Programming Embedded Systems

An Introduction to Time-Oriented Programming

Version 4.0

Frank Vahid
University of California, Riverside

Tony Givargis
University of California, Irvine

Bailey Miller
University of California, Riverside

UniWorld Publishing, Lake Forest, California

Version 4.0, August, 2012.

Formatted for electronic publication.

ISBN 978-0-9829626-4-0 (e-book)

UniWorld Publishing

www.programmingembeddedsystems.com
info@programmingembeddedsystems.com

Contents

Chapter 1: Introduction
Chapter 2: Bit-Level Manipulation in C
Chapter 3: Time-Ordered Behavior and State Machines
Chapter 4: Time Intervals and Synchronous SMs
Chapter 5: Input/Output
Chapter 6: Concurrency and Multiple SynchSMs
Chapter 7: A Simple Task Scheduler
Chapter 8: Communication
Chapter 9: Utilization and Scheduling
Chapter 10: Programming Issues
Chapter 11: Implementing SynchSMs on an FPGA
Chapter 12: Basic Control Systems
Chapter 13: Basic Digital Signal Processing
Book and Author Info

Chapter 1: Introduction

The first computers of the 1940s and 1950s occupied entire rooms. The 1960s and 1970s saw computers shrink to the size of bookcases. Continued shrinking in the 1980s brought about the era of personal computers. Around that time, computers also became small enough to be put into other electrical devices, such as into clothes washing machines, microwave ovens, and cash registers. In the 1990s, those computers became known as embedded systems.

What is an embedded system?

An ***embedded system*** is a computer embedded within another device. The embedded computer is composed of hardware and software sub-systems designed to perform one or a few dedicated functions. Embedded systems are often designed under stringent power, performance, size, and time constraints. They typically must react quickly to changing inputs and generate new outputs in response. Aside from PCs, laptops, and servers, most systems that operate on electricity and do something intelligent have embedded systems. Simple embedded system examples include the computer in a clothes washing machine, a motion-sensing lamp, or a microwave oven. More complex examples include the computer in an automobile cruise control or navigation system, a mobile phone, a cardiac pacemaker, or a factory robot. (Wikipedia: Embedded_Systems)

Examples of embedded systems include computers in simple systems like blinking tennis shoes or coffee makers, to more complex systems like mobile phones or automated teller machines.

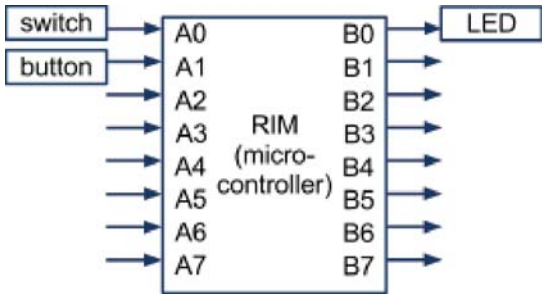
Try: List three embedded systems that you interact with regularly.

Wikipedia: iPhone
Wikipedia: IP_Phone
Wikipedia: PlayStation
Wikipedia: Amazon_Kindle
Wikipedia: SetTopBox
Wikipedia: HVAC_Control_System
Wikipedia: Engine_Control_Unit
Wikipedia: Flight_Control_System

Each year over 10 billion microprocessors are manufactured. Of these, about 98% end up as part of an embedded system.

The example statement "B0 = A2 && A1 && A0" sets the microcontroller output B0 to 1 if inputs A2, A1, and A0 are all 1. The "`while (1) { <statements> }`" loop is a common feature of a C program for embedded systems and is called an ***infinite loop***, causing the contained statements to repeat continually.

