasks, as well as required reliability. Use broadcast variables when possible, especially if the variable transfers information to more than one task. If a semaphore protocol is used for a data path between three or more tasks, it will require separate variables for each destination task. It will also complicate the protocol-handling routines. Try to minimize the number of semaphore variables. This reduces the potential for state lock and minimizes the time needed for the data transfer.

When all paths have been identified, described, and have a protocol assigned to them, document the name, description, originating task, destination task and protocol of each data path in a master variable list. Figure 5 is an example of a master variable list for the clock.
<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>CurrentTime Broadcast: Current time</td>
<td>Time</td>
<td>Alarm</td>
</tr>
<tr>
<td></td>
<td>in BCD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Display</td>
<td>Display</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>CurrentAlarm Broadcast: Current alarm</td>
<td>Alarm</td>
<td>Display</td>
</tr>
<tr>
<td></td>
<td>in BCD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>SlowSetTime Semaphore: Slow Set</td>
<td>Command</td>
<td>Time</td>
</tr>
<tr>
<td></td>
<td>command</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>command</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>SlowSetAlarm Semaphore: Slow Set</td>
<td>Command</td>
<td>Alarm</td>
</tr>
<tr>
<td></td>
<td>command</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>command</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td>Snooze Semaphore: Snooze command</td>
<td>Command</td>
<td>Alarm</td>
</tr>
<tr>
<td>8.</td>
<td>SoundAlarm Broadcast: Sound Alarm</td>
<td>Alarm</td>
<td>Sound</td>
</tr>
<tr>
<td></td>
<td>buzzer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.</td>
<td>2HzTiming Broadcast: 2Hz timebase</td>
<td>Display</td>
<td>Sound</td>
</tr>
<tr>
<td></td>
<td>flag</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10.</td>
<td>AlarmSet Broadcast: Alarm set function</td>
<td>Key</td>
<td>Display</td>
</tr>
<tr>
<td></td>
<td>active</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11.</td>
<td>TimeSet Broadcast: Time set function</td>
<td>Key</td>
<td>Time</td>
</tr>
<tr>
<td></td>
<td>active</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12.</td>
<td>AlarmOnOff Broadcast: Alarm enabled/</td>
<td>Command</td>
<td>Display</td>
</tr>
<tr>
<td></td>
<td>disabled</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13.</td>
<td>AMPMMilitary Broadcast: 12/24 hour</td>
<td>Command</td>
<td>Display</td>
</tr>
<tr>
<td></td>
<td>select flag</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14.</td>
<td>DigitDrive Broadcast: LED digit drive</td>
<td>Display</td>
<td>Port</td>
</tr>
<tr>
<td></td>
<td>to port</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15.</td>
<td>SegmentDrive Broadcast: LED segment</td>
<td>Display</td>
<td>Port</td>
</tr>
<tr>
<td></td>
<td>drive to port</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16.</td>
<td>DigitDrive Broadcast: Button select</td>
<td>Key</td>
<td>Port</td>
</tr>
<tr>
<td></td>
<td>drive to port</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17.</td>
<td>PortOpen Broadcast: Port available</td>
<td>Display</td>
<td>Key</td>
</tr>
<tr>
<td></td>
<td>to button</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18.</td>
<td>KeyPress Broadcast: Button return</td>
<td>Key</td>
<td>Port</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19.</td>
<td>Buzz Broadcast: Speaker drive to port</td>
<td>Sound</td>
<td>Port</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20.</td>
<td>Key1,2,3,4,5 Broadcast: Debounced key</td>
<td>Key</td>
<td>Command</td>
</tr>
<tr>
<td></td>
<td>press</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 5:**
Example Master-Variable List for Alarm Clock

Notice the following regarding the variable list:

1. The type of variable is predominantly Broadcast, to prevent state lock conditions.
2. There is a PortOpen variable between the Display and Key tasks. This coordinates the activity between the two tasks as they share the DigitDrive lines. Ideally, a separate task should arbitrate access to the port, but here we used a simpler system and allowed the Display task to arbitrate, due to its regular timing.
3. The 2HzTiming variable between the Display and Sound tasks is designed to warble the tone during the alarm sound. A separate counter could have been used in the sound task, but a single flag was used because it was smaller.
4. There are no Buffer variables used because there are no multibyte transfers. If we add a serial port for setting and reading values, then a Buffer will be needed between the Serial-Command Task and the Serial Transmit, as described above.
Timing Analysis

The next step in the system-level design is to analyze the each task’s timing requirements. The purpose of the timing analysis is to determine the base rate at which the overall loop must call the individual state machines to ensure fast enough response times to accomplish their functions. The goal is to determine the system’s timing Tick and the skip rate for each of the tasks. We start by establishing the desired, minimum and maximum polling rates for all the system tasks. Once we have these, we can determine a Tick. The Tick should be fast enough for the fastest task or tasks. The skip rates are then the number of ticks that must pass for each call of the slower tasks.

For example, look at the task list from the task-definition section. Note that timing and timing accuracy are now included.

- **Time** Real-Time clock function, 1Hz rate. Accuracy of increment event +/-10%
  Accuracy of increment rate +/-0%
- **Alarm** 1Hz rate, same accuracy as Time task.
- **Display** Seven Segment, 7-Digit LED Display, Scanned by Software
  To prevent flicker, 60 Scans/Sec Min; Scan Rate = 7 digits * 60 or 420 Calls/Sec. Min. Note digit 7 is scan of keyboard
  Accuracy of +10%/-0%
  2Hz modulation during alarm time, +/-10%
- **Sound** 1-2 kHz rate, 1.5kHz +/-33%. 2Hz modulation +/-10%
- **Key** 5 buttons, 3 press/second maximum, 8 scans to debounce. Assume all 5 buttons scanned each pass of display: so 360 / 8 = 45 key/sec (Faster than 3 key/sec)
  Timing driven by display meets timing requirements
- **Command** Auto repeat function operates at a 20 Hz rate +/-10%
  Delay to auto repeat is 3 seconds +/-10%
  Also must respond to button press at maximum rate of 3 Hz +10%/-0%

Table 1 shows all of this organized as into a chart, with maximum and minimum timing.

<table>
<thead>
<tr>
<th>Task</th>
<th>Optimum Timing</th>
<th>Minimum Timing</th>
<th>Maximum Timing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time Task</td>
<td>1 S</td>
<td>1 S</td>
<td>1 S</td>
</tr>
<tr>
<td>Alarm Task</td>
<td>1 S</td>
<td>1 S</td>
<td>1 S</td>
</tr>
<tr>
<td>Display Task</td>
<td>2.38 mS scan</td>
<td>2.14 mS</td>
<td>2.38 mS</td>
</tr>
<tr>
<td>Display Task</td>
<td>250 mS flash</td>
<td>225</td>
<td>275</td>
</tr>
<tr>
<td>Sound Task</td>
<td>333 µS</td>
<td>250 µS</td>
<td>500 µS</td>
</tr>
<tr>
<td>Sound Task</td>
<td>500 mS modulation</td>
<td>450 mS modulation</td>
<td>450 mS modulation</td>
</tr>
<tr>
<td>Key Task</td>
<td>Tied to Display</td>
<td>Tied to Display</td>
<td>Tied to Display</td>
</tr>
<tr>
<td>Command Task</td>
<td>50 mS</td>
<td>45 mS</td>
<td>55 mS</td>
</tr>
<tr>
<td>Command Task</td>
<td>3S</td>
<td>2.7S</td>
<td>3.3S</td>
</tr>
</tbody>
</table>

**Table 1:** Example Timing Analysis
The Tick for each task can now be determined. Typically, the smallest value will be the best choice for the system. However, a smaller Tick may be needed if any of the tasks are not related to the minimum tick by an integer multiple. For example, a task requiring a time of 2 mS and one requiring a time of 3 mS are not related by an integer. However, a 1 mS Tick is related to both by an integer value and is therefore a good candidate for the minimum common Tick. The rates must be related by an integer. However, remember that the timing accuracy is also listed. This gives some variation in the task timing, which can make an integer relationship possible.

