AC 2008-315: EMBEDDED SOFTWARE DESIGN METHODOLOGY TO HELP STUDENTS SUCCEED IN THE REAL WORLD

Keith Curtis, Microchip Technology Inc.

© American Society for Engineering Education, 2008

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 code that is reusable. 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 moderately complex, solid code.

Please note that while a state machine based methodology is a good design technique for small 8 and 16 bit applications, it is not an alternative for every RTOS application. It can produce good results for 8- and small 16-bit microcontroller designs of 5-20 tasks. However, it becomes unwieldy for larger designs. The designer should consider a full RTOS design for significantly more complex applications.

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, or 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 (IT) 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 operating, the next step is to provide communications between the tasks to share information and coordinate activity. This is collectively referred to as intertask communications. 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, multi-tasking 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 of the state machines, 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
3. Implementation and Testing.

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

Implementation and testing gets down to the 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, plus 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 of taking 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 priority- handling requirements of the system. These sections are important, as they determine the requirements for the design to 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 I/O 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 as well.

When this list is compiled, group the functions into tasks. The first impulse is often to give every function its own task. While this may seem to simplify the design and be 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, reducing 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.

The second impulse is to group all related functions into only two or three tasks. This also 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 -- allowing 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 concurrently must 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.

To help illustrate this point, the following list shows one possible grouping of functions from a typical embedded application (the functions have been grouped into tasks). The reasoning behind each choice is listed.

Task 1 Low-Speed SPI Receive Function

Low-Speed SPI Transmit Function

Reasons:

- Receive and transmit functions execute synchronously to each other
- Functions may be asynchronous to other functions
- Functions may operate simultaneously with other functions
- Functions have a different polling-frequency requirements than other functions

Task 2 LED Display-Scanning Function

Time, command, and error code display functions

Reasons:

- Time, command and error functions are an extension of the display function
- The collection of functions is asynchronous to the other function
- All of the functions control the same peripheral

Task 3 Keyboard-Scanning Routine

Reasons:

- Function has different timing and is asynchronous to other functions

Task 4 Error-Buzzer Routine

Key-press ding routine

Alarm buzzer routine

Reasons:

- Functions are grouped, due to their mutually exclusive operation
- Functions have different timing and are asynchronous to other functions
- All functions control a common peripheral (the speaker)
- Functions operate as subroutines to other functions

Task 5 Command Decoder

Reasons:

- Function calls other functions as subroutines
- Function shares some of its timing with other tasks, but is not completely synchronous to any task

As mentioned above, the SPI receive and transmit functions are synchronous and should reside in a common task for simplicity. Separate tasks for the functions unnecessarily complicate their communication and timing. The sound functions have compatible timing, are mutually exclusive and all control the same peripheral, so they can also reside in a common task. However, the LED and keyboard-scan routines are asynchronous to all other tasks, so they should reside in separate tasks. Finally, the command decoder has its own task, due to its use of the other tasks as subroutines and its asynchronous relationship with them. While this example is simple, it does serve to illustrate design decisions made while distributing functions into tasks.

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 as to the type or size of data path -- it just needs to show the general requirements for the communications, plus the source and destination tasks.

Intertask Communications

Before diving into intertask communication layout, a quick overview of the three common types of IT variable protocols 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. 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 IT variable holding the display information for the LED display task. The display task scans only the current display data out to the LED display.

There is no problem if changes occur too quickly or the variable is only partially updated during the scan, because the human eye is not fast enough to see the irregularities. Therefore, the communication meets both requirements for a broadcast variable. One hundred percent reliability is not required and partial updates are tolerable.

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 a command code from the keyboard to the command decoder. When the keyboard determines that a valid key has been pressed, it updates the data portion of the IT variable and sets a flag indicating valid data. The command decoder task then retrieves the data from the IT variable and acknowledges the transfer by clearing the valid data flag. Handshaking on the data transfer insures that the command is reliably transferred once, and only once, from the keyboard task.

An example of a non-data semaphore is an error signal from the SPI-communication task to the command-decoder task. When the SPI task determines that a reception error occurred, it sets an error semaphore indicating the error. The command decoder then clears any command in process, buffers up a report for transmission during the next SPI communication and clears the semaphore to acknowledge receipt of the error. The SPI task is then free to reset and prepare for the next communication. In this example, the semaphore's function is to coordinate the response of both tasks to an SPI error.

