



# Programming Embedded Systems

## An Introduction to Time-Oriented Programming

**Frank Vahid and Tony Givargis**

[www.programmingembeddedsystems.com](http://www.programmingembeddedsystems.com)

[info@programmingembeddedsystems.com](mailto:info@programmingembeddedsystems.com)

Version 2 December 10, 2010

ISBN 978-0-9829626-0-2

UniWorld Publishing

Copyright © 2011 Frank Vahid and Tony Givargis

[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](#)

[Chapter 7: A Simple Task Scheduler](#)

[Chapter 8: Communication](#)

[Chapter 9: Utilization and Scheduling](#)

[Chapter 10: Coding Issues](#)

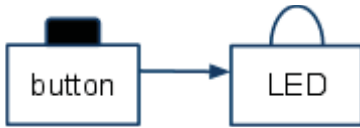
[Chapter 11: Implementing SynchSMs on an FPGA](#)

[Chapter 12: Basic Control Systems](#)

[Chapter 13: Basic Digital Signal Processing](#)

[Book and Author Info](#)

We can build a simple system that is composed of a push button and an LED connected as shown below. When the button is pressed, the LED will illuminate.



This simple system falls short of being an embedded system because it lacks computing functionality. For example, the system can't be easily modified to toggle the LED each time the button is pressed, or to illuminate the LED when the button is pressed AND a switch is in the on position. A component executing some computing functionality is a key part of an embedded system.

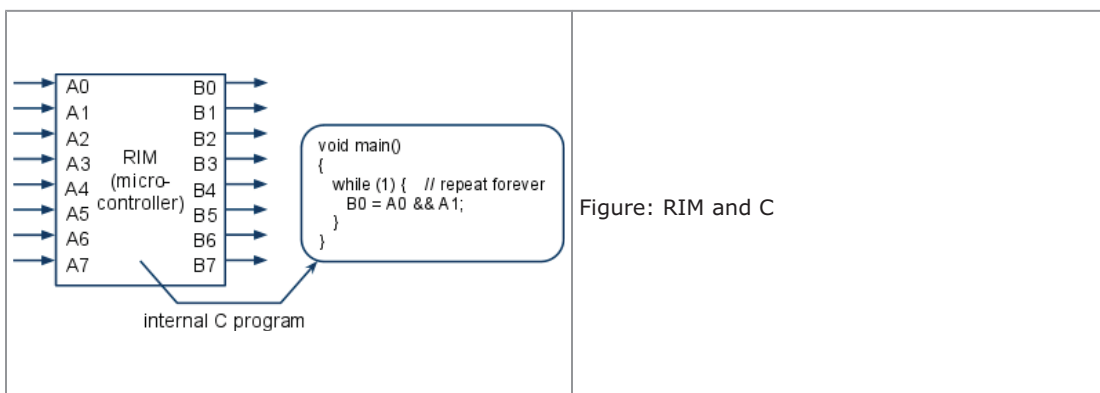
## Microcontroller

A **microcontroller** is a programmable component that reads digital inputs and writes digital outputs according to some internally-stored program. Hundreds of different microcontrollers are commercially available, such as the PIC, the 8051, the 68HC11, or the AVR. A microcontroller contains an internal program memory that stores machine code generated from compilers/ assemblers operating on languages like C, C++, Java, or assembly language.



([Wikipedia: Microcontroller](#) [Wikipedia: Atmel AVR](#) [Wikipedia: C language](#))

We will use an abstraction of a microcontroller, referred to as **RIM** (Riverside-Irvine Microcontroller), consisting of eight bit-inputs A0, A1, ..., A7 and eight bit-outputs B0, B1, ..., B7, and able to execute C code that can access those inputs and outputs as implicit global variables.



The example statement "B0 = A0 && A1" sets the microcontroller output B0 to 1 if inputs A0 and A1 are both 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. The system is now 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.