From the list above, we know that the minimum Tick is 250 µS. All other tasks are either integer multiples or have sufficient tolerance in their timing to fit an integer multiple of this tick.

Given the time Tick, skip rates can be calculated for each of the other tasks.

- **Time**: 1000 / .25  Skip rate is 4000:1
- **Alarm**: 1000 / .25  Skip rate is 4000:1
- **Display**: 2.25 / .25  Skip rate is 9:1
- **Sound**: .25/.25  Skip rate is 1:1
- **Key** tied to display
- **Command** 50/.25  Skip rate is 400:1
  3000/.25  Skip rate is 12000:1

From this example, the Time and Alarm tasks will be called once every 4,000 passes through the main loop. The display task will be called every 9 passes through the main loop, and the 2 Hz flash flag will be toggled every 1,000 passes. It is important to note that 1,000 is not divisible by 9, so the rate of 1000:1 is not achievable. However, on the next multiple of 9, the flag can be acted upon. Even though it is not an even integer, the rate of 1000:1 can be generated, it will just have a jitter of +/- 5 Ticks.

One other point to note is that not all of the timing accuracy numbers were present in the description in appendix A. One of the important features of a top-down design approach is that it not only organizes the project data for the actual writing, but also highlights any missing information that can be found and documented before it is needed.

One final timing-related issue is to determine if there is a need for synchronized I/O. This is driven primarily by the application’s timing requirements. If all changes to the system I/O must be synchronized, the individual tasks should communicate their changes to the peripherals via an interim variable. These variables can then be transferred as a group to the peripherals at the end of the loop. If there is no need to synchronize the system’s I/O, then the tasks may control the peripherals directly. The only requirement at this point is to note the specific condition in the timing documentation.
Setting Priorities

The final system-design step is to analyze the system priorities in order to identify the tasks and states with tight timing requirements, as well as the priorities required for the various system conditions. Also, identify tasks that may be mutually exclusive at some time during their execution. Variability in a task’s execution time or its priority should also be identified for later use. Remember, the more flexibility that can be found in the system, the more options will be available when writing the priority-handler. Carefully document the specific requirements of each task and any flexibility in its timing. For this example, we should note the following:

- The Time task has the highest priority at all times. The rate must never change, however, the exact time of the increment can be variable so, even though it has a high priority, it can be deferred from one tick to the next provided that rate of increment is not affected.

- The sound task must not be deferred when it is generating a tone. Even a 1-2% shift is discernable, so it should have a high priority when active.

- The Alarm task is only a priority if it is enabled and not active. If enabled and active, the alarm has already passed and its priority will drop.

- The Key and Command tasks are relatively low-priority tasks and can be deferred without concern, because the rate of polling is higher than required. Note: the ability to call the Key task is driven by the Display task, so deferring the Key task means that it must wait for the next call by the Display task to prevent interference with the Display-task timing.

Component-Level Design

Now that the system-level design is complete, it is time to design the components specified at the system level. State-machine designs are needed for each system task. Variable definitions and handling routines are needed for task-to-task communications. The system and skip-timer routines must be designed and a priority-handler must be designed.

State-Machine Design

Component-level state machine design is the process of determining which type of state machine to use and laying out the number of states required, in addition to their functions, inputs, outputs, and the conditions that cause a change from one state to another.

To determine which type of state machine to use, ask the following simple questions:

1. Does the state machine do something different each time it is called?  
   If the answer is yes, then use an execution-indexed state machine.

2. Does the state machine do the same thing each time it is called, just on different data?  
   If the answer is yes, then use a data-indexed state machine.

3. Does the state machine do something different most of the time it is called, while repeating the same function on different data some of the time?  
   If the answer is yes, then use a hybrid state machine

To understand each type of state machine design, let us start with a task state machine that does something different each time it is called. Note that the task may require that the state machine occasionally do the same thing repeatedly. As long as the data operated upon does not change, it is still considered an execution-indexed state machine.
The first step of the design process for an execution-indexed state machine is to make a preliminary list of states, with a description of the action to be taken during the state.

To start the state-machine design, create a preliminary list of states. For example, the state machine for the Command task would look like the following:

<table>
<thead>
<tr>
<th>State</th>
<th>Function performed in state</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Idle no function, waiting for a button press</td>
</tr>
<tr>
<td>2.</td>
<td>AlarmSet reset autorepeat timers for alarm set functions</td>
</tr>
<tr>
<td>3.</td>
<td>TimeSet reset autorepeat timers for time set functions</td>
</tr>
<tr>
<td>4.</td>
<td>AlarmFast set FastSetAlarm semaphore and wait for confirmation</td>
</tr>
<tr>
<td>5.</td>
<td>AlarmSlow set SlowSetAlarm semaphore and wait for confirmation</td>
</tr>
<tr>
<td>6.</td>
<td>TimeFast set FastSetTime semaphore and wait for confirmation</td>
</tr>
<tr>
<td>7.</td>
<td>TimeSlow set SlowSetTime semaphore and wait for confirmation</td>
</tr>
<tr>
<td>8.</td>
<td>Toggle1224 Toggle12 / 24 hour flag</td>
</tr>
<tr>
<td>9.</td>
<td>Repeatdelay Wait initial 3 second delay</td>
</tr>
</tbody>
</table>

When the list is complete, ask an additional few questions about the design:

- Does the state machine require an error state?
- Does the state machine have a default state?
- Do all of the states come from another state?
- Do all the states go to another state?

In this example, the state machine is monitoring key-press inputs and generating semaphore command flags as an output. If an undefined combination of key-press inputs is received, an error condition can occur, so an error state is therefore reasonable. Additionally, there is always the possibility that the state variable may become corrupted, so a default state is prudent. All other states have a clear prior and next state, so the questions have been answered.

The next step is to define the conditions necessary for the transition from state to state, as shown in Figure 6.
### Figure 6:

#### Conditions Necessary for the Transition from State-to-State in Alarm-Clock Example

The next step is to make a list of inputs required for the state machine. For this design, the inputs include key press flags from the Key task. When the input variable list is complete, compare it to the data-flow diagram generated in the previous step to confirm that all input-data paths have been accounted for. If not, make the appropriate additions to the data-flow diagram and the master variable list.