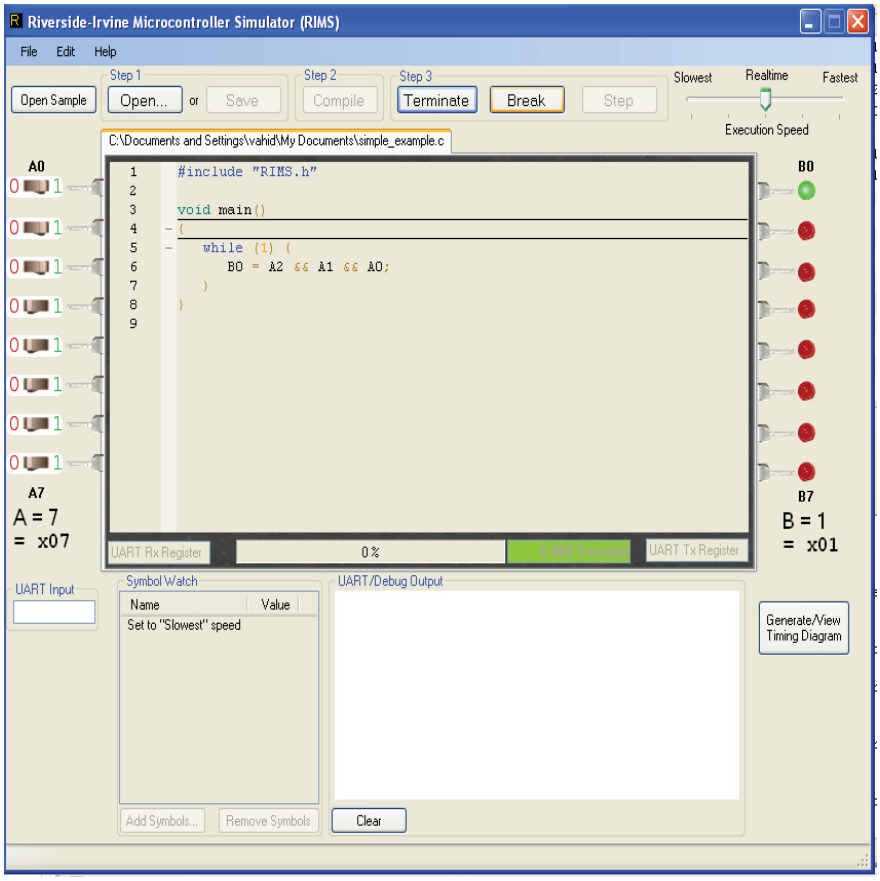
We can use a microcontroller to add functionality to the earlier simple system to create an embedded system. The term embedded system, however, commonly refers just to the compute component. The switch and buttons are examples of ***sensors***, which convert physical phenomena into digital inputs to the embedded system. The LED is an example of an ***actuator***, which converts digital outputs from the embedded system into physical phenomena. (Wikipedia: Sensor) (Wikipedia: Actuator)



The RIM connected to switches, buttons, and LEDs.

RIMS

RIMS (RIM simulator) is a graphical PC-based tools that supports C programming and simulated execution of RIM. RIMS is useful for learning to program embedded systems. A screenshot of RIMS is shown below. The eight inputs A0-A7 are connected to eight switches, each of which can be set to 0 or 1 by clicking on the switch. The eight outputs B0-B7 are connected to eight LEDs, each of which is red when the corresponding output is 0 and green when 1.



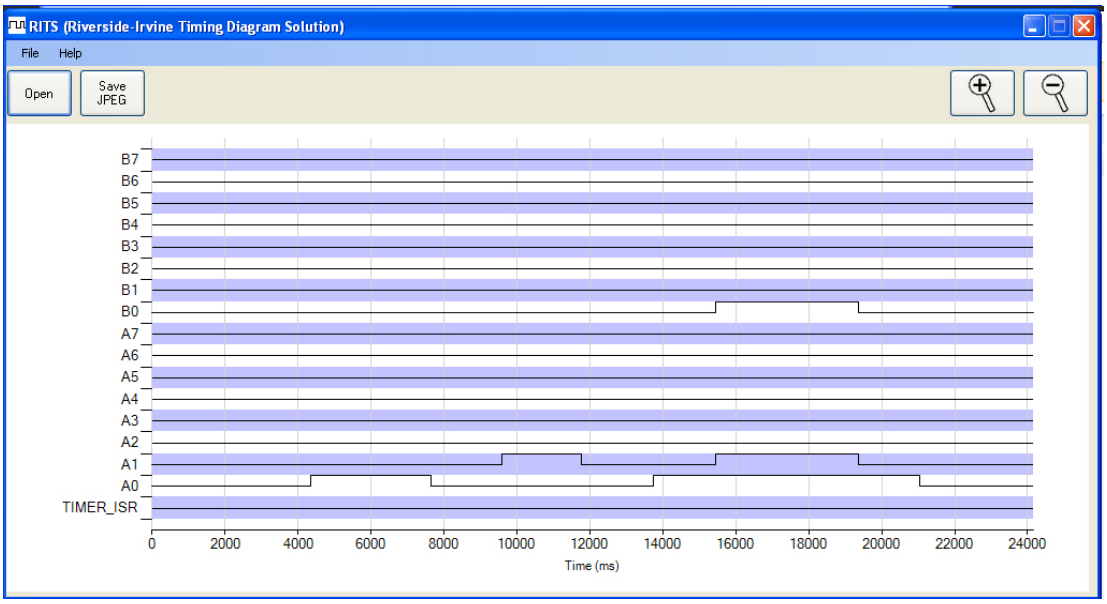
Try: Circle all ten events in the above timing diagram.