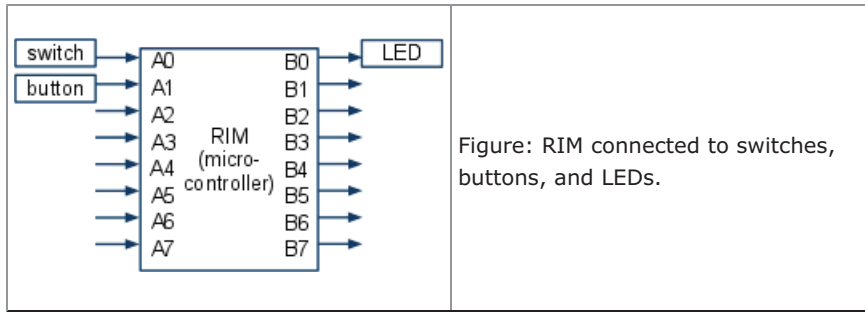


Figure: RIM connected to switches, buttons, and LEDs.

Try: Write a C program for RIM that sets B0=1 whenever any of A0, A1, or A2 is 1.  
 Try: Write a C program for RIM that sets B0=1 whenever the number of 1s on A0, A1, A2, and A4 is 2 or more. Hint: First count the number of 1s, using a variable to keep the count.

**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 8 inputs A0-A7 are connected to 8 switches, each of which can be set to 0 or 1 by clicking on the switch. The 8 outputs B0-B7 are connected to 8 LEDs, each of which is red when the corresponding output is 0 and green when 1.

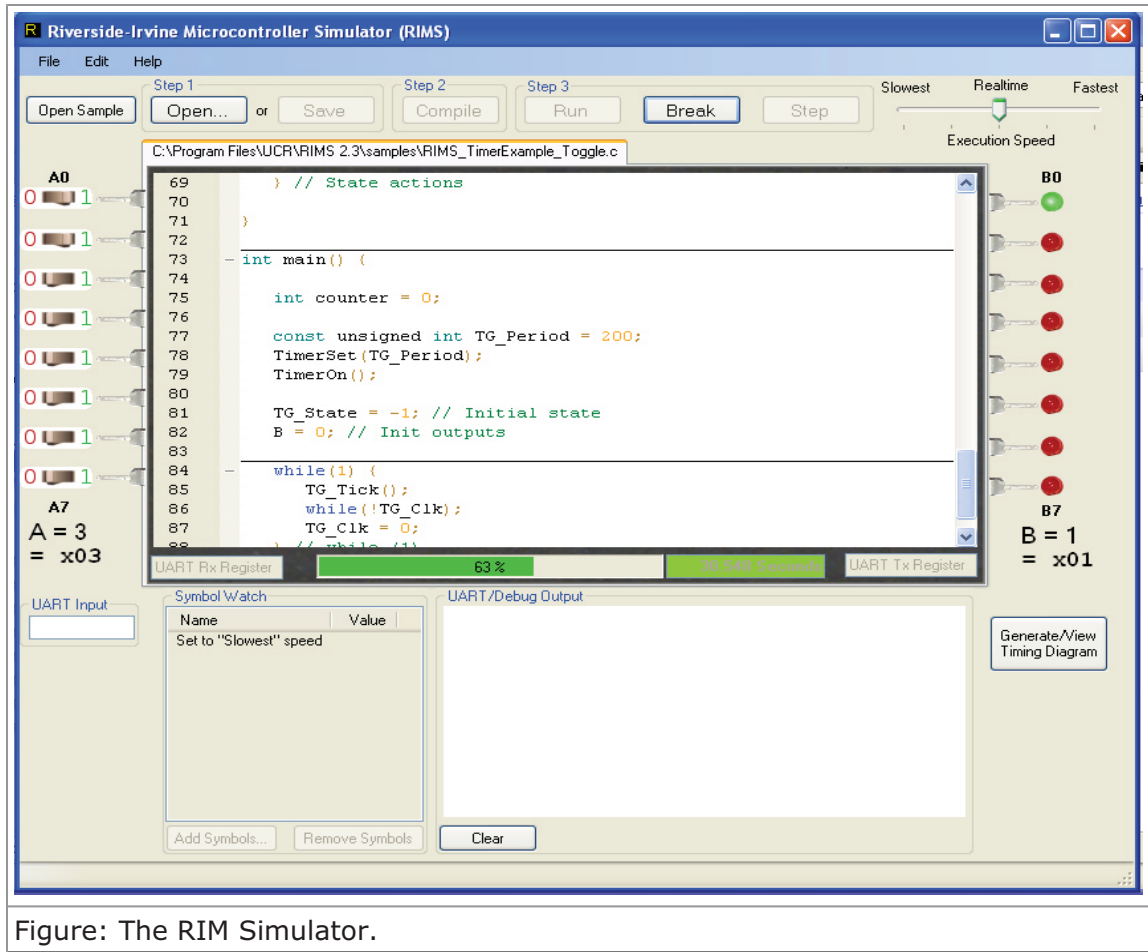


Figure: The RIM Simulator.