After the input variables are accounted for, define the outputs of the state machine. Additionally, note the states that affect the output variables. In the example state machine, these include the command semaphores and three broadcast flags (12/24 hour display, Alarm On/Off and Time/Alarm). Listing the states that control these variables yields the list shown in Figure 7.
### Figure 7:

List of States the Control Alarm-Clock Input Variables

<table>
<thead>
<tr>
<th>State Name</th>
<th>Outputs from the state</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Idle</td>
<td>no output</td>
</tr>
<tr>
<td>2. AlarmSet</td>
<td>Time/Alarm Clear</td>
</tr>
<tr>
<td>3. TimeSet</td>
<td>Time/Alarm Set</td>
</tr>
<tr>
<td>4. AlarmFast</td>
<td>FastSetAlarm</td>
</tr>
<tr>
<td>5. AlarmSlow</td>
<td>SlowSetAlarm</td>
</tr>
<tr>
<td>6. TimeFast</td>
<td>FastSetTime</td>
</tr>
<tr>
<td>7. TimeSlow</td>
<td>SlowSetTime</td>
</tr>
<tr>
<td>8. Toggle1224</td>
<td>AMPM Military</td>
</tr>
<tr>
<td>9. Repeatdelay</td>
<td>no output</td>
</tr>
<tr>
<td>10. Error</td>
<td>no output</td>
</tr>
<tr>
<td>11. Default</td>
<td>no output</td>
</tr>
</tbody>
</table>

Note that the Time/Alarm flag is set when setting the alarm, but cleared at any other time. The only time we set the alarm value is when the Alarm Set button is pressed, so why do we need to generate an additional flag? All we really need is to pass the Alarm Set button flag to the Display task and looking at the data-flow diagram, we see that Alarm Set already goes there. We do not need to create a new flag and we can delete it from the design. All we need to do is make the appropriate changes to the data-flow diagram and the notes for this state machine.

Note the actions in the error and default states. The default state is called when the state variable does not decode to a valid state. The error state is called whenever an invalid button combination is pressed. In either state, the state machine treats the error as a syntax or soft error and returns to the Idle state. To the user, the bad combination will appear to be ignored and the interface continues as before, as if the error did not occur. However, if the error had been a hard error, the Default and Error states are the location for error-correction routines, which would then clear the error, coordinate its recovery with the other state machines in the system and return to normal operation when the fault is cleared.

Finally, document the actions taken in each state of the state machine in a compact shorthand for inclusion in the code listing as comments, as shown in Figure 8. Pseudo-code for any specific steps or algorithms should also be included, as should the state-change conditional statements and any special I/O functions.
IdleState

If (Alarm set button = closed) State = AlarmSetState
If (Time set button = closed) State = TimeSetState
If (Snooze button = closed) State = SnoozeStage
If ((Alarm set button & Time set Button) = closed) State = ErrorState

AlarmSetState

Delaycount = 12000 / 400 // skip rate for auto repeat delay / skip rate for command task
If (Alarm set button = open) State = IdleState
If (Fast set button = closed) State = AlarmFastState
If (Slow set button = closed) State = AlarmSlowState
If (Snooze button = closed) State = ErrorState
If ((Fast set button & Slow set button) = closed) State = ErrorState

TimeSetState

Delaycount = 12000 / 400 // skip rate for auto repeat delay / skip rate for command task
If (Time set button = open) State = IdleState
If (Fast set button = closed) State = TimeFastState
If (Slow set button = closed) State = TimeSlowState
If (Snooze button = closed) State = Toggle1224State
If ((Fast set button & Slow set button) = closed) State = ErrorState

AlarmFastState

If (FastSetAlarm = false) // wait for previous command semaphore to clear
    FastSetAlarm = true
    State = RepeatdelayState

AlarmSlowState

If (SlowSetAlarm = false) // wait for previous command semaphore to clear
    SlowSetAlarm = true
    State = RepeatdelayState

TimeFastState

If (FastSetTime = false) // wait for previous command semaphore to clear
    FastSetTime = true
    State = RepeatdelayState

TimeSlowState

If (SlowSetTime = false) // wait for previous command semaphore to clear
    SlowSetTime = true
    State = RepeatdelayState

SnoozeState

If ((Snooze button = open)&(Snooze = false) // wait for previous command to clear
    State = IdleState
    Snooze = true

Toggle1224State

If (Snooze button = open)
    State = IdleState
    AMPMMilitary = ! AMPMMilitary

RepeatdelayState

If (delaycount > 0) delaycount--
If (delaycount = 0)&(Alarm set button closed)&(Fast set button closed)
    State = AlarmFastState
If (delaycount = 0)&(Alarm set button closed)&(Slow set button closed)
    State = AlarmSlowState
If (delaycount = 0)&(Time set button closed)&(Fast set button closed)
    State = TimeFastState
If (delaycount = 0)&(Time set button closed)&(Slow set button closed)
    State = TimeSlowState

ErrorState

If ((Fast set button & Slow set button & Snooze button = open) &
    (Time set button or Alarm set button = open)) State = IdleState

DefaultState

State = IdleState

---

**Figure 8:**

List of Actions Taken in Each State of the State Machine
This completes the component-level design of the task state machine. From this point, the design is really just implementing the actual code. All the information required and the state machine’s design flow have been determined and documented. In fact, the documentation that was created during the design process makes a very good header for the actual code in that it defines exactly how the state machine works and the system-interface requirements.

Using our clock example, the Display task would be a good data-indexed state machine. This task performs the same function each time it is called—converts the current digit data into a 7-segment format, outputs the data to the port, and then selects the appropriate digit-enable output to turn on. The other functions are grouped into the task. Simply select which block of digit data is converted, whether or not the display is blanked, and which individual enunciators are lit.

To begin, let’s start by reviewing the functions in the Display Task:

- Display Task
  - Display Time in AM/PM
  - Display Time in Military
  - Display Alarm in AM/PM
  - Display Alarm in Military
  - Flash Time Display at 2 Hz rate
  - Display Alarm ON/OFF

Don’t forget that the task must also arbitrate access to the port so the Key task can use the digit-select lines to scan the system buttons.

Looking over the functions, we can group them into four main areas:

1. One determines the source of the data to be displayed (Alarm or Time)
2. The second determines the format of the display (AMPM or military)
3. The third group are blanking functions (Flash and the blank 7th digit for the Key function)
4. The fourth just drives the alarm on/off enunciator.

The source-control function is simply a data multiplexer. We will handle it with a simple conditional when accessing the data. The format functions will drive how the data is decoded into seven-segment data. The flash function will require additional logic to control the rate of the flash, and an additional state to create one blank time for the Key task.