Testing

Written code should be tested for correctness. One method is to generate different input values and then observe if output values are correct. To test code implementing "B0 = A0 && !A1", *all possible input value combinations* of A1 and A0 can be generated: 00, 01, 10, and 11. Using RIMS, switches can be clicked to generate each desired input value. First switches for A1 and A0 can both be set to, then A0 can be set to 1, then A0 can be set to 0 and A1 to 1, and finally both A1 and A0 can be set to 1. B0 should only output 1, and hence B0's LED should only turn green, in the second case.

For most code, there are too many possible input combinations to test all of them. Testing should cover ***border cases*** such as all inputs being 0s and all inputs being 1s, and then several *sample normal cases*. For example, completely testing "B0 = A0 && A1 && A2 && A3 && A4 && A5 && A6 && A7" would require 256 unique input value combinations. Border and sample testing might instead test two borders, A7..A0 set to 00000000 (output should be 0) and to 11111111 (output should be 1), and then a few (perhaps a dozen) sample normal cases like 00110101 or 10101110. If code has branches, then good testing also ensures that every statement in the code is executed at least once, known as *100% code coverage*. (Wikipedia: Software Testing) (Wikipedia: Software Debugging)

RIMS records all input/output values textually over time. That text can be analyzed for correct code behavior, rather than observing RIMS LEDs. Pressing the "Generate/View Timing Diagram" button while a program is running (or in a "break" status) automatically saves those textual input/output values in a file and then runs the timing-diagram viewing tool called ***RITS*** (Riverside-Irvine Timing-diagram Solution).



RITS timing diagram from running the "B0 = A0 && !A1" program.The user can zoom in or out using the "+" and "-" buttons on the top right, and scroll using the scrollbar at the bottom. The user can save the currently shown portion of the timing diagram using the "Save JPEG" button.

```
}
```

(Note: bit #0 refers to the least significant bit, *LSB*, so bit #2 is the third bit from the right)

Try: Write a single C statement for RIM that sets B to A except that bit #7 and bit #6 are set to 1s.

A ***mask*** is a constant value having a desired pattern of 0s and 1s, typically used with bitwise operators to manipulate a value. In the above examples, 0x0F, 0xF0, 0xF7, and 0x04 are masks. Masks are typically used based on the following ideas (below, assume "a" is a single bit):

- To force a bit position to 0, AND with a mask having 0 in that position (a & 0 = 0)
- To force a bit position to 1, OR with a mask having 1 in that position (a | 1 = 1)
- To pass a bit position through, AND with a mask having 1 in that position (a & 1 = a), or OR with a mask having 0 in that position (a | 0 = a)

Masks are sometimes defined as constant variables:

```
const unsigned char MaskLoNib1s = 0x0F;
B = A | MaskLoNib1s; // Passes high nibble, sets low nibble to 1111
```

The term *mask* comes from the role of letting some parts through while blocking others, like a mask someone wears on his face letting the eyes and mouth through while blocking other parts.

Two more bitwise operators are commonly used:

- ***<< : left shift***
- ***>> : right shift***

For unsigned integer types, shift operators move their first operand's bits left/right by the number of positions indicated by their second operand (the shift amount), as shown:

```
0x0F << 2:          0x0F >> 3:
   00001111            00001111
<< 2                >> 3
------------        ------------
   00111100            00000001
```

Note that vacated positions on the right (for left shift) or left (for right shift) have 0s shifted in. Below are some examples of using shift operators:

```
B = A << 1; // Sets B7 to A6, B6 to A5, ..., B1 to A0, and B0 to 0
B = A >> 4; // Sets B7..B4 to 0000, and B3..B0 to A7..A4
B = A & (0x0F << 2); // Passes A's 4 middle bits to B
```

The shift amount should be between 0 and the number of left-operand bits, inclusive.

Try: Compute "B = A << 6;" and "B = A >> 5" for A being 0xFE.

Try: Test the above shift operator examples using RIMS, by creating a distinct program for each.