C code can be written in the center text box. "#include "RIMS.h" is required atop C files for RIMS. The user should first press the "Save" button to save the C code, then press "Compile" (to translate the C code to executable machine code, which is hidden from the user), then "Run". The running time number (light green box under the C code) shows how long the C program has run.

The user can click on the switches on the left to change each of RIM's eight input values to 0 or 1. RIM's eight output values, written by the C code, set the LEDs on the right each to green (for 1) or red (for 0).

The Execution Speed slider (upper right) can be moved left to slow execution; the "Slowest" setting causes an arrow to appear next to each C statement as it executes, and the user can click "Add Symbols" (bottom) to see the current value of any input, output, or variable in the C code. Pressing "Break" stops the run, and then "Step" executes one C statement; "Continue" resumes running.

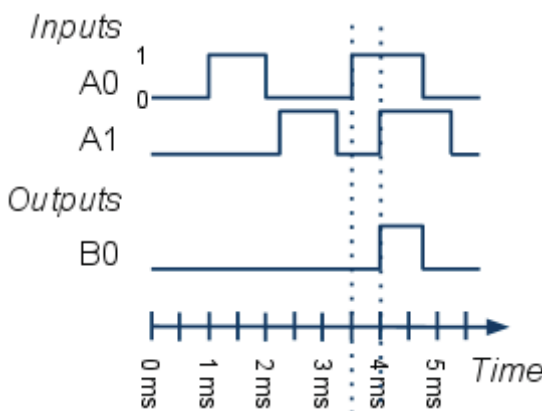
For debug/test, the C code can include print statements like: 'puts("Hello");'. Printed items appear in the Debug Output text box (bottom right).

Numerous samples that introduce features can be found by pressing "Open Sample" (upper left). Other features will be described later.

Do: Download and install RIMS. Write and run on RIMS the C programs you wrote earlier, testing each by setting the input switches and observing the output LEDs.

## Timing diagrams

An embedded system operates continually over time. A common representation of how an embedded system functions (or should function) is a timing diagram. A **timing diagram** ([Wikipedia: Digital Timing Diagram](#)) shows time proceeding to the right, and plots the value of bit signals as either 1 (high) or 0 (low). The figure below shows sample input values for the earlier embedded system [C example](#) that continually computes  $B0 = A0 \ \&\& \ A1$ . A0 is 0 from time 0 ms to 1 ms, when it changes to 1. A0 stays 1 until 2 ms, when it changes to 0. And so on. The 1 and 0 values are labeled for signal A0, but are usually implicit as for A1.



The timing diagram shows that B0 is 1 during the time interval when both A0 and A1 are 1, namely between 4 ms and 5 ms.

Vertical dotted lines are sometimes added to help show how items line up (as done above) or to create distinct timing diagram regions.

```

switch( x )
{
    case 0 : B = 0x77; break; // 0111 0111 (0)
    case 1 : B = 0x24; break; // 0010 0100 (1)
    case 2 : B = 0x5d; break; // 0101 1101 (2)
    //...
    case 9 : B = 0x6f; break; // 0110 1111 (9)
    default: B = 0x6b; break; // 0101 1011 (E for Error)
}
}

```

The remainder of the program is shown below.

```

int main()
{
    unsigned char temperature;
    while( 1 )
    {
        temperature = A;          // note that A's MSB is hardwired to 0
        temperature /= 10;        // only interested in the 10s
        display_7_seg(temperature); // send to display
    }
}