As with the execution-indexed state machine, start by defining states for the state machine, as shown in Figure 9:

<table>
<thead>
<tr>
<th>State</th>
<th>Name</th>
<th>Function performed in state</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Seconds</td>
<td>display the ‘ones’ seconds digit</td>
</tr>
<tr>
<td>2.</td>
<td>TenSec</td>
<td>display the ‘tens’ seconds digit</td>
</tr>
<tr>
<td>3.</td>
<td>Minutes</td>
<td>display the ‘ones’ minutes digit</td>
</tr>
<tr>
<td>4.</td>
<td>TenMin</td>
<td>display the ‘tens’ minutes digit</td>
</tr>
<tr>
<td>5.</td>
<td>Hours</td>
<td>display the ‘ones’ hours digit</td>
</tr>
<tr>
<td>6.</td>
<td>TenHrs</td>
<td>display the ‘tens’ hours digit</td>
</tr>
<tr>
<td>7.</td>
<td>Blank</td>
<td>Blank to allow Key task to scan buttons.</td>
</tr>
</tbody>
</table>

**Figure 9:**
Example List of Defined State-Machine States
The next step is to determine the state transitions. However, in this design, state change is not conditional—the state just increments from Seconds to Blank, and then rolls over to Seconds. If an error occurs, or the state variable is out of range, we can simply reset the state to the Blank state to reset the state variable. If the error condition persists, the state machine will become locked in the blank state. This can be the indication to the user that the system needs to be power cycled to clear an error.

Figure 10 shows that the I/O list for the state machine is also relatively simple:

<table>
<thead>
<tr>
<th>State</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-6</td>
<td>CurrentTime</td>
<td>DigitDrive</td>
</tr>
<tr>
<td></td>
<td>CurrentAlarm</td>
<td>SegmentDrive</td>
</tr>
<tr>
<td></td>
<td>AlarmSet</td>
<td>PortOpen</td>
</tr>
<tr>
<td></td>
<td>AMPMMilitary</td>
<td>2HzTiming</td>
</tr>
<tr>
<td></td>
<td>AlarmOnOff</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>PortOpen</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 10:**
I/O List for State Machine

Finally, just as in the execution-indexed design, document the function of the state machine in a compact short hand for inclusion in the code listing as comments. Figure 11 shows an example list. Pseudo-code for any specific steps or algorithms should also be included, as should the state change conditional statements and any special I/O functions.

DigitDrive = off
SegmentDrive = off
State++
2HzTiming = Flash
If (state < 7)
   PortOpen = False
   If (Flash != True)
      If (AlarmSet = true)
         Temp = CurrentAlarm[State]
      Else
         Temp = CurrentTime[State]
   Else
      Temp = CurrentTime[State]
   SegmentDrive = BCD2SegConvertArray[AMPMMilitary][Temp]
   DigitDrive = DigitSelectConvertArray[State]
Else
   PortOpen = True

**Figure 11:**
Example Documentation of State-Machine Function
The basic design of the data-indexed state machine is complete. It is a relatively simple routine, which performs the same function each time it is called with only the data acted upon changing. Notice that a new communication variable was introduced into the design—the Flash variable. The reason this variable has not yet been defined is because we have yet to define all the timing variables that will be part of the timing-control section of the design. However, we needed it here to show how the 2Hz flag is generated, so we are foreshadowing the process slightly. Also note that the 2HzTiming variable is just a copy of Flash. We could use the Flash variable to drive the 2Hz Timing function and save a bit, but by leaving the 2HzTiming flag in the design, we open up the capability to disconnect the flash rate from the tone modulation at a latter point in the design. This only costs a bit, so the expense is not significant.

The third form of state machine, the Hybrid, is simply an execution-indexed design with one or more states that are data-indexed state machines. As the design of both the Execution- and Data-indexed versions have been covered in depth, there is no reason to repeat the information for the hybrid. However, one point should be made concerning the Hybrid form—it is best to use two separate state variables in a hybrid design. While a single variable may save data storage, the cost in parsing the variable negates any incremental savings in the design. Using separate state variables also makes the design simpler to read, understand and troubleshoot.

**Variable Design**

Using the documentation from the system-level design, variable definition is a straightforward process. Simply define a variable for each of the data paths and add any flags or handling routines as specified by the type of variable. The variables should be grouped together based upon data path and function. Related variables, such as semaphores with their data and buffers with their pointers, should be given a common naming prefix to aid in their identification.

The variable definitions should then be gathered together in a header or “.H” file. Large and complex designs may require more than one header file. If so, group variables by task and use descriptive file names. The header file should also include documentation on each data path’s protocol and a list of all routines used to implement any handshaking used with the protocol. Placing this information in a single location is very helpful not only in the design process, but also for anyone who may come along behind you to modify or maintain the software.

Using subroutines to implement protocol handshaking has several advantages and should be considered for all designs, even if the protocol subroutines may only be used in one location. Reasons for this are as follows:

1. Subroutines allow the variable-handling functions to be developed and tested separately.
2. Variable structures and protocol subroutines are modular and can be reused.
3. Temporary variables with simpler protocols can be substituted during testing of the state machines by simply changing the name of an include file.

When all variables have been declared, create an include file with prototypes for all protocol subroutines. The actual code will be generated at the next level. However, a template file is convenient for generating the final subroutines and any intermediate testing subroutines that may be required during testing. Figure 12 shows the .H and .inc files for the alarm clock example.
VARIABLE HEADER FILE:

// Time related communications variables
INT CurrentTime[6];  // Array of 6 BCD values [hrs:min:sec]  Broadcast
BIT AMPMMilirary;  // 12-24 hour flag 0=12 1=24   Broadcast
BIT SlowSetTime;  // Time slow set cmd, 1 = inc time 1 min Semaphore
BIT FastSetTime;  // Time fast set cmd, 1 = inc time 20 min Semaphore

// Alarm related communications variables
INT CurrentAlarm[6]; // Array of 6 BCD values [hrs:min:00]  Broadcast
BIT AlarmOnOff;  // alarm enable bit 1 = on    Broadcast
BIT AlarmSet;  // display time versus alarm, 1 = alarm  Broadcast
BIT SlowSetAlarm;  // Alarm slow set, 1 = inc alarm by 1 min Semaphore
BIT FastSetAlarm;  // Alarm fast set, 1 = inc alarm by 20min  Semaphore
BIT Snooze;  // Alarm snooze, 1 = snooze command Semaphore
BIT SoundAlarm;  // Alarm tone enable, 1 = generate tone  Broadcast

// key Task communications variables
BIT Key1;  // key1 pressed Time set function Broadcast
BIT Key2;  // Key2 pressed Alarm set function Broadcast
BIT Key3;  // Key3 pressed Fast set function Broadcast
BIT Key4;  // Key4 pressed Slow set function Broadcast
BIT Key5;  // Key5 pressed Alarm snooze Broadcast

// port communications variables
BIT Port_open;  // 1 = port avail, enables Key Task Broadcast
CHAR DigitDrive;  // 6 port bits tied to digit drivers Broadcast
CHAR SegmentDrive;  // 7 port bits tied to segment drivers Broadcast
BIT KeyPress;  // port bit tied to button common Broadcast
BIT Buzz;    // port bit tied to speaker Broadcast

// Timing variables
BIT Flash;   // Disp blanking, 1 = blank Broadcast
BIT 2HzTiming   // modulation for alarm tone Broadcast

// example state variable definitions
ENUM CommandState(Idle, AlarmSet, TimeSet, AlarmFast, AlarmSlow, TimeFast,
                   TimeSlow, Toggle1223, Repeatdelay, Error, Default);
ENUM DisplayState(Seconds, TenSec, Minutes, TenMin, Hours, TenHrs, Blank);