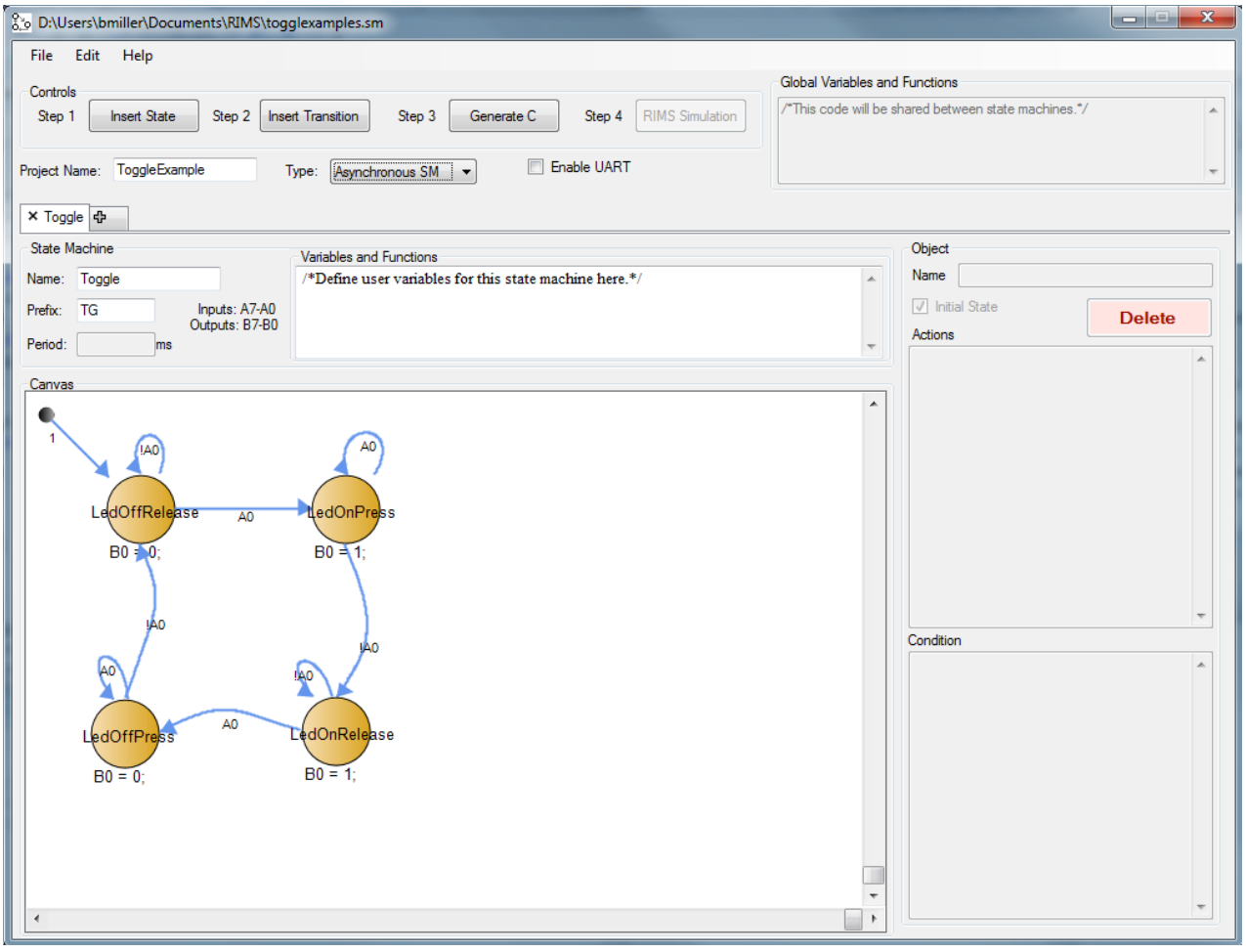
Notice that the SM model and the earlier C code have the same behavior. However, the SM more explicitly captures the desired time-ordered behavior. This straightforwardness can be further seen by trying to extend the SM.

Try: Extend the SM to sound an alarm if A1A0=11 is reached by a sequence other than 00, 10, 11.

The extension can be achieved by adding transitions leaving Wait00 and Wait10, where each transition checks for "A1 && A0" and points to an "Alarm" state that sets B1=1 (all other states should set B1=0). The transitions leaving Wait00 and Wait10 that point back to the same state would also need to have their conditions refined to not include the "A1 && A0" case.

RIBS

The ***RIBS*** (Riverside-Irvine Builder of State machines) tool supports graphical state diagram capture of SMs.



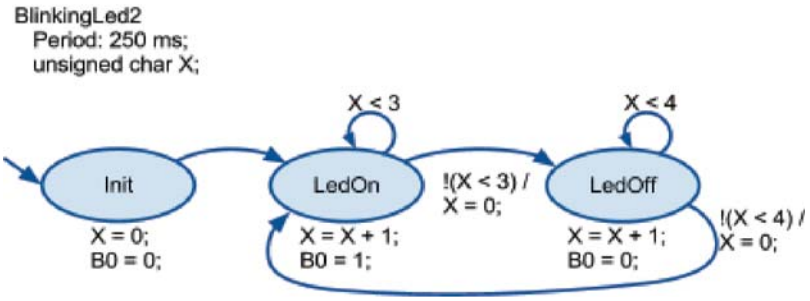
A user can insert states and insert transitions between states. The user can click on a state and write C code for the state's actions (in the text box on the right). Likewise, the user can click on a transition and write C conditions (bottom-right text box). The SM begins by transitioning to an initial state (from the shown black dot). Pressing "Generate C"

Choosing a period for different time intervals

Sometimes one system involves multiple different time intervals. For example, consider a system that repeatedly blinks an LED on for 500 ms and off for 1 second. The system involves two different intervals: 500 ms, and 1 second. The system can be captured as a synchSM having a 500 ms period and three states: LedOn, LedOff1, and LedOff2. The idea is to choose the period as the greatest common divisor of the required time intervals, and then use multiple states (or counting within a state) to obtain the actual desired interval.