Semaphores are a simple way of synchronizing events between tasks. However, care must be taken when semaphores crosslink tasks (crosslinking occurs when two tasks have semaphores passing in both directions between them). The problem is that semaphore crosslinking 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 final protocol is for buffer variables. These IT variables are generally reserved for communication between tasks, where the rate of data handling is significantly different between the two tasks. A good example is the communication between the command decoder and the SPI task. The SPI task is limited in the rate at which it can process data, due to its external timing requirements. The command decoder, 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 decoder does not usually have a problem keeping up with the data received from the SPI task. It handles the commands, one character at a time, as they are received. The problem occurs in the command decoder's response to the commands. The command decoder cannot sit idle, waiting for the SPI task to transmit its response one character at a time. If it did, it might miss commands from the keyboard or other important events in the system. Using a buffer protocol between it and the SPI output function, the command decoder can queue up large responses for the next transmit and then move on to other activities. Adding an additional buffer for incoming data also ensures that received data is transferred, even if the command-decoder task falls behind the incoming SPI function.

A buffer variable is typically a block of memory large enough to hold a reasonable amount of data, with 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 reset 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 is retrieved.

Diagramming the Data Flow

Now that the tasks are defined, the communication requirements of the system 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 the 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.

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, as well as minimizing 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.

Finally, review the variable list for potential state-lock conditions. Remember, this condition occurs when two or more state machines have semaphore variables linking then in both directions. If semaphores link through a third task, they may also become locked. Therefore, check not only tasks that are directly linked, but also tasks that are linked to the two in question.

Timing Analysis

The next step in the system-level design is to analyze the timing requirements of each system task. 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 minimum Tick and the skip rate for each of the tasks.

For example, look at the task list from the task definition section. Note that timing accuracy is also included.

- SPI Low Speed SPI Receiver/Transmitter Task Capable of 1K Bits Per Second

  Sample Rate = 4*1000 or 4000 Calls/Sec to Capture the Clock Falling Edge
  Sample Rate = 1000 Calls/Sec to Initially Acquire Enable Input Falling Edge
  Accuracy of 2 percent or better is required.

- Display Seven Segment, 8-Digit LED Display, Scanned by Software
  To prevent flicker, 60 Scans/Sec Min; Scan Rate = 8 digits * 60 or 480 Calls/Sec. Min.
  Accuracy of 10 percent or better.

- Keyboard 4 by 5 Key Matrix, Scanned by Rows
  10 Scans/Sec/Row, Scan Rate = 5 * 10 or 50 Calls/Sec Min.
  Accuracy of 40 percent or better.

- Sound Direct Drive of the Speaker.
  300 Hz, 1.0 kHz, 2.0 kHz; Nyquist = 2*Max. Frequency, or 4000 Calls/Sec.
  Accuracy of 5 percent or better.

- Command Minimum Command Length = 4 Bytes at 1Kbits/sec; Min. Call Rate = 1000 / 8 / 4 or 31.25 Calls Per Second Minimum

  Accuracy of 50 percent or better.

The Tick for each task can now be calculated using:

Tick = 1 / (calling Rate)

- SPI 1000 uS +/-2% for enable, 250 uS +/-2% for clock edge
- Display 4.16 mS max. all states
- Keyboard 20 mS max. all states
- Sound 1.67 mS, 500 uS, 250 uS +/-5% all states
- Command 33 uS +/- 50% all states

Typically, the smallest Tick will be the minimum common tick 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 3mS 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, the minimum Tick is 250 uS. 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.

- SPI 1000/ 250, Skip Rate is 1:1 for the Clock and 4:1 for the Enable Input
- Display 4200 / 250, Skip Rate is 16.8:1 or 16:1
- Keyboard 20000 / 250, Skip Rate is 80:1
- Sound 1666 / 250 Skip Rate is 6.66:1 or 7:1 for 300 Hz Tone (286 Hz, -4.3%)

  500 / 250 Skip Rate is 2:1 for 1 kHz Tone

  250 / 250 Skip Rate is 1:1 for 2 kHz Tone
- Command 33000 / 250 Skip Rate is 132:1

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 latter 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.

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 design, IT variable and protocol routine definitions, system and skip-timer routine design and priority-handler design.

State Machine Design

Component-level state machine design is the process of laying out the number of states required, their function, inputs, outputs and conditions that cause a change from one state to another.

The first step of the design process is to make a preliminary list of states, listing the general action to be taken during the state.

It should be noted that it is not necessary for the state machine to change state every pass. In fact, some states are idle states in which the state machine may wait for a long time for some outside event. Other states may be repeated to perform the same function on a group of data variables. In fact, some state machines have only a single state, such as a state machine that scans an LED display. The software that is executed each time is the same. It is the data used by the software that is indexed by the state variable. In practice, most state machine designs will be a combination of data and execution indexing. The next example shows a combination of data and execution indexing in the design of a low-speed, SPI communication task state machine.