// example variables for a Buffer variable type
INT OutPoint_SerialTX  // retrieval pointer for Serial Transmit buffer
INT InPoint_SerialTX  // storage pointer for Serial Transmit buffer
CHAR Buffer_SerialTX[10]  // storage ram for Serial Transmit buffer

VARIABLE INCLUDE FILE

// Convenient definition for alarm = time compare
#define Time_eq_alarm  (CurrentTime[TenHrs]== CurrentAlarm[TenHrs])&
                     CurrentTime[Hours]== CurrentAlarm[Hours])&
                     CurrentTime[TenMin]== CurrentAlarm[TenMin])&
                     CurrentTime[Minutes]== CurrentAlarm[Minutes])
#define TimeSetButton  Key1
#define AlarmSetButton Key2
#define FastSetButton  Key3
#define SlowSetButton  Key4
#define SnoozeButton  Key5
#define closed   true
#define open   false

// Note: example of a buffer variable function prototypes
VOID StoreInBuffer(CHAR Datain);
CHAR RetrieveFromBuffer();
BIT TestEmptyBuffer();
BIT TestFullBuffer();

Figure 12:
Example Code Listing for Alarm Clock

State Lock

An integral part of the variable design must be a strategy for handling variables that have the potential to create a state-lock condition. The best strategy for handling state lock is to prevent it from happening in the first place. If possible, simply avoid crosslinking two or more state machines with semaphores. If cross-linked controls are needed, consider changing one of the semaphores to either a broadcast or a buffer protocol.
If cross-linked semaphores cannot be avoided, try adding qualifications to the semaphore protocol that disables or defers the transfer, if the receiving state machine is not in the expected range of states. If synchronization between two or more functions is required, consider creating an intermediate task that combines the functions in a single state machine.

```cpp
If (ReceiveState == Idle) {
    SyncSemaphore = true;
    SendState = NextState;
} else

If this communication is not critical, consider a recovery method that interrupts the system when state lock occurs. One method is to add a time-out timer to the protocol. If the receiving state machine does not acknowledge in a specified time, declare an error and reset. Another method is to create a watchdog timer to monitor the number of calls to the state machine without a state change and declare an error after a reasonable number.

```cpp
If (SyncSemaphore = true) {
    SemaphoreWDT--;  
    If (SemaphoreWDT == 0) {
        SemaphoreWDT = DefaultValue;  
        SystemError = true;  
        SendState = LockError; 
    }
} else SyncSemaphore = true;
```

The only method that will not work is wishing that the problem would not occur. The precautions/recovery systems for handling state lock are similar to the default state in a state machine. Designers hope they are not needed, but when they are, there is no substitute. Whatever method is selected, remember to make the appropriate updates to the state machine design and the intertask-variable list.

**Timing Control**

In the timing-analysis section of this paper, the timing requirements and skip rates for each of the task state machines was defined. At this level, the designer should decide if the Timing functions will be included in the state machines, themselves, or as separate timing functions.

If the skip timer is to be part of the state machine, the skip-timer logic must be added to the start of each task state machine and the appropriate timer variables should be defined. Figure 13 shows an example of this.
Void Time_Task(){
    if ((Time_skip_timer--)==0){ // check Time Task
        Time_skip_timer=1000;
        //Statemachine code
    }
}

Void Alarm_Task(){
    if ((Alarm_skip_timer--)==0){ // check Alarm Task
        Alarm_skip_timer=1000;
        //Statemachine code
    }
}

Void Display_Task(){
    if ((Display_skip_timer--)==0){ // check Display Task
        Display_skip_timer=9;
        //Statemachine code
    }
}

Void Command_Task(){
    if ((Command_skip_timer--)==0){ // check the Command Flag
        Command_flag=true;
        Command_skip_timer=400;
        //Statemachine code
    }
}

**Figure 13:**
Example of a State-Machine Timer System

If the timer is to be an external routine, then a timing handler with the appropriate timer variable and timeout flags should be defined, as shown in Figure 14.
TimerRoutine(){
    if (TMR0_INT_FLAG == 1) Error_state; // if set at incoming, error
    
    if ((Time_skip_timer--)==0){ // check Time Task
        Time_flag=true;
        Time_skip_timer=1000;
    }
    if ((Alarm_skip_timer--)==0){  // check Alarm Task
        Alarm_flag=true;
        Alarm_skip_timer=1000;
    }
    if ((Display_skip_timer--)==0){  // check Display Task
        Display_flag=true;
        Display_skip_timer=9;
    }
    if ((Flash_skip_timer--)==0){   // check the Flash Flag
        Flash = !Flash & SoundAlarm; // if alarm sounding, toggle Flash
        Flash_skip_timer=500;
    }
    if ((Command_skip_timer--)==0){ // check the Command Flag
        Command_flag=true;
        Command_skip_timer=400;
    }
    else{
        while (TMR0_INT_FLAG == 0); // wait for the roll over
        TMR0_INT_FLAG = 0;
    }
}

Void Time_Task(){
    If (Time_Flag==true) {
        Time_Flag = false;
        //Statemachine code
    }
}

Void Alarm_Task(){
    If (Alarm_Flag==true) {
        Alarm_Flag = false;
        //Statemachine code
    }
}

Void Display_Task(){
    If (Display_Flag==true) {
        Display_Flag = false;
        //Statemachine code
    }
}

Void Command_Task(){
    If (Command_Flag==true) {
        Command_Flag = false;
        //Statemachine code
    }
}

Figure 14: Example of a Centralized Timer System

In either case, update all documentation to include the new flags and timer variables.
Documentation for the variables should also include the state machine associated with the timer
and its expected range values.
Priority-Control System

Using the priority information from the system level, an algorithm and design for the priority handler can now be chosen. The handler can either be a central system, or included as part of the state machine’s design. The specific implementation is somewhat dependant upon the type of priority system used and the designer’s coding style.

The following is a collection of some of the simpler priority-handling systems. Their descriptions include a short explanation of how they can be applied to some of the previous examples. These systems are not mutually exclusive -- they can be combined to create more complex priority-control systems, or be used standalone for simpler systems.

Passive Priority-Handling

The idea behind a passive priority-handling system is to stagger the starting values of a select group of state-machine skip timers. To be a candidate for this system, the state machines should have skip rates that are related by an integer ratio. Offsetting the starting values then prevents the timers from timing out on the same pass through the main loop and minimizes the number of state machines that are likely to execute a state on any pass through the loop.

Let’s examine the skip-timer values for the Time, Alarm and Command tasks from our Alarm clock example:

<table>
<thead>
<tr>
<th>Task</th>
<th>Skip timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>4000:1</td>
</tr>
<tr>
<td>Alarm</td>
<td>4000:1</td>
</tr>
<tr>
<td>Command</td>
<td>400:1</td>
</tr>
</tbody>
</table>

The Skip-timer values have a common factor of 400, so if we offset the initial skip-timer values by a differing amount less than 400, they will not timeout on the same pass through the loop.

<table>
<thead>
<tr>
<th>Task</th>
<th>Initial Skip timer value</th>
<th>Reload value on timeout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>4000</td>
<td>4000</td>
</tr>
<tr>
<td>Alarm</td>
<td>4001</td>
<td>4000</td>
</tr>
<tr>
<td>Command</td>
<td>402</td>
<td>400</td>
</tr>
</tbody>
</table>

The Alarm task will now timeout on the pass immediately following the Time task, which should work well as the alarm task needs to check the time after the Time task increments each second. The Command task still times out at the correct rate—it is just set to time out after the other tasks on the rare occasions when the Time, Alarm and Command time outs coincide.

Time Remaining

The time-remaining priority system tries to call the optimum combination of state machines to fill the time in each Tick. Prior to calling a specific task state machine, the priority-handler task determines the time remaining in the Tick and the time required to execute the state machine’s current state. If sufficient time remains, the state machine is called. If not, the state machine’s execution is deferred to the next pass through the loop.
To implement this algorithm, the system needs two things:

1. An autonomous hardware-based timer to monitor the time remaining
2. Accurate tables of the execution times for all states in all state machines

The hardware timer is configured to free-run at the microcontroller’s instruction rate, and its rollover period should be equal to and synchronized to the system Tick. Its value at any given time will then be equal to the number of instruction cycles remaining in the Tick. A priority handler which includes a routine to compare the timer count against the execution time for each state machine’s current state is then written, to determine if enough time remains to execute the state. If sufficient time does not remain, the state machine is deferred, allowing the next state machine in the loop to try. If sufficient time does remain, the state machine is called. If no state machine can execute in the time remaining, the Tick runs out at the bottom of the loop, waiting for the end of the Tick. Figure 15 shows an example of a time-remaining system.