Try: A system should repeatedly blink an LED on for 750 ms and off for 1 second. Capture the behavior as a synchSM, clearly specifying the period.

Counting within a state is better than using multiple states when the desired interval is much larger than the SM period. The following shows the above blinking LED example using a variable for counting.

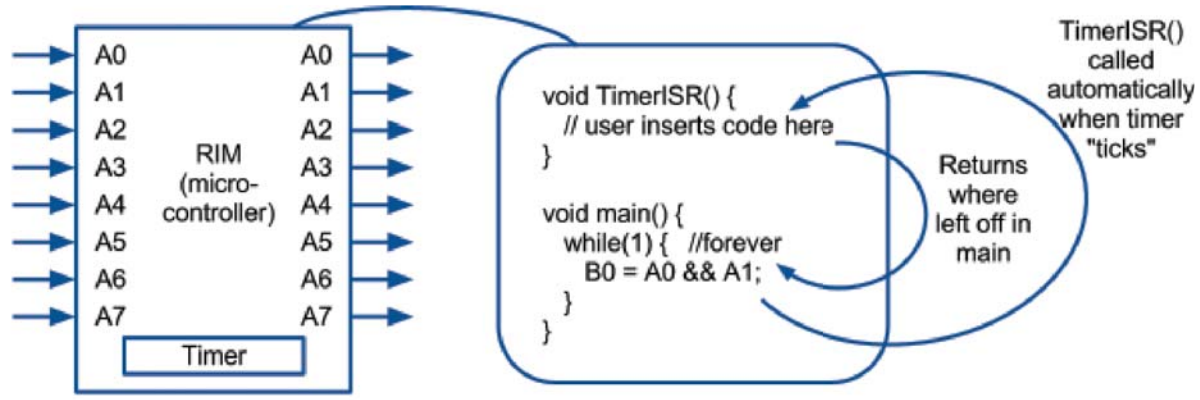


The advantage of the counting approach is clearer for a longer off interval, such as 5 seconds.

Converting a synchSM to C

Microcontrollers with timers

Microcontrollers come with one or more timers to measure time intervals. A ***timer*** is a hardware component that can be programmed to tick at a user-specified rate, such as once every 100 ms. (Wikipedia: Programmable Interval Timer). When the timer ticks, it interrupts the microcontroller's execution. ***Interrupt*** means to temporarily stop execution of the main C code and jump to a special C function known as an ***interrupt service routine (ISR)***. (Wikipedia: Interrupt Service Routine). When that ISR function finishes executing, execution resumes where it previously stopped in the main C code.



Timer can be set to "tick" every T ms. At each timer tick, RIM stops executing the main code, calls TimerISR() automatically, and then resumes executing the main code where it previously stopped.

In RIMS, the ISR is called **TimerISR.** It can be defined by the user as follows:

```
void TimerISR() {
   // user inserts code here
}
```

Every time the hardware timer ticks, the TimerISR function gets called *automatically* by the microcontroller. The user can insert code into the ISR that should be executed whenever the timer ticks. The user's own main code should *never* call the TimerISR function directly.

The user sets the timer's tick rate by calling another RIMS built-in function, **TimerSet**(Period), where Period is an unsigned short indicating the tick period in milliseconds. To activate the timer, the user calls **TimerOn**().