```

## Bit manipulation in C

### Operators

An important programming skill to be mastered by an embedded C programmer is the ability to manipulate bits within an integer variable. For example:

```

unsigned short x;
unsigned short r;

x = x | 0x04;          // set bit #2 of x to '1'
x = x & 0xf7;         // clear bit #3 of x to '0'
r = x & 0x10 ? 1 : 0; // get bit #4's value and store it in r

```

Note that the least significant bit (LSB) is always bit #0. Bit manipulation requires the use of bitwise operators and masks. **Bitwise operators** of C are **and** (&), **or** (|), **not** (~), and **xor** (^), not to be confused with the boolean operators of C, which are &&, ||, and ! (there is no boolean xor operator). The difference between bitwise and boolean operators is that the bitwise operators operate on individual bits in the operands, while boolean operators operate on the value of the operands. Thus, "(0x0f & 0xf0)" evaluates to 0 because 00001111 & 11110000 is 00000000 (e.g., the bits in location 7 are 0 and 1, whose ANDing yields 0; likewise for bit 6, bit 5, etc.), whereas "(0x0f && 0xf0)" evaluates to 1, because a non-zero value ANDed with a non-zero value is 1. Likewise, if unsigned char x has the value of 1, then ~x may be "11111110", while !x is just 0 (the "!" of any non-zero value becomes zero).

Try: Write a single C statement for RIMS that sets each output in B to the complement of the corresponding input A.

Shifting is sometimes necessary when manipulating bits. In C, the **left shift** operator, `<<`, returns a value obtained by shifting its left operand by the number of places indicated by its right operand, shifting 0s into the rightmost bits. Thus, if `X` is presently 00000110 in binary, then "`X << 2`" returns 00011000. The **right shift** operator, `>>`, behaves similarly. Thus, setting `B` to `A` with the two nibbles swapped, i.e., `B3-B0 = A7-A4` and `B7-B4 = A3-A0`, can be achieved by:

```
B = (A >> 4) | (A << 4);
```

## Masks

A **mask** is a constant value having a desired pattern of 0s and 1s, typically used with bitwise operators to manipulate a value. Above, "`0x04`", "`0xf7`", and "`0x10`" were masks. For example, to copy RIM's eight inputs to the eight outputs while forcing `B0` to 0, an appropriate mask would be `0xfe` (1111 1110), used as follows: "`B = A & 0xfe`";. This statement achieves the desired result because considering the 0th bit, any value in `A0` ANDed with 0 will yield 0. To copy RIM's inputs to outputs while forcing `B0`, `B1`, `B2`, and `B3` to 1s, the statement "`B = A | 0x0f`" could be used.

Try: Write a single C statement for RIMS that sets `B3-B0` to `A3-A0` and sets `B7-B4` to 0s.

Try: Write a single C statement for RIMS that sets `B3-B0` to `A5-A2` and sets other output bits to 0s.

Masks are sometimes defined as constant variables:

```
const unsigned char MaskLoNibls = 0x0F;
```

```
B = A | MaskLoNibls;
```

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

## Bit access functions

It can be useful to define C functions that perform common bit manipulation tasks.

To set the  $k$ 'th bit to 1 within an integer variable  $x$ , we construct the mask  $m$  containing a **1** in position  $k$  and 0s in all other positions. Then, we combine  $x$  with  $m$  using the bitwise OR (`|`) operator. Because any bit value ORed with 1 yields 1, then bit  $k$  of  $x$  becomes 1, while other bits remain unchanged because any bit value ORed with 0 yields the original bit value. The following C function performs the set to 1 operation. This function is designed for the unsigned char data type. Similar functions can be created for other integer types.

```
unsigned char set_bit1(unsigned char x, int k) {
    return (x | (0x01 << k));
}
```

The above code makes use of the left shift operator (`<<`) to construct the mask. When  $k$  is 0, the constant `0x01` (00000001) is unchanged. When  $k$  is 1, the constant 00000001 is shifted 1 position to the left, resulting in 00000010. And so on.