```c
Void PriorityHandler(){
    If (Time_Flag==true) {
        if (HWTimer_value > TimeTask_Time2execute[Time_state_variable]) {
            Time_Flag = false;
            Time_Task();
        }
    }
    If (Alarm_Flag==true) {
        if (HWTimer_value > AlarmTask_Time2execute[Alarm_state_variable]) {
            Alarm_Flag = false;
            Alarm_Task();
        }
    }
    If (Display_Flag==true) {
        if (HWTimer_value > DisplayTask_Time2execute) {
            Display_Flag = false;
            Display_Task();
        }
    }
    If (Command_Flag==true) {
        if (HWTimer_value > CommandTask_Time2execute[Command_state_variable]) {
            Command_Flag = false;
            Command_Task();
        }
    }
}
```

**Figure 15:**
Example of a Time-Remaining System

While the time-remaining system is useful for efficient use of execution time, it can also be very cumbersome to write and maintain due to the requirement for accurate tables of execution time. The time-remaining algorithm can also make the system-response timing erratic and can introduce beat-frequency behavior in the system timing. Note that the display task does not need an array of execution times, as it is a Data-indexed state machine. Remember that data-indexed state machines perform the same functions each time they are called. It is only the data that is indexed, so all that is needed is a worst-case execution time.
Variable Order

The variable-order prioritizing algorithm is designed to respond to the changing needs of the system and concentrate resources where needed. To accomplish this, each state machine maintains a variable that indicates the relative importance of its current activity. At the start of each pass, these variables are combined mathematically to create an overall state-of-the-system value. The state-of-the-system value is then used as an index into a master SWITCH statement. Each CASE of the master switch statement contains a calling list of the system’s state machines, prioritized in a different order and optimized for the specific system condition. This allows the system to call the state machines in the order best suited to specific system states, as well as disable those state machines that can safely be ignored.

To create a variable-order system, the designer must have a list of all possible system states with the resources required for each state. A good example of the variable-order system is a system with two main states—LINE and LOCAL. In the LINE system state, the display, sound and keyboard tasks can be disabled or ignored, allowing priority execution to be shifted to the more important serial and serial command-decoding tasks. Conversely, in the LOCAL state of the system, the serial tasks are ignored, allowing the system to concentrate on the keyboard and display-related tasks of the user interface.

The design work required for the variable-order system can be a little more complicated to build and test. However, the variable-order system is quite powerful for its minimal cost in execution time. Designers should be careful with the design of variable-order systems. Completely disabling a user-interface state machine can create a condition where it is possible to change into a system state but not return from it, effectively killing the user interface. Figure 16 is an example variable-order system.

```c
switch(system_state)
{
    case line:    serial_in_task();
                 serial_command_task();
                 control_task();
                 serial_out_task();
                 break;
    case standby: keyboard_task();
                  button_command_task();
                  break;
    case local:   keyboard_task();
                  button_command_task();
                  control_task();
                  display_task();
                  break;
}
```

Figure 16:
Example of a Variable-Order System

“Excuse Me”/“Excuse You” Priority System

A variation of the variable-order system is the “Excuse Me”/”Excuse You” system. In this system, tasks decide whether to defer execution based on the state of other tasks in the system. Whether the decision to defer occurs in the deferred task or in the task being deferred to determines if the system is “Excuse Me” or “Excuse You.”
In an “Excuse Me” implementation, individual states of a task are written to voluntarily defer their execution or state change based upon the state of a higher-priority task. An example is the decision by the serial-transmit state machine to delay its transmission of data if the Serial-receive task is not in its idle states. This takes advantage of the transmit task’s more flexible start timing to “Excuse Me” out of the loop for the short period while the receive task is busy. Figure 17 shows an example of the Excuse-Me logic in a serial-transmit task.

```c
void SerialTransmit()
{
    switch (SerialTxState)
    {
        case Idle:
            RS485Driver = false;
            if ((DataAvailable == true) & (SerialRxState == Idle)) {
                ResetCRC();
                SerialTxState = EnableDriver;
            } else SerialTxState = SendCRC;
            break;
        case EnableDriver:
            RS485Driver = true;
            if ((TestEmptyBuffer() == false) & (TXRegEmpty == true)) {
                Temp = RetrieveFromBuffer();
                CalcCRC(Temp);
                TXREG = Temp;
            } else SerialTxState = SendCRC;
            break;
    }
}
```

Figure 17:
Example of Excuse-Me Logic in a Serial Transmit Task

In an “Excuse You” implementation, a broadcast flag driven by a high-priority task forces lower-priority tasks to defer all execution until the higher-priority state machine completes a specific function. An example is an Error-recovery task using an “Excuse You” flag to force all of the state machines within a system to defer execution indefinitely, until the error-recovery task has reset a corrupted variable and verified the status of all of the task’s state machines.

The difference between these two variants may seem trivial. However, the important concepts to note are an “Excuse Me” decision is made by the task deferring its own execution or state change based upon the state of one or more other state machines. An “Excuse You” decision to defer execution of a task, on the other hand, is made by the task being deferred to and may not be based solely on state.

**Parent/Child Priority System**

In the parent/child priority system, the calling of low-priority tasks is conditional to an enable flag controlled by one or more parent tasks. Typically, the child is a subroutine task that is infrequently used, and only in response to a higher-priority task’s need. Using the enable flag, high priority tasks that need the services of the state machine set the flag for as long as the state machine is needed, and then clear it to recover the execution time. Often, the child state machine may clear its own enable flag when it has completed its task, as a semaphore to the parent task that has completed its function.
A good example of this type of system is to configure the sound task as a child task. When a tone is needed, the alarm task simply sets the state variable to the starting state of the tone state machine, and then sets the enable flag. While the tone task is running, the command-decoder task can minimize its execution to free up execution cycles for the child task. Then, the alarm is disabled and the alarm task state machine clears the child task’s enable flag, in order to disable the task. Figure 18 shows an example of parent-child logic for Parent-Child Logic for the alarm and tone tasks.