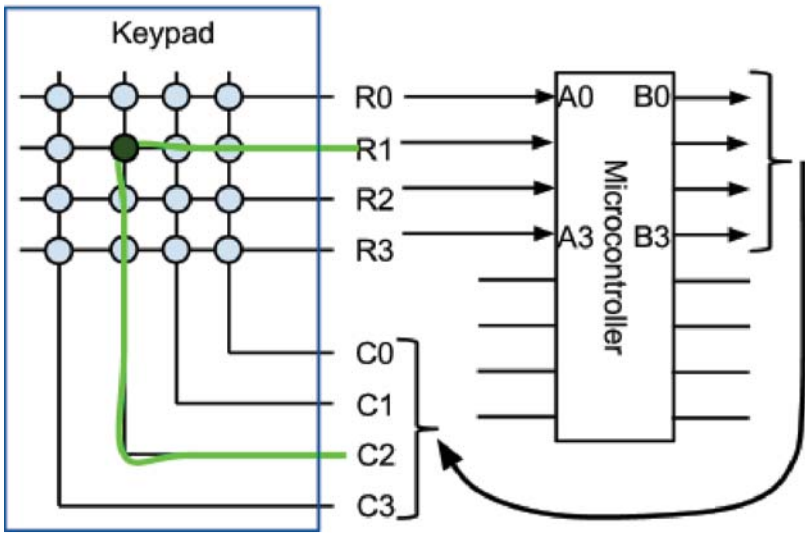
We use the ISR to set a global flag to 1. A ***flag*** is a global variable used by different parts of a C program to communicate basic status information with one another. (Wikipedia: Flag (computing)). The user's main C code can thus monitor the flag's value, waiting for it to become 1, to determine that the timer has ticked. For example, the following code would toggle B0 every 1 second:

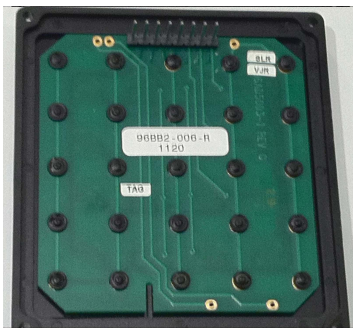
```
#include "RIMS.h"

volatile unsigned char TimerFlag = 0;

void TimerISR() {
   TimerFlag = 1;
}

void main() {

   B = 0;//Initialize output

   TimerSet(1000); // Timer period = 1000 ms (1 sec)
   TimerOn();      // Turn timer on

   while (1) {
      B0 = !B0;            // Toggle B0
      while(!TimerFlag) {} // Wait 1 sec
      TimerFlag = 0;
```

Similarly, a microcontroller with too few inputs may expand its effective inputs via time-multiplexed reading of input registers (in conjunction with an external multiplexor), or via rapid scanning, which is the input-version of refresh. To illustrate rapid scanning, consider a keypad. A keypad is a device consisting of several push buttons arranged in a two-dimensional grid, as in the figure on the right with 16 buttons arranged in a 4x4 grid. If each button had its own pin, 16 pins would be required on the keypad as well as on a microcontroller that reads the keypad -- in general, an NxM keypad would require N*M pins. Reducing that number of pins is important, especially for larger keypads such as a PC's keyboard.

Similarly, a microcontroller with too few inputs may expand its effective inputs via time-multiplexed reading of input registers (in conjunction with an external multiplexor), or via rapid scanning, which is the input-version of refresh. To illustrate rapid scanning, consider a keypad. A keypad is a device consisting of several push buttons arranged in a two-dimensional grid, as in the figure on the right with 16 buttons arranged in a 4x4 grid. If each button had its own pin, 16 pins would be required on the keypad as well as on a microcontroller that reads the keypad -- in general, an NxM keypad would require N*M pins. Reducing that number of pins is important, especially for larger keypads such as a PC's keyboard.

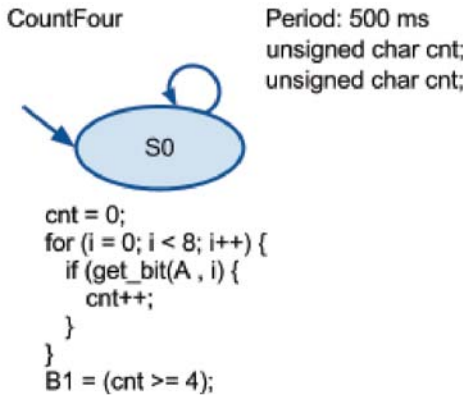
Instead, keypads commonly have only N+M pins, or 8 pins for the above 4x4 keypad, arranged as shown in the figure below. Each button (drawn as a circle) is a passive button that when pressed connects one R terminal with one C terminal.





A microcontroller can poll each button one at a time, a process known as *scanning*, to detect whether that button is pressed. The scheme works as long as the keypad is scanned at a rate much faster than button presses. The microcontroller connects outputs B3..B0 to the keypad's C3..C0 terminals, and inputs A3..A0 to the keypad's R3..R0 terminals. Those A3..A0 inputs must be configured as pull-up, meaning each will be read as 1 (even when disconnected) unless a 0 is written.