To instead set the  $k$ 'th bit to 0 within an integer variable  $x$ , we construct the mask  $m$  containing a **0** in position  $k$  and 1s in all other positions. Then, we combine  $x$  with  $m$  using the bitwise AND (`&`) operator. Because any bit value ANDed with 0 yields 0, bit  $k$  of  $x$  becomes 0, while other bits remain unchanged because any bit value ANDed with 1 yields the original bit value. The following C

## Chapter 3: Time-Ordered Behavior and State Machines

### Introduction

**Time-ordered behavior** is system functionality where outputs depend on the *order* in which input events occur. Consider a simple electronic lock with two inputs A1 and A0 coming from switches, and output B0. B0=0 locks a device, while B0=1 unlocks it. To unlock, a user must first set the switches such that A1A0 are 00, then 10, and 11. Any other sequence leading to 11, such as 00 then 01 then 11, does not unlock the device. The system has time-ordered behavior due to reacting to events ordered in time (such behavior is sometimes called **reactive** because it reacts to events). However, C was not designed for time-ordered behavior. Like most programming languages, C uses a sequential instruction computation model, which consists of a list of statements, and whose execution consists of continually executing instructions one after another, until the end of the statements is reached. That model is good for capturing algorithms that transform input data into output data, known as *data processing*. That model is less well suited for capturing time-ordered behavior.

Try: Capture the simple lock behavior using C (do not look at the below C code).

The following (less than ideal) C code strives to capture the above desired time-ordered behavior using sequential instructions:

```
void main()
{
    B0 = 0; // start with lock set
    while (1) {
        while (!(A1 && !A0)) {}; // wait for first unlock step 00
        while (!(A1) && (!A0)) {}; // wait while 00
        if (A1 && !A0) { // 10 is correct second unlock step
            while (A1 && !A0) {}; // wait while 10
            if (A1 && A0) { // 11 is correct third unlock step
                B0 = 1; // unlock
                while (A1 && A0) {}; // wait while 11
                B0 = 0; // lock again
            }
        }
    }
}
```

Although the code may work, one can begin to see why sequential instructions are not well-suited for time-ordered behavior. The while statements and nested if statements are awkward. Extending the C code for modified behavior could be difficult, as for the following.

Try: Extend the above code to sound an alarm by setting another output B1 to 1 if A1A0=11 is reached by any other sequence.

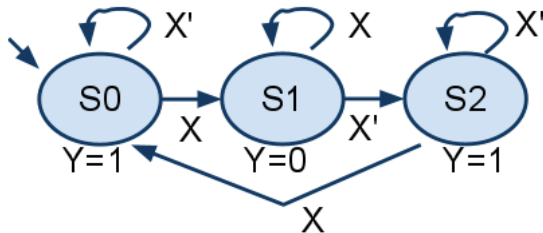
There are many different ways to extend the code for the above modified behavior. Any such extension makes the code much harder to understand. The lesson is this: Forcing time-ordered

behavior onto a sequential instruction computation model is challenging. Instead, a state machine computation model is better suited for time-ordered behavior.

## State machines

Various state machine computation models are used in different aspects of computing. Common features of **state machine models** are a set of single-bit inputs and outputs, a set of states with actions, a set of transitions with conditions, and an initial state. State machines are commonly drawn graphically, resulting in a *state diagram*. ([Wikipedia: State diagram](http://en.wikipedia.org/wiki/State_diagram)).

Inputs: X Outputs: Y



The above figure shows a state machine with input X and output Y; with three states S0, S1, S2 having actions Y=1, Y=0, and Y=1 respectively; with six transitions labeled either X or X'; and with the initial state being S0 as indicated by the arrow pointing from nothing to S0.

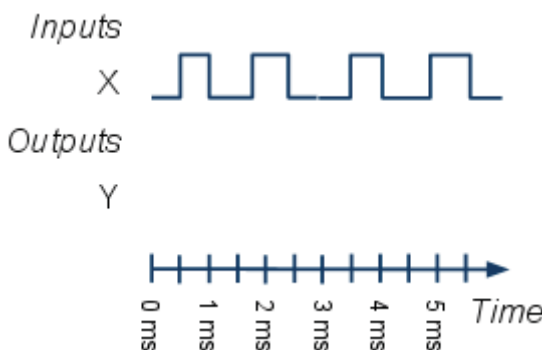
A system described by a state machine "executes" as follows. The system is always "in" some state, called the current state. Initially, the specified initial state is made the current state and its actions are executed. The following process, which we call a single **tick** of the SM, then occurs:

- The current state's outgoing transitions are checked to see which one has a true condition; one and only one should be true, else the state machine has not been properly described.
- The system's new current state is set to the state pointed to by that transition (which may be the same as the previous current state). That state's actions are then executed.