```
// Alarm Task
case AlarmTest: if (Time_eq_alarm) {
    SoundState = StartState;
    SoundEnable = true;
}
If (AlarmOnOff == false) {
    SoundEnable = false;
    AlarmState = Disabled
} break;

void SoundTask() {
    if (SoundEnable == true) {
        //State machine code here
    }
}
```

**Figure 18:**
Parent-Child Logic for Alarm and Tone Tasks

**Implementation and Testing**

This section of this paper will deal with actual software writing. This includes recommendations on project organization and development flow as they apply to multiple state machine design. It will also include strategies for state machine and integration testing.

**Project Organization**

Organize the project in the same way that the design is organized—break it up by tasks, with the source, test and documentation files for each task separated into subdirectories. This system is cleaner than running one big directory. It also saves time searching for files and significantly reduces the risk of accidentally overwriting another task’s files. Figure 19 is an example development-file structure for an alarm clock.
Figure 19:
Example Development-File Structure for Alarm Clock

Development Flow

A good place to start development is to build the variable-handling subroutines. Every other part of the system uses this section, so having it built, tested and documented provides a valuable reference document for the balance of the design. A copy of the file containing the original protocol-routine prototypes should be retained for use in developing test software. Figure 20 shows some example buffer-interface routines.

```c
Void Store(unsigned char data_out) {
    Ser_Data[Ser_Data_in] = data_out;
    Ser_Data_in++;
    if (Ser_Data_in > 8) Ser_Data_in = 0;
}

unsigned char Retrieve(void) {
    unsigned char temp;
    temp = Ser_Data[Ser_Data_out];
    Ser_Data_out++;
    if (Ser_Data_out > 8) Ser_Data_out = 0;
    return temp;
}

unsigned char Test_empty(void) {
    if (Ser_Data_in == Ser_Data_out)  return 1;
    else return 0;
}

unsigned char Test_full(void) {
    unsigned char temp;
    temp = Ser_Data_in + 1;
    if (temp > 8) temp = 0;
    if (temp == Ser_Data_out)  return 1;
    else return 0;
}
```

Figure 20:
Example Buffer-Interface Routines
Next, start building each state machine in a separate file. Even if your development system does not support multiple files, write each state machine separately and then merge the files in the integration step. Using separate files simplifies the task of debugging and allows the designer to concentrate on the specific section without wading through the other state machines in the design. Having the task state machines in separate files also keeps the software modular, and makes its reuse easier because all relevant routines are grouped together in a known-good working condition.

To help with coding, format the software for a task state machine in the form of a C SWITCH statement. Even if the language tool does not support the SWITCH instruction, the SWITCH command format is very compatible with state-machine design and organizes the software in a readily accessible format. Additionally, try to keep the software for each state to a single page for readability. If a state runs to two or three pages, it may be time to consider breaking it into several smaller states.

**Testing**

The best testing solution is to incrementally test the software as it is written. When each stage of the state machine design is written, test it thoroughly. Then, add the next piece, and test it thoroughly. Follow this pattern throughout development. In fact, the order in which pieces of the state machine are written should be influenced by how they can be tested. The following is an example development flow with test points specific to state machines included.

- Build the state-decoding software. Test it thoroughly by passing it at every possible value, not just the values with states associated with them. This is also a good time to experiment with different decoding algorithms to optimize for size or speed. Keep notes of what happens in response to out-of-bounds values for later integration testing. Figure 21 shows example state-decoder and –list for the alarm clock.