To start the state machine design, create a preliminary list of states:

State 1 Idle State: Wait for Enable to go Low

State 2 Shift Out State: Output Data Bit, 8-bits of Data (Indexed Data)

State 3 Shift In State: Latch Input Data Bit, 8-bits of Data (Indexed Data)

State 4 Repeat State: Test for End, Enable High=End of Communication, Clock High = Another Byte

When the list is complete, ask a 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 clocking data in and out based on an external clock, so the possibility exists for timing errors. 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.

Note that states 2 and 3 are repeated for 8 bits. While the bits are different, the functions of "shift out" and "shift in" are common. These are the state machine's indexed data states. To handle the data's index, either a separate counter must be employed, or the state variable will have to decode 16 values into two states. Typically, a separate counter is preferred as it makes the state variable decoding simpler and the added data overhead is smaller than the additional software required to decode the more complex state variable.

The next step is to define the conditions necessary for the transition from state to state.

State 1: Stay in State 1, unless the enable line goes low and then move to State 2.

State 2: Wait for the clock to go high and then move to State 3

State 3: Wait for the clock to go low and, if 8 bits are complete, move to State 4.
f not, move to State 2.

State 4: If the enable input goes high, move to State 1.
If the clock input goes high, move to State 2.
If neither input goes high in 20 mS, move to the error state.

Error: Set the ERROR TYPE variable to Timing-Error, set the ERROR FLAG,
ait for acknowledgement and then move to State 1

Default: Set the ERROR TYPE variable to State-Error, set the ERROR FLAG, wait or acknowledgement and then move to State 1

The next step is to make a list of inputs required for the state machine. For this design, the inputs include the clock input, the enable input, the data input and the input buffer from the command-decoder 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 SPI-task state machine, the outputs include an ERROR FLAG semaphore, a broadcast ERROR TYPE variable, a single data-output bit and a buffer for the received data. Listing the states that control these variables yields the following list:

State 1 No Output

State 2 Serial Output Bit = data_byte(bit counter)

State 3 No Output

State 4 Output Buffer = Received data_byte

Error ERROR TYPE = Timing, ERROR FLAG Semaphore = True

Default ERROR TYPE = State, ERROR FLAG Semaphore = True

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 the wait for the rising edge of the enable signal is overly long. In either state, the state machine generates an ERROR FLAG, waits for acknowledgement and then returns to the idle state. Both error flags could have been semaphore variables, but a single error semaphore with a broadcast “error type” variable sends the same message and is less complex to implement.

Finally, document the state machine in a compact short hand for inclusion in the code listing as comments. Pseudo-code for any specific steps or algorithms should also be included.

State 1 **IDLE**

Data Output Bit = 0

If Enable Input = 0 then State2, retrieve next data byte for transmit

State 2 **SHIFT OUT**

Output Data Bit = transmit data_bit(bit counter)

If Clock Input = 0, then State3

State 3 **SHIFT IN**

Receive data_bit(bit counter) = input data_bit

Increment Bit Counter

If Clock Input = 1, then State 2

If Bit Counter = 8, then State 4

State 4 **REPEAT**

Store receive data to input buffer

If Clock = 1, then retrieve next data byte for transmit and go to State2

If Enable = 1 then State1

If Timer > 20 mS then Error

Error **TIMING ERROR**

Set error type to Communications Error

Set Error Flag
If Acknowledge, then State1

Default **STATE ERROR**

Set error type to State Error

Set Error Flag
If Acknowledge, then State1

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 controls as specified by the protocol. The IT 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.

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 generating any intermediate testing subroutines that may be required during testing.

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 disable or defer 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.

If the communications are 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.

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 IT variable list.

Timers

In he timing-analysis section, the timing requirements and skip rates for each of the task state machines was defined. At this level, the designer should decide if the timer 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 state machine may need separate states for each delay. If the timer is to be an external routine, then a timing handler with the appropriate timer variable and timeout flags should be defined. 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 machines' 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 the 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 minimizes the number of state machines that are likely to time out and actually execute a state on any pass through the loop.

In the following example, the skip rates of three tasks are related to each other by multiples of four. Staggering the start of their skip timers by 1 count guarantees that the tasks will never be executed on the same pass through the loop.

The skip rates are Keyboard = 80:1, LED Display = 16:1, Command Decoder = 132:1. At startup:

- The keyboard skip timer starts at 0, forcing it to time out on the 1st pass in 4
- The LED display skip timer starts at 1, forcing it to time out on the 2nd pass in 4
- The command-decode skip timer starts at 2, forcing it to time out on the 3rd in 4

Because each skip timer will reload with its full skip rate when it decrements to zero, the three tasks will retain their staggered execution times and continue to execute on separate passes through the loop.

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 decoding a specific state, the state machine for each task determines the time remaining in the Tick and the time required to execute its current state. If sufficient time remains, the state machine will decode the state variable and execute the state. If not, the state machine will defer execution to the next state machine by returning without executing the current state.