With the hardware interfacing complete, an algorithm for scanning the keypad can be created. The first step is to write a function to read whether a particular button (say the button at row 2 and column 1) is pressed, assuming at most one button is pressed at a



The transition from S0 back to S0 implements the "while (1)" loop. Selecting the same period as the BlinkLed and ThreeLeds tasks, namely 500 ms, enables straightforward round-robin execution of all three tasks when converted to C. A TickFct_CountFour() can be written for the CountFour synchSM, and the main code's while loop will simply be extended with a third task:

```
...
while (1) {
   TickFct_BlinkLed();     // execute one tick of the BlinkLed synchSM
   TickFct_ThreeLeds();    // execute one tick of the ThreeLeds synchSM
   TickFct_CountFour();    // execute one tick of the CountFour synchSM
   while (!TimerFlag){}    // wait for timer period
   TimerFlag = 0;          // lower flag raised by timer
}
```

For the above to work, the original sequential instructions should have been run to completion (within the infinite loop, of course). If they weren't run to completion, then the behavior should first be re-captured as an SM instead such that each state's actions run to completion.

When translating any single-state synchSM to C where that state has a single true-conditioned no-action transition pointing back to the state (as for CountFour above), a reasonable simplification is to eliminate the state and transition code in the synchSM's tick function, just listing that state's actions. For example, the CountFour synchSM may be converted to the following tick function:

```
unsigned char cnt;
void TickFct_CountFour() { // single-state synchSM
  cnt=0;
  for (i=0; i<8; i++) {
     if (GetBit(A, i)) {
        cnt++;
     }
  }
  B1 = (cnt >= 4);
}
```

Global versus local variables

system (OS), but can instead be included directly in user code as above, which is especially useful in the absence of an OS.

To ease learning of ISRs, our earlier code just had the ISR set a flag, but with the reader now more comfortable with ISRs, we now put the scheduler code directly into the ISR. The main() function's "while (1)" loop will be empty; all activity occurs in the ISR.

If a function exists to put the processor into a low-power mode while not executing any tasks and instead waiting for a timer tick, then that function can be called in main's "while (1)" loop. RIM has such a function named "Sleep();", which we insert into that loop.

The below program portion highlights the main code and TimerISR code in the task scheduler approach:

```
void TimerISR() {
   unsigned char i;
   for (i = 0; i < tasksNum; ++i) { // Heart of the scheduler code
      if ( tasks[i].elapsedTime >= tasks[i].period ) { // Ready
         tasks[i].state = tasks[i].TickFct(tasks[i].state);
         tasks[i].elapsedTime = 0;
      }
      tasks[i].elapsedTime += tasksPeriodGCD;
   }
}

int main() {
   unsigned char i=0;
   tasks[i].state = -1;
   tasks[i].period = 500;
   tasks[i].elapsedTime = tasks[i].period;
   tasks[i].TickFct = &TickFct_ThreeLED; //function TickFct_ThreeLED not
shown

   TimerSet(tasksPeriodGCD);
   TimerOn();

   while(1) {
      Sleep();
   }
   return 0;
}
```

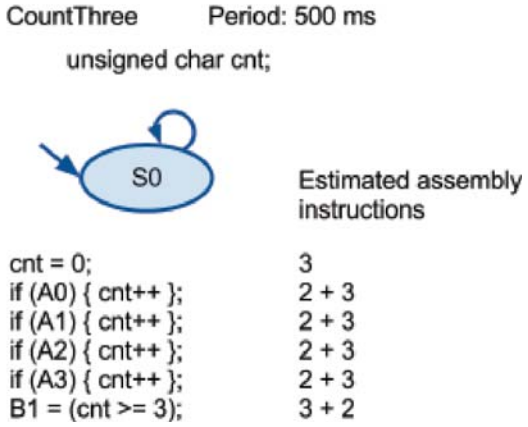
The following shows the complete code for the LedShow system with different-period tasks, including the definition of both tasks, and the scheduler code. To keep the scheduler code close together, the code uses function declarations for the tick functions, with the longer function definitions appearing after the main function.

```
#include "RIMS.h"

typedef struct task {
   int state; // Current state of the task
   unsigned long period; // Rate at which the task should tick
```

programmer may choose to ignore the exception, meaning certain ticks would be skipped.

Analyzing code for timer overrun

A programmer can manually analyze code to estimate whether a timer-overrun exception might occur on a microcontroller. Consider the following single-state synchSM task named CountThree, with a period of 500 ms, that sets B1 to 1 if A0-A3 have three or more 1s:



Ticks are separated by 500 ms. The question is whether state S0's actions execute in less than 500 ms on a particular microcontroller. Suppose a (very slow) microcontroller M executes 800 assembly-level instructions per second, meaning 1 sec / 800 instr = 0.00125 sec/instr. We must estimate the number of assembly instructions to which S0's actions translate. Very roughly, we estimate that each assignment statement ("cnt=0", "cnt++", "B1=") translates to 3 assembly instructions, each *if* statement translates to 2 instructions, and each comparison ("cnt >=3") translates to 2 instructions, as shown in the figure above. Then S0's actions translate to 28 instructions. On microcontroller M, 28 instructions will require 28 instr * 0.00125 sec/instr = 0.035 sec = 35 ms. Because 35 ms is much less than 500 ms, we can estimate that timer overrun will not occur.