```c
Void Command() {
    Switch(CommandState) {
        Case IdleState: Break;
        Case AlarmSetState: Break;
        Case TimeSetState: Break;
        Case AlarmFastState: Break;
        Case AlarmSlowState: Break;
        Case TimeFastState: Break;
        Case TimeSlowState: Break;
        Case SnoozeState: Break;
        Case Toggle1224State: Break;
        Case RepeatdelayState: Break;
        Case ErrorState: Break;
        Default: Break;
    }
}
```

**Figure 21:**
State Decoder and State List for Command Task
• Next, build the conditionals for state transitions. Test all combinations of conditions, even the combinations that should not be possible. Take notes for use in integration testing. Figure 22 shows the state transitions for the alarm clock’s IDLE state.

Void Command()
{
    Switch(CommandState) {
        Case IdleState:
            If (AlarmSetButton == closed)
                CommandState = AlarmSetState;
            If (TimeSetButton == closed)
                CommandState = TimeSetState;
            If (SnoozeButton == closed)
                CommandState = SnoozeState;
            Break;
    }

    Figure 22:
    State Transitions for the IDLE State

• Add the error-detection logic and test it by forcing all potential error conditions and combinations of error conditions, even if they are mutually exclusive. Again, take notes. Figure 23 shows the error-detection logic and error-recovery state for the alarm clock.

    Case AlarmSetState:
        If ((FastSetButton == closed) &
            (SlowSetButton == closed)
            CommandState = ErrorState;
        Break;
    Case ErrorState:
        If ((FastSetButton == open) &
            (SlowSetButton == open) &
            (SnoozeButton == open) &
            ((TimeSetButton == open) +
            (AlarmSetButton == open)))
            CommandState = IdleState;
        Break;

    Figure 23:
    Error-Detection Logic and Error-Recovery State

• Add the state-specific functions. Test for random behavior by starting the state machine out of sequence. Take notes. Figure 24 shows state-specific functions in the Snooze state.
Case SnoozeState: If ((SnoozeButton == open) &
(Snooze == false)) {
  State = IdleState;
  Snooze = true;
}
Break;

Figure 24:
State-Specific Functions in the Snooze State

- Add the input and output sections. Test them by building an interface using the protocol prototypes that allows the convenient passing of values. Test with expected and out of range values. Take notes.

- Implement and test the skip timers. Test the state machine for its response to timer values that are above and below the skip values. Also, test carry conditions on variable byte boundaries, and take notes.

Notes are important. At integration testing, the notes will be a valuable tool for diagnosing problems. Additionally, save all test software. Old test code is a great template for creating new test blocks, which saves development and debugging time.

Timing

When a state machine is complete, collect a database of execution times by state. When writing the priority section, the execution-time information will be valuable, so be accurate. At the end of testing, the data should include the CALL to RETURN execution time for every state in the state machine, including the time for quick RETURNS generated when the skip timer is not zero, as well as abnormal conditions such as state-lock watchdog timeouts.

Integration Testing

When all state machines are complete and tested, integrate the system by adding one state machine at a time. By adding one state machine at a time, the number of potential problems is reduced and it narrows the search area for bugs. Even if the state machine added is not related to the existing set, only add one at a time. There is any number of ways in which state machines can interact that were never intended by the designer.

Typically, problems at integration come from one of two sources:
1. One or more variables are being corrupted.
2. Broadcast variables are causing unintended synchronization between tasks.

The first problem arises when two state machines use the same memory for different variables. This can occur if a variable name is inadvertently reused, or if a memory pointer is out of its range. This problem is the best argument for well-documented Header files. The best way to find this problem is to start mapping the variables to memory space, and look for the overlap. Also, look back through the individual state-machine testing notes. This was the reason for testing out-of-bound values and impossible combinations.
To check synchronization problems, test the individual state machines with test-protocol subroutines that delay the suspect broadcast variable by a ramping delay. The ramping delay will move the time at which a given variable changes its value in each successive pass through the state machine. When the variable, and its problem timing are identified, double-buffer the variable in a state before the problem state to fix its transition time.

**Optimizations**

The state machine framework developed using these techniques will ultimately be larger than standalone software. There is no way to avoid this. However, some techniques can be used to minimize the overall software size.

**Compiler Research**

In general, if you are working with a compiler, experiment with the compiler to determine how it responds to a specific coding style. When the compiler converts your code into assembly and then the hex file, it makes assumptions on the optimum way to implement the software. Knowing how a compiler will implement IF THEN, DO WHILE or SWITCH statement can give the designer significant control over the size of the hex file. It is well worth the effort to explore how a compiler responds to variations in coding.

Another area to examine is the compiler’s optimizer, which looks for specific inefficiencies in the software and makes substitutions for improved speed, size or both. Knowing what the optimizer is looking for and what it substitutes should be understood before enabling optimization.

One final note on optimizers -- when testing, leave the optimization off. Finding a problem is difficult enough without have to trace through the changes that the optimizer has made.

**State Machines**

Another good candidate for optimization in assembly-language implementations are the state-machine state decoders. This is the one section of a state machine that is executed on every pass. Improvements in its speed can produce dramatic results.

One optimization is to decode the most often used states first. For example, assign the idle state a value of zero. The state decoder can use the Zero status flag as a quick idle-state decode before moving on to range checking and decoding of the other states. Figure 25 shows state decoding with fast IDLE decode and jump table.
Another good optimization is to implement the state decoder as a jump table. Jump tables have little overhead and are very fast to execute. However, remember the default state has to be called for all unused state values. A jump table quickly becomes inefficient if half of the entries lead to the same address. Range checking the state variable and consolidating states into contiguous values can help, but remember to balance the speed of a jump table against the inefficiencies in size of the redundant entries.

Another state machine optimization is to make use of subroutine states in the state machine. If two or more states or groups of states perform the same functions, replace both with a single state or group of states and define a return variable to hold the return state.

Variables

The type and definition of variables used in the system can also have a significant effect on how the compiler implements the final firmware. Some possible variable optimizations include:

1. Grouping Booleans in a UNION with a CHAR or INT. This allows pre-loading of the Booleans with a single assignment.

2. Functions that employ bit-wise operations between Boolean variables may execute faster if the variables are declared in the same bit position of different bytes.

3. Math functions will execute faster if they have a common variable type for all input variables. Different variable types must be converted to a common, typically larger, type every time the functions executes, which slows execution and increases size.

4. If variable space is limited, pack several smaller bit variables into a single byte or word variable. However, remember the additional software overhead needed to access the variables.
Most of the optimizations mentioned are obvious and will probably yield only small improvements. However, small improvements repeatedly applied can make a significant difference.

**Conclusions**

The techniques described in this paper are designed to simplify multitasking firmware design using interleaved state machines. While the concepts are simple, this methodology does require a perspective shift. The added complexity of developing two or more state machines and making them work together should not be underestimated. However, if used carefully and methodically, this method makes it possible to create firmware that is modular, reusable, small, fast to develop, easy to test and capable of multitasking.

One final note -- while this is a good design technique, it is not an alternative for every RTOS application. This methodology can produce good results for 8- and small 16-bit microcontroller designs of 5-20 tasks. However, it becomes unwieldy for larger designs. A full RTOS should be considered for designs that are more complex.

###

*Note: The Microchip name and logo are registered trademarks of Microchip Technology Inc. in the U.S.A. and other countries. All other trademarks mentioned herein are property of their respective companies.*
Appendix A

General Specifications for an Alarm Clock

Hardware:

DISPLAY:
1. Six digit, 7 segment displays used to display hours, minutes and seconds of time plus the hours and minutes of the alarm time.
2. A single LED indicator for AM/PM indication. PM = LED lit.
3. A single LED indicator for Alarm Active indications. Active = LED lit.

Notes:
   During the alarm active, when the speaker is generating its tone, the display should flash at a 2 Hz rate that is synchronous to the tone generation. For convenience, it is assumed that the display will be active during the break in the tone and inactive during the tone.
   Will be used in a 60 Hz power environment, so the display must be scanned (all digits) at a rate of greater than 60 Hz. The display should not have a discernable flicker or flash, except in the display of the alarm which will flash as specified above.

SPEAKER:
   A 1-2 kHz tone, modulated ON-OFF at a approx. 2 Hz (+/-10%) rate to indicate the alarm condition. The rate and frequency of the tone should be constant, without discernable drop out or warble.

INPUT KEYS:
1. KEY1: Time Set function, used in combination with Fast and Slow Set functions to set time. If held, will cause Fast and Slow Set functions to increment current time.
2. KEY2: Alarm Set function, used in combination with Fast and Slow Set functions to set alarm time. If held, will cause Fast and Slow Set functions to increment current alarm time.
3. KEY3: Fast Set function, must be used in combination with Time Set or Alarm Set functions. First press will increment either time or alarm time by 20 minutes and clear seconds (time only). If held more than 3 seconds, the key will increment time or alarm at a 20 Hz rate.
4. KEY4: Slow Set function, must be used in combination with Time Set or Alarm Set functions. First press will increment either time or alarm time by 1 minute and clear seconds (time only). If held more than 3 seconds, the key will increment time or alarm at a 20 Hz rate.
5. KEY5: Snooze function, can be used singly or in combination with the Time Set functions.
   a) If Time Set key is held, pressing this key will toggle the Military/AMPM display function for the current time. Holding this key has no function.
   b) If the Alarm is active and the alarm time has passed, this key pressed singly will snooze the alarm for 10 minutes. Holding this key has no function. If held until alarm time, to snooze, the key must be released and then pressed to snooze again.

MICROCONTROLLER:
The clock will be implemented with the physically smallest microcontroller available with sufficient I/O to accommodate the required functions.

PIN CONNECTIONS:
<table>
<thead>
<tr>
<th>RA0</th>
<th>RA1</th>
<th>RA2</th>
<th>RA3</th>
<th>RA4</th>
<th>RA5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blank display (7 segment decoder input)</td>
<td>Latch display (7 segment decoder input)</td>
<td>Speaker (high = on)</td>
<td>Keypress (low = active)</td>
<td>PM indicator</td>
<td>Alarm enabled indicator</td>
</tr>
<tr>
<td>RC0</td>
<td>RC1</td>
<td>RC2</td>
<td>RC3</td>
<td>RC4</td>
<td>RC5</td>
</tr>
<tr>
<td>D0/Dig6/Fastset key</td>
<td>D1/Dig5/Slowset key</td>
<td>D2/Dig4/Alarmset key</td>
<td>D3/Dig3/Timeset key</td>
<td>Dig2/Snooze key</td>
<td>Dig1/Alarm on/off switch</td>
</tr>
</tbody>
</table>