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 should be configured to free-run at the microcontroller's instruction rate, and its rollover period should be equal to the system Tick. Its value at any given time will then be equal to the number of instruction cycles remaining in the Tick. Each state machine includes software to compare the timer count against the execution time for its current state to determine if enough time remains to execute the state. If sufficient time does not remain, the state machine returns to the calling loop, allowing the next state machine in the loop to try. If sufficient time does remain, the state is decoded and executed. 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.

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 introduce beat-frequency behavior in the system timing.

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 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. Good examples of the variable-order system are the two system states LINE and LOCAL. In the LINE system state, the LED display, sound or keyboard tasks can be disabled or ignored, allowing priority execution to be shifted to the more important SPI and command-decoding tasks. Conversely, in the LOCAL state of the system, the SPI task is ignored, allowing the system to concentrate on the command decoder and the user interface.

The design work required for the variable-order system can be a little more complicated to design 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.

“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 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 would be the decision by the keyboard state machine to delay its decoding of a key-press if the SPI or command-decoder tasks are not in their idle states. This takes advantage of the keyboard task’s slower, more flexible timing to “Excuse Me” out of the loop for the short period while the other two tasks are busy.

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 the command-decoder task using an “Excuse You” flag to force the keyboard and sound tasks to stop execution until it has completed its response to a command from the SPI task. Because the response to SPI communication has tighter timing requirements than the keyboard or sound tasks, the keyboard and sound tasks can be safely delayed until the command decoder completes its response. The command decoder task is creating an “Excuse You” condition to disable the other tasks.

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 solely based 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 it 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 command-decoder task simply sets the state variable to the starting state of the tone required, 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, when the tone is complete, the tone state machine clears its enable flag to disable itself and signal the command decoder task to resume its normal operation. This not only frees up time in the Tick, but also eliminates the need for an idle task in the tone state machine, which saves program space.

Implementation and Testing

This section 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.

Development Flow

A good place to start the development is to build the variable-handling subroutines. This section is used by every other part of the system, 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.

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.

- 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.

- 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.

- Add the state-specific functions. Test for random behavior by starting the state machine out of sequence. Take notes.

- 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. Adding one state machine at a time limits the number of problems and 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 is the state machine state decoder, the one section of a state machine that is executed on every pass. Improvements in its speed can produce dramatic results.

One good 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.

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 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 may 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.

Note: All trademarks mentioned herein are property of their respective companies.

References

Over the course of my career developing embedded applications, I tried a number of different systems—relying on complex interrupt systems, building simple version of the RTOS systems and even foreground/background context switching using interrupts. However, the simplicity and flexibility was what kept bringing me back to state machine-based design. As a result, I have slowly evolved this design technique over the course of my career, primarily as an aide in the timely development of projects.

As part of this evolution, I have also conducted an extensive search to find either books or papers on the subject of multitasking in small microcontrollers. Unfortunately, while books on RTOS solutions (and operating system fundamentals) are plentiful, I found absolutely nothing on the subject of using multiple state machines for multitasking.

Finally, when I started working for Microchip, I decided to formalize the methodology and present it in a paper at CMP's Embedded Systems Conference in Chicago. At this presentation, I meet a number of engineers who all had a very similar experience to my own. They had developed very similar methodologies, and had similarly failed to find any published references on the subject.

This convinced me that there was a need for this methodology to be published and, as I currently work in a position which specializes in training customers (embedded designers), I had a forum in which to propagate the information. I therefore developed a four-hour class based upon the subject and proceeded to teach the technique to customers and conference attendees in both the US and Europe.

Invariably, when speaking with attendees following the presentation, the typical responses were:

1. Experienced engineers typically evolve a design methodology very similar to this
2. They have yet to find any publication on the subject.

Years later, after presenting the material to hundreds of engineers, I was approached by an acquisitions editor for Elsevier who talked me into committing the methodology to paper in the form of a book. This book is presently available and, to my knowledge, it remains the only book on the subject of multitasking with state machines in small microcontrollers, even though the practice appears to be an almost universal method for low-cost embedded multitasking design.

This unique contradiction of a very prevalent technique, with almost no publications in support of it, still remains a mystery today. While I will admit there are probably several texts on the subject, my own periodic search of the Internet has yet to produce anything other than information that I have published. However, for completeness, I will include here references to both my book and the papers I have published on the subject.

1. “Embedded Multitasking With Small Microcontrollers,” by Keith Curtis. ISBN: 075067918, Publisher: Elsevier

2. “Embedded Multitasking with Small MCUs,” by Keith Curtis. Microchip Technology Inc., www.embedded.com.

3. “Doing Embedded Multitasking with Small Micros,” by Keith Curtis, Microchip Technology Inc., www.mechatropolis.com