The ***utilization*** of a microcontroller is the percentage of time that the microcontroller is actively executing tasks:

Utilization = (time executing / total time) * 100%

For the above, the utilization during a 500 ms time window is the measure of interest, because every 500 ms window is identical. During a 500 ms window, microcontroller M executes S0's actions in 35 ms, so its utilization is computed as 35 ms / 500 ms = 0.07, or 7%. The microcontroller is said to be ***idle*** for the remaining 93% of the time.

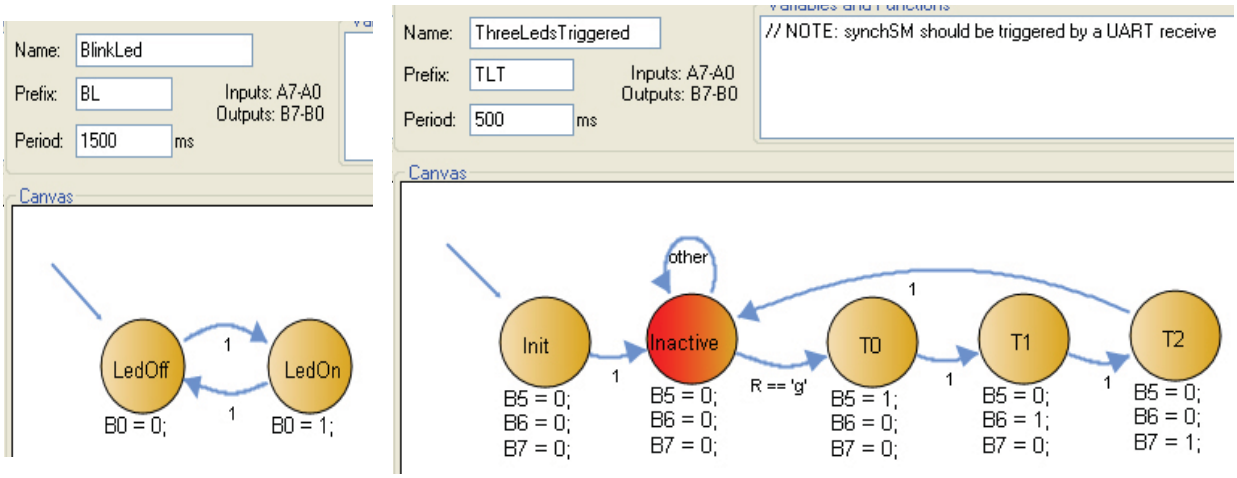
Utilization analysis usually ignores the additional C instructions required to implement a task in C, such as the switch statement instructions in a tick function, or the instructions involved in calling a tick function itself. For typical-sized tasks and typical-speed microcontrollers, the number of such "overhead" instructions is negligibly small. The analysis does not consider the C instructions that simply wait for the next tick ("while (1) Sleep();"); the processor is considered to be "idle" during that time.

A state's actions may include loops, function calls, branch statements, and more, as below:

The timing diagram on the left shows TG with polling, executing every 100 ms. The ticks for state S0 poll the input A0 looking for a change from 0 to 1, and the ticks for state S2 for a change from 1 to 0. The timing diagram on the right shows a triggered TG. TG goes inactive while waiting for an event on A0, and does not execute during that time, reducing microcontroller utilization.

Using existing special hardware that detects events on pins can yield the additional benefit of sampling the pin faster than sampling achieved by polling done by a task on a microprocessor.

A triggered synchSM can be implemented in C on a microcontroller with some extensions to the earlier scheduler. (Note: RIBS does not currently have support for triggered SMs). RIMS does *not* currently have built-in support for detecting events on pins, but RIMS does have built-in support for detecting a UART character receive, and thus a UART receive can be used to trigger a synchSM. Thus we shall introduce an example having a synchSM triggered by a Uart receive. Consider a modification of an earlier example (that was used to demonstrate scheduler code) where one task blinks B0 repeatedly, and another task lights three LEDs B5, B6, B7 in sequence once whenever the letter 'g' is received by the UART: \



The following code modifies the earlier scheduler code to support the triggered synchSM. Key changes are in bold and described in the comments.

```
#include "RIMS.h"

typedef struct task {
   signed   char state;
   unsigned long period;
   unsigned long elapsedTime;
   unsigned char active; // 1: active, 0: inactive
```