The process then repeats. Ticks are assumed to take a small but non-zero amount of time. SM ticks are assumed to occur at a much faster rate than events, such that no events are missed.

Thus, a system described by the above state machine initially starts in state S0 and sets Y=1. Assume input X is 0. On each tick while X is 0, the system's current state is set to S0 and the output Y is set to 1; this is called *staying* in state S0. When X changes to 1, then on the next tick the system's current state becomes S1, which sets output Y to 0. The system stays in S1 until X becomes 0 again, causing the current state to be set to S2, which sets Y to 1. When X changes to 1, the system's current state becomes S0 again.

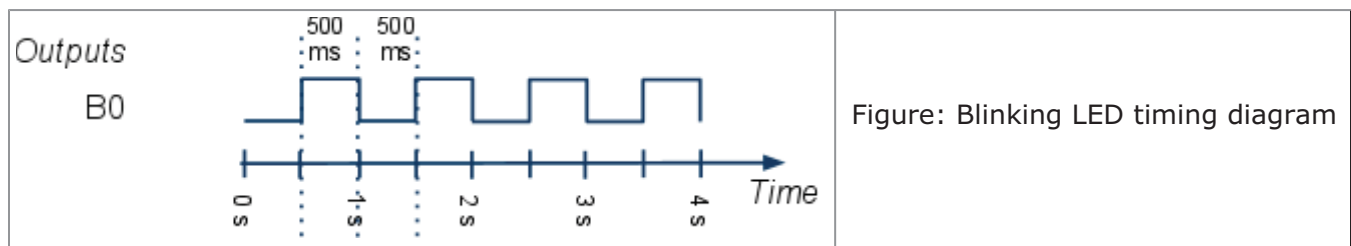
Try: Draw the Y signal for the given X signal and the above state machine.



# Chapter 4: Time Intervals and Synchronous SMs

## Introduction

In addition to time-ordered behavior, embedded systems commonly must carry out time-interval behavior. **Time-interval behavior** is system functionality where events must be separated by specified intervals of real time. Consider a system that should repeatedly blink an LED on for 500 ms and off for 500 ms. "500 ms" is a time interval. Time intervals have a magnitude (e.g., "500") and a real-time unit (e.g., "milliseconds" or "ms"). The following timing diagram illustrates the behavior, assuming B0 connects to the LED. Time intervals are commonly listed explicitly between vertical lines, as shown.

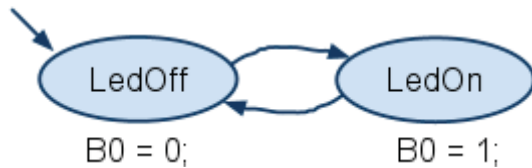


## Synchronous SMs

### Time intervals for outputs

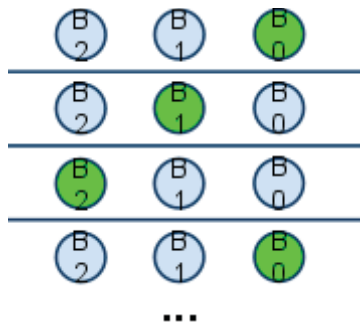
SMs can be extended to support time-interval behavior. In the previous chapter, the tick of an SM was assumed to take an infinitely small but non-zero amount of time. The tick rate can instead be set to a specific real-time rate such as 500 ms, known as the SM's **period**. An SM with a real-time tick period is called a **synchronous SM**, or **synchSM**. The blinking LED can be captured as a synchSM:

```
BlinkingLed
  Period: 500 ms;
```

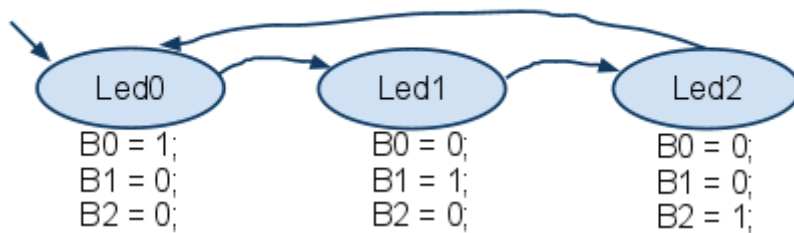


Entering a state causes (nearly) immediate execution of its actions, after which the system waits for 500 ms, due to the 500 ms tick period, before evaluating transitions and entering the next state.

Another example is a system that lights one of three LEDs connected to B0, B1, and B2, one LED per second in sequence, such that the lit LED appears to move (and wrap around). Such a system might be found in a highway construction sign, indicating that traffic should move to the left.



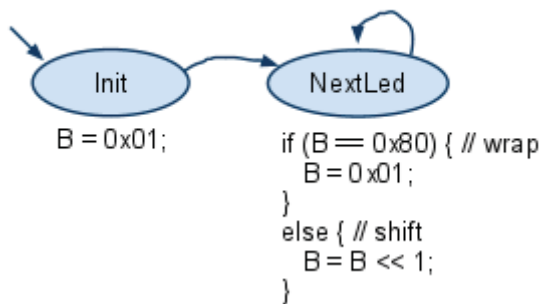
ThreeLeds  
 Period: 1000ms;



Try: Create a festive light display system that works by controlling eight electric sockets (B7-B0), into each of which a light strip may be plugged in. When activated (A0=1), the system generates the following patterns for 1 second each: 00000000, 11111111, 11110000, 00001111, repeat. When deactivated, the system turns off all sockets within one second.

The ThreeLeds synchSM could be extended for eight LEDs by using eight states, but a better approach makes use of bit manipulation methods (see Chapter 2) as follows:

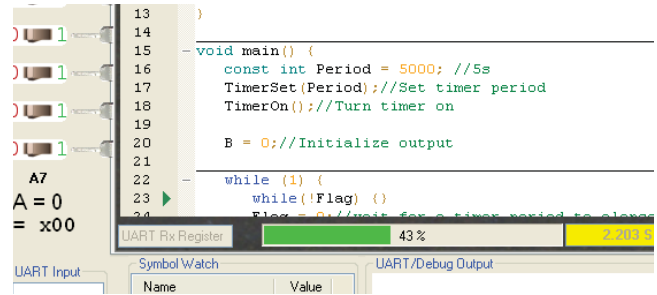
SequenceLeds  
 Period: 1000 ms;



Note how the "1" bit moves from B0 to B1 to B2 ... to B7, once per second, wrapping back to B0 again.

Try: A festive light display controls 8 light bulbs (B7-B0). A0 activates the system. A1 chooses a mode. When A1=0, the lights blink all-on and all-off 1 second each. When A1=1, the lights give the illusion of a ball bouncing back and forth: 10000000, 01000000, ..., 00000001, 00000010, ..., 10000000, repeat. Use bit manipulation methods for the bouncing ball mode, rather than using a separate state for different output combinations.

RIMS (see "Figure: The RIM Simulator" from Chapter 1) graphically animates the timer using a rectangle under the C code. As time passes, the percentage of the timer's period that has passed is displayed ("43%" in the figure on the right) and a green bar fills the timer rectangle. When one timer period has passed and a timer tick thus occurs, RIMS calls TimerISR().



Running the above C code, a user can view the timer behavior by pressing "break" and then pressing "step" repeatedly, noting that the current instruction arrow stays at the "while (!TimerFlag)" statement until the timer period passes, at which point the current statement automatically changes to the TimerISR (the user may have to scroll up to see the current statement), which sets TimerFlag=1. When TimerISR reaches its end, the user will note that the current statement automatically jumps back to the "while (!TimerFlag)" statement; because TimerFlag is now 1, execution then proceeds past the while statement.

Try: Run the above C code on RIMS. Observe the timer display under the C code, showing the timer counting up to the timer period. Use break and step buttons to see how the ISR is called. Use RIMS' symbol watch to watch the value of the TimerFlag variable.

## Converting a synchSM to C on a microcontroller with a timer

A synchSM can be translated to C code for a microcontroller with a timer. The code is similar to that for an SM, with additional code to initialize and start the microcontroller timer to the synchSM's period, and code that ensures that the synchSM's tick C function is only called when the timer ticks via use of a flag. The following shows C code for the BlinkingLed (BL) example.

```

unsigned char TimerFlag; // raised by ISR, lowered by main code

void TimerISR() {
    TimerFlag = 1;
}

enum BL_States { BL_LedOff, BL_LedOn } BL_State;

void BL_Tick() {

    switch( BL_State ) { //transitions
        case -1:
            BL_State = BL_LedOff; //Initial state
            break;
        case BL_LedOff:
            BL_State = BL_LedOn;
            break;
        case BL_LedOn:
            BL_State = BL_LedOff;
            break;
    }

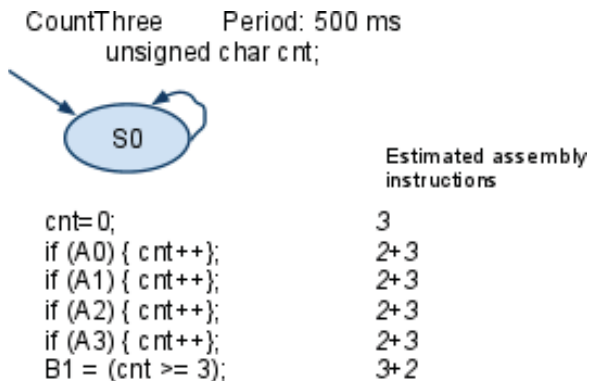
    switch (BL_State) { //actions
        case BL_LedOff:

```

Automatically-detected incorrect execution is known as an **exception**. We call the above a **timer-overflow exception**. The programmer decides how to handle the exception depending on the application. The programmer may output an error message or turn on an output "error" LED. The programmer may modify the system by lengthening SM periods, decreasing SM actions, decreasing the number of SMs, etc. In some systems, the programmer may have the system automatically restart itself. In other systems, the 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-overflow 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:

$$\text{Utilization} = (\text{time executing} / \text{total time}) * 100\%$$

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 (!timerFlag);"); 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:

CountThree    Period: 500 ms  
 unsigned char cnt, i;



*Estimated assembly instructions*

cnt = 0;	3
for (i=0; i<4; i++) {	+ 3 + 4*(2 + 3
if (get_bit(A, i)) {	+ 2 + ?
cnt++;	+ 3
}	)
}	
B 1 = (cnt >= 3);	+ 3 + 2;

For a *for* loop, the analysis should include the loop initialization ("i=0": 3 instrs), plus the loop control instructions ("i<4", and "i++", or 2 + 3), and should also multiply the instructions-per-loop-iteration by the number of iterations. The above loop iterates 4 times. If the number of loop iterations is data dependent, an upper bound on the number of iterations should be used.

For a function call, the analysis should determine the instructions executed within the function. Above, the number of instructions for the call to function `get_bit` (defined in Chapter 2) is listed as "?". Examining the statements within the `get_bit` function itself, we might estimate 10 instructions.

For the *if* statement, we must consider the *worst case*, which for this statement would mean the branch is taken and thus "cnt++" is executed. In general, in the presence of branches (if-else statements), we must consider the maximum number of instructions that might be executed for any values of the branch conditions.

Thus, the total worst-case number of assembly instructions that execute for S0's actions are  $3+3+4*(2+3+2+10+3) + 3+2 = 51$  instrs. On a microcontroller M that requires 0.00125 sec/instr, the worst case execution time for those instructions is  $51 \text{ instr} * 0.00125 \text{ sec/instr} = 63.75 \text{ ms}$ . Again considering a 500 ms window, the utilization is  $63.75 \text{ ms} / 500 \text{ ms} = 12.75\%$ .

As should be clear from above, the **worst-case execution time** (or **WCET**) of a synchSM task is determined as the time to execute the worst-case number of instructions for any possible tick of the synchSM. ([Wikipedia: WCET](#)) WCET is the value of concern regarding timer overrun.

For a task consisting of a multi-state synchSM, the analysis requires determining the worst-case number of instructions among all states, and then using the state having the largest possible number of instructions as the WCET. For example, consider the CountThree synchSM written using two states: