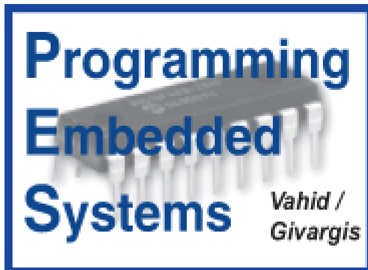


Programming Embedded Systems

An Introduction to Time-Oriented Programming

Version 3.0

Frank Vahid
University of California, Riverside

Tony Givargis
University of California, Irvine

UniWorld Publishing, Lake Forest, California

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

Version 3.0, January, 2012.

Formatted for electronic publication.

Copyright © 2012, 2011, 2010 Frank Vahid and Tony Givargis. All rights reserved.

ISBN 978-0-9829626-2-6 (e-book)

UniWorld Publishing

www.programmingembeddedsystems.com
info@programmingembeddedsystems.com

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

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

Copyright © Frank Vahid and Tony Givargis 2011

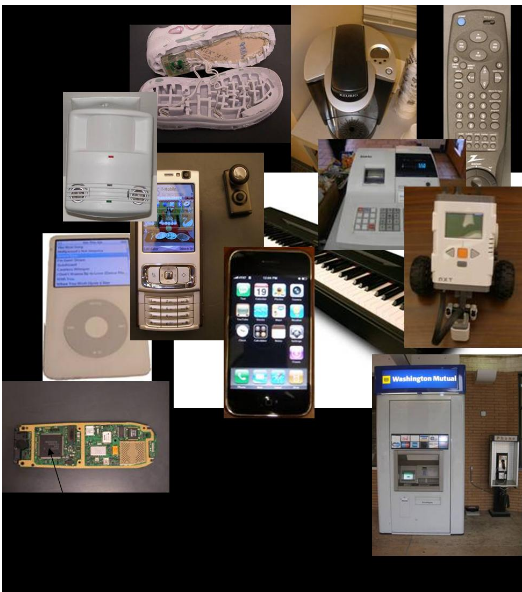
Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

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.

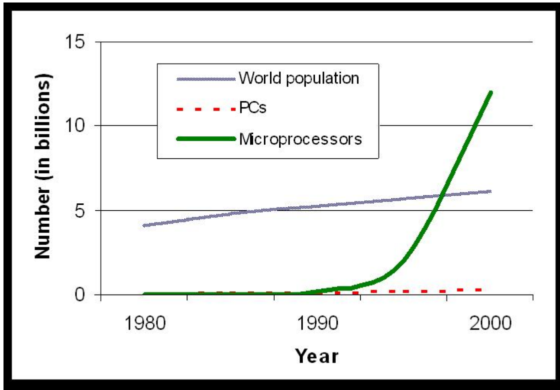
Try: List three embedded systems that you interact with regularly.

(Wikipedia: Iphone Wikipedia: PlayStation Wikipedia: SetTopBox Wikipedia: Engine_Control_Unit Wikipedia: HVAC_Control_System Wikipedia: IP_Phone Wikipedia: Flight_Control_System Wikipedia: Amazon_Kindle)

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

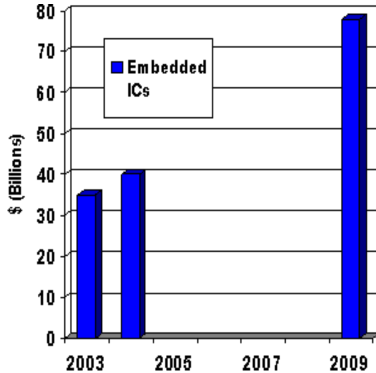
Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu



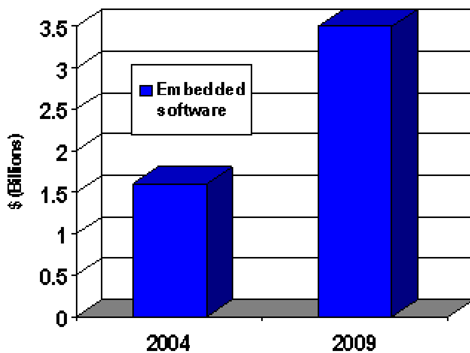
The microprocessors produced per year is growing exponentially, mostly destined for embedded systems.

Source: Study of Worldwide trends and R&D programmes in Embedded Systems by FAST GmbH. ftp://ftp.cordis.europa.eu/pub/ist/docs/embedded/final-study-181105_en.pdf



Gross sales of ICs destined for embedded systems is growing each year.

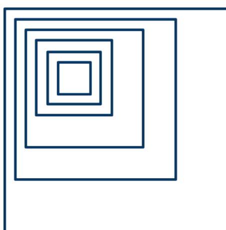
Source: BCC, Inc.



Gross sales of embedded software is also growing rapidly.

Source: BCC, Inc.

Integrated circuits (a.k.a. ICs or chips), on which microprocessors are implemented, have been doubling in transistor capacity roughly every 18 months, a trend known as ***Moore's Law*** (Wikipedia: Moore's Law). Such doubling means: (1) that a same-size system (e.g., a cell phone) gets more capable, and (2) that a same-capability system can be made smaller (halved every 18 months) thus enabling new inventions (e.g., computerized pills that can be ingested) (Wikipedia: Motes).



IC size shrinking in half every 18 months; note the reduction after just 5 * 18 months = 90 months or 7.5 years.

Try: Fold a sheet of paper in half as many times as you can. Each fold corresponds to IC size shrinking in 18 months. Notice how size shrinks dramatically after just a few folds.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

Try: Think of a new invention that would be enabled by a microprocessor that is the size of a speck of dust, self-powers for years, has ample memory, and can communicate wirelessly.

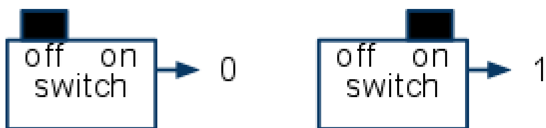
Basic components

A system with electrical components uses wires with continuous voltage signals. A useful abstraction is to consider only two voltage ranges, a "low" range (such as 0 Volts to 0.3 Volts) that is abstracted to 0, and a "high range" (such as 0.7 Volts to 1.2 Volts) that is abstracted to 1. A ***bit*** (short for "binary digit") is one digit of such a two-valued item. A bit that can change over time is called a digital ***signal***. "Digital" refers to the signal having discrete rather than continuous possible values. (Wikipedia: Digital signal).

Switch and push button

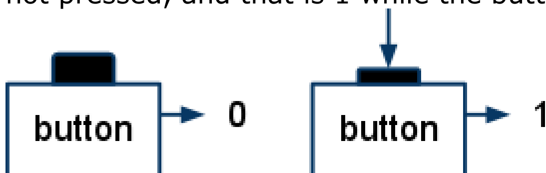
A switch is an electromechanical component with a pair of electrical contacts. The contacts are in one of two mechanically controlled states: closed or open. When closed, the contacts are electrically connected. When open, the contacts are electrically disconnected. (Wikipedia: Switch).

In digital system design, it helps to think of an abstraction of a switch. A switch is a component with a single bit output that is either a 0 or 1 depending on whether the switch is in the off or on position.



A push button operates similar to a simple switch, having a pair of electrical contacts and two mechanically controlled states: *closed* or *open*. Unlike a simple switch, the push button enters its closed state when it is being *pressed*. The moment the pressing force is removed, the push button reverts to and remains in its open state. (Wikipedia: Push_button).

An abstraction of a push button is a component with a single bit output that is 0 when the button is not pressed, and that is 1 while the button is pressed.



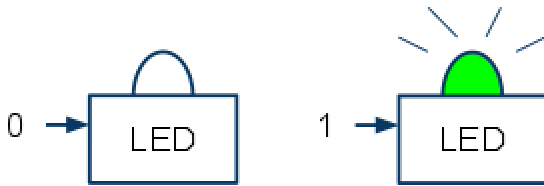
LED

A light emitting diode (LED) is a semiconductor with a pair of contacts. When a small electrical current is applied to the LED contacts, the LED illuminates. (Wikipedia: LED).

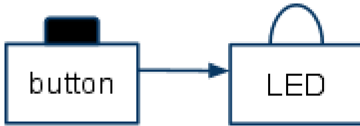
An abstraction of an LED is a component with a single bit input that can be either 0 or 1. When the input is 0, the LED does not illuminate. When the input is 1, the LED illuminates.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu



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 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.

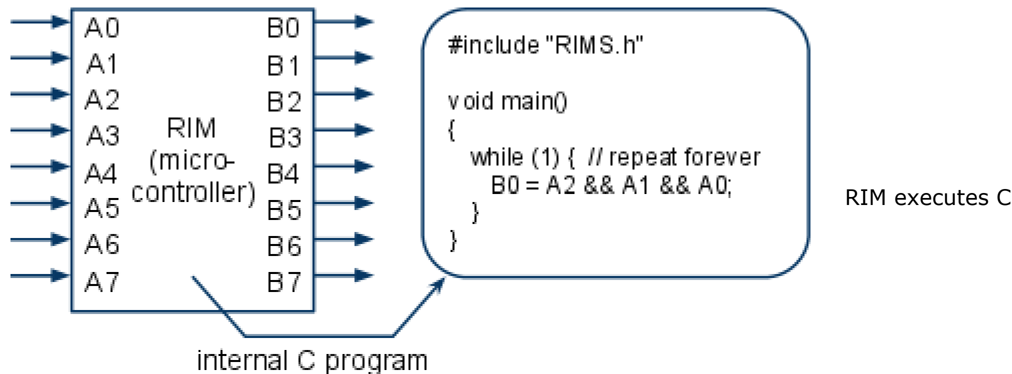
Microcontroller

A ***microcontroller*** is a programmable component that reads digital inputs and writes digital outputs according to some internally-stored program that computes. 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.

A "PIC" microcontroller

(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 (this book assumes reader proficiency with C programming).



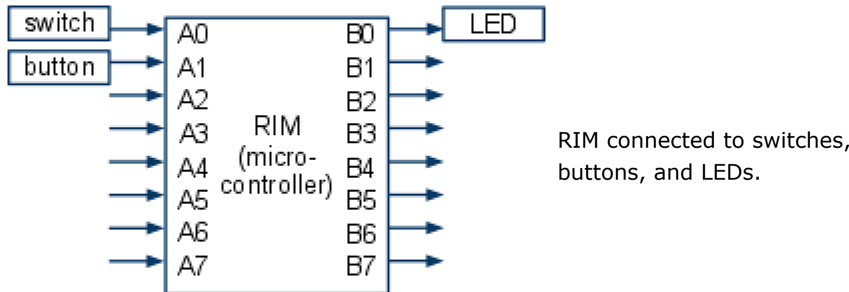
RIM executes C

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

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

for embedded systems and is called an ***infinite loop***, causing the contained statements to repeat continually.

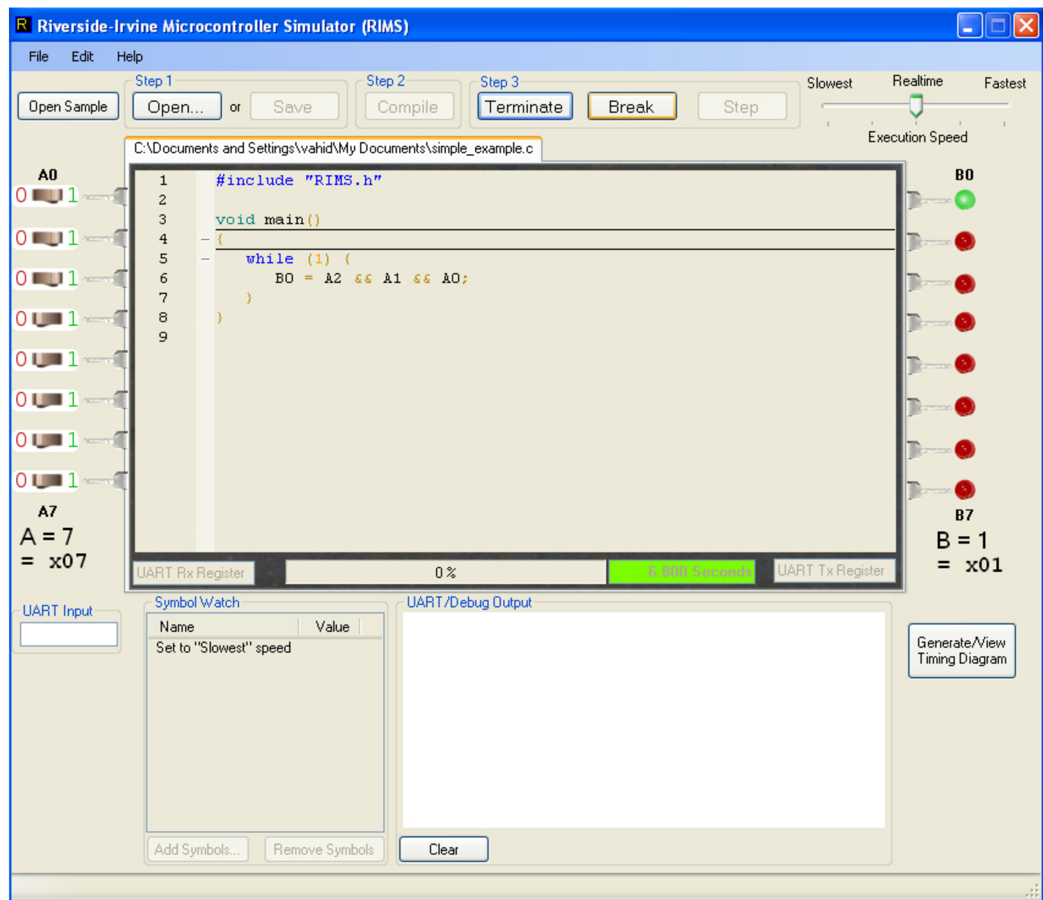
We can use a microcontroller to add functionality to the earlier simple system to create an embedded system. The term embedded system, however, commonly refers just to the compute component. The switch and buttons are examples of ***sensors***, which convert physical phenomena into digital inputs to 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)



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.



C code can be written in the center text box. *#include "RIMS.h" is required atop all C files for RIMS*. The user can first press the "Save" button to save the C code, then press "Compile" (to translate the

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

C code to executable machine code, which is hidden from the user), then press "Run" (after which the button name changes to "Terminate" as in the above figure).

While the program is running, 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 running C code, set each LED on the right to green (for 1) or red (for 0). When done, the user should press "Terminate".

Try: Download, install, and execute RIMS (see www.programmingembeddedsystems.com). Note that a default C program appears in the center text box. Replace the default C program by the program below. Press "Save" and name the file "example1.c", then press "Compile", press "Run", and then click switches for A2, A1, and A0 all to 1, noting that B0 becomes 1. Press "Terminate" when done.

```
#include "RIMS.h"

void main()
{
   while (1) {
      B0 = A1 && A0;
   }
}
```

Try: Write a C program for RIM that sets B0=1 whenever the numbers of 1s on A2, A1, and A0 is two or more (i.e., when A2A1A0 are 011, 110, 101, or 111). Run the program in RIMS to test the program.

Pressing "Break" temporarily stops the running (and the button changes to "Continue") and shows an arrow next to the current C statement, and then each press of "Step" executes one C statement. Pressing "Continue" resumes running. Pressing "Terminate" ends the program, and re-enables editing of the C code. A box under the C code shows how many seconds the C program has run.

Try: For the above program named "example1.c", set A1A0 to 00, run the program, and press "Break". Press "Step" 5 times and observe the arrow pointing to the current statement after each press. Now change A1A0 to 11, and then press "Step" several times until B0 changes. Press "Continue" to resume running. Press "Terminate" to end the running.

RIMS' Execution Speed slider (upper right) can be moved left to slow running speed; the "Slowest" setting causes an arrow to appear next to each C statement as it executes.

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

The user can click "Add Symbols" (at bottom of RIMS) to see the current value of any input, output, or global variable in the C code.

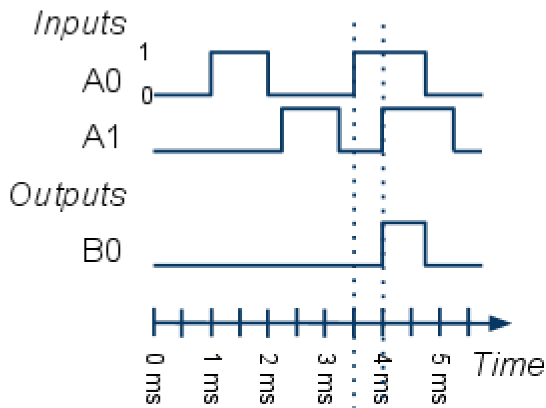
Numerous samples that introduce features can be found by pressing "Open Sample" (upper left). Other features will be described later.

Timing diagrams

An embedded system operates continually over time. A common representation of how an embedded system operates (or should operate) 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 above example.c program that continually computes B0 = A1 && A0. 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.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu



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.

Try: Draw a timing diagram showing all possible combinations of three single-bit input signals A0, A1, and A2. Use vertical dotted lines to delineate each combination.

Try: A program should set B0 to 1 if exactly two of A0, A1, and A2 are 1. Draw a timing diagram illustrating this behavior.

A change from 0 to 1 or from 1 to 0 on a bit signal is called an ***event.*** The above figure has 10 events. If a signal changes from 0 to 1, the event is called ***rising***. 1 to 0 is called ***falling***.

Try: Circle all 10 events in the above timing diagram.

Testing

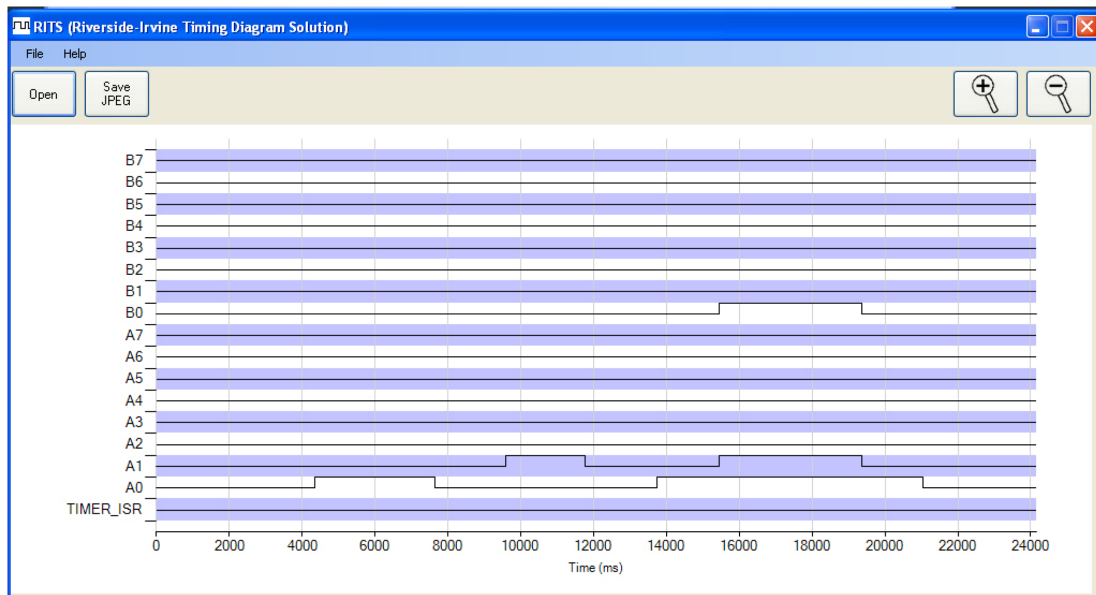
Written code should be tested for correctness. One method is to generate different input values and then observe if output values are correct. To test code implementing "B0 = A0 && !A1", **all possible input value combinations** of A1 and A0 can be generated: 00, 01, 10, and 11. Using RIMS, switches 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).

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu



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.

The saved text file is an industry standard vcd (value change dump) file (Wikipedia: Value Change Dump file). A vcd file can also be read by many other timing diagram tools. RITS can also be run on its own, and can open vcd files generated by other tools.

Ideally, input value combinations, known as ***test vectors***, could be captured in a file and then input to a tool rather than each input value combination being generated by clicking on switches. For tool simplicity and ease of use, RIMS does not presently support such file input.

Try: Save, compile, and run example1.c from above. Click the input switches to achieve the following values: A1A0=00, then 01, then 00, then 10, then 00, then 01, then 11, then 00. Press "Break", then press the "Generate/View Timing Diagram" button, causing a timing diagram window to appear, and observe how the timing diagram corresponds to the output values you observed just prior (and should closely match the above RITS figure). Press RITS' "Save JPEG" button to save the timing diagram to a JPEG file. Open the saved JPEG file using a picture viewing tool (not included with the RI tools). Finally, back on RIMS, press the "Terminate" button.

RIMS.h provides three functions for printing to the "Debug output" text window:

- puts(x): Prints a string x. Example:
  - `puts("Hello.\n");`
- putc(y): Prints a character y. Example:
  - `putc('H');`
- puti(z): Prints an integer z. Example:
  - `puti(209);`

Exercises

1. Write RIM C code that sets B0 to 1 only if A0-A3 are all 1s or if A4-A7 are all 1s (or if both situations are true). Using border and sample input value combinations, test the written code with RIMS, and generate a timing diagram showing the test results.

2. A car has a sensor connected to A0 (1 means the car is on), another sensor connected to A1 (1 means a person is in the driver's seat), and a sensor connected to A2 (1 means the seatbelt is fastened). Write RIM C code for a "fasten seatbelt" system that illuminates a warning light (by

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

setting B0=1) when the car is on, a driver is seated, and the seatbelt is *not* fastened. Test the written code with RIMS for all possible input combinations of A2, A1, A0, and generate a timing diagram showing the test results.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

Chapter 2: Bit-Level Manipulation in C

C is a popular programming language in embedded systems due to its simplicity and efficiency, but C was not originally created for embedded systems. C was created in 1972 for mainframe and desktop computers, which typically manipulate data in files such as integer or character data. In contrast, embedded systems commonly manipulate bits, in addition to other data types. This chapter describes C's built-in data types and discusses how to manipulate bits in C (Wikipedia: C language). Most of the discussion applies equally to the C++ language.

Language	% of embedded systems programmers using language for most of current project (2006)
C	51%
C++	30%
Assembly	8%
Java	3%
BASIC	1%

Source: www.embedded.com, 2006 State of Embedded Market Survey.

C data types

Several C data types are commonly used to represent integers in embedded system programs:

Type	Width	Range	Notes
signed char **unsigned char**	8 8	-128 to +127 0 to 255	
signed short **unsigned short**	16 16	-2^{15} to $+2^{15}-1$ 0 to $+2^{16}-1$	2^{16} is 65,536
signed long **unsigned long**	32 32	-2^{31} to $+2^{31}-1$ 0 to $+2^{32}-1$	2^{32} is about 4 billion
~~*signed int*~~ ~~*unsigned int*~~	*N* *N*	-2^{N-1} to $+2^{N-1}-1$ 0 to $2^{N}-1$	*Though commonly used, we avoid these due to undefined width*

Thus, a variable whose value may only range from 0 to 100 might be best declared as "unsigned char". A variable whose value may only range from -999 to 999 might be best declared as "signed short". Note that "unsigned" data types represent positive integers, while "signed" types represent negative and positive integers. If a variable is used to represent a series of bits (rather than a number), then an unsigned type should be used.

Embedded systems commonly deal with **1-bit** data items. C does not have a 1-bit data type, which is unfortunate. Thus, 1-bit items are typically represented using an unsigned char, e.g., "unsigned char myBitVar". The programmer only assigns the variable with either 0 or 1, e.g., "`myBitVar = 1;`",

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

even though the variable could be assigned integers up to 255. Checking whether such a variable is 1 or 0 is typically done without explicit comparison to 1 or 0, and is instead done as "`if (myBitVar)`" or "`if (!myBitVar)`".

Below are some example variable declarations:

```
unsigned char  ucI1;
unsigned short usI2;
signed   long  slI3;
unsigned char  bMyBitVar;
```

A common practice is to name variables with a lower-case prefix indicating the data type, as above -- uc, us, and ul for unsigned char, short, and long; sc, ss, and sl for signed char, short, and long; or b for bit -- to help ensure that larger constants or variables aren't assigned to smaller variables. For clarity of short examples, this book often skips that practice, but programmers of larger programs should consider it.

char is called such because it is commonly used in desktop programming to represent the integer value of an 8-bit ASCII *character*. Note however that char is actually an integer. Also, 8-bits is sometimes called a *byte*.

In C, the word "signed" is optional for a signed type, so "char I1" is the same as "signed char I1". However, for program clarity, we avoid that shortcut. Also, the word "int" may follow the words short or long, but that word is superfluous so we usually omit it.

Unfortunately, although the above widths are quite common, C actually defines the above widths as *minimum* widths, so a compiler could for example create a long as 64 bits. Thus, a programmer should never assume an exact width, e.g., a program should not increment an "unsigned char" and expect it to roll over from 255 to 0, because the char could be 16 bits. Another unfortunate fact is that C allows a variable to be declared merely as type "int", where the width is compiler dependent. *Due to the unpredictability of int, we avoid using the int type entirely*. Following these conventions improves code ***portability***, which is the ability to recompile code for a different microprocessor without undesirable changes in program behavior.

The underlying representation of each data type is binary. For an 8-bit unsigned char ucI1:

```
ucI1 = 1;   // underlying bits will be 00000001
ucI1 = 12;  // underlying bits will be 00001100
ucI1 = 127; // underlying bits will be 01111111
ucI1 = 255; // underlying bits will be 11111111
```

In binary, the rightmost bit has weight 2^0, the next bit 2^1, then 2^2, etc. In other words, from left to right, the 8 bits have weights 128, 64, 32, 16, 8, 4, 2, and 1. 0001100 is thus 8 + 4 = 12.

Signed data types in C use two's complement representation. At the bit level, a variable of type *char* set to 127 would have an internal representation of 01111111, while -128 would be 10000000, and -1 would be 11111111. For the curious reader -- the binary representation of a negative number in two's complement can be obtained by representing the number's magnitude in binary, complementing all the bits, and adding 1. For example, -1 is 00000001 (magnitude is 1) --> 11111110 (complement all bits) --> 11111110+1 = 11111111 (add 1). -128 is 10000000 (magnitude is 128) --> 011111111 (complement all bits) --> 011111111+1 = 100000000 (add 1). Note that the eighth bit will always be 1 for a negative 8-bit number and is thus called the *sign bit*. The programmer generally need not deal directly with the binary representations of signed numbers,

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

because compilers/assemblers automatically create the proper constants (e.g., "myVar = -1;" would result in the two's complement representation being stored in myVar). However, knowing whether an item is signed or unsigned is important when assigning values, and for determining a variable's range. (Wikipedia: Two's complement)

In RIMS, the microcontroller's inputs and outputs are implicitly defined as global variables "unsigned char A0;", "unsigned char A1;", ..., "unsigned char B0;", etc. As mentioned above, an item intended to represent a single bit, such as B0 in RIMS, should be set to only 0 or 1, e.g., "B0 = 1;".

RIMS also implicitly defines two additional global variables A and B:

```
unsigned char A; // built-in variable A, representing RIM's 8 input pins
                 // as a single 8-bit variable

unsigned char B; // built-in variable B, representing RIM's 8 output pins
                 // as a single 8-bit variable
```

Thus, setting all of RIM's outputs to 1s can be accomplished by one statement, "B = 255;" (255 is "11111111" in binary), rather than the eight statements: "B0 = 1; B1 = 1; ...; B7 = 1;" To set RIM's outputs to the number 7 in binary (00000111), the statement "B = 7;" could be used. To set RIM's outputs to RIM's inputs, the statement "B = A;" could be used. Such grouping of bits is uncommon in desktop programming, but quite standard in C environments for microcontrollers.

Because B is a global variable, a program can write as well as read that variable. However, A is automatically written by the microcontroller and should never be written by a program, only read. In RIMS, writing to A results in a runtime error, causing execution to terminate.

Try: Write a C program in RIMS that repeatedly executes "B = 7;" Note that outputs B2, B1, and B0 become 1s, because 7 is 00000111 in binary. Set the input switches such that A3=1, A2=0, A1=0, and A1=1, with the other inputs 0, and note below the input pins that RIMS indicates the value of A to be 9, because 00001001 is 9 in binary.

Try: Write a C program for RIMS that sets B equal to A plus 1.

Try: Write a C program for RIMS that sets B = 300. Note that the value actually output on B is not 300, because an unsigned char's range is 0 to 255.

Variables can be *initialized* when declared, e.g., "unsigned char i1 = 5;".

The keyword "const," short for constant, can precede any variable declaration, e.g., "**const unsigned char i1 = 5;**". A ***constant variable***'s value cannot be changed by later code. A constant variable *must* therefore be initialized when declared. Above, 5 is a constant, and i1 is a constant variable.

Try: A car has a sensor that sets A to the passenger's weight (e.g., if the passenger weighs 130 pounds, A7..A0 will equal 10000010). Write a RIM C program that enables the car's airbag system (B0=1) if the passenger's weight is 105 pounds or greater. Also, illuminate an "Airbag off" light (by setting B1=1) if weight > 5 pounds but weight < 105 pounds.

Hexadecimal

Commonly an 8-bit unsigned item isn't used as a number but rather just a eight distinct bits. For example, if RIM's eight outputs connected to eight light bulbs and we wanted to light all the bulbs, we could write "B = 255;" (because 255 is 11111111 in binary), but "255" does not directly convey our intent. Ideally, we could write "B = b11111111;" or something similar, but C unfortunately has

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

no binary constant support. But fortunately, C does support hexadecimal constants, which are close to the ideal.

Hexadecimal (or "***hex***") is a base 16 number, where each digit can have the value of 0, 1, ..., 8, 9, A, B, C, D, E, or F. A is ten, B is eleven, C is twelve, D is thirteen, E is fourteen, and F is fifteen. Hex values have the following binary representations: **0:0000, 1:0001, 2:0010, 3:0011, 4:0100, 5:0101, 6:0110, 7:0111, 8:1000, 9:1001, A:1010, B:1011, C:1100, D:1101, E:1110, F:1111**. In the C language, a hex constant is preceded by *"0x"*. Thus, "0xFF" represents "11111111" in binary. *Each hex digit corresponds to four bits* (four bits is called a ***nibble***). "0xff" may also be used; hex constants are not case sensitive. (Wikipedia: Hexadecimal)

Thus, "B = 0xFF;" sets all RIM outputs to 1s. The intent of 0xFF is clearer than 255. Likewise, "B = 0xAA" sets the outputs to 10101010, having clearer intent than "B = 170;".

Try: Write a single statement for RIM that sets B7-B4 to 1s and B3-B0 to 0s, using a hex constant.
Try: Write a single statement for RIM that sets B0 to 1 if all eight A inputs are 1s, using a hex constant.

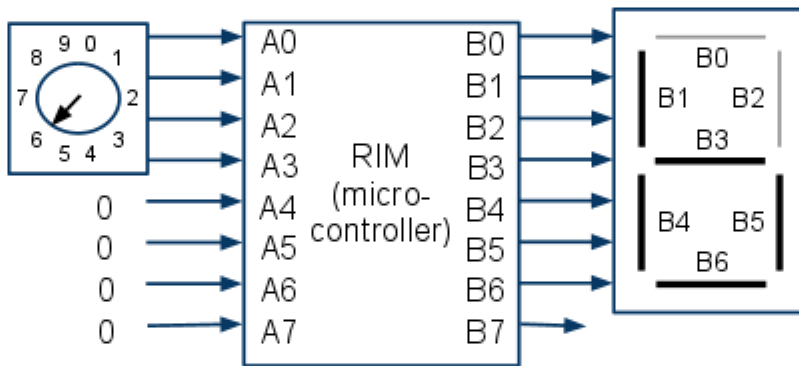
Example: The following program sets B to 00000000 when A1A0=00, to 01010101 when A1A0=01, to 10101010 when A1A0=10, and to 11111111 when A1A0=11:

```
#include "RIMS.h"

/* Set B to 00000000 when A1A0=00, to 01010101 when A1A0=01,
   to 10101010 when A1A0=10, and to 11111111 when A1A0=11 */
void main()
{
   while (1) {
      if (!A1 && !A0) {
         B = 0x00; // 0000 0000
      }
      else if (!A1 && A0) {
         B = 0x55; // 0101 0101
      }
      else if (A1 && !A0) {
         B = 0xAA; // 1010 1010
      }
      else if (A1 && A0) {
         B = 0xFF; // 1111 1111
      }
   }
}
```

Example: Consider the following embedded system with a dial that can set A3..A0 to binary 0 to 9, and a 7-segment display (Wikipedia: 7-Segment Display) connected to B6..B0 as shown:

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu



Below is a (partial) RIM C program that appropriately sets the display for the given dial position:

```
#include "RIMS.h"

void main()
{
   while (1) {
      switch( A )
      {
         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)
      }
}
```

Try: Complete the above program using hex constants to produce the correct display for dial settings 3..8. Test in RIMS.

Bitwise operators and masks

An important programming skill for an embedded C programmer is manipulating bits within an integer variable. For this, C's bitwise operators are needed:

- ***& : bitwise AND***
- ***| : bitwise OR***
- ***^ : bitwise XOR***
- ***~ : bitwise NOT***

Bitwise operators operate on the operands' corresponding bits, as shown:

```
0x0F & 0xF2:     0x0F | 0xF2:    0x0F ^ 0xF0:        ~0x0F:
  00001111         00001111        00001111        00001111
& 11110010       | 11110010      ^ 11110010      ~
-----------      -----------     -----------     -----------
  00000010         11111111        11111101        11110000
```

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

Try: Compute "0xAA *OP* 0x33" for OP being &, then |, then ^.

In contrast, Boolean operators &&, ||, and ! (there is no Boolean XOR operator) treat operands as zero (false) or non-zero (true). So "0x0F & 0xF0" (bitwise AND) evaluates to 0 because each AND of corresponding operand bits evaluates to 0, whereas "0x0F && 0xF0" (Boolean AND) evaluates to 1 because both operands are non-zero. Likewise, if unsigned char x has the value of 1, then ~x is "11111110", while !x is just 0 (the "!" of any non-zero value becomes zero).

Below are some examples of using bitwise operators:

```
B = ~A; // Invert each bit in A, e.g., 00001111 becomes 11110000
B = A & 0x0F; // Sets B3..B0 to A3..A0, and B7..B4 to 0000
B = A | 0xF0; // Sets B3..B0 to A3..A0, and B7..B4 to 1111
B = A & 0xF7; // Sets B to A, except that bit #3 is cleared to 0
B = A | 0x04; // Sets B to A, except that bit #2 is set to 1
x = x | 0x04; // For unsigned char x, sets bit #2 to 1

if (A & 0x03) { // Sets B0 to 1 if A1A0 are 11
   B0 = 1;
else {
   B0 = 0;
}
```

(Note: bit #0 refers to the least significant bit, or *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:

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

```
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.

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

Example: The following program treats A7..A0 as one 4-bit binary number and A3..A0 as another 4-bit binary number, and outputs the sum of those two numbers on B:

```
#include "RIMS.h"

const unsigned char LoNib1s = 0x0F;
const unsigned char HiNib1s = 0xF0;

unsigned char op1;
unsigned char op2;

void main()
{
   while (1) {
      op1 = A & LoNib1s; // 0000a3a2a1a0
      op2 = (A & HiNib1s) >> 4; //a7a6a5a40000 --> 0000a7a6a5a4
      B = op1 + op2;
   }
}
```

Try: Run the above program. Press "Break", then add symbols op1 and op2 to symbols being watched. Press "Continue" and then move the speed slider to "Slowest." Now set values on A3..A0 and A7..A4 and observe their values in the symbol watch area, e.g., set A3..A0 to 0011 and set A7..A0 to 0100, and note that op1 is 3 and op2 is 4, with B being 7.

Try: A parking lot has eight spaces, each with a sensor connected to A7..A0 (1 means a car is detected in the space). Spaces A7 and A6 are reserved handicapped parking. Write a RIM C program that: (1) Sets B0 to 1 if both handicapped spaces are full, and (2) Sets B7..B5 equal to the number of available non-handicapped spaces.

Shifting can be performed on signed integer types too, but we do not recommend such use. Such shifting was previously popular because shifting a binary number left or right is equivalent to

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

multiplying or dividing by 2, respectively (just as shifting a decimal number left or right is equivalent to multiplying or dividing by 10), and shifting could result in faster code execution than the slower * and / operations on some processors. However, modern compilers automatically replace * and / by shifts when possible, so today the programmer can emphasize understandable code rather than such low-level speedup attempts. For the curious reader, shifting a signed number performs an "arithmetic" shift that preserves the number's sign, rather than a "logical" shift that merely shifts all bits. We will never shift signed types. (Wikipedia: Arithmetic Shift Wikipedia: Logical Shift)

Bit access functions

Defining C functions that perform common bit manipulation tasks can be quite useful.

The following function returns a value in which the k'th bit of an unsigned char x is set to 1:

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

The mask 0x01 is shifted left k positions to get the sole 1 bit into the k'th position, and then bitwise ORed with x so that in the result the k'th bit is 1 (because single-bit a | 1 = 1) while x's bits pass through to the remaining positions (because a | 0 = a). If k is 2, the mask will be shifted to become 00000100. Recall that the rightmost bit is position 0, not position 1. Similar functions can be created for short or long integer types.

The following function returns a value in which the k'th bit of an unsigned char x is set to 0:

```
unsigned char set_bit0(unsigned char x, unsigned char k) {
   return (x & ~(0x01 << k));
}
```

The mask 0x01 is shifted left k positions to get the sole 1 bit into the k'th position, and then bitwise complemented using "~" to yield a mask with a 0 in the *k*'th bit and 1s in the other bits. For example, when *k* is 1, the shifted constant will be 00000010, which will then be complemented into 11111101. The resulting mask is bitwise ANDed with x, so that in the result the k'th bit is 0 (because single-bit a & 0 = 0), while x's bits pass through to the remaining bits (because a & 1 = a).

Example: The following statements set B to A except for setting B7 to 0:

```
unsigned char tmp;
tmp = A;
tmp = set_bit0(tmp, 7);
B = tmp;
```

The above functions can be used to create another function that takes as a parameter the bit value to be set .

```
unsigned char SetBit(unsigned char x, unsigned char k, unsigned char b) {
   return (b ? set_bit1(x, k) : set_bit0(x, k));
}
```

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

The function uses C's ternary ***conditional operator (?:)*** , which returns its second operand when the first operand is non-zero, else it returns its third operand. So for "m = (n<5) ? 44 : 99;", if n < 5 then m will be assigned 44, else m will be assigned 99.

An example of using the SetBit function involves setting all B bits to the value of A0:

```
unsigned char i;
for (i=0; i<8; i++) {
   B = SetBit(B, i, A0);
}
```

Finally, the following function gets (rather than sets) the value of a particular bit in an integer variable:

```
unsigned char GetBit(unsigned char x, unsigned char k) {
   return ((x & (0x01 << k)) != 0);
}
```

The function creates a mask *m* containing a 1 in position *k* and 0s in all other positions, then performs a bitwise AND to pass the k'th bit of x through, resulting either in a zero result if the k'th bit was 0, or a non-zero result if the k'th bit was 1. The function compares the result with 0, then returning either a 1 or a 0.

Example: A parking lot has eight parking spaces, each with a sensor connected to input A. The following program sets B to the number of occupied spaces, by counting the number of 1s using the GetBit function:

```
#include "RIMS.h"

unsigned char GetBit(unsigned char x, unsigned char k) {
   return ((x & (0x01 << k)) != 0);
}

void main()
{
   unsigned char i;
   unsigned char cnt;
   while (1) {
      cnt=0;
      for (i=0; i<8; i++) {
         if (GetBit(A, i)) {
            cnt++;
         }
      }
      B = cnt;
   }
}
```

Note that the above bit access functions do not perform error checking (e.g., you can attempt to set the 9th bit of a variable).

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

The examples using the SetBit and GetBit functions may seem inefficient due to computing the mask, but today's optimizing compilers handle these very efficiently. Furthermore, the ***inline*** keyword can be prepended to each function declaration (e.g., "inline unsigned char GetBit(...)") to encourage compilers to inline the function calls (though many compilers would do so anyways). Inlining means to replace a function call by the function's internal statements. Compiler optimizations may then eliminate most of the statements within the GetBit and SetBit functions.

A programmer may wish to copy the bit-access functions to the top of a file whose program performs bit manipulation, as in the below example program that copies A's bits to B's bits in reverse order (with function SetBit performing the set to 1 and 0 directly rather than using the set_bit1 and set_bit0 functions),

```
// Bit-access functions
inline unsigned char SetBit(unsigned char x, unsigned char k, unsigned char b) {
   return (b ? x | (0x01 << k) : x & ~(0x01 << k));
}
inline unsigned char GetBit(unsigned char x, unsigned char k) {
   return ((x & (0x01 << k)) != 0);
}

void main(){
   unsigned char i;
   while(1){
      for (i=0; i<8; i++) {
         B = SetBit(B, 7-i, GetBit(A,i));
      }
   }
}
```

(Wikipedia: Bit Manipulation   Wikipedia: Bitwise Operation   Wikipedia: Mask   Wikipedia: ?: Operator)

Try: Write a C program for RIMS that sets B0=1 if a sequence of three consecutive 1s appears anywhere on input A (e.g., 11100000 and 10111101 have such sequences, while 11001100 does not), using a C *for* loop and the GetBit() function.

Exercises

Assume the RIMS environment for all exercises below.

1. Write a C program to set B3-B0 to A7-A4, and B7-B4 to A3-A0.
2. Write a C program that treats A1A0, A3A2, and A5A4 as three 2-bit unsigned binary number. The program should output the sum of those three numbers onto B.
3. Write a C program that rotates input A right once and outputs the result on B (rotate right is the same as shift right, except the LSB becomes the MSB). Use a variable X to store A, and use the right shift operator to avoid having to explicitly set each bit with a unique statement.
4. Write C statements that set B to the reverse of A, such at B7 = A0, B6 = A1, etc. Rather than writing 8 assignment statements, instead write a for loop that makes use of the GetBit and SetBit functions.
5. Write a C program that interprets the input A as an 8-bit unsigned binary number representing a temperature in Fahrenheit, and outputs the temperature in Celsius on B .

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

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, then 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 sequence other than the correct unlocking 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

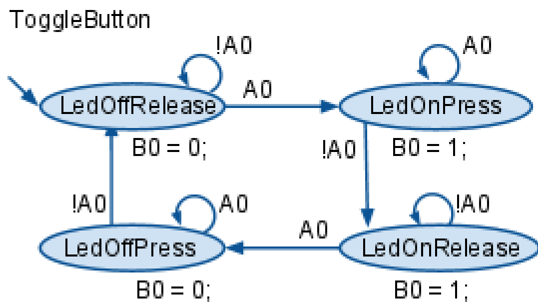
Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

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

State machines

State machines are a computation model intended for describing time-ordered behavior. Numerous kinds of state machines exist. Common features of ***state machines*** 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: Finite state machine) (Wikipedia: State diagram).



The above figure shows a state machine (assume bit input A0 and bit output B0); with four states LedOff, LedOnPress, LedOn, and LedOffPress, each having actions B0 = 0, B0 = 1, B0 = 1, and B0 = 0, respectively; with eight transitions labeled either A0 or !A0; and with the initial state being LedOff as indicated by the arrow pointing from nothing to LedOff.

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 once -- above, the initial state is LedOff and its action B0=0 is executed once. 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 transition has a true condition; one and only one should be true, else the state machine has not been properly created.
- The system's new current state is set to the state pointed to by that transition (that state may be the same as the previous current state). Th new current state's actions are then executed once. Above, supposing A0 was 1, the tick would cause transition to state LedOnPress and its action B0=1 would be executed once.

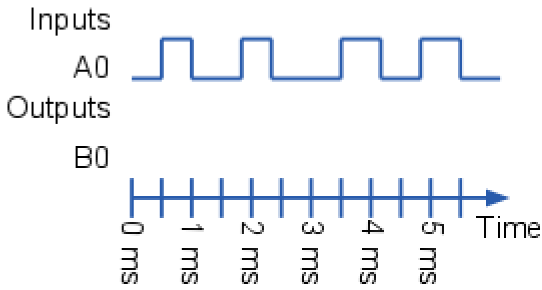
The ticking process then repeats. Ticks take a small unknown non-zero amount of time. Ticks are assumed to occur at a much faster rate than input events, such that no events are missed.

Continuing the above example, while A0 stays 1, each tick will set the system's current state to LedOnPress and execute B0=1 once; this is called *staying* in state LedOnPress. When A0 changes to 0, then on the next tick the system's current state becomes LedOn, with action B0=1. The SM stays in that state until A0 changes to 1, causing transition to LedOffPress, with action B0=0. And so on.

Try: Draw the B0 signal for the given A0 signal and the above state machine.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

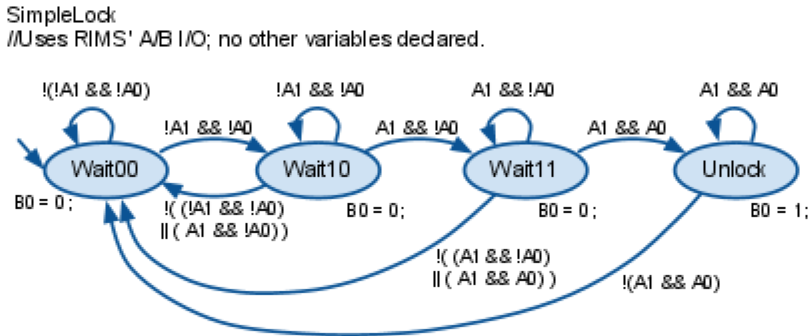


(The above system happens to carry out *toggle* functionality, wherein a system output changes between two values. Some household lights utilize a single button that toggles a light between on and off).

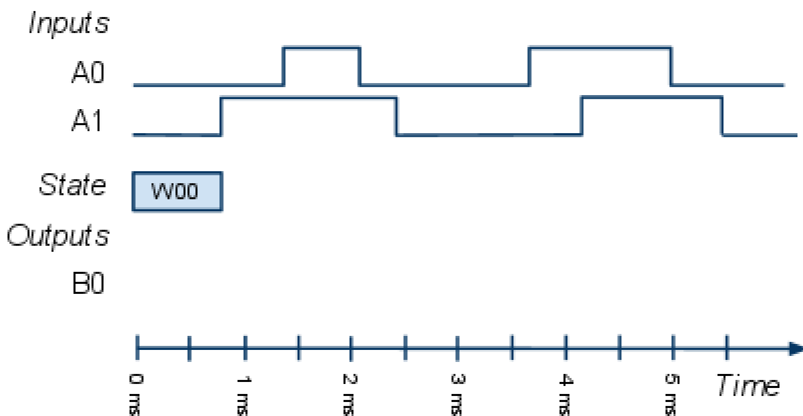
Transitions leaving a particular state should have mutually exclusive transitions (otherwise, which of two true transitions should be taken?). A transition need not have a condition specified, which means the condition is always true (thus, that transition should be the only one leaving a particular state). A state may have multiple actions, such as "B0 = 1; B1 = 1;" or may have no actions at all.

We will use a particular variation of a state machine model, referred to in this book just as an ***SM***, intended for creating C programs that support time-ordered behavior. The SM has C variables declared, rather than inputs and outputs; we'll continue to use RIMS' implicitly-declared A and B input and output variables though. The SM's state actions consist of C statements, and the SM's transitions consist of C expressions.

Using an SM, the earlier simple lock example can be captured as follows.



Try: For the above SM, indicate the current state throughout the time period shown below for the following input signal values (the initial state Wait00 is shown as W00). Also show the B0 signal value.



Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

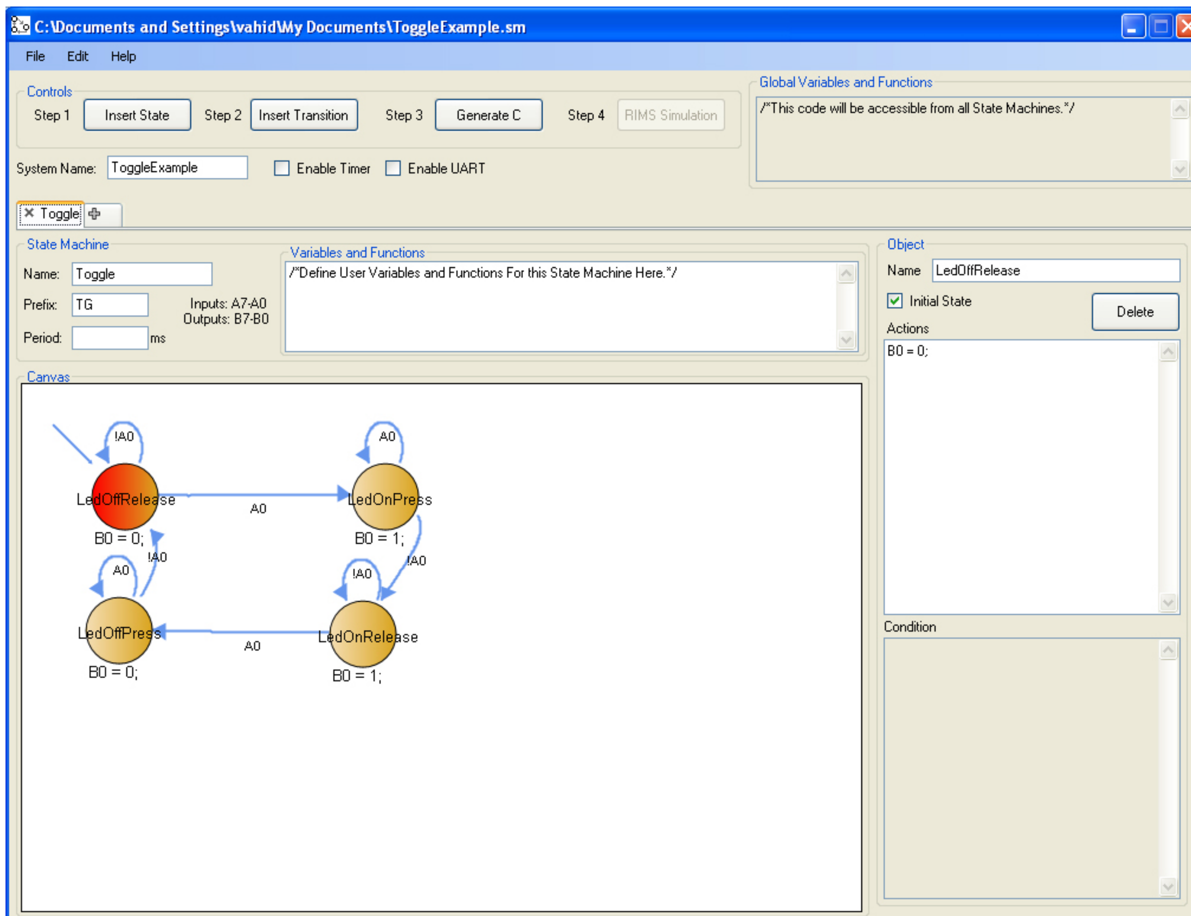
Notice that the SM model and the earlier C code have the same behavior. However, the SM more explicitly captures the desired time-ordered behavior. This straightforwardness can be further seen by trying to extend the SM.

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). Pressing "Generate C" automatically generates C code for RIMS. Pressing "RIMS Simulation" automatically starts RIMS and runs the generated C code on RIMS, where the user can set input switches and observe output LEDs, while RIBS highlights the currently-executing state.

Try: Run the RIBS tool and create the above-described Toggle state machine. Unselect the "Enable Timer" checkbox, and delete any text from the "Period" text box (these items will be described later). Press "Generate C" and then press "RIMS

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

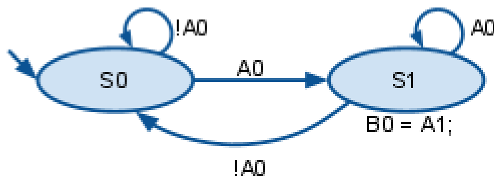
Simulation", which causes the RIMS tool to automatically execute. With both the RIBS and RIMS tools visible on your screen, click on the A0 switch on RIMS several times and notice how the current state changes in RIBS.

Try: Capture an SM for a three-LED sequencer. Initially B0's LED is on. When A0 rises, B0 turns off and B1 turns on. When A0 rises again, B1 turns off and B2 turns on. When A0 rises again, B2 turns off and B0 turns on again. And so on. Try capturing the behavior in C directly first, and then instead capture it as an SM (use a distinct state for each distinct output situation). Test the SM using RIBS.

Try: Extend the three-LED sequencer by reversing the sequence if A1=1, else following the earlier sequence if A1=0. Try extending the C code first (hard!), and then the SM.

Converting an SM to C

Because microprocessors typically have C compilers but not SM compilers, converting an SM to C is necessary. Using a standard method for converting an SM to C enhances the readability and correctness of the resulting C code. The following illustrates such a method for the given SM named Latch (abbreviated as LA), which saves (or "latches") the value of A1 onto B0 whenever A0 is 1.




```
#include "RIMS.h"

enum LA_States { LA_s0, LA_s1 } LA_State;

void LA_Tick()
{
  switch(LA_State) { // Transitions
      case -1:  // Initial transition
         LA_State = LA_s0;
         break;
      case LA_s0:
         if (!A0) {
            LA_State = LA_s0;
         }
         else if (A0) {
            LA_State = LA_s1;
         }
         break;
      case LA_s1:
         if (!A0) {
            LA_State = LA_s0;
         }
         else if (A0) {
            LA_State = LA_s1;
         }
         break;
      } // Transitions
```


Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

```
  switch(LA_State) { // State actions
      case LA_s0:
         break;
      case LA_s1:
         B0 = A1;
         break;
   } // State actions
}

void main() {

   B = 0; // Initialize outputs
   LA_State = -1; // Indicates initial call

   while(1) {
      LA_Tick();
   }
}
```

While the code may at first glance look imposing, it follows a simple pattern. The first line creates a new enum data type LA_States defined to have possible values of "LA_s0" and "LA_s1." That same code line declares global variable LA_State to be of type LA_States. ***enum*** is a C construct for defining a new data type (in contrast to built-in types like char or short) whose value can be one of an "enumerated" list of values. The programmer provides the enumeration list ("{ LA_s0, LA_s1 }" above). Each item in the list becomes a new constant, with the first item having value 0, the second item having value 1, etc.

The main function sets the current state to a value of -1 (which is a small trick used to indicate the start of execution, because items in the enumeration list are numbered 0 or greater), initializes outputs, and then enters the normal infinite "`while (1)`" loop, which just repeatedly calls a function LA_Tick().

LA_Tick carries out one "tick" of the SM. For the current state, the first switch statement in LA_Tick carries out the appropriate transitions for the current state; if LA_State is -1, the transition is to the SM's initial state. The second switch statement then carries out the appropriate actions for that new current state.

Try: Run the above code in RIMS. Press "Break" and then repeatedly press "Step" (setting A0 accordingly), observing how the code executes each tick of the state machine.

Capturing behavior as an SM and then converting to C using the above method may result in more code than capturing behavior directly in C. However, more code does not always mean worse code. The C code generated from an SM may be more likely to be correct, may be more easily extendible and maintainable, and has other benefits that will be seen later.

Variables, statements, actions, and conditions in SMs

Variables

An SM can have variables declared at the SM scope (i.e., not within a state), which can be accessed by all actions and conditions of the SM. For example, a variable "unsigned short UnlockCnt;" could be declared. Above, you can think of the A0, B0, and similar items as having been declared as "unsigned

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

char" variables. When a user adds a variable declaration, the variable should be initialized. While the variables could be initialized when declared (e.g., unsigned short UnlockCnt=0;), it is better practice to use a new state named "Init" to carry out initializations of variables as well as of outputs. In this way, not only are all initializations in one place, but the system can be re-initialized merely by having a transition point back to the Init state.

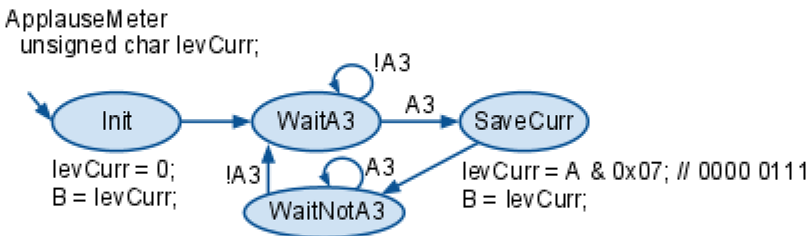
When converting to C, the synchSM variables can be declared as global variables above the synchSM's tick function.

Statements

The statements that can appear in actions can include more than just assignment statements. Any C statements can appear, such as an if-else statements and even some for loops. However, for our purposes, **statements in actions should *NEVER* wait on an external input value**, such as the statement "`while (!A0) {};`". Such behavior should be captured as states and transitions so that *all time-ordering information is visible at the transition level*.

Try: Create an SM that counts the number of times that A0 rises, and outputs that number onto B. Declare, initialize, and use a variable "Cnt" to maintain the count.

Example: Consider an applause-meter system intended for a game show. A sound sensor measures sound on a scale of 0 to 7 (0 means quiet, 7 means loud), outputting a three-bit binary number, connected to RIM's A2-A0. A button connected to A3 can be pressed by the game show host to save (when A3 rises) the current sound level, which will then be displayed on B. The system's behavior can be captured as an SM with a variable, as shown.



Note that the SM's Init state has a transition leaving it but with no associated condition. *A transition with no shown condition is shorthand for a transition having a "1" condition, meaning the condition is always true.* Obviously, such a transition can be the only transition leaving a state.

Try: Extend the SM so that pressing A4 would output the maximum level seen so far onto B. Use another variable to store the maximum.

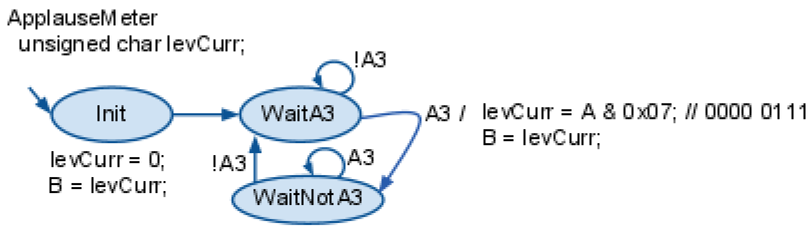
Mealy/Moore actions

The above state machine model associates actions with states only, known as a ***Moore-type state machine***. A ***Mealy-type state machine*** allows actions on transitions too. A Mealy-type SM can make some behaviors easier to capture.

For example, notice in the Applause Meter example that the SaveCurr state is used just to perform actions, after which the SM immediately transitions to the WaitNotA3 state. Using Mealy actions, the SM could be simplified to the following:

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

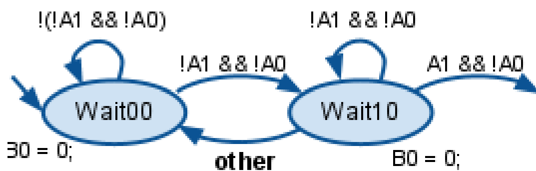


Try: For the SimpleLock SM, maintain a global variable "unsigned short Fail;" that counts every time that the system leaves state Wait00 but then returns to that state *without* having been unlocked. Hint: create a new initial state to initialize Fail, and then increment the variable on the appropriate transitions.

When translating to C, a transition's actions appear in the transition switch statement, in the appropriate if-else branch.

Conditions

Conditions in an SM must be C expressions. Exactly one of a state's leaving transitions must have a condition that evaluates to true at a given time, no more, no fewer. Note in the earlier SimpleLock SM that we sometimes had transitions going to next states such as a transition with "!A1 && !A0" and another transition with "A1 && !A0, and then a transition having a condition that covered all *other* possibilities, such as "! (!A1 && !A0) || (A1 && !A0))". To simplify the SM's appearance, a special condition called ***other*** may be associated with one transition leaving a state, which is shorthand for the opposite of the ORing of all conditions from the remaining leaving transitions of that state. An example for part of the SimpleLock SM is shown:

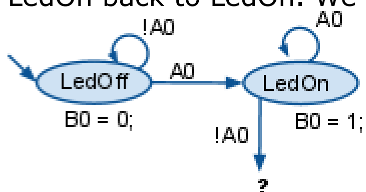


When translating to C, "other" may be implemented as a last "else" branch (with no expression) in the state's transitions if-else code.

How to capture behavior as an SM

Capturing behavior as an SM is an art. The following process may help. Consider the earlier-described toggle example that toggled an LED between on and off each time a button was pressed. A first step is to list the obvious states of the system, adding actions as appropriate:

The second step of the process is to add transitions to each state to achieve the desired behavior. In doing so, sometimes we encounter the need for more states. We add a transition !A0 to LedOff pointing back to LedOff, and another transition A0 pointing to LedOn. We add a transition A0 from LedOn back to LedOn. We then add a transition !A0, but to where should it point?



Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

If that transition points to LedOn, we can never leave that state. If it points to LedOff, the LED turns off (B0=0), but we want the LED on until the next button press (A0). Thus, we need states that keep the LED on, one while the button is pressed, and one while the button is released. We thus rename LedOn to LedOnPressed, add a state LedOnReleased, and point that transition to LedOnReleased. We proceed to create transitions for LedOnReleased, and similarly encounter the need for two LED off states, ultimately resulting in the toggle SM earlier in the chapter.

The third step is to mentally check the behavior of the captured SM. For each state, we should check that *exactly* one transition's condition will be true, modifying the conditions or adding transitions if necessary. We should also check that the behavior is as desired; drawing a timing diagram may help.

Testing an SM

Testing time-ordered behavior requires generating good *sequences* of test vectors, in contrast to earlier chapter's examples where the outputs depended solely on the current input values. Minimally, test vectors should ensure that *each state and each transition is executed at least once*. Furthermore, if a state's action code has branches, then test vectors should also ensure that every statement is executed at least once. Even more ideally, *every path* through the SM would also be tested. Designing good test vectors can take much effort and is as important as capturing a good SM.

For the ThreeLEDs SM above, good test vectors would cover each state and each transition, as follows:

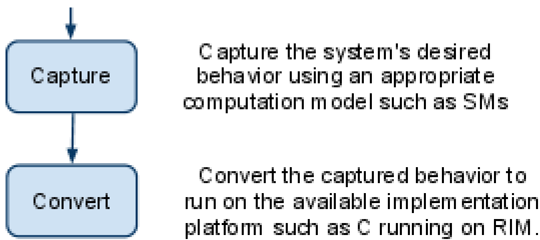
A0	Covers
0	S0_Output, S0_Output's !A0 transition
1	S0_Output's A0 transition, S1_Wait, S1_Wait's A0 transition
0	S1_Wait's !A0 transition, S1_Output, S1_Output's !A0 transition
etc...	

Capture/convert process

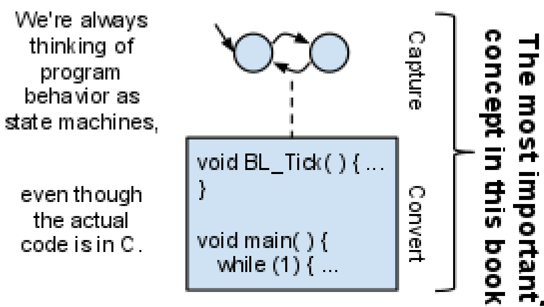
The above sections described a two-step process that is common in disciplined embedded programming. The first step is to ***capture*** the desired behavior using a computation model appropriate for the desired system behavior, such as SMs. The second step is to ***convert*** that captured behavior into an implementation, such as C that will run on a microprocessor. The conversion is typically very structured and automatable. The capture/convert process will be used in subsequent chapters for more complex behavior and can result in code that is more likely to be correct, that is more maintainable, and that has many other benefits compared to behavior that is captured directly in C's sequential instruction model. *The capture/convert design process is perhaps one of the most important concepts in disciplined programming of embedded systems.*

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu



Even in the absence of a tool like RIBS, programmers can (and should) capture time-ordered behavior as an SM, typically drawing the SM on paper first, and then converting to C. All modifications are done by changing the SM (capture), and then updating the C code (convert) An experienced programmer can work with SMs in C without always having to see a state diagram, and can see or draw a state diagram from C code. ***The concept of thinking of program behavior as state machines, even though the actual code is in C, is the most important concept in this book.***



Try: For the below C code, draw the corresponding SM state diagram.

```
#include "RIMS.h"

unsigned char x;

enum EX_States { EX_S0, EX_S2, EX_S1 } EX_State;

void EX_Tick() {
   switch(EX_State) { // Transitions
      case -1:
         EX_State = EX_S0;
         break;
         case EX_S0:
         if (1) {
            EX_State = EX_S1;
         }
         break;
      case EX_S2:
         if (A3) {
            EX_State = EX_S2;
         }
         else if (!A3) {
            EX_State = EX_S1;
         }
```

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

```
         break;
      case EX_S1:
         if (!A3) {
            EX_State = EX_S1;
         }
         else if (A3) {
            EX_State = EX_S2;
            x = A & 0x07;
            B = x;
         }
         break;
      default:
         EX_State = EX_S0;
   }

   switch(EX_State) { // State actions
      case EX_S0:
         x = 0;
         B = x;
         break;
      case EX_S2:
         break;
      case EX_S1:
         break;
      default:
      break;
   }
}

void main() {
   EX_State = -1; // Initial state
   B = 0; // Init outputs
   while(1) {
      EX_Tick();
   }
}
```

You should have obtained a state diagram identical to the earlier ApplauseMeter example, just with different names.

Note that the RIBS tool does not run a C compiler on the C code in actions/conditions/declarations, but rather just generates a new C program that contains that code. Any syntax errors will only be determined upon running a C compiler on that program, requiring a RIBS user to correlate the error message to the SM code.

Try: For the earlier Toggle example in RIBS, introduce a C syntax error by removing the semicolon after the action in the initial state. Save, generate C, then press "RIMS Simulation". Note the error message that is generated, and strive to correlate that with the RIBS synchSM.

The SM model in this chapter is a basic state machine model specifically intended to aid the capture of time-ordered behavior and for conversion to C. Many other state machine models exist, such as UML state machines ([Wikipedia: UML state machine](https://en.wikipedia.org/wiki/UML_state_machine)). Some state machine models have a more formal mathematical basis, but translation to C is more cumbersome and the code harder to

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

maintain. Another common category of computation model involves dataflow models, which are well suited to digital signal processing applications but are beyond our scope.

Exercises

Capture each system as an SM. Use RIBS and RIMS as appropriate below. (Be sure RIBS' "Enable timer" box is unchecked).

1. A doorway is for exit only. Sensors A0, A1, A2 detect a person passing through. A proper exit causes A2A1A0 to be 000, then 100, then 010, then 001, then 000. Any other sequence causes a buzzer to sound (B0=1) until 000.

2. An automatic door at a store has a sensor in front (A0) and behind (A1). The door opens (B0=1) when a person approaches from the front, and stays open as long as a person is detected in front or behind. If the door is closed and a person approaches from the behind, the door does not open. If the door is closed and a person approaches from the front but a person is also detected behind, the door does not open, to prevent hitting the person that is behind.

3. A dimmer light system has increase (A0) and decrease (A1) buttons. B sets the light intensity, 0 is off, 255 is the maximum intensity, and:
   (a) Pressing both buttons does nothing.
   (b) Pressing both buttons immediately turns the light off.

4. An amusement park ride has sensor mats on the left (A0) and right (A1) of a ride car. A ride operator starts the ride by pressing a button (A7); each unique press toggles the ride from stopped to started (B0=1) and vice-versa. If anyone leaves the ride car and steps on a sensor mat, the ride stops and an alarm sounds (B1=1). The alarm stops sounding when the person gets off the sensor mat. The only way for the ride to restart is for the operator to press the button again. The ride never starts if someone is on the sensor mat.

Curved arrows are sometimes drawn in a timing diagram to show that one event ***triggers*** (meaning "causes") another. In the above timing diagram, the first rising A1 triggers the change of state from Wait00 to Wait10, and the first rising A0 triggers the change of state from Wait10 to Unlock and also triggers the rise of B0.

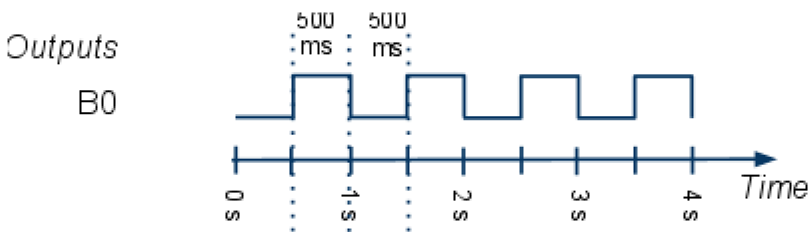
Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

Chapter 4: Time Intervals and Synchronous SMs

Introduction

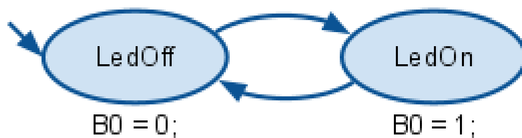
In addition to time-ordered behavior, embedded systems commonly must carry out time-interval behavior. ***Time-interval behavior*** is system functionality where events must be separated by specified intervals of real time. Consider a system that should repeatedly blink an LED on for 500 ms and off for 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 a small unknown 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***. We call an SM with a real-time tick period a ***synchronous SM***, or ***synchSM***. The blinking LED can be captured as a synchSM:

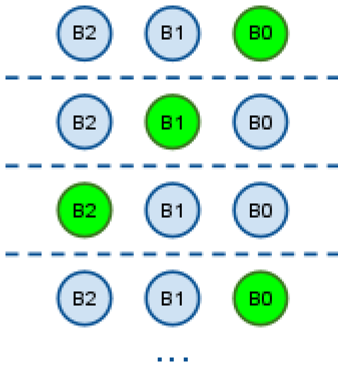


Entering a state causes (nearly) immediate execution of its actions, after which the system waits for 500 ms, due to the 500 ms tick period, before evaluating transitions and entering the next state.

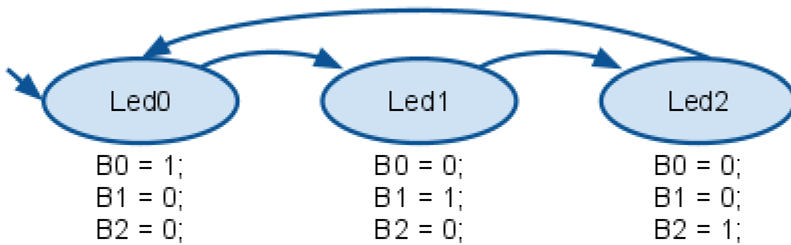
Another example is a system that lights one of three LEDs connected to B0, B1, and B2, one LED per second in sequence, such that the lit LED appears to move (and wrap around). Such a system might be found in a highway construction sign, indicating that traffic should move to the left.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

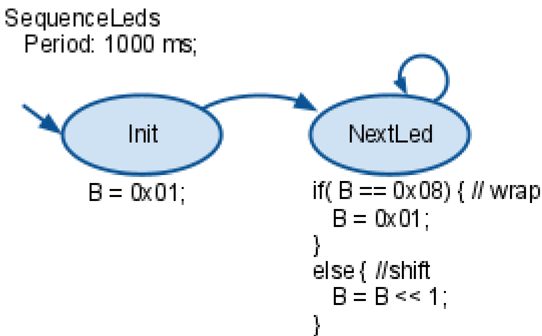


ThreeLeds
Period: 1000ms



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 earlier 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:



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, ..., 1000000, repeat. Use bit manipulation methods for the bouncing ball mode, rather than using a separate state for different output combinations.

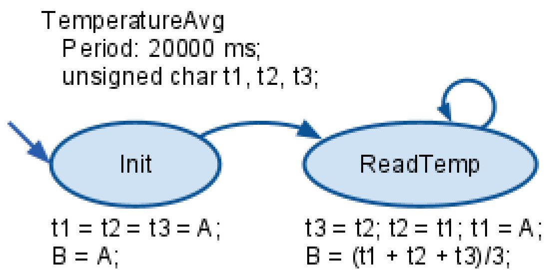
Time intervals for inputs

The above examples showed the usefulness of time intervals for outputs. Time intervals are also used for inputs. Consider a system that should output the average of the last three temperature

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

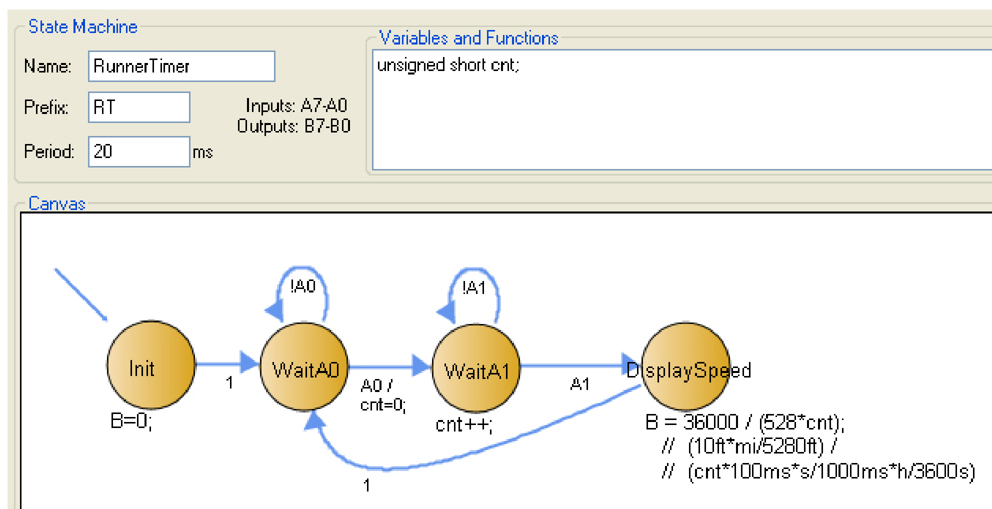
readings, where readings are taken every 20 seconds. Assume a temperature sensor has an 8-bit output connected to input A. The system behavior can be captured as the following synchSM:



The synchSM has a period of 20000 ms, which is 20 seconds. Three variables hold the past three temperature readings. t1 holds the current reading, t2 the previous reading, and t3 the reading previous to that. State Init initializes the variables and the output to the current temperature. 20 seconds later, state ReadTemp sets t1 to the current read temperature, after having updated the t2 and t3 variables. It then computes and outputs the average (which will be rounded due to integer division). The transition leaving ReadTemp will be taken every 20 seconds, and thus temperature reads and output updates will occur every 20 seconds.

Try: A car has a fuel-level sensor whose value appears in binary on input A, 0 meaning empty and 100 full. A system checks the fuel level every 10 seconds, and displays the level graphically to a driver by illuminating 5 light segments B4-B0: 11111 means full, 11110 means 80% full, 11100 60% , 11000 40% , 10000 20% , and 00000 means empty. When 20% full or less, the system also turns on a low-fuel warning light (B7).

Variables are commonly used in conjunction with time intervals. For example, consider computing the speed of marathon runners as they pass two sensors, separated by 10 feet, and connected to A0 and A1. Suppose we set the synchSM period to 100 ms for timing precision. The synchSM should start incrementing a counter when A0 is 1, and stop when A1 is 1, computing and displaying the runner's speed. Such a synchSM is shown below.



Try: Create a simple reaction timer. A user presses a button (A0) to reset the system. A few seconds later, an LED illuminates (B0), after which the user should press a second button (A1) as soon as possible. The system then displays the reaction time on B, in binary and x10 ms. Use a 10 ms synchSM period.

Copyright © Frank Vahid and Tony Givargis 2011

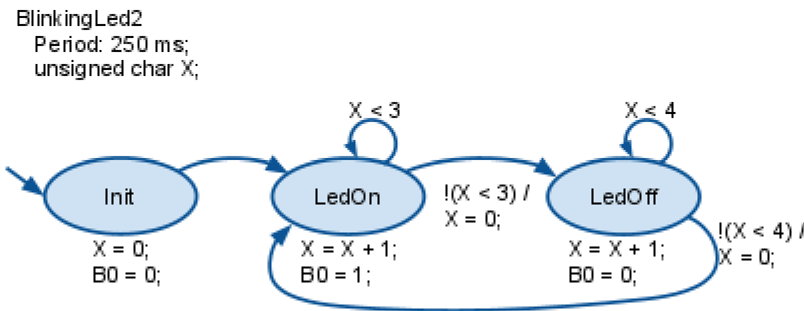
Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

Choosing a period for different time intervals

Sometimes one system involves multiple different time intervals. For example, consider a system that repeatedlys blink 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.

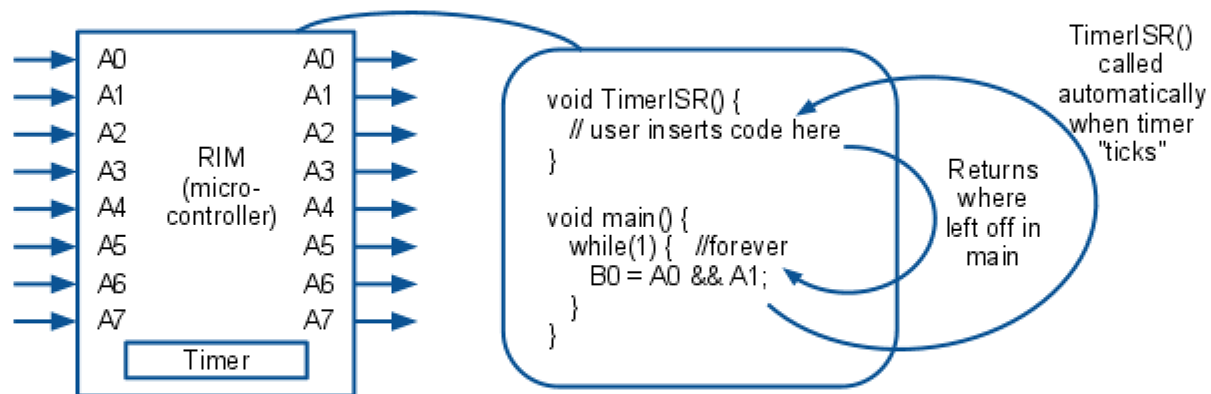


The advantage of the counting approach is clearer for a longer off interval, such as 5 seconds.

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.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

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**().

While programmers sometimes insert various statements into timer ISRs, *we will use ISRs only in a disciplined way -- our ISR will 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:

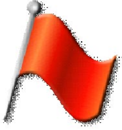
```
#include "RIMS.h"

volatile unsigned char TimerFlag = 0;

void TimerISR() {
   TimerFlag = 1;
}

void main() {

   B0 = 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;
      //NOTE: better style would use a synchSM
      //This example just illustrates use of an ISR and flag
   }
}
```

The user's main code initializes the timer period to 1 second (1000 ms) and turns the timer on. The code enters the while(1) loop, toggles B0, and waits for the timer flag to become 1. Though it appears the flag could never be set to 1 due to the code never leaving the "while(!TimerFlag) {}" loop, the functionality of an ISR is such that when the timer ticks, the microcontroller automatically stops executing that while loop, jumps to TimerISR() and thus sets the flag to 1, and then returns to that while loop. Upon that return, the flag value will have changed, so the while loop will exit and the remaining statement will be executed, which clears the flag back to 0, and then the while(1) loop repeats (causing another toggle and another wait).

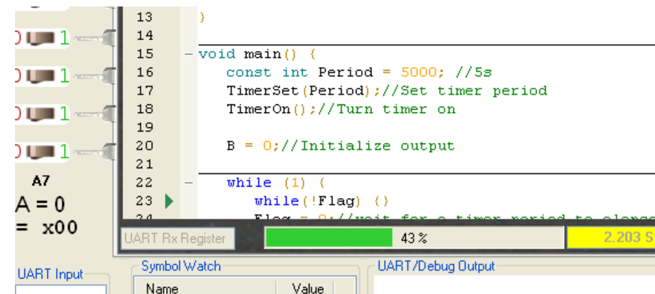
Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

Setting a flag to 1 is called *raising* the flag, and setting it to 0 is called *lowering* the flag.



The TimerFlag variable has been defined using the *volatile* keyword. Modern C compilers optimize C programs for better performance. A C compiler might analyze the above code and determine that timerFlag is initialized to 0 and that TimerFlag is changed by two functions, namely TimerISR and main. Since TimerISR is not called within the program, the compiler may incorrectly conclude that TimerFlag is always 0. This can lead to a problematic compiler transformation. The C compiler might transform the statement "while (!TimerFlag);" to "while (1)", causing an infinite loop and thus incorrect execution. By defining TimerFlag as ***volatile***, the compiler will assume that timerFlag may be changed at any time by some external entity, thus it is "volatile" in nature. As a result, the compiler will not make a transformation such as the one just described. An experienced programmer asks: How would the compiler interpret my code? Based on insight in and knowledge of compilers, the programmer writes programs that are portable, less prone to undesired compiler transformations, efficient, maintainable, compact, and so on. Embedded systems programmers benefit from being knowledgeable of compiler technologies, microcontroller architectures, and operating systems.



RIMS 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 in RIMS, 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 function is only called when the timer ticks, via use of a flag. The following shows C code for the earlier BlinkingLed (BL) example:

```
#include "RIMS.h"

volatile unsigned char TimerFlag=0; // raised by ISR, lowered by main code

void TimerISR() {
```

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

```
   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 ) { //State actions
      case BL_LedOff:
         B0 = 0;
         break;
      case BL_LedOn:
         B0 = 1;
         break;
   }
}

void main() {
   B = 0; //Init outputs
   TimerSet(500);
   TimerOn();
   BL_State = -1; // Indicates initial tick function call
   while (1) {
      BL_Tick();            // Execute one tick of the BL synchSM
      while (!TimerFlag){}  // Wait for BL's period
      TimerFlag = 0;        // Lower flag raised by timer
   }
}
```

Try: Use RIMS to observe the behavior of the above code.

Try: Capture the ThreeLEDs synchSM using RIBS, click "Generate C", and then click "RIMS Simulation". Observe the correspondence between the microcontroller timer ticks in RIMS and the synchSM ticks in RIBS.

In addition to or instead of using RIBS' animation of the current state, debugging synchSMs can be aided by temporarily adding print statements. The following code adds "puts" statements to print the current state. We have also introduced a bug somewhere in the code.

```
#include "RIMS.h"
```

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

```
volatile unsigned char TimerFlag=0; // 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_LedOff;
         break;
      case BL_LedOn:
         BL_State = BL_LedOff;
         break;
   }

   switch (BL_State ) { //State actions
      case BL_LedOff:
         puts("BL_State: BL_LedOff\n");
         B0 = 0;
         break;
      case BL_LedOn:
         puts("BL_State: BL_LedOn\n");
         B0 = 1;
         break;
   }
}

void main() {
   B = 0; //Init outputs
   TimerSet(500);
   TimerOn();
   BL_State = -1; // Indicates initial tick function call
   while (1) {
      BL_Tick();             // Execute one tick of the BL synchSM
      while (!TimerFlag){}  // Wait for BL's period
      TimerFlag = 0;         // Lower flag raised by timer
   }
}
```

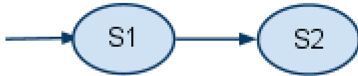
Try: Run the above code, and note that the LED does not blink. Observe the printed strings to understand the basic problem and thus find and fix the bug in the code. Also, use the built-in debug features of RIMS, namely Break and Step, to observe how the code is executing.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

State actions should never wait

We earlier stated that *statements inside a state should never include statements that wait.* synchSMs clearly illustrate why this requirement exists. Consider the following part of a synchSM with a 100 ms period:



B0 = 1;
while (!A0); // wait
B0 = 0;

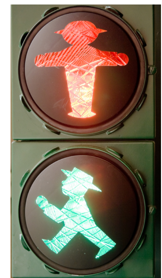
If the synchSM enters state S1 and is waiting at the while statement when another 100 ms passes, how should the synchSM proceed? Should it move on to state S2, or should it continue to wait at the while statement? synchSM behavior becomes indeterminate if state actions include waits.

A state's statements should thus be written such that they execute and reach their end, a feature known as ***run to completion***.

A key assumption in a synchSM is that every state's statements always run to completion faster than the synchSM's period. Actions are assumed to run in a small but non-zero time.

Functions can be declared in a synchSM and called by any state's statements, as long as functions always run to completion. Functions thus serve as a programming convenience to eliminate redundant code in different states (e.g., the earlier-defined GetBit function may be used in the actions of multiple states), or to make a state's actions more clear or less cluttered (e.g., a state's actions may consist of the statement:"B = computeNum1sOnA()" where that function is defined at the top of the synchSM.

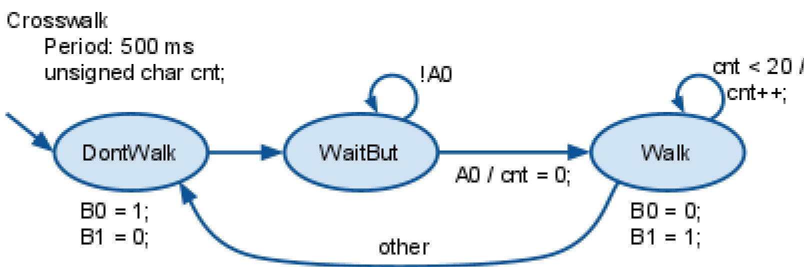
Example



Consider a simple crosswalk system. The system initially illuminates a don't-walk symbol (B0=1). When a pedestrian presses button A0, the system illuminates a walk symbol (B1=1) for 10 seconds.

The system can be captured as the following synchSM. We choose a period of 500 ms to detect button presses (meaning a button press should be at least 500 ms long to be consistently detected). We use a cnt variable to stay in state Walk for 10 seconds, by counting to 20 (20 * 500 ms = 10 seconds).

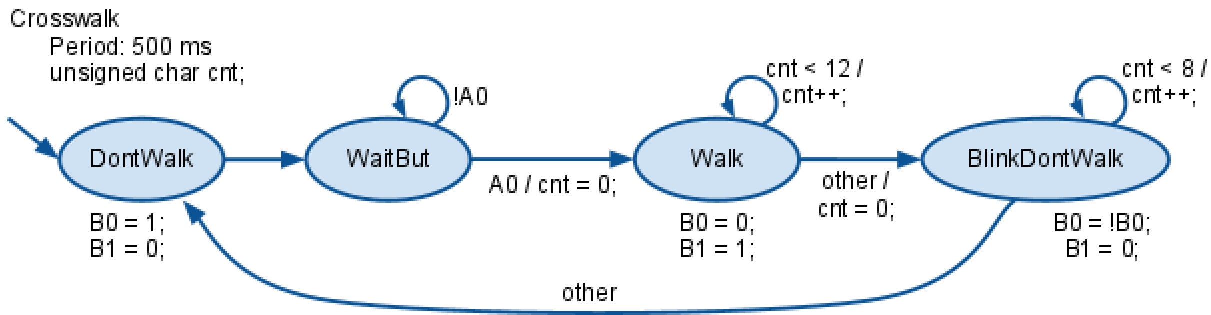
Crosswalk
Period: 500 ms
unsigned char cnt;



A better system might warn the pedestrian by blinking the don't walk symbol for the last 4 seconds, captured as follows.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu



Note how we use the synchSM's period for several timing purposes: to read the input button at 500 ms intervals, to stay in Walk for 6 seconds, to blink the don't walk symbol on/off for 500 ms each, and to stay in BlinkDontWalk for 4 seconds.

Try: Improve the crosswalk system by requiring at least 15 seconds to pass between the end of one crossing and the beginning of the next (so cars have a chance to drive).

Exercises

Capture each system as a synchSM. Use RIBS and RIMS as desired.

1. A baby monitor system detects motion using a sensor (A0=1). The system should sound an alarm (B0=1) if no motion is detected for at least 60 seconds. A button (A1) or detected motion resets the system.

2. Create a festive lights display with 8 light bulbs (B7-B0). When activated (A0=1), the appearance is of two balls bouncing off each other, as follows: 10000001, 01000010, 00100100, 00011000, 00100100, 01000010, 10000001, repeat. Each output configuration lasts for one second. Use bit-manipulation methods rather than numerous states. When deactivated, the display turns off all bulbs within 1 second.

3. A simple display driver writes to a display having 3 rows and 3 columns, for 9 pixels. B1-B0 specifies a current row and B3-B2 a current column. B5-B4 represent a 2-bit pixel shading (00 is white, 01 light gray, 10 dark gray, and 11 black) for the current row and column. An array "unsigned char F[3][3]" holds the desired pixel values, each element storing either 0, 1, 2, or 3. The driver should repeatedly write the pixel values to the display, 1 pixel per 20 milliseconds. Even though writing a pixel only temporarily lights the pixel, by such rapid repeated writing, a person viewing the display might see what appears to be 9 continually-lit pixels due to persistence of the display and of his eyes. Hint: Create nested loops using state machine constructs.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

Chapter 5: Input/Output

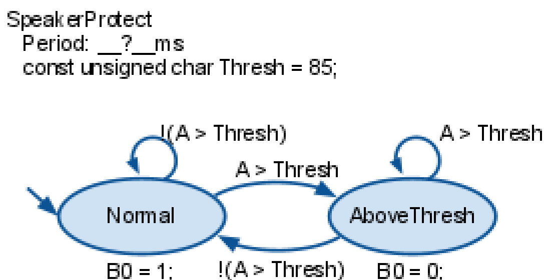
Several detailed issues must be considered when embedded systems interact with inputs and outputs.

Sampling of inputs

Data inputs

Reading a sensor at a specified period is called ***sampling***, the period is the *sampling rate*, and each data item read is a *sample*. (Wikipedia: Sampling). Choosing a good sampling rate is important. An earlier example sampled a temperature sensor once every 20 seconds.

Consider a system that prevents an audio system speaker from being damaged, by disabling the speaker (B0=0) if the input audio level exceeds a threshold of "85" as detected by an 8-bit audio-level sensor connected to A. A synchSM is shown.



The response must be fast enough to prevent damage. A 25 second rate is clearly too slow. Assuming that speaker damage may occur if audio exceeds the threshold for 500 ms, we might choose a synchSM period of 100 ms.

Ideally, a system would sample the input as fast as possible. However, when a synchSM is implemented as C on a microcontroller, the instructions do take time. If the input sampling rate is too fast, then the instructions to carry out a synchSM tick's transitions and state actions may not complete before the next tick, possibly resulting in erroneous execution. Therefore, *the synchSM period needs to be chosen to be as large as possible* while still safely satisfying system requirements, in order to ***reduce microcontroller utilization***. The need for such reduction will become even more critical when implementing multiple synchSMs on the same microcontroller. A later chapter discusses utilization further. Due to this need to reduce microcontroller utilization, we generally do not use SMs (which do not have a period, in contrast to synchSMs, and thus execute as fast as possible on the microcontroller); SMs were introduced primarily as a stepping-stone towards understanding synchSMs.

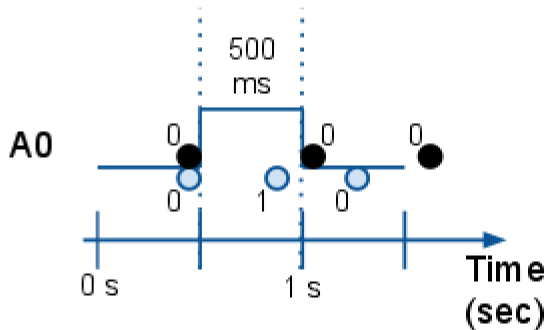
Try: Create a synchSM with a period of 20 ms and having extensive actions such that the actions do not complete within 20 ms on RIM. Try executing the synchSM using RIBS/RIMS and observing incorrect results.

Event inputs

An earlier-stated assumption was that an SM's tick rate was on a faster scale than events, such that events would never be missed. Choosing the largest possible synchSM period thus requires explicitly thinking about the smallest separation of events that we wish to detect. The ***minimum***

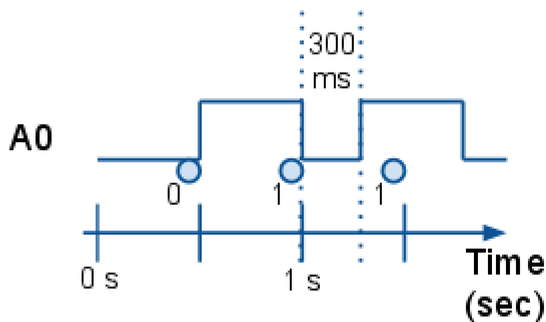
Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

event separation time is the smallest time between any two input events. *Choosing a synchSM period less than the minimum event separation time guarantees that all events of interest will be detected*. Consider detecting button presses in a toggle button system (see previous chapter) where a button press lasts more than 500 ms (1/2 second). If the sampling rate is slightly greater than 500 ms, then the rising and falling events on A0 in the below figure could be missed, as shown by the black circles representing the times at which the signal was sampled. All three samples are 0. If the sampling rate is less than or equal to 500 ms, as shown by the lightly-filled circles in the below figure, then all events are detected. The first sample is 0, the second is 1, and the third is 0, so all the rising and falling events on A0 were detected, albeit delayed slightly from when the events actually occurred. Any shifting in time of those lightly-filled circles (maintaining their horizontal separation) cannot miss a 500 ms event.



Try: Consider a sensor in a street used to count the number of cars that pass over the sensor. If the sensor detects just a single point above it, the shortest car is 6 feet long, and the fastest speed to be considered is 200 mph, what is the minimum event separation time for a single car passing over the sensor?

Minimum event separation time for input items such as for a button or for a car sensor includes not only button presses or car detections, but also the time *between* such items. If button presses need only be separated by 300 ms, then a sampling rate greater than 300 ms could miss a button release, as shown in the figure below.



The minimum event separation time is actually 300 ms, and thus the sample rate should be less than 300 ms.

Latency

Some input events trigger new output events or output data. The time between the input event and the new output is called ***latency***. (Wikipedia: Latency(Engineering)). In the earlier toggle button system, the time between the start of a button press (A0 rising) and the toggled LED output is the system latency. Reduced latency is usually desired. While reducing microcontroller utilization demands a longer synchSM period, reducing latency demands a shorter period. For the toggle button system, a 300 ms latency might be undesirable, so a synchSM period of perhaps 50 ms might be

Copyright © Frank Vahid and Tony Givargis 2011

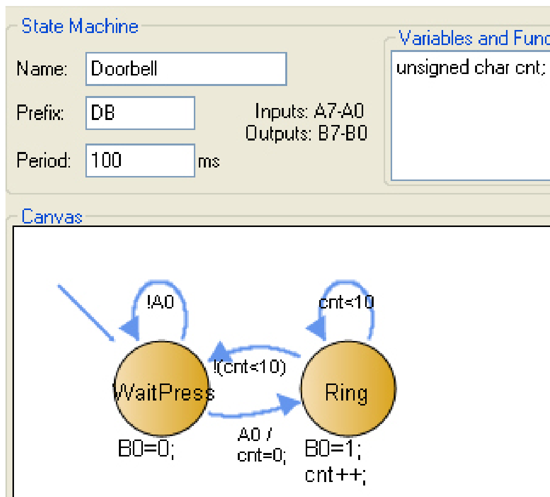
Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

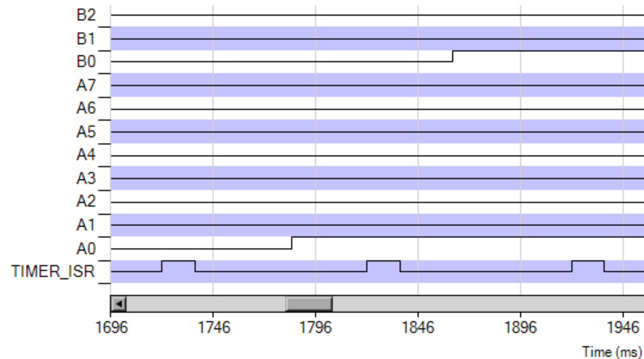
chosen instead to minimize latency at the expense of more microcontroller utilization. Observers indicate that 100 ms latency may be slightly noticeable to humans, but 50 ms may not be (see Ganssle: A Guide to Debouncing).

Example: Consider an electronic doorbell, with a button connected to A0, and B0 connecting to a bell. Consider the following timing features. The minimum button press length considered is 400 ms. The minimum separation between presses is 500 ms. The maximum latency between a press and the start of the bell ringing should be 100 ms. When a valid button press is detected, the bell should ring for 1 second and then stop ringing until the next distinct button press. The following table lists how each timing feature constrains the synchSM:

Timing feature	Constraint
Minimum press length 400 ms	Period should be < 400 ms
Minimum press separation 500 ms	Period should be < 500 ms
Maximum latency between press and bell 100 ms	synchSM period should be < 100 ms, and state sequence should ensure latency <= 100 ms
Bell rings for 1 sec	Period should evenly divide 1000 ms
(Nearly always present) Minimize processor utilization	Period should be as large as possible

Based on the constraints, the largest possible period is 100 ms. Other possible periods are 50, 25, 20, 10, ... (divisors of 100), but we should try the largest period first. We can create the following synchSM:





We evaluated the synchSM and note that the bell ring will occur 100 ms or less after A0 becomes 1, as also validated by the timing diagram obtained using RIBS/RIMS/RITS.

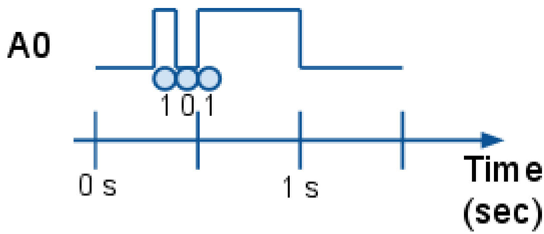
Input conditioning

Sensors are not perfect. The values they provide to a microcontroller commonly must be adjusted to reduce the impact of such imperfections; such adjustment is known as ***input conditioning***. A common input conditioning task is button debouncing. A real button, being a mechanical device, physically bounces a small amount when it is initially pressed, just like a bowling ball dropped to the ground bounces a small amount before coming to rest on the ground. The figure below shows an example of a button's signal, connected to A0, bouncing when pressed once. If the sampling of the

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

input signal is fast, the system may incorrectly treat the button as having been pressed twice, as shown.



Button debouncing is the task of ignoring the bouncing on the signal from a button so that a single press is interpreted by the system as a single press and not as multiple presses. A simple solution is to sample at a slower rate than the bouncing. Modern buttons typically do not bounce for more than 10-20 ms (Ganssle: A Guide to Debouncing), and thus a minimum of 50 ms sampling period is likely sufficient for such buttons; the actual best period depends on the button being used.

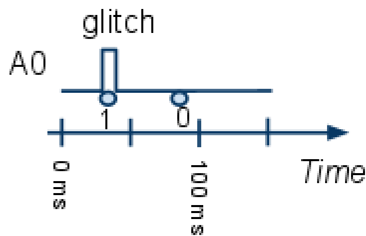
For the above Doorbell example, the debounce requirement introduces another row in the features table:

Debounce achieved by sampling no faster than every 50 ms.	Period should be >= 50 ms

The earlier-chosen period of 100 ms would satisfy the above. Note that the only other valid period would be 50 ms.

Try: For the ToggleButton system from the previous chapter, assume the button may bounce for up to 20 ms. Assume the the maximum latency between a press and toggle is 50 ms. Assume button presses last at least 250 ms and are separated by at least 500 ms. Choose a good period for the ToggleButton synchSM. Also indicate why a much smaller period and a much larger period would be bad.

More generally, ***filtering*** involves ignoring certain input events. A system's inputs may be subject to noise, such as electromagnetic interference (***EMI***) from nearby electrical products (e.g., a powerful vacuum cleaner or plasma TV). EMI can cause sensors or wires to output unintentional 1s (or 0s) for very brief periods of time, known as *spurious signals*, *spikes*, or ***glitches***. While a slower sampling rate reduces the probability of detecting a glitch, a glitch could still be detected as a 1 if the sample is coincidentally taken during the glitch, as shown in the figure below.



A more powerful solution is to require that a 1 be detected for multiple consecutive samples, such as two or three samples, before being confirmed as a legitimate 1. Sampling should obviously be longer than the maximum glitch duration. For a 50 ms sampling rate of a button and glitches that last at most 10 ms, we might require that two consecutive samples be 1 before treating the input as a 1. Note that such a consecutive sample requirement impacts period selection related to minimum event separation time, decreasing the maximum allowable period; requiring two consecutive samples cuts the maximum allowable period in half.

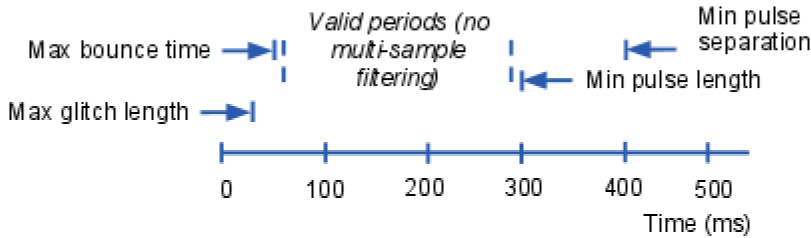
Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

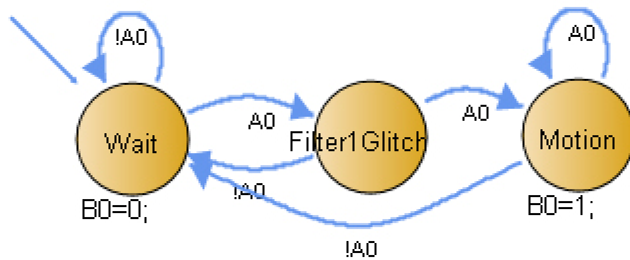
Filtering just reduces the likelihood of glitches being interpreted as actual events, but does not eliminate the possibility entirely. For example, in the above glitch figure, if a second glitch happened to occur 50 ms after the first glitch, the filtering approach would interpret the two consecutive 1s as a valid event.

Note also that filtering increases latency.

Example: Consider a synchSM whose only purpose is to condition the signal on input A0 coming from a motion sensor (A0=1 should mean motion is sensed), into a clean signal on output B0 such that B0=1 indicates motion. Timing features are summarized in the below figure.



From the figure, we see that, without consecutive sample filtering, the period must be greater than 50 ms and less than 300 ms. If we introduce filtering that requires two consecutive 1 samples, the period must be less than 300 / 2 = 150 ms (and still greater than 50 ms). We may choose 100 ms as the period. The input conditioning synchSM is shown below:



Try: Capture the above synchSM in RIBS, specifying a period of 100 ms. Press "Generate C" and "RIMS Simulation", then set RIMS' speed to Slowest. Next, generate 1s on A0, some 1s longer than two 100 ms timer ticks (see the green bar below RIMS' C text box), noting that B0 becomes 1 after some latency, and some 1s shorter -- noting that those shorter A0 pulses (usually) get filtered out and never appear on B0. (You might also try generating two glitches that just happen to appear right at the timer ticks, causing them to unfortunately pass through to B0 as a 1).

Glitches could cause a 1 to temporarily become a 0, and so we might also require that two consecutive 0s be detected before treating the input as having changed from a steady 1 to a 0.

Input conditioning may have to be considered for various sensor types. A switch may generate glitches when moved. A motion sensor or sound sensor may be very sensitive to EMI and thus generate glitches. A temperature sensor may output spurious values on occasion. Handling imperfect inputs must be balanced with conserving microcontroller use; the impact of incorrectly interpreting a spurious input value may not be negative enough to warrant excessive microcontroller use.

The above describes software solutions to sensor imperfections. Designers sometimes instead use hardware solutions to reduce the complexity of software, such as purchasing a sensor with a cleaner output (e.g., less bounce), insulating sensors and wires to reduce glitching from noise, introducing capacitors to filter glitches, etc.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

Outputs

Steady output

The simplest form of output is to generate a steady 0 or 1 bit value. Such a value typically will enable or disable an output device. For example, an output connected to an indicator LED may be 1 (device is powered on) or 0 (device is powered off). Such outputs are assigned to the desired value in appropriate states of the synchSM. A related form of output is a steady integer value output in binary. Previous chapters had examples of such outputs.

Care should be taken to avoid generating output glitches when values should instead be steady. Consider the code below on the left that repeatedly counts the number of 1s on A and writes the count to B:

```
#include "RIMS.h"

unsigned char
 GetBit(unsigned char x, int k) {

   return ( (x & (0x01 << k) != 0 );
}

void main() {
   unsigned char i;
   B = 0; // initialize output
   while (1) {
      B=0;
      for (i=0; i<8; i++) {
         if (GetBit(A, i)) {
            B++;
         }
      }
   }
}
```

```
#include "RIMS.h"

unsigned char
 GetBit(unsigned char x, int k) {

   return ( (x & (0x01 << k)) != 0 );
}

void main() {
   unsigned char i;
   unsigned char cnt; // use to avoid
                      // glitches on B
   B = 0; // initialize output
   while (1) {
      cnt=0;
      for (i=0; i<8; i++) {
         if (GetBit(A, i)) {
            cnt++;
         }
      }
      B = cnt;
   }
}
```

The code on the left will result in glitches on B as 1s are found and B is incremented. The code on the right uses a variable cnt to count, and then updates B when done.

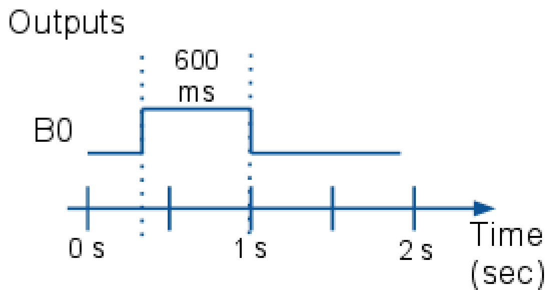
Try: Run the code on the left in RIMS and observe the glitches, especially as more inputs are set to 1; you may have to move RIMS' Execution Speed slider to a slower setting to see the glitches. Next, run the code on the right, and note that no glitches occur.

Pulse output

Embedded systems sometimes must generate an output pulse. A ***pulse*** is a change on a signal from 0 to 1 and back to 0 again, holding the 1 for a particular time interval, known as the pulse *duration*. The figure below shows a single pulse. The pulse duration is 600 ms. Unlike a steady output, a pulse is typically used to trigger an output device. For example, one embedded device may output a pulse, signaling a second embedded device to increment its counter.

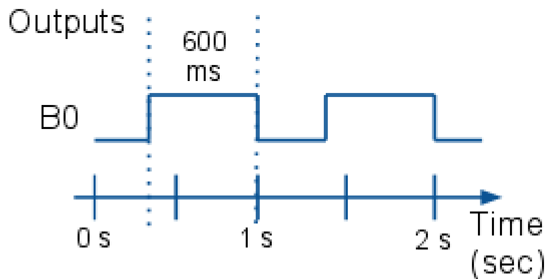
Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu



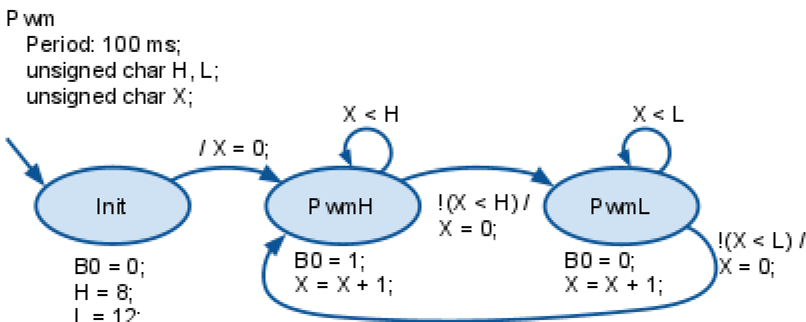
Pulse width modulation

Embedded systems sometimes generate a periodic sequence of pulses. The figure below shows two pulses from such a sequence. The first pulse is shown with dotted lines surrounding it. The pulse duration is 600 ms. The second pulse is also 600 ms. Signal B0 is a ***periodic signal*** because its pattern repeats, in this case with a period of 1 second. The signal is low for the first 400 ms of the period and then high for 600 ms. The ***duty cycle*** is the percentage of time the signal is high during the period, in this case 600/1000 = 60%. A signal with a duty cycle of 50% is called a ***square wave***.



Try: Draw a waveform for a periodic signal with a 500 ms period and an 80% duty cycle.

A pulse width modulator (PWM) is a programmable component that generates pulses to achieve a specified period and duty cycle. A PWM can be captured as a synchSM and implemented in C on a microcontroller. Assume a PWM allows a period that is a multiple of 1 second (e.g., 2 seconds) and a duty cycle that is a multiple of 10% (e.g., 40%). To implement this PWM, a synchSM can be defined with a period of 100 ms. Variables H and L store the number of synchSM ticks to hold the signal high and low, respectively; for 2 seconds and 40%, H would be 2000ms*0.40 = 800 ms, meaning 8 ticks, and L would be 12.



One use of a PWM is to control a DC (direct current) motor (Wikipedia: DC motor). The motor spins when its input is 1 and coasts when its input is 0. A PWM can be used to achieve the desired spinning speed. Suppose a DC motor will spin at a maximum of 1000 revolutions per minute (rpm) if its input is held at 1. Instead, the input can be set by a PWM with a period of 100 ms and a duty cycle of

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

50% to achieve spinning at 500 rpm. A 40% duty cycle spins the motor at 400 rpm. Many common products, such as a cordless drill, use a PWM for such purposes. On the average, a PWM's output, in terms of energy, is equivalent to its duty cycle. (Wikipedia: Pulse Width Modulation)

Another use of a PWM is to generate a tone on a speaker. A square wave can be created (50% duty cycle) with a period selected for the desired tone frequency. Consider generating music notes. A musical note can be generated using a signal of a particular frequency. Frequencies for some notes are shown in the table below. These are low notes, corresponding perhaps to the leftmost keys on a piano (for reference, "middle C" is C_4 which is 261.63 Hz).

Note	Freq. (Hz)	Period (sec)
C_0	16.35	0.061
D_0	18.35	0.054
E_0	20.60	0.049
F_0	21.83	0.046
G_0	24.50	0.041
A_0	27.50	0.036
B_0	30.87	0.032
C_1	32.70	0.031

The required intervals are half of the notes' periods. The greatest common denominator of the required intervals is 1 ms. Suppose each switch A0, A1, ... corresponds to notes C_0, D_0, ..., with higher inputs having priority over lower inputs. An SM can be constructed with a period of 1 ms that spends X ticks in a state that outputs 1 and X ticks in a state that outputs 0. Before those two states, another state can set X to the interval required by the input switch. (Note: RIMS does not presently support periods less than 20 ms). Tones generated by square waves sound rough (sine waves sound smoother).

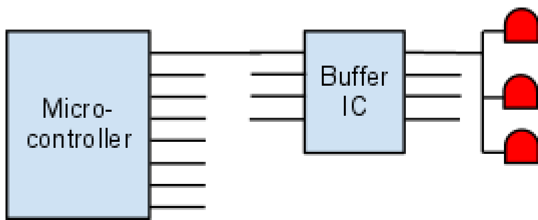
In the music example, another consideration is the duty cycle. A duty cycle of 50% will deliver the maximum power to the speaker. At 50% duty cycle, the speaker diaphragm will travel inwards and outwards in equal amounts, and to the maximum extent, generating a loud sound. A duty cycle of less than 50% will move the diaphragm inwards (or outwards) less, and hence will generate a softer sound. Interestingly, a duty cycle of more than 50% will also generate a softer sound (with respect to a 50% duty cycle). Can you see why?

I/O electrical issues

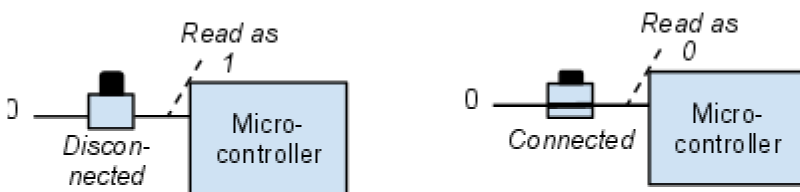
Most digital circuits (sensors, actuators, microcontrollers, and other logic devices) are designed to easily connect together. In other words, the output of one device (e.g., a temperature sensor) can directly be connected to the input of another device (e.g., a microcontroller). In some cases, such direct connections may not work. A common case is that a microcontroller output cannot sufficiently drive the device/devices to which it connects. Suppose a microcontroller's output has a max voltage of 5 V and a max current of 25 mA. That output clearly cannot directly drive a DC motor that requires 12 V and 1 A. Likewise, that one output cannot drive 5 LEDs that each require 15 mA. A ***buffer IC*** may be added to a microcontroller output to increase the voltage and/or current drive capability, such as a 74HC125 IC. Conversely, a device like a particular LED may require a lower voltage (and may be damaged by too high a voltage); a *load resistor* may be added to reduce a microcontroller's output voltage to the desired voltage. Similarly, a device that connects to a microcontroller's input may require a load resistor.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu



Microcontrollers commonly support two configurations for an input pin, normal and pull-up. In the normal configuration, if a pin is not driven with a 0 or a 1 (e.g., if the pin in not unconnected), reading the pin yields an indeterminate value; the pin should thus always be driven. In contrast, in the ***pull-up configuration***, if a pin is not driven with a 0 or a 1, reading the pin yields a 1. A pull-up configuration is useful for a *passive button*, for example, wherein pressing the button causes a connection between the button's terminals and hence a connection between 0 and a microcontroller input pin (so the pin is read as 0), while releasing the button causes disconnection (so the pin is read as 1).



If a microcontroller does not have internal pull-up support, one can add a resistor (typically 1K to 5K Ohms) from the pin to the positive terminal of the power supply, creating the pull-up condition.

Careful attention to I/O electrical characteristics specified in a microcontroller's datasheet and basic electronics knowledge are necessary to properly connect microcontrollers with other physical components.

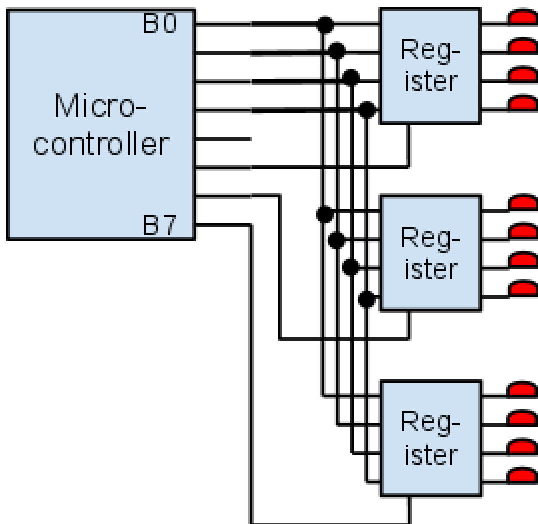
Dealing with too few pins

Time-multiplexed output with registers

A common problem is that a more outputs are needed than are available on a microcontroller. For example, consider wishing to drive 12 LEDs but with only 8 microcontroller output pins. A solution is to add external registers (such as a 74HC173) to extend the number of outputs. Below, the program would first set B3-B0 to the values for the top four LEDs and then pulse B5 to store those values in the top 4-bit register. The program would then set B3-B0 to the values for the middle four LEDs and then pulse B6. Likewise for the bottom four LEDs, pulsing B7.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu



The above approach is known as ***time-multiplexed output***, wherein the desired output data is divided into pieces and then sent one piece at a time.

Time-multiplexed output with rapid refresh

In some cases, a microcontroller can time-multiplex output without storing the data in external registers. For example, the above 4-bit registers could each be replaced by a group of four transistors, each group controlled by B5, B6, or B7 (a transistor passing its input to its output if the control is 1, else disconnecting its output). Because the LEDs don't fade off instantly when a 1 is removed, and because a human's eyes and brain have limited ability to see very fast visual events, an LED may appear to be on even if it is driven by a 1 only intermittently. As long as each LED is ***refreshed*** (meaning rewritten with the appropriate value) frequently enough, typically about 50-100 times per second, it can appear to a human to be constantly lit; if the refresh rate is too slow, the human will see flickering. (Rapid refresh is akin to the classic entertainment act of keeping spinning plates balanced on sticks). Furthermore, the brightness of LEDs can be adjusted by varying the refresh rate and/or the duration of 1s.

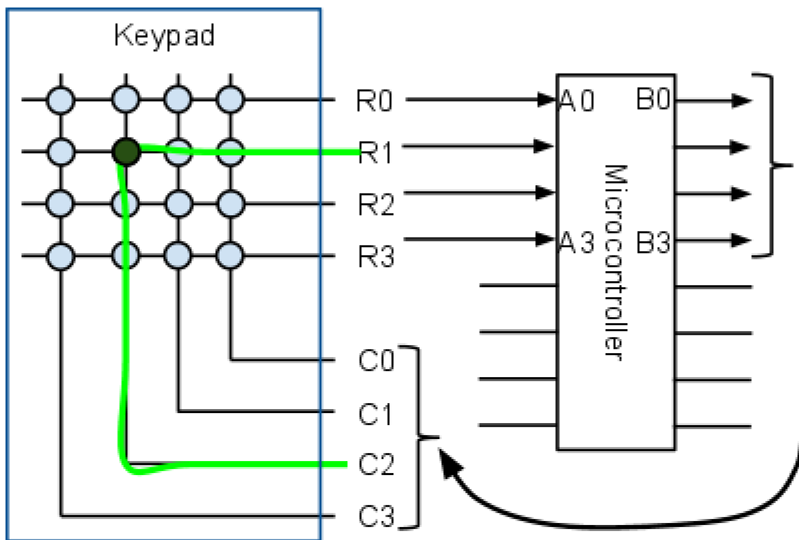
Time-multiplexed input: Keypad example

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.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu



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 time.

```
// Returns 1 if the key at row/col is pressed. Returns 0 otherwise.
unsigned char GetSingleButton(unsigned char row, unsigned char col)
{
   /* Set B3..B0 outputs to 1 except the bit at position 'col',
      ensuring B7..B4 are not modified. */
   B = (B | 0x0f) & ~(1 << col);

   /*  Now read the input pin at position 'row'. If the button at
       row/col is pressed, it will go from pull-up state (1) to 0. */
   return ((A & (1 << row)) == 0 ? 1 : 0);
}
```

The function can now be used in a simple scanning function. This function, when called, will iterate over the buttons, from left to right and top to bottom, and return an index of 1, 2, 3, 4 ... 16, corresponding with the first button that is found to be pressed, or returning 0 if no button was pressed.

```
/* Returns an index (1 through 16) corresponding to the button pressed,
  or 0 if no button is pressed. Scanning order is left to right and
  top to bottom, returning when finding the first pressed button. */

unsigned char ScanKeypad()
{
   unsigned char i, j;
```

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

```
    for (i=0; i<4; i++) {
        for (j=0; j<4; j++) {
            if (GetSingleButton(i, j)) {
                return (i * 4) + j + 1;
            }
        }
    }
    return 0;
}
```

This scanning algorithm does not detect multiple button presses; instead, the algorithm returns the pressed button that is closest to the top and left, giving such buttons higher priority. If multiple button presses must be detected, the function needs to maintain a list of buttons that are discovered to be pressed, and return that list to the caller at the end of a full scan.

Copyright © Frank Vahid and Tony Givargis 2011

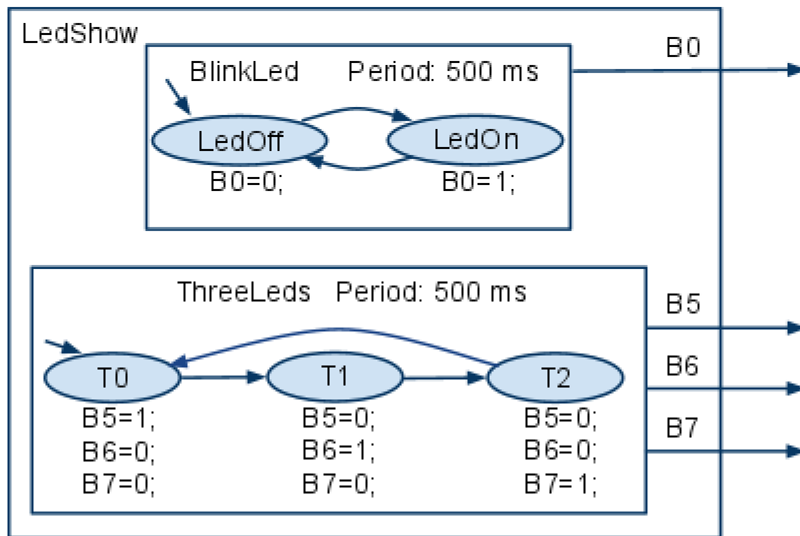
Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

Chapter 6: Concurrency and Multiple SynchSMs

A ***task*** is a unique continuously-executing behavior, such as the task of toggling an LED whenever a button is pressed, or the task of sounding an alarm when motion is sensed. Earlier chapters described systems each implementing just one task, captured as sequential instructions in C, as an SM, or as a synchSM. ***Concurrently*** means to carry out multiple tasks at the same time. Referring to multiple tasks implies that the tasks are concurrent.

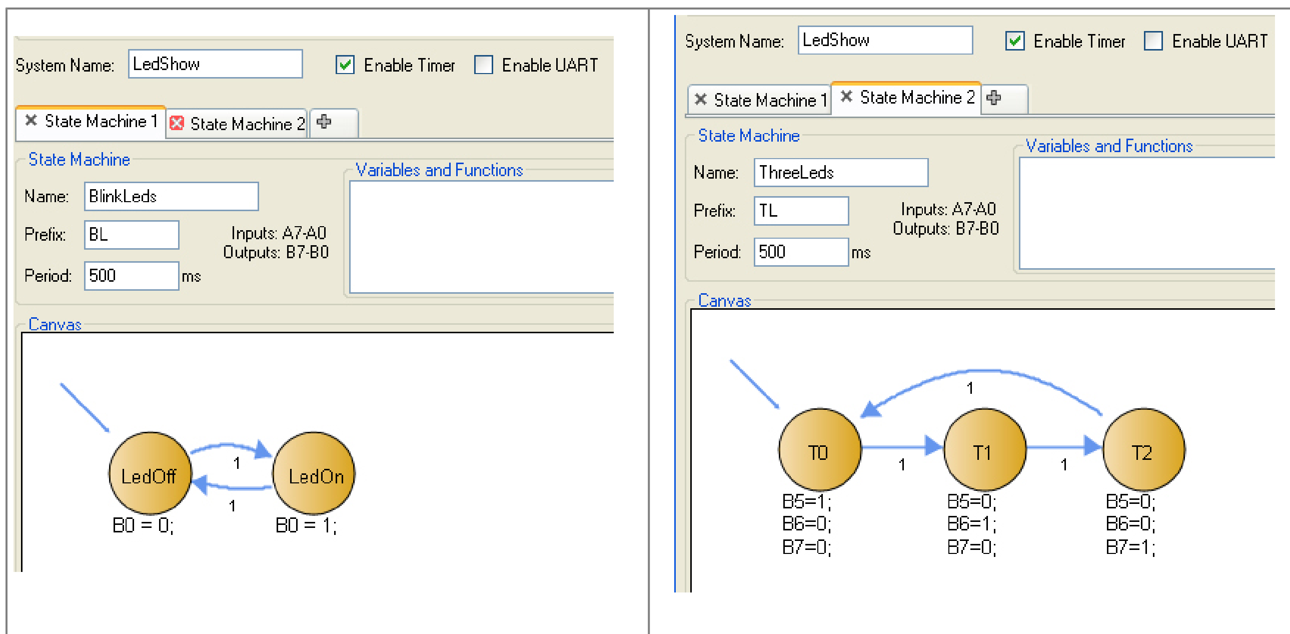
Multiple synchSMs

Many systems implement multiple tasks. For example, a system named LedShow may blink an LED connected to B0 on for 500 ms and off for 500 ms, repeatedly. That same system may also light three LEDs connected to B7B6B5 in the sequence 001, 010, and 100, 500 ms each, repeatedly. While one could try to capture the system's behavior with a single task, a simpler approach uses two tasks, each a synchSM. The following block diagram illustrates. A ***block diagram*** shows each task as a block (a rectangle), and uses a directed line to show that a block writes to an output (or reads from an input).



RIBS allows capture of multiple tasks using multiple tabs. The left image below has the first tab selected, and the right image has the second tab selected, for a system named LedShow.

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu



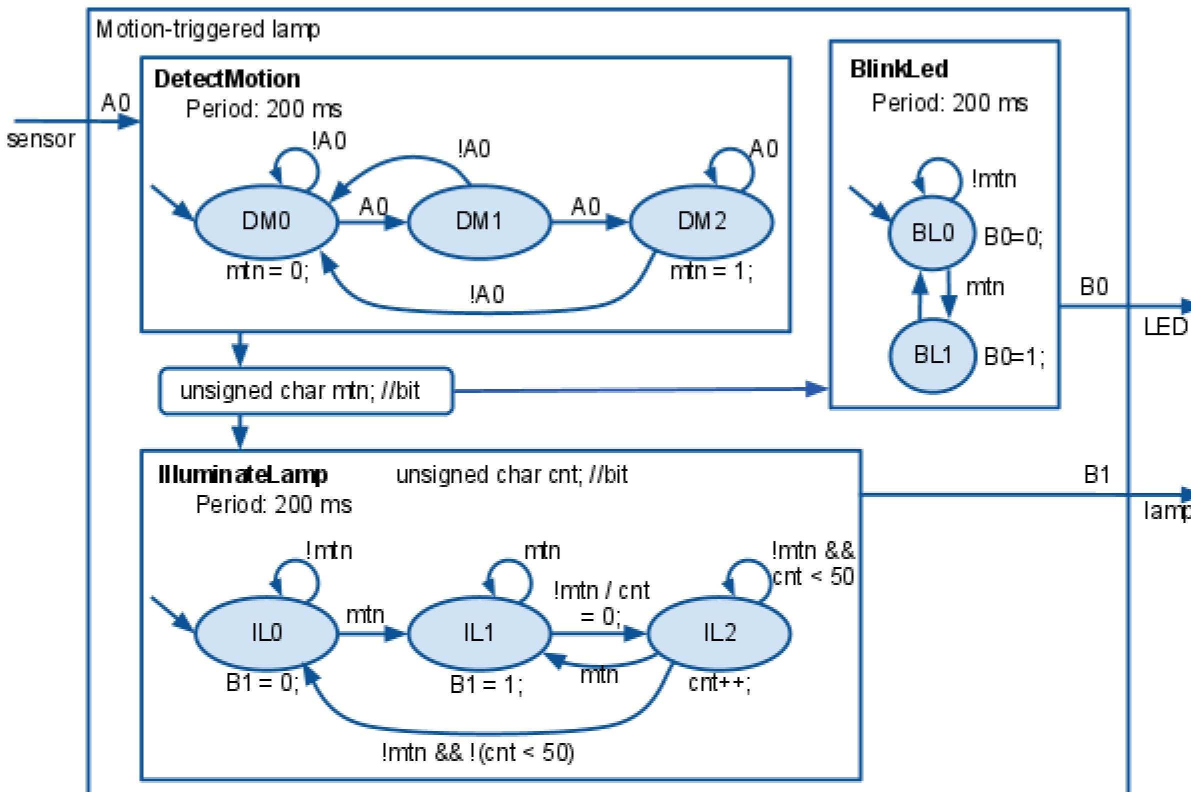
Try: Capture the LedShow system's two synchSMs using RIBS as above . Press "Generate C" and then "RIMS Simulation". Click on each tab in RIBS to see each synchSM executing, and note how each corresponds to RIMS' LED outputs.

Shared variables

Sometimes a system's behavior is best captured as multiple tasks even though the tasks are not entirely independent. In such cases, the tasks may share variables to communicate information between the tasks. For example, consider a motion-triggered lamp system with a motion sensor connected to A0. The system defines motion as A0=1 for two consecutive 200 ms samples. When motion is detected, the system should illuminate a lamp (by setting B1 to 1), keeping the lamp on for 10 seconds past the last detected motion. The system should also blink a small LED (connected to B0) for 200 ms on and off while motion is detected. Trying to capture all that behavior with a single synchSM would be challenging. A simpler capture approach uses three tasks as shown below.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu



The DetectMotion task detects motion and informs the other tasks of the detected motion by setting a shared variable mtn. The other two tasks read mtn and respond accordingly: BlinkLed blinks the LED, and IlluminateLamp turns on the lamp for the required time.

Note the clear "separation of concerns." DetectMotion is concerned only with conditioning the A0 input. BlinkLed is concerned only with blinking the LED while motion is detected, and Illuminate lamp only with keeping the lamp on for the appropriate time. These tasks were simple to capture, easy to understand, and straightforward to independently modify -- e.g., motion could be redefined in the DetectMotion task as A0=1 for 3 samples, without requiring any changes to the other two tasks.

Note that a block diagram shows concurrent tasks and global variables. The directed lines of the block diagram (from DetectMotion to mtn, and from mtn to IlluminateLamp) indicate whether the global variable is written or read by a task. Those directed lines should not be confused with the directed lines of an SM, which indicate flow of control from one state to another. A common mistake is to draw directed lines in a block diagram with the intention of describing flow of control, which makes no sense because all items in a block diagram execute concurrently.

Try: A portable electric heater system should turn on heat (B0=1) when the actual temperature (A3-A0) is less then the desired temperature (A7-A4). To let users see that the heater is on, the system should blink B1 on for 500 ms and off for 500 ms. Capture the system using multiple tasks and a shared variable.

__Only one task should have actions that write to a shared variable.__ Otherwise, if two tasks write to a shared variable, the resulting behavior may be indeterminate -- if both tasks simultaneously write different values, the resulting value is not defined and may depend on arbitrary implementation factors, such as which tick function is called first when the tasks are implemented in C.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

Converting multiple synchSMs to C

SynchSM tasks

Multiple tasks described as synchSMs can be converted to C using nearly the same technique as for a single synchSM. Assume two synchSMs have the same period, such as the earlier BlinkLed (BL) and ThreeLeds (TL) synchSMs having periods of 500 ms. Their main code looks as follows:

```
#include "RIMS.h"

//LedShow C code, having two tasks

enum BL_States { BL_LedOff, BL_LedOn } BL_State;
enum TL_States { TL_T0, TL_T1, TL_T2 } TL_State;

volatile unsigned char TimerFlag=0;

void TimerISR() {
   TimerFlag = 1;
}

void BL_Tick() {
   ... // Standard switch statements for SM
}

void TL_Tick() {
   ... // Standard switch statements for SM
}

void main() {
   B = 0; //init outputs
   TimerSet(500);
   TimerOn();
   BL_State = -1;
   TL_State = -1;
   while (1) {
      BL_Tick();             // Execute one tick of the BlinkLed synchSM
      TL_Tick();             // Execute one tick of the ThreeLeds synchSM
      while (!TimerFlag){}   // Wait for timer period
      TimerFlag = 0;         // Lower flag raised by timer
   }
}
```

The BL_Tick and TL_Tick functions would be defined indentically to synchSMs converted to C in the earlier chapter. Every 500 ms, the BL_Tick and the TL_Tick functions will be called, and thus each synchSM will proceed one tick every 500 ms. The order in which the two functions are called in the main function is arbitrary.

Note that the implementation in C does not perfectly execute the two tasks concurrently, but instead the ThreeLeds task executes slightly after the BlinkLed task. However, the two tasks execute each tick quickly and thus appear to a user to execute concurrently. In fact, even when a microcontroller executes just one task, that task is supposed to execute instantly after each timer tick but in

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

reality involves some delay due to non-zero execution time of microcontroller instructions; for the concurrent tasks, the second task just experiences a slightly longer delay. Such serialization of concurrent tasks is standard in computing systems (desktop or embedded) and is called *multi-tasking*. Executing each task for every period is a straightforward form of what is called ***round-robin*** task execution.

Recall the earlier requirement that a synchSM's state actions should never include statements that wait, but rather should run to completion. The need for this requirement is clearly evident in the above C code; if any of the tasks included a wait, the other tasks might not get a chance to execute before 500 ms passes.

Try: For the above LedShow example captured in RIBS, press "Generate C", then press "RIMS Simulation". On RIMS, press "Break", and examine the C code, noticing the two tick function definitions, and their calls from main(). Press "Step" repeatedly and observe how one tick function gets called, followed by the other tick function. Press "Continue" and notice that the serialization of the two tasks is unnoticeable to the human eye.

The reader may be wondering how to convert C two synchSMs having different periods into C; the next chapter deals with that issue.

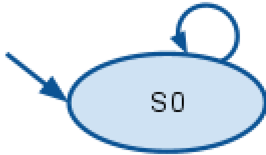
Other task types

Some tasks may originally be captured as an SM or just sequential code rather than a synchSM. For conversion to C along with synchSM tasks, each non-synchSM task can be first rewritten as a synchSM. An SM merely needs to be given a period. Sequential code can be rewritten as a single-state synchSM having that code as its actions, and having a single true-condition transition pointing back to itself. For example, consider the following sequential code task:

```
unsigned char cnt;
unsigned char i;
while (1) { // Repeatedly look for four 1s on A
   cnt=0;
   for (i=0; i<8; i++) {
      if (GetBit(A, i)) {
         cnt++;
      }
   }
   B1 = (cnt >= 4);
}
```

(Assume the GetBit function is defined). To execute that task in C along with the BlinkLed and ThreeLeds tasks, the task can be rewritten as a single-state synchSM:

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu



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 CF_Tick() can be written for the CountFour synchSM, and the main code's while loop will simply be extended with a third task:

```
...
while (1) {
   BL_Tick();              // execute one tick of the BlinkLed synchSM
   TL_Tick();              // execute one tick of the ThreeLeds synchSM
   CF_Tick();              // 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 CF_Tick() { // single-state synchSM
  cnt=0;
  for (i=0; i<8; i++) {
     if (GetBit(A, i)) {
        cnt++;
     }
  }
  B1 = (cnt >= 4);
}
```

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

Global versus local variables

When converting to C, a variable global to multiple synchSMs can be declared as a C global variable. Note that synchSM local variables were in an earlier chapter (for a single synchSM) converted to C global variables, but that approach may not work for multiple synchSMs if multiple synchSMs have variables with the same names, or if a synchSM has a variable with the same name as a global variable. One solution is to convert a synchSM's local variables to C static local variables inside the synchSM's tick function. Declaring the synchSM's variables within the tick function ensures the variables won't conflict with other synchSM C variable names or C global variable names. However, variables within a C function are normally temporary, existing only during the function call. To cause those variables to be permanent and thus maintain their values across function calls, as required to correctly execute a synchSM, the variables declarations have the keyword "static" prepended. For the above motion lamp example, the variables would be declared as follows:

```
unsigned char mtn; // Global variable for synchSMs converted to C global

void DM_Tick() { // DetectMotion synchSM tick function
   ...
}

void IL_Tick() { // IlluminateLamp synchSM tick function
   static unsigned char cnt; // Local synchSM variable converted to
                             // C static local
   ...
}
```

In RIBS, ideally the "Generate C" tool would automatically prepend the keyword static to any synchSM local variables. However, RIBS just copies the C code in the variables text box into the generated C program. Thus, when capturing multiple synchSMs using RIBS, the programmer must declare synchSM variables as static, else the generated C may not behave correctly because the variables will not retain their values across calls to the synchSM's tick function.

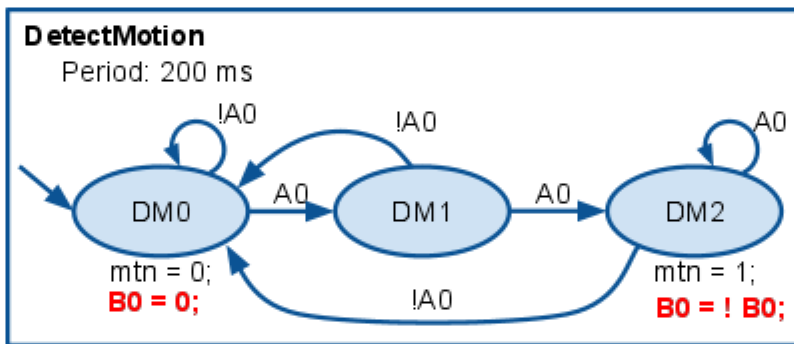
Keeping distinct behaviors distinct

An important skill is recognizing from a system's desired behavior description whether there are naturally concurrent behaviors, and then creating a separate synchSM for each.

Some people try to merge concurrent behaviors into a single task to reduce code. Such merging results in unnatural capture of desired behavior, ultimately resulting in programs that are harder to understand, harder to maintain, and contain more bugs. For example, in the earlier motion triggered lamp example, one might be tempted to merge the blinking LED behavior with the detect motion behavior as follows:

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu



The functionality of the DetectMotion task may now be harder to understand. Furthermore, if we later decided to modify the blinking pattern to be on for 400 ms and off for 200 ms, the task could be difficult to modify. In contrast, modifying the original BlinkLed task would be easy. The lesson is:

Less code does not mean better code

Try: A store entry system has a person sensor connected to A0. When a person is detected, the system generates a tone by setting B0 to 1 for 1000 ms. To indicate that the system is on, the system always blinks an LED by setting B1 to 0 for 500 ms and to 1 for 500 ms, repeatedly.

The above system behavior is most easily captured as two distinct synchSMs. Using a 500 ms period for each enables straightforward round-robin processing in C.

Obviously, not all system behavior is most naturally captured as concurrent tasks. Sometimes a single task is best.

Try: A highway construction sign can inform drivers to move left, right, or both left and right. If A1A0=10, move left is indicated by setting LEDs connected to B5B4B3 to 001, 010, and 100, repeatedly for 500 ms each. If A1A0=01, move right is indicated by setting B2B1B0 to 100, 010, 001. A1A0=11 indicates both move left and right, with the sequences operated in synch. A1A0=00 turns off all LEDs.

Because the various behaviors are actually synchronized with one another, capturing the behaviors as a single synchSM turns out to be easiest for the above system.

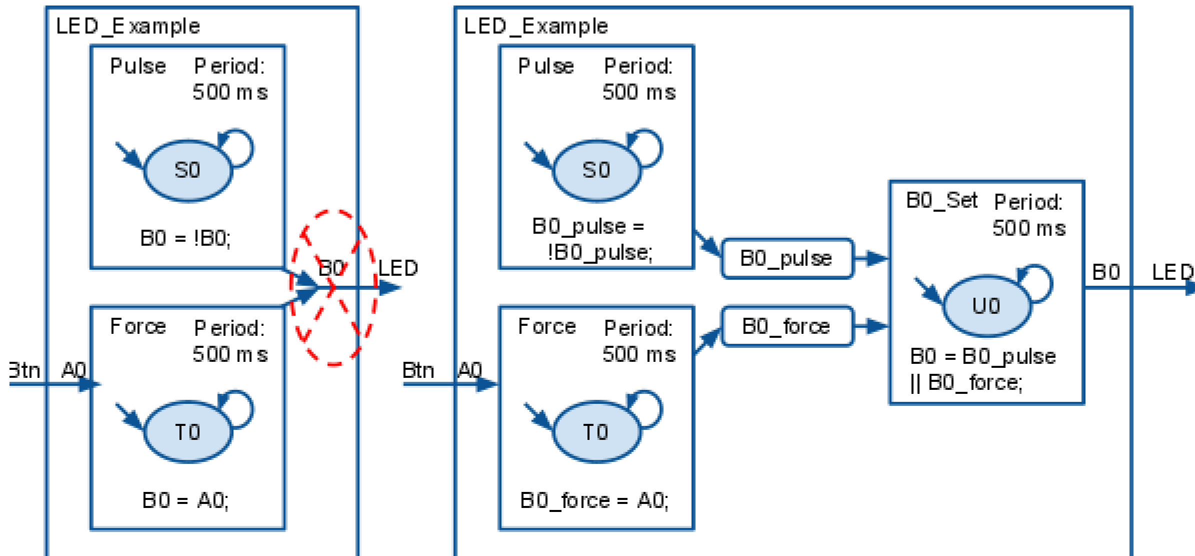
Sometimes behaviors read the same input but are still independent. For example, a system may have a button connected to A0. When the button is pressed, a tone is generated by setting B0 to 1 for 1 second, and an LED is blinked twice by setting B1 to 1, then 0, then 1, then 0, for 500 ms each. The system is most easily captured using two synchSMs, each reading A0 and responding appropriately.

Two tasks writing to one global variable

We commonly desire to capture two tasks that write to one global variable (or to RIM's B output, which is a global variable), but only one task should write to a global variable, as stated earlier. For example, consider an LED (connected to B0) that should blink 500 ms on and 500 ms off, but that can be forced to stay on by a switch connected to A0. We might try to capture this system as two synchSMs as below. Task Pulse sets B0=!B0 every 500 ms, and task Force sets B0=1 if A0=1. But two tasks should not write to a shared variable. The solution is to introduce another task, B0_Set, to manage B0's value. Task Pulse writes to a new shared variable B0_pulse, and task Force writes to a new shared variable B0_force. Task B0_Set defines what B0's value should actually be, based on the two shared variables. For this system, B0_Set should repeatedly set B0 to the OR of the two variables, i.e., "B0 = B0_pulse || B0_force".

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu



A similar situation arises when one task writes to part of group output B and another task writes to another part; even though each task writes to a unique subset of B, the two tasks are still writing to the same shared variable B, which is not allowed. For example, suppose Task1 writes 1010 to B's low nibble and Task2 writes 0101 to B's high nibble. Earlier examples would have Task1 write to B using a mask, e.g., "B = (B & 0xF0) | 0x0A;", which preserves the high-nibble while zero'ing out the low nibble and then writing 1010 ("A" in hex) to the low nibble. Task2 might have "B = (B & 0x0F) | 0x50. However, both tasks should not write to shared variable B, even though we can see that they don't interfere with one another -- allowing such writing could cause problems later if one of the tasks is changed. A solution is to add a third task, B_Set, that manage's B's value. In this case, we might make clear that Task1 writes to the low nibble and Task2 writes to the high nibble, by having Task1 write to a new shared variable B_low and Task2 write to B_high. Then B_Set would repeatedly set "B = (B_high << 4) | B_low;".

Two tasks may *read* an input or a shared variable; the restriction is against two tasks *writing* to a shared variable.

Note that using an additional task to manage a share variable's value increases system latency. As such, a faster period may be desired for that task (see next chapter for discussion of different-period tasks). Note also that the task may be a single-state synchSM and thus, when converted to C, may have a simplified tick function as described earlier.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

Chapter 7: A Simple C Task Scheduler

The previous chapter simplified task execution in C by using the same period for every task. On the other hand, different periods are sometimes desired. For example, consider the earlier LedShow system where the BlinkLed task should blink on and off not for 500 ms each but rather 1500 ms each, whereas the ThreeLeds task still lights LEDs for 500 ms each. BlinkLed could be modified to count three states of 500 ms each, but it is more intuitive to just change BlinkLed's period from 500 ms to 1500 ms. This change is easy to make on the block diagram; the challenge arises when converting to C.

Converting different-period tasks to C

Different-period tasks can be converted to C via a method that uses new variables for counting timer ticks. The timer period can be set to some small value, and then the appropriate number of timer ticks can be counted to determine whether to call a task's tick function on a particular pass through the main code's "while(1)" loop. The following code illustrates for the LedShow example with the BlinkLed (BL) task having a 1500 ms period and the ThreeLeds (TL) task having a 500 ms period.

```
//LedShow C code

#include "RIMS.h"

enum BL_states { BL_LedOff, BL_LedOn } BL_state;
enum TL_states { TL_T0, TL_T1, TL_T2 } TL_state;

volatile unsigned char TimerFlag=0;

void TimerISR() {
   TimerFlag = 1;
}

void BL_Tick() {
   ... // standard switch statements for SM
}

void TL_Tick() {
   ... // standard switch statements for SM
}

...
void main() {
   unsigned long BL_elapsedTime=0;
   unsigned long TL_elapsedTime=0;
   const unsigned long timerPeriod = 100;

   B = 0; //init outputs
   TimerSet(timerPeriod);
   TimerOn();
   BL_state = -1;
```


Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

```
   TL_state = -1;
   while (1) {
      if (BL_elapsedTime == 1500) { // 1500 ms period
         BL_Tick();          // Execute one tick of the BlinkLed synchSM
         BL_elapsedTime = 0;
      }
      if (TL_elapsedTime == 500) {  // 500 ms period
         TL_Tick();          // Execute one tick of the ThreeLeds synchSM
         TL_elapsedTime = 0;
      }
      while (!TimerFlag){}    // Wait for timer period
      TimerFlag = 0;          // Lower flag raised by timer
      BL_elapsedTime += timerPeriod;
      TL_elapsedTime += timerPeriod;
   }
}
```

The code sets the timer period to 100 ms. Each time through the "while(1)" loop, the elapsed time variables are increased by 100. When BL_ElapsedTime is 1500, BL_Tick will be called. When TL_ElapsedTime is 500, TL_Tick will be called. Note that the timer period MUST evenly divide all task periods. Choosing the task period as the *greatest common divisor (GCD)* ensures this. The GCD of 1500 and 500 is 500, and thus the above example could have the timer period set to 500 instead of 100.

Slightly better code might use constants for the task periods:

```
void main() {
   const unsigned long BL_period=1500;  // 1500 ms
   const unsigned long TL_period=500;   // 500 ms
...
     if (BL_elapsedTime == BL_period) {
...
...
```

Unfortunately, adding the elapsedTime variables for counting timer periods results in cluttered main code, especially if there are tens of tasks. Furthermore, each task has several items declared in different places (state variable, period, elapsed time counter, etc.) The code is becoming harder to maintain. A more structured approach is needed.

Creating a task structure in C

Good programming involves keeping related items together. In the above example, we can see that each task, such as BlinkLed (BL), has several related items:

- A state variable: BL_state
- An SM tick function: BL_Tick()
- A period: BL_period (1500)
- An elapsed time variable: BL_elapsedTime

But these items are spread around the C code. C provides a mechanism to collect related items in one place. A ***struct*** (short for *structure*) is a C construct that allows several variables to be grouped together under a single name. We define a new type for a task using the struct shown:

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

```
typedef struct {
   signed   char state;         // Task's current state
   unsigned long period;        // Task period
   unsigned long elapsedTime;  // Time elapsed since last task tick
   int (*TickFct)(int);         // Task tick function
} task_t;
```

typedef is a C construct that defines a new data type, in contrast to the built-in C data types like int and char. Variables can be defined as being of the new type. Our new type is called "task_t". In C/C++, programmers commonly append an "_t" to newly created types, making it easy to recognize these identifiers (names) as being data types. New variables of type "task_t" can be declared as follows:

```
task_t BL_task;
task_t TL_task;
```

Each variable has several fields (declared in the struct) that can be written and read by using a period "." (known as *dot notation*) as if each field had been declared as a separate variable. For example:

```
BL_task.state = -1;          // Indicates initial state
BL_task.period = 1500;       // Tick function should be called every 1500 ms
BL_task.elapsedTime = 1500;  // Time since last tick; initial 1500 causes tick
                             //    function to be called at program start
```

The last field in the above struct has a syntax that some readers may not be familiar with. That field is for a ***function pointer***, which can be set to point to an existing function, in the above case a function having one parameter of type short int, and returning a value of short int type also. That field can be set as follows:

```
BL_task.TickFct = &BL_Tick;
```

The "&" in front of BL_Tick obtains the memory address of the BL_Tick function, to which BL_task.TickFct is set. Then, the BL_Tick function can be called as follows:

```
BL_task.state = BL_task.TickFct(BL_task.state);
```

The above assumes BL_Tick has been redefined to take the current state as a parameter and to return the next state. Redefining BL_Tick as such eliminates the need to declare state variables as global variables. Thus, tick functions will take an int parameter and have a return type of int, as follows:

```
int BL_Tick(int state) {
   switch(state)
      ...
         state = ...
      ...
   return state;
}
```

Although we strive to avoid use of integer type "int," int is used here because a C enum creates constants of int type; short or long does not work for some compilers.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

All items related to the BlinkLed task are now grouped in a single variable, BL_task. Likewise, all items related to ThreeLeds are grouped into TL_Task. However, the real usefulness of such grouping becomes clear if we put all a system's tasks into an array:

```
task_t *tasks[] = { &BL_task, &TL_task };  /* initialized array of
                                              pointers to tasks */
const unsigned short numTasks = sizeof(tasks) / sizeof(task_t*);
     /* variable automatically set to number of tasks in array "tasks" */
```

The first line of code defines the variable "tasks" to be an array of pointers to "task_t". That statement initializes the array to contain pointers to the two previously-defined tasks . Note that in C, if the number of elements in an array is not specified (i.e., []), the C compiler will use the number of items in the initialization list to automatically set the size. In our example, the C compiler will make the array "tasks" large enough to accommodate two pointers.

The second line of code defines a constant "numTasks", which will be used by later code to know how many tasks are in the array. We could have written that line as "`const unsigned short int numTasks = 2;`", but such hard-coding of the size could lead to inconsistent code if we add another task to the array but forget to change the 2 to a 3. Thus, we instead use a C construct called sizeof to automatically initialize "numTasks". ***sizeof*** returns the size of its operand in bytes. In our example, sizeof(tasks) will return a number, lets say N bytes, needed to store our 2 pointers. However, depending on the microcontroller in use, the size of a pointer may be 2, 4, or even 8 bytes. Therefore, we need to divide the size of the array by the size of a single pointer to determine the number of elements in the array. We accomplish this by dividing N by sizeof(task_t*), i.e., the size of a single pointer. This expression will thus return the *number of elements* in the array rather than the number of bytes. The code is also portable, meaning regardless of the microcontroller, the result will always be correct. To illustrate, assume that a pointer requires 4 bytes, so a call to sizeof(task_t*) will return 4. The "tasks" array would occupy 8 bytes in memory, so the call to sizeof(tasks) will return 8. Hence, the expression sizeof(tasks) / sizeof(task_t) will yield 2, as expected.

One last change is needed for the above code to work. Namely, the tasks must be defined as:

```
static task_t BL_task;
static task_t TL_task;
```

The C keyword "static" has been prepended to the task declarations. In C, a variable declared within a function (including the "main" function) is allocated temporary memory in the program's memory space, lasting only the duration of the call to a function. Prepending "static" to a variable declaration caused allocation of permanent memory intead. We need the tasks to have a known (permanent) memory address so that we can construct an array of them, namely the above-defined "tasks" array.

Code for a Simple Task Scheduler

The benefit of the above efforts becomes evident in the main code. We can now simply use a for loop that processes each task in a round-robin manner:

```
while(1) {
   // For each task, call task tick function if task's period is up
   for (i=0; i < numTasks; i++) {
      if (tasks[i]->elapsedTime == tasks[i]->period ) {
         // Task is ready to tick, so call its tick function
         tasks[i]->state = tasks[i]->TickFct(tasks[i]->state);
```

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

```
            tasks[i]->elapsedTime = 0; // Reset the elapsed time
         }
         tasks[i]->elapsedTime += timerPeriod; // Account for below wait
      }
      while (!TimerFlag); // Wait for next timer tick
      TimerFlag = 0;
   }
```

Note that the above scales easily for more tasks, e.g., 5 tasks, 10 tasks, or even 100 tasks; *the for loop code stays exactly the same*. The for loop code carries out what is known as a scheduler. A ***scheduler*** determines when each task should be executed in a multiple-task system. A scheduler is commonly found inside the code for an operating system (OS), but can instead be included directly in user code as above, which is especially useful in the absence of an OS.

In the above C code, to access some data (such as BL_task) via a pointer (such as tasks[0]), we use the pointer dereferencing operator of C "->".

The following shows the complete code for the LedShow system with different-period tasks, including the definition of both tasks, and the scheduler code.

```
#include "RIMS.h"

volatile unsigned char TimerFlag=0;

void TimerISR(void) {
   TimerFlag = 1;
}

typedef struct {
   signed   char state;
   unsigned long period;
   unsigned long elapsedTime;
   int (*TickFct)(int);
} task_t;

enum BL_States { BL_LEDOFF, BL_LEDON };

int BL_Tick(int state) {
   switch(state) { // Transitions
      case -1:
         state = BL_LEDOFF;
         break;
      case BL_LEDOFF:
         state = BL_LEDON;
         break;
      case BL_LEDON:
         state = BL_LEDOFF;
         break;
      default:
         state = -1;
         break;
   }
```

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

```
   switch(state) { // State actions
      case BL_LEDOFF:
         B0 = 0;
         break;
      case BL_LEDON:
         B0 = 1;
         break;
      default:
         break;
   }
   return state;
}

enum TL_States { TL_ONE, TL_TWO, TL_THREE };

int TL_Tick(int state) {
   switch(state) { //Transitions
      case -1:
         state = TL_ONE;
         break;
      case TL_ONE:
         state = TL_TWO;
         break;
      case TL_TWO:
         state = TL_THREE;
         break;
      case TL_THREE:
         state = TL_ONE;
         break;
      default:
         state = -1;
         break;
      }

   switch(state) { //State actions
      case TL_ONE:
         B5 = 1;
         B6 = 0;
         B7 = 0;
         break;
      case TL_TWO:
         B5 = 0;
         B6 = 1;
         B7 = 0;
         break;
      case TL_THREE:
         B5 = 0;
         B6 = 0;
         B7 = 1;
         break;
      default:
```

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

```
         break;
   }
   return state;
}

void main()
{
   const unsigned long BL_period = 1500;
   const unsigned long TL_period = 500;

   const unsigned long GCD = 500;

   unsigned char i; // Index for scheduler's for loop

   static task_t        task1,  task2;
   task_t *tasks[] = { &task1, &task2 };
   const unsigned short numTasks = sizeof(tasks) / sizeof(task_t*);

   task1.state       = -1;
   task1.period      = BL_period;
   task1.elapsedTime = BL_period;
   task1.TickFct     = &BL_Tick;

   task2.state       = -1;
   task2.period      = TL_period;
   task2.elapsedTime = TL_period;
   task2.TickFct     = &TL_Tick;

   TimerSet(GCD);
   TimerOn();

   while(1) {
      for ( i = 0; i < numTasks; ++i ) {
         if ( tasks[i]->elapsedTime == tasks[i]->period ) {
            // Task is ready to tick, so call its tick function
            tasks[i]->state = tasks[i]->TickFct(tasks[i]->state);
            tasks[i]->elapsedTime = 0; // Reset the elapsed time
         }
         tasks[i]->elapsedTime += GCD; // Account for below wait
      }
      while(!TimerFlag); // Wait for next timer tick
      TimerFlag = 0;
   }
}
```

The above code can serve as the basis for most systems having multiple synchSMs. It can be copied and pasted into RIMS, and then the tick functions replaced by one's own synchSMs (commonly more than just two).

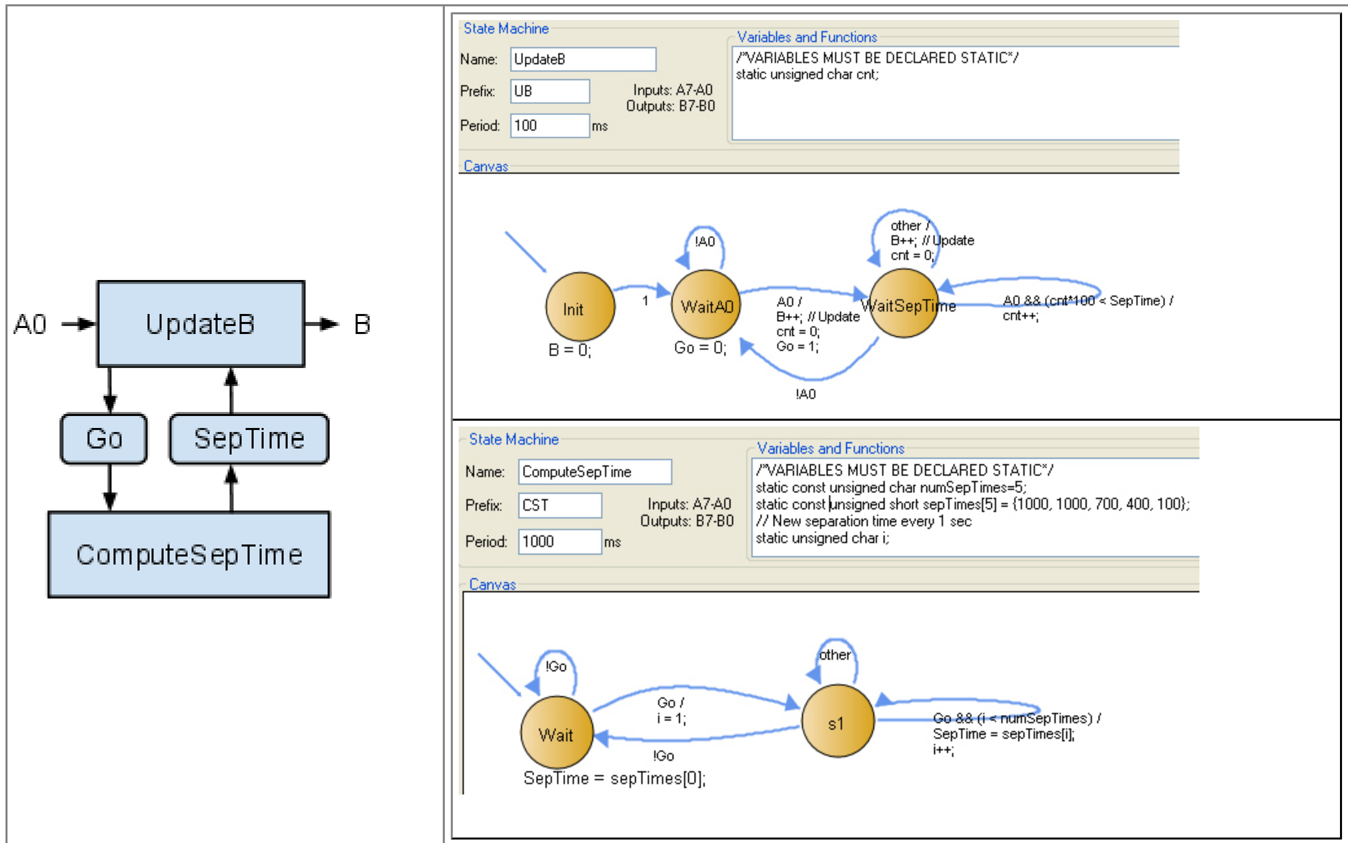
Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

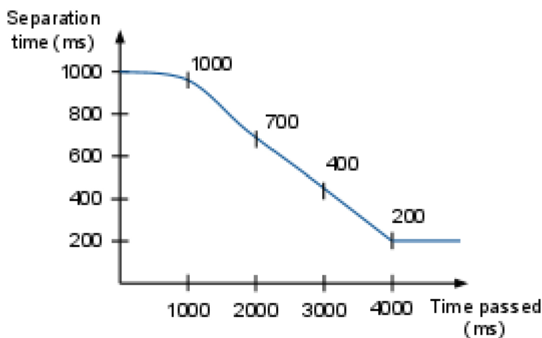
Example of concurrent multi-period synchSMs

A common time-oriented behavior involves a varying-rate update. For example, holding a button may cause a light to fade on, fading on slowly at first and then more rapidly after some seconds until fully on. A clock may have a button to increment the current time, incrementing slowing at first and then increasingly rapidly. A gas pump may initially pump gas fast until approaching a prepaid amount, then pumping slower and slower until hitting that amount. And so on.

One elegant method for such varying-rate update uses two synchSMs as shown below.



SynchSM UpdateB has simple behavior of waiting until A0 is 1, then updating B at the rate SepTime (UpdateB's period should be selected to be a divisor of all possible SepTime values). Meanwhile, synchSM ComputeSepTime computes SepTime, adjusting SepTime every 1 second, per the following separation-time graph:



Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

When A0 is initially 1, B should be updated every 1000 ms (the separation time is 1000 ms); the longer A0 is held at 1 (the more time passed), the faster the updates, until after A0 is 1 for four seconds, after which the rate stays constant at 200 ms. The graph is represented in ComputeSepTime as the array sepTimes with the shown values for each 1000 ms of time passed. Separation times could be updated faster than every 1000 ms, yielding more array items and a faster period for ComputeSepTime. Different separation-time graphs would be represented using different values.

If we tried to capture the varying-rate functionality as a single synchSM, we would likely create a confusing (and possibly incorrect) synchSM. Using concurrent synchSMs with different periods and shared variables yielded an elegant easy-to-modify system.

Copyright © Frank Vahid and Tony Givargis 2011

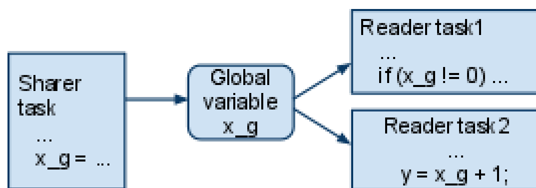
Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

Chapter 8: Communication

Communication is the sharing of information by one task with another. Communication can be carried out using a variety of methods.

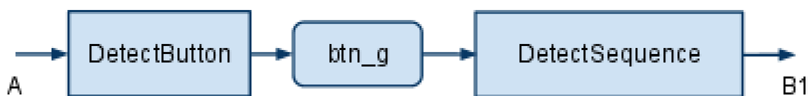
Shared variable

The most basic communication method involves one task writing to a global variable and another task (or multiple tasks) reading that variable -- the variable is ***shared***. A global variable is accessible by multiple tasks, in contrast to a local variable, which is only accessible to one task. We follow the practice that only one task may write to a global variable, while any number of tasks may read that variable. The shared variable method of communication is commonly used when one task (the sharer task) needs to expose internal data or state to other tasks (the reader tasks). The sharer task writes that data to a global variable read by the reader tasks.



An example is the earlier motion-triggered lamp, wherein task DetectMotion writes to a global variable *mtn* to let other tasks know when motion is currently being detected. Another task IlluminateLamp reads that variable and illuminates a lamp when the variable is 1. Other tasks could read the variable and respond by blinking an LED, sounding a tone, counting the number of times motion was detected, etc.

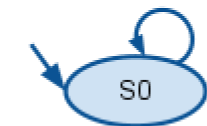
Example: Consider a door-lock system with five input buttons numbered 5, 4, 3, 2, 1, connected to A5, A4, A3, A2, A1. Task DetectButton detects which button is pressed. The task samples with a period of 100 ms and writes the currently pressed button to global variable unsigned char *btn_g*, writing 0 when no button is pressed or if multiple buttons are pressed, and writing either 5, 4, 3, 2, or 1 when a single button is pressed. The task is captured as a one-state synchSM, and happens to illustrate use of a synchSM that calls a function. Another task DetectSequence samples btn_g every 100 ms and toggles the door lock (B0=1 locks the door) whenever the task observes a particular three-button sequence. The task is captured using a multiple-state synchSM, which happens to have the correct button sequence stored in an array constant.



Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

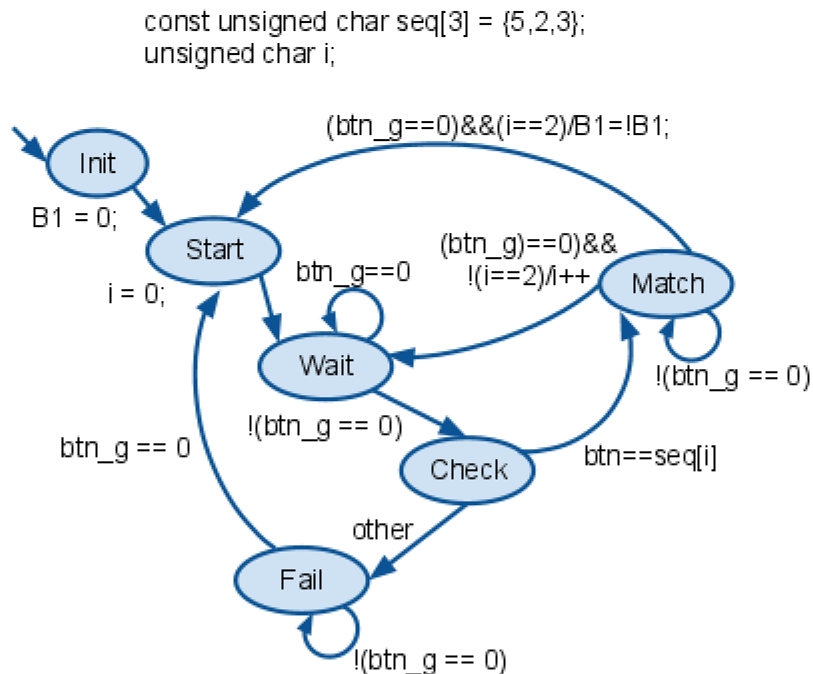
DetectButton Period: 100 ms



btn_g = DB_GetBtn();

```
// function declaration
unsigned char DB_GetBtn() {
  unsigned char btn;
  switch ((A&0x3E ) >> 1) {
    case 0x00: btn = 0; break;
    case 0x01: btn = 1; break;
    case 0x02: btn = 2; break;
    case 0x04: btn = 3; break;
    case 0x08: btn = 4; break;
    case 0x10: btn = 5; break;
    default: btn = 0; break;
  }
  return btn;
}
```

DetectSequence Period: 200 ms



A good practice is to name global variables using a "_g" suffix, to distinguish them from a task's local variables, as was done above for "btn_g".

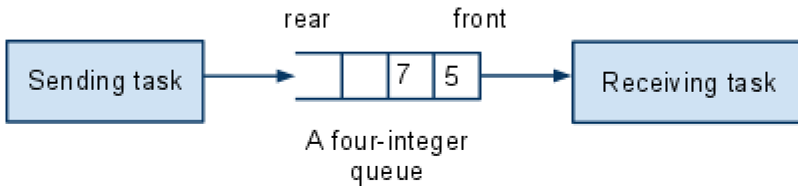
Message passing via a queue

Message passing is the communication behavior of sending a data item from a sender task to a receiver task such that the item is treated as a distinct message that is transferred. In contrast, a task writing a value to a global shared variable has no assurance that a reader task was actually able to read the value; perhaps the reader task was busy performing some other computation. For example, careful analysis of the above door-lock system using a shared variable reveals the problem that fast button presses could be missed by the DetectSequence task. Using a faster period for DetectSequence could reduce problems, but eliminating such timing interdependencies among tasks is preferable. Thus, a communication method is needed wherein a sender task can place a "message" somewhere for a receiving task (e.g., "button 5 was pressed"), such that the message persists in that place until the receiver task can read that message.

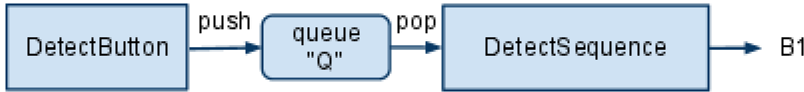
A queue is commonly used to support message passing communication. A ***queue*** can hold up to N data items. Data is always written (called a queue ***push***) to the rear of the queue, while data is always read (called a queue ***pop***) from the front of the queue. A pop also removes the data item from the queue. A queue is like a "line" at a grocery store: New customers arrive at the line's rear, and are processed from the line's front. The term ***FIFO*** is commonly used instead of queue, short for "first-in first-out," which describes how data is inserted and read/removed. A queue with no data items is said to be ***empty*** and cannot be popped (until after a push), while a size N queue with N data items inserted is said to be ***full*** and cannot be pushed (until after a pop). A queue is sometimes called a ***buffer.*** Items in a queue are said to be ***queued*** or ***buffered***.

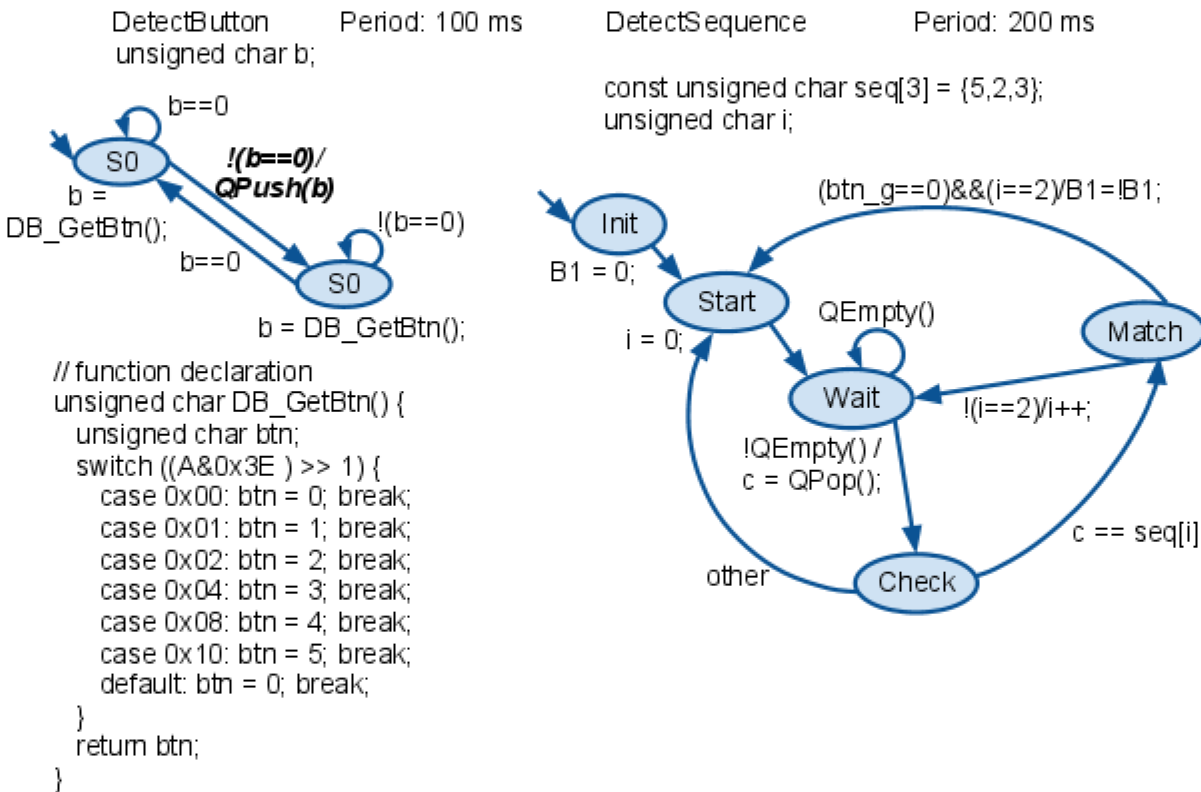
Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu



The earlier door-lock system could use a queue as in the following pseudo-coded tasks.





The queue leads to clearer communication of button presses between the tasks. For example, the DetectSequence task no longer needs to detect changes of btn_g from 0 to non-zero or vice-versa. Also, a queue of size N could buffer up to N button presses, accomodating fast button presses. Button presses while the queue is full would be ignored by the above system.

A queue can be implemented in C by defining a new structure type and some functions:

`typedef struct _Q4uc {`
`   unsigned char buf[4];`
`   unsigned char cnt;`
`} Q4uc;`
	New type defn Q4uc for queue of 4 unsigned char items.
	Sample declaration:
	`Q4uc btnQ; // new queue to hold buttons`

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

```
void Q4ucInit(Q4uc *Q) {
   (*Q).cnt=0;
}
```

Function to call before using a newly-declared queue.
Sample call:

```
   Q4ucInit(&btnQ);
```

```
unsigned char Q4ucFull(Q4uc Q) {
   return (Q.cnt == 4);
}
```

Function that returns true if queue is full.
Sample call:

```
   if (!Q4ucFull(btnQ)) { // not full, OK to
push ...
```

```
unsigned char Q4ucEmpty(Q4uc Q)
{
   return (Q.cnt == 0);
}
```

Function that returns true if queue is empty
Sample call:

```
   if (!Q4ucEmpty(btnQ)) { // not empty, OK to
pop ...
```

```
void Q4ucPrint(Q4uc Q) {
   int j;
   puts("Q4uc contents: \n");
   for (j=0; j<4; j++) {
       puts("Item "); puti(j);
       puts(": ");
       puti(Q.buf[j]);
       puts("\n");
   }
}
```

Function to print contents of queue. Useful for debugging.
Sample call:

```
   Q4ucPrint(btnQ);
```

```
void Q4ucPush(Q4uc *Q,
              unsigned char
item) {
   if (!Q4ucFull(*Q)) {
      (*Q).buf[(*Q).cnt] = item;
      (*Q).cnt++;
   }
}
```

Function that pushes item onto rear of the queue.
Should never be called with a full queue.
Note that queue *address* is passed so that the function can change the queue.
Sample call:

```
   unsigned char btn;
   btn = 3;
   if (!Q4ucFull(btnQ)) {
      Q4ucPush(&btnQ, btn); ...
```

```
unsigned char Q4ucPop(Q4uc *Q)
{
   int i;
   unsigned char item=0;
   if (!Q4ucEmpty(*Q)) {
      item = (*Q).buf[0];
      (*Q).cnt--;
      for (i=0; i<cnt; i++) {
         // shift fwd
         (*Q).buf[i]=
            (*Q).buf[i+1];
```

Function that pops item from front of the queue and returns that item.
Shifts the items forward so the second item moves to the front.
Should never be called with an empty queue.
Note that queue *address* is passed so that the function can change the queue.
Sample call:

```
   unsigned char recBtn;
```

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

``   }
   }
   return(item);
}```
	``if (!Q4ucEmpty(btnQ)) {
	   recBtn = Q4ucPop(&btnQ); ...```

(For large queues having hundreds or thousands of items, a faster queue implementation does not shift items in the pop function, but instead treats the buf array as a circular buffer and maintains head and tail variables, in addition to the cnt variable, to indicate the current front and rear of the queue respectively.)

Try: A store's person-entry system sounds a single tone by setting B0=1 for 500 ms and B0=0 for 500 ms to alert a store clerk every time a person enters (A0 rising) a store. A0 events (0 to 1, or 1 to 0) can occur as rapidly as every 200 ms. Because the tone behavior lasts 1000 ms, and a person entering could be detected every 400 ms, person entering events should be queued. Capture the system's behavior using a task DetectPerson, a global queue of size 4, and a task GenTone. Note that the queue will only be storing 1s, each 1 corresponding to a unique person-entering event. First use psuedo-code for the queue-related operations, and then replace the psuedo-code by code that uses the above C typedef and functions.

The queue struct together with the functions that operate on the struct are known collectively as an ***abstract data type*** or ***ADT***, also known as an ***object***. An important ADT concept is that the struct's internal variables should NEVER be accessed directly by a user's code; the user's code can only call the pre-defined functions. A key advantage of C++ and other "object-oriented" languages over C is better support for ADTs.

The above-defined queue supports what is known as ***non-blocking message passing***, wherein the sender task does not wait for the receiver task to receive the message. Blocking message passing can be accomplished by the sender always following a push by a wait for an empty queue before proceeding; in that case, a queue size of one is sufficient. Note that using a queue of size one is more advanced than writing/reading one shared variable, due to the queue's push and pop functionality.

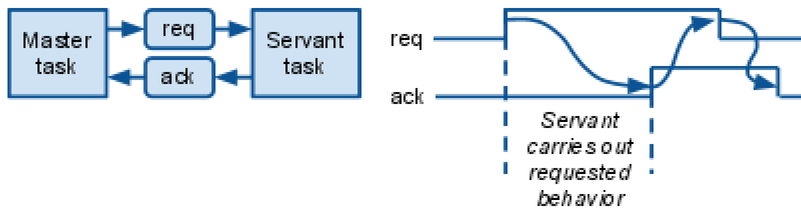
The designer selects the size of the queue. A larger queue prevents messages from being "lost" due to a sender task being unable to push a full queue, which can occur if the sender task pushes at a faster rate than the receiver task pops. In the person-entering example above, because a queue of size 4 is used, if 5 or more people quickly enter consecutively, some person-entering events may not be queued and hence lost. However, a larger queue uses more memory, and thus the designer must make a tradeoff between using memory and losing messages. To guide that tradeoff, the designer commonly performs some simple estimations of the rates of the sending and receiving tasks. In the person-entering example, a designer may estimate that people typically enter in groups of up to 4 people with larger groups being less uncommon and hence OK for losing messages. Furthermore, sometimes a designer does not want more than X messages being queued; in the person-entering example, the designer may limit the queue to size 4 to prevent the generated tones from lagging too many seconds behind when a person actually enters the store.

Other communication concepts

Handshake control: Shared variables are sometimes used to carry out handshake control between two tasks. A ***handshake*** is a method for task X to cause task Y to carry out some behavior. A handshake involves two single-bit global variables, which we'll call *req* (for request) and *ack* (for acknowledge), as follows: Task X raises req, task Y responds by raising ack, task X responds by lowering req, and task Y responds by lowering ack. Task X is known as the *master* and task Y as the *servant*. For example, task X may monitor input buttons and upon detecting a particular button sequence may request task Y to open a door, which may take several seconds. Task X raises a global

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

variable open_req. Task Y responds by opening the door, which may take many seconds. Once the door is opened, task Y informs task X that the behavior has been completed by raising open_ack. Task X lowers open_req, after which task Y lowers open_ack, thus completing the handshake. The earlier door-lock system may be extended by having DetectSequence open or close a door, using tasks OpenDoor and CloseDoor and handshakes; while waiting for the door to open or close, the task might ignore button presses.



Sometimes a handshake is used when the master task needs to read data from the servant. For example, task Y may maintain the past 100 samples of an input sensor, and have the ability to output their average when requested. Task X may request the average by raising avg_req, after which task Y computes the average, writes the result to a shared variable avg_data, and then raises avg_ack. Task X can then read avg_data, after which it lowers avg_req, causing task Y to then lower avg_ack. Note that avg_data is only guaranteed to be correct when avg_ack is high; the handshake enables two tasks to correctly transfer data even if those tasks execute at different rates. A similar handshake procedure can be used if task X needs to write data to the servant.

Note that handshake control combined with a global data variable can implement blocking message passing.

UARTs

A microcontroller task may need to communicate data with another physical device such as a liquid crystal display or another microcontroller. Data can be communicated using the microcontroller's output pins. However, to conserve limited output pins, microcontrollers typically include hardware, called a ***UART***, that automatically transmits eights bits of data serially, meaning one bit at a time, over a single pin. UART stands for universal asynchronous receiver/transmitter. When dealing with UARTs, receive is typically written as ***rx***, and transmit as ***tx***. RIMS has a UART that can send data over an additional tx pin, and receive data over an additional rx pin. The UART must first be activated by the RIMS built-in function UARTOn().

To transmit data, a program writes to a special global variable T of type unsigned char. Upon being written, the contents of T will be shifted out serially over the tx pin; however, rather than showing the UART tx pin, for viewing convenience, RIMS instead displays the transmitted data by its equivalent ASCII character in the "UART/Debug Output" text box. Such transmission takes time (being one bit at a time), during which time the program should not modify T. Thus, before writing to T, the program must ensure the UART is ready for transmitting by checking that a global flag variable, TxReady, is 1. TxReady is automatically set to 1 by the UART hardware while busy transmitting, and to 0 when done transmitting and thus ready for T to be written again. Writing without checking may corrupt a transmission in progress. The following program repeatedly sends 01100001 (0x61) serially over the UART output pin, thus displaying multiple "a" characters (the ASCII character for 0x61) in the UART output text box:

```
#include "RIMS.h"

void main() {
   UARTOn(); // activate UART
   while (1) {
```


Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

```
      while (!TxReady); // wait for UART transmit ready
      T = 0x61; // transmit 01100001 serially over UART output pin
   }
}
```

For receiving, instead of showing the rx pin, RIMS has a "UART input" text box in which a user may type characters, whose 8 bits (per ASCII) are received serially into a special global variable R. The UART informs the program when new data has been received into R by automatically calling an ISR function "void RxISR()" that the programmer must define. Our convention is to have that ISR set a flag that the program may then read to determine that new data was received by the UART.

The following program repeatedly waits until the UART receives new data ("while (!rxFlag);") and writes that data to output B ("B = R;").

```
#include "RIMS.h"

volatile unsigned char RxFlag = 0;
void RxISR() { // Called automatically when UART receives new data
   RxFlag = 1;
}

void main()
{
   UARTOn();
   while(1){
      while (!RxFlag);//wait until UART receives new data
      RxFlag = 0;
      B = R; // write received data to B output
   }
}
```

Try: Run the above program, type characters into the UART text box, and note that the character's ASCII value appears on B until the next character is typed.

The UART hardware automatically detects when R is read by the program, after which the UART hardware may overwrite R when new data is received. Until R is read by the program, the UART hardware will not overwrite R. Such built-in microcontroller functionality may be hard for new programmers to remember.

Try: In the above program, replace "B = R;" by "putc(R);" and run the program, noting that multiple characters typed in the UART input box are printed in the Debug Output box. Next, replace "putc(R);" by "putc('x');" and run the program. Note that typing a character into the UART input box causes 'x' to be printed once, but typing more characters does not cause further printing of 'x'. The reason is that the UART will not write to R a second time because the program did not read R, so the RxISR does not get called a second time, and thus the program gets stuck at the "while (!rxFlag);" statement. Verify this by breaking and stepping the program.

Common UART-related bugs include forgetting to turn on the UART (via the UARTOn function call), or failing to read R when data is there, thus preventing the hardware from calling the RxISR function.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

Chapter 9: Utilization and Scheduling

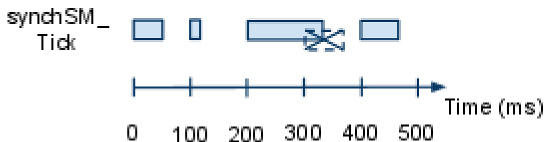
Earlier chapters assumed actions completed in a short non-zero time that was less than the synchSM period. For systems with extensive actions in a state, or with numerous tasks sharing the same microcontroller, the assumption may not hold. An embedded systems programmer may have to pay attention to the actual execution duration of actions relative to synchSM periods to ensure the system utilizes a microcontroller appropriately.

Timer overrun

In our approach to converting a synchSM to C, the main function calls the synchSM's tick function, waits for timerFlag to be set to 1 by TimerISR, sets timerFlag to 0, and repeats:

```
void TimerISR(void) {
   timerFlag = 1;
}

void main() { ...
  while (1) {
     if (<synchSM should execute>) {
         synchSM_Tick();
     }
     while(!timerFlag);
     timerFlag = 0;
  }
}
```

Thus, if TimerISR ever gets called with timerFlag still being 1, this means that the microcontroller was still executing the synchSM's tick function when the next timer tick occurred. The situation on the microcontroller is illustrated by the following timing diagram:



The synchSM has a period of 100 ms. Every 100 ms, the tick function executes some actions, whose duration varies depending on the current synchSM state. At time 200 ms, the tick function's actions required 120 ms, and thus the tick at 300 ms will not have an effect based on the way our C code is written. The next tick that will be noticed will be at 400 ms.

A simple check added to the TimerISR function detects this incorrect execution situation:

```
void TimerISR(void) {
   if( timerFlag ) {
      // Timer-overrun exception has occurred; possibly add code to handle it
   }
   timerFlag = 1;
}
```


Copyright © Frank Vahid and Tony Givargis 2011
Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

Automatically-detected incorrect execution is known as an ***exception***. We call the above a ***timer-overrun 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-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:

CountThree Period: 500 ms

unsigned char cnt;

	Estimated assembly instructions
cnt = 0;	3
if (A0) { cnt++ };	2 +3
if (A1) { cnt++ };	2 +3
if (A2) { cnt++ };	2 +3
if (A3) { cnt++ };	2 +3
B1 = (cnt >= 3);	3 +2

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.

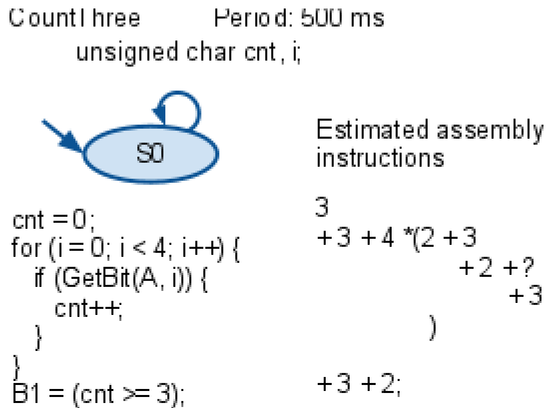
Utilization analysis usually ignores the additional C instructions required to implement a task in C, such as the switch statement instructions in a tick function, or the instructions involved in calling a tick function itself. For typical-sized tasks and typical-speed microcontrollers, the number of such "overhead" instructions is negligibly small. The analysis does not consider the C instructions that

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

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:



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 GetBit (defined in Chapter 2) is listed as "?". Examining the statements within the GetBit 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 instr * 0.00125 sec/instr = 63.75 ms. Again considering a 500 ms window, the utilization is 63.75 ms / 500 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:

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

CountThree Period: 500 ms

unsigned char cnt, i;

```
cnt = 0;
for (i = 0; i < 4; i++) {
   if (get_bit(A, i)) {
      cnt++;
   }
}
```

```
B1 = (cnt >= 3);
```

S0's actions translate to 3+3+4*(2+3+2+10+3) = 46 instrs, while S1's actions translate to 3+2 = 5 instrs. 46 instrs is larger than 5 instrs. On microcontroller M, the WCET is 46 instr * 0.00125 sec/ instr = 57.5 ms. For a time window of 500 ms, the worst-case utilization of M is 57.5 ms / 500 ms = 11.5%.

Any conditions and actions appearing on transitions should also be considered during analysis. One approach determines the worst-case number of instructions (due to conditions and actions) among all of a state's *incoming* transitions, and then adds that number to the state's actions. This approach makes sense when considering how a tick function, when translated to C, first executes a switch statement for transitions, and then executes a switch statement for the determined next state.

Analysis leading to utilization over 100% indicates that timer overrun will occur. For example, suppose the single-state for-loop version of CountThree is to run on an (extremely slow) microcontroller that executes 100 instructions per second, meaning 0.010 sec/instr. Because S0's actions require 51 instructions, its WCET is 51 instr * 0.010 sec/instr = 510 ms. The microcontroller utilization would be 510 ms / 500 ms = 102%.

Several remedies can be considered:

- Use a slower synchSM period. For example, we could change the synchSM period from 500 ms to 1000 ms. Utilization thus becomes 510 ms / 1000 ms = 51%.
- Rewrite the actions to be more efficient. Converting to the original single-state CountThree version that did not use a for-loop or a function call results in worst-case instructions of 28, so WCET = 28 instr * 0.010 sec/instr = 280 ms, and thus utilization of 280 ms / 500 ms = 56%.
- Split the actions among two or more states. Converting to the two-state CountThree version yields WCET of 46 instr * 0.010 sec/instr = 460 ms, and a worst-case utilization of 460 ms / 500 ms = 92%.
- Reduce system functionality, thus eliminating some actions
- Use a faster microcontroller. Using a microcontroller that requires 0.00125 sec/instr leads to a WCET of 51 instr * 0.00125 sec/instr = 63.75 ms, and thus a utilization of 63.75 ms / 500 ms = 12.75%.
- Or, a programmer may determine that the missed synchSM ticks do not pose a serious problem, and could leave the system as is.

For the purpose of introduction, the above discussion uses very slow microcontrollers executing only 100 or 800 instructions per second. Typical microcontrollers will execute 10,000 to 10,000,000 instructions per second. Microcontroller utilizations for the above example tasks would thus be well below 0.1%, as expected for such trivially simple tasks.

Note that the above approach to estimating assembly instructions is approximate. A more accurate method may involve examining the actual assembly instructions generated by a compiler. Some tools exist to automatically estimate WCET of C or assembly code, but they are not yet commonly

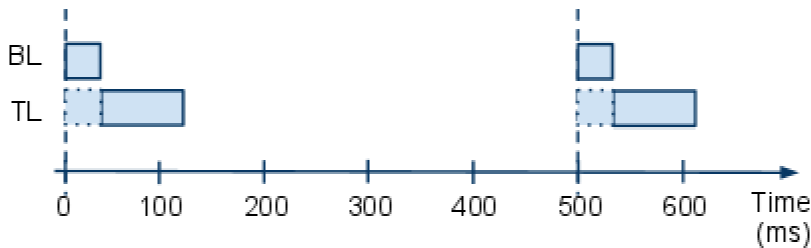
Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

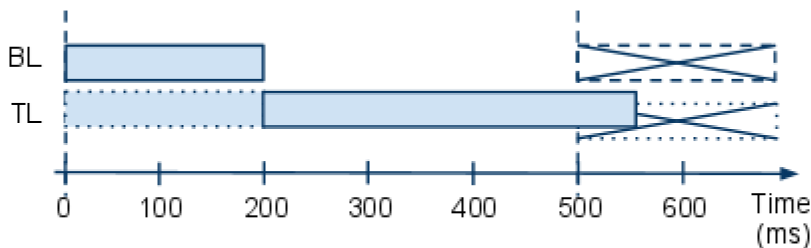
used. Due to the approximate nature of WCET estimation, programmers might strive to keep utilization well below 100%, e.g., perhaps below 80%.

Utilization for multiple tasks

Microcontroller utilization analysis is even more commonly done when multiple tasks are implemented on a single microcontroller. Consider the LedShow system (Chapter 6), which has two tasks, BlinkLed and ThreeLeds, each with a 500 ms period. Suppose microcontroller M executes 100 instr/sec, or 0.010 sec/instr. Suppose we determine BlinkLed's worst-case instructions per tick to be 3 instr, meaning a WCET on microcontroller M of 3 instr * 0.010 sec/instr = 30 ms, and we determine ThreeLeds worst-case instructions per tick to be 9 instr, for a WCET of 9 instr * 0.010 sec/ instr = 90 ms. The time window of interest is 500 ms, because the two tasks tick once every 500 ms. Thus, the microcontroller utilization will be (30 ms + 90 ms) / 500 ms = 24%. The timing diagram below illustrates the microcontroller's utilization for the two tasks executing on microcontroller M. In the timing diagram, a dotted block represents a ready task that isn't executing, and a solid block represents a task that is executing. At time 0, both tasks are ready to execute, per the convention that tasks should tick at system startup. Our earlier-introduced simple task scheduler may execute BL first, which takes 30 ms, followed by TL, which takes 90 ms. At time 120 ms, the microcontroller becomes idle (it is executing no task), until time 500 ms when the pattern of execution repeats.



Utilization above 100% would mean that the microcontroller cannot execute the tasks within the given period. For example, suppose that BL's WCET was instead 200 ms, and TL's WCET was 350 ms. Then utilization would be (200 ms + 350 ms) / 500 ms = 550 ms / 500 ms = 110%. The timing diagram below illustrates. BL executes for 200 ms, then TL for 350 ms. The microcontroller timer ticks every 500 ms. At time 500 ms, a timer tick occurs, but the scheduler code misses it, so neither BL nor TL will tick at 500 ms. At 550 ms, the scheduler code clears timerFlag and waits for a timer tick. The next tick that will be detected will be at 1000 ms.

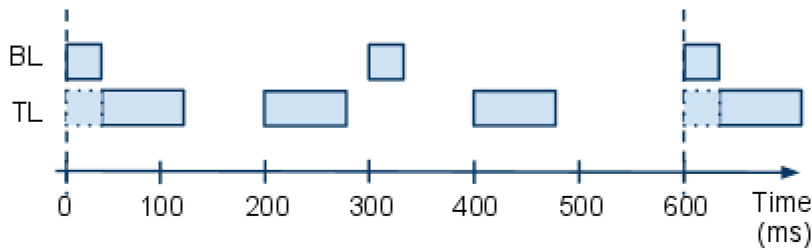


Possible remedies are the same remedies as for a single synchSM. Additionally, the remedy of "reducing functionality" may also include eliminating an entire synchSM from the system.

When two or more tasks have different periods, the time window of interest is the ***hyperperiod*** of the tasks, which is the least-common-multiple (LCM) of the tasks' periods. For example, suppose BL's period is 300 ms and TL's period is 200 ms. LCM(300 ms, 200 ms) = 600 ms. Thus, every 600 ms, the pattern of execution will repeat, as shown in the timing diagram below (with BL's WCET being 30 ms, and TL's being 90 ms).

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu



During the hyperperiod of 600 ms, BL will execute 600 ms / 300 ms = 2 times, while TL will execute 600 ms / 200 ms = 3 times, as shown above. The utilization during the hyperperiod is (2*30ms + 3*90ms) / 600 ms = 330 ms / 600 ms = 55%.

Summarizing, utilization for tasks T1...Tn executing on a microcontroller M is determined as follows:

- Determine M's sec/instr rate -- call this R
- Analyze each task Ti to determine its worst case number of instructions per tick, then multiply that number by R to determine Ti.WCET
- Determine the hyperperiod H as LCM(T1.period, T2.period, ..., Tn.period)
- Utilization = ((H/T1.period)*T1.WCET + (H/T2.period)*T2.WCET + ... + (H/Tn.period)*Tn.WCET) / H (*100%). Note that H/Ti.period is the number of times that Ti executes during hyperperiod H.

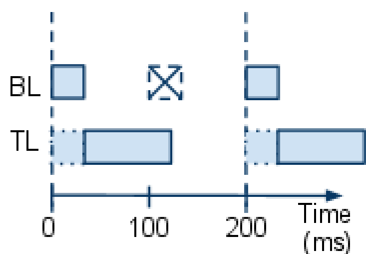
Utilization and timer overruns

The relationship of utilization and timer overruns for our scheduler code is as follows.

- Utilization > 100%: Timer overrun will occur
- Utilization < 100%
    - Single task: Timer overrun won't occur
    - Multiple tasks: Timer overrun may still occur

For utilization > 100%, timer overrun will occur (assuming the worst-case execution times actually occur).

For utilization < 100%, in a system with just a single task, timer overruns won't occur. In a system with multiple tasks, timer overruns may still occur, however. For example, suppose BL's period and WCET are (100, 30) respectively (in ms), and TL's are (200, 90). The timing diagram below shows that the tick at 100 ms is missed because of TL's execution, even though utilization is well below 100%.



Such timer overrun can be detected through further analysis, namely by noting that at every hyperperiod (starting with 0), all tasks will be ready and thus all will execute one after the other, representing the worst-case execution situation within a hyperperiod. Thus, checking if the sum of WCETs for all tasks exceeds the smallest period of any task will indicate whether a timer overrun will occur in that worst-case situation.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

Jitter

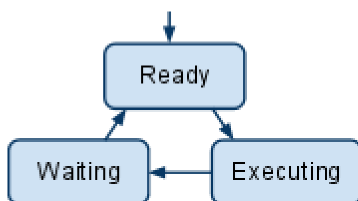
A task's ***jitter*** is the delay between the time that a task was *ready* to start executing and the time that it *actually* starts executing on the microcontroller. Jitter can be determined by drawing a timing diagram that shows ready and executing tasks for the tasks' hyperperiod, as earlier. Jitter is caused by two or more tasks being ready; the scheduler executes one of those tasks, causing jitter in the other tasks. In an earlier timing diagram, TL experiences 30 ms of jitter every third time it executes, due to the scheduler choosing to execute BL first. TL's worst-case jitter is thus 30 ms, and TL's average jitter is 10 ms. BL experiences no jitter. Different scheduling decisions may result in different jitter. If our scheduler code had chosen to execute TL first rather than BL when both tasks were ready at time 0 ms (and every 600 ms after then), then TL would experience no jitter, but BL would experience 90 ms of jitter every second time it executes due to interference from TL, for a worst-case jitter of 90 ms and an average jitter of 45 ms. If minimizing worst-case or average jitter is the goal, then giving BL priority when scheduling would be a better choice.

Avoiding timer overruns is thus not the only goal when scheduling tasks to execute on a microcontroller. Other goals, such as reducing jitter, and meeting deadlines (to be discussed), are also important. Attention should thus be paid to creating good scheduling approaches.

Scheduling

Note that a task implemented on a microcontroller can have one of three statuses at a given time:

- *Waiting*: The task should not execute because it is waiting for its period to elapse.
- *Ready*: The task's period has elapsed and the task is ready to execute, but the microcontroller hasn't yet started executing the task.
- *Executing*: The microcontroller is executing the task; for a synchSM, the microcontroller is executing the synchSM's tick function.



All tasks are initially considered ready when the system is started (namely, at time 0). Thus, each task should execute once at startup, and then wait until its period elapses to execute again.

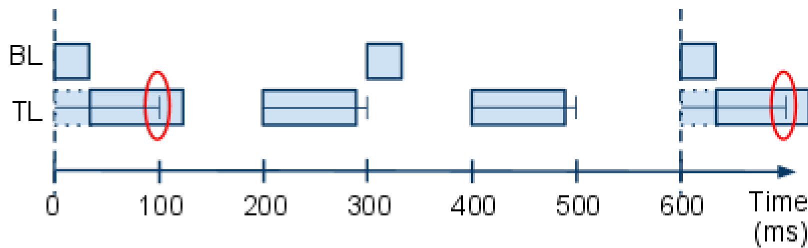
Scheduling is the job of choosing which of several ready tasks to execute. Scheduling is necessary because a microcontroller can only execute one task at a time. A scheduler is the code responsible for scheduling tasks. We designed a simple scheduler in an earlier chapter. In the earlier timing diagram of BL and TL tasks, when both tasks were ready at time 0 ms (and 600 ms), the scheduler chose to execute BL first, but it could have chosen TL instead. Scheduling does not impact utilization, but does impact features of task execution, such as jitter, and meeting deadlines. Jitter was discussed earlier. We now discuss deadlines.

A task's ***deadline*** is the time by which a task *must complete after becoming ready.* Else, the system has "missed a deadline" and is considered to have not executed correctly. For example, in the earlier example with tasks BL (period 300 ms, WCET 30 ms) and TL (period 200 ms, WCET 90 ms), suppose TL has a deadline of 100 ms. The period means that TL will be ready at time 0 ms, 200 ms, 400 ms, etc. The deadline means that when TL becomes ready at each of those times, TL should execute *and complete* by 100 ms, 300 ms, 500 ms, etc. We can include the deadline on a timing diagram by drawing a line from the ready time to the deadline time, as shown below. If a programmer does not specify a deadline for a task, then by default the deadline is set equal to the period (and is usually

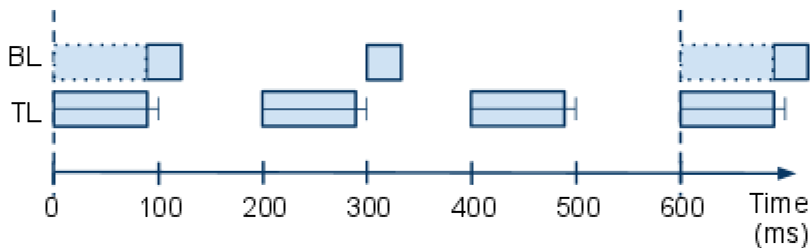
Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

not drawn), meaning the task should at least complete before the next time the task becomes ready. We see that TL misses a deadline during its first execution of each hyperperiod, due to jitter caused by the scheduler executing BL before TL.



If the scheduler had executed TL before BL, the following execution would have resulted:



Jitter in the second schedule is larger than in the first schedule, but no deadlines are missed. The example demonstrates the impact of scheduling on jitter and meeting deadlines.

Several scheduling approaches exist, and can be divided into static-priority and dynamic-priority approaches. ***Priority*** is an ordering of tasks indicating which ready task should execute first. When multiple tasks are ready, a scheduler executes the highest-priority task first.

A ***static-priority scheduler*** assigns a priority to each task before the tasks begin executing, and those priorities don't change during runtime.

- A common approach assigns highest priority to tasks with *shortest deadlines*. The intuition is that such tasks may miss deadlines if not executed first, as in the above BL/TL example.
- If all tasks have their deadlines equal to their periods (as is often the case), then the above approach translates to assigning highest-priority to *shortest-period tasks.* This scheduling approach is called ***rate-monotonic scheduling (RMS)***, named because priorities are based on the task's period or rate, with priorities assigned in a monotonically-increasing manner according to those rates.
- Another approach assigns highest priority to *shortest-WCET tasks*. This approach may be better at reducing jitter, since first executing short tasks reduces jitter of other ready tasks. The approach requires knowledge of task WCETs, which aren't commonly known. The approach is useful if the microcontroller speed is such that missed deadlines are rare.
- Some schedulers allow a programmer to manually assign priorities to tasks. This approach is useful if the programmer knows that reducing jitter or meeting deadlines for certain tasks is more important than for other tasks. For example, a task controlling a car's speed may be more important than a task controlling a car's passenger-compartment air temperature, regardless of the periods, deadlines, or WCETs of those tasks.

Our earlier simple task scheduler code, replicated below, processes tasks in the order they appear in an array, and thus gives priority to tasks that appear earlier in the array.

```
for ( i = 0; i < numTasks; ++i ) {
    if ( tasks[i]->elapsedTime == tasks[i]->period ) { // task is ready
       tasks[i]->state = tasks[i]->TickFct(tasks[i]->state);
       tasks[i]->elapsedTime = 0;
```

Copyright © Frank Vahid and Tony Givargis 2011

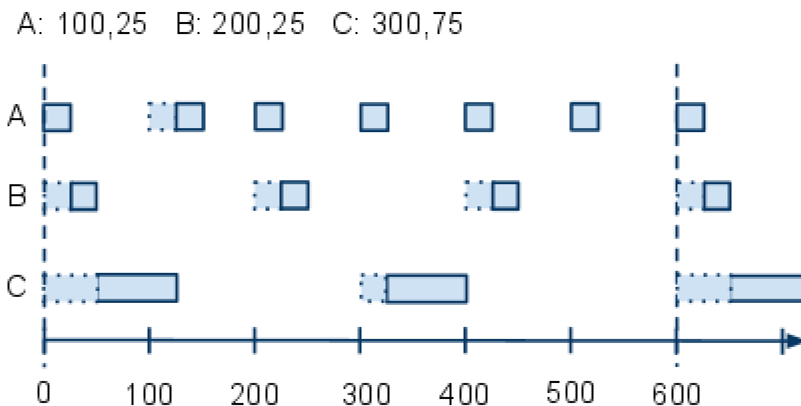
Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

```
        }
        tasks[i]->elapsedTime += GCD;
    }
```

A programmer can thus insert tasks into the array in order of priority. A more general approach may introduce new fields to the task structure to store a deadline value, a WCET value, or a programmer-assigned priority number, and then a function can be called that sorts the tasks array by a particular field before the main code's "while (1)" loop.

The simple task scheduler code will experience timer overrun if a task is executing when the next timer tick occurs and another task should become ready, in which case the other task's execution may be missed entirely rather than just delayed. A more sophisticated scheduler moves the scheduler code from the main function to inside the timerISR function. In addition to the tasks array, the scheduler maintains another list of ready tasks, where the list implements a priority queue (Wikipedia: Priority queue). When the timer ticks and timerISR is called, rather than just setting a flag, the scheduler code checks all tasks and adds any newly ready tasks to the priority queue, where the queue keeps tasks ordered by their priority. Such a scheduler has the advantage of avoiding timer overruns; rather than missing task ticks, the scheduler adds those tasks to the list and will execute them later. The disadvantage is that such a scheduler requires more code and has more instruction overhead for each timer tick. The reader is encouraged to try implementing this more sophisticated scheduler.

Consider a system with three tasks A, B, and C, with "period, WCET" values as shown in the figure below, and deadlines equal to periods. Assume a priority-queue-based scheduler. RMS scheduling will give A highest priority, then B, then C. The figure shows that B experiences average jitter of 25 ms and C of (50+25)/2 = 37.5 ms, but no deadlines are missed.



Note that at time 100 ms, the microcontroller was busy executing C when A became ready. The priority-queue-based scheduler does not miss that tick, but rather adds A to the priority-queue, and then executes A when C completes.

Try: Schedule the above tasks A, B, and C assuming the priority order C, B, A (C has highest priority, A lowest). Indicate whether any deadlines are missed, and list average jitter per task.

A ***dynamic-priority scheduler*** determines task priorities as the program runs, meaning those priorities may change. A common dynamic approach assigns ready tasks with the *earliest deadlines* the highest priority. The intuition is that ready tasks with the nearest deadline are more likely to miss the deadline if not execute first. The approach is known as ***earliest deadline first (EDF)***. Dynamic-priority schedulers may reduce jitter and missed deadlines, at the expense of more complex scheduler code. EDF scheduling is commonly considered when some tasks are triggered by events (to be discussed) rather than all tasks being periodic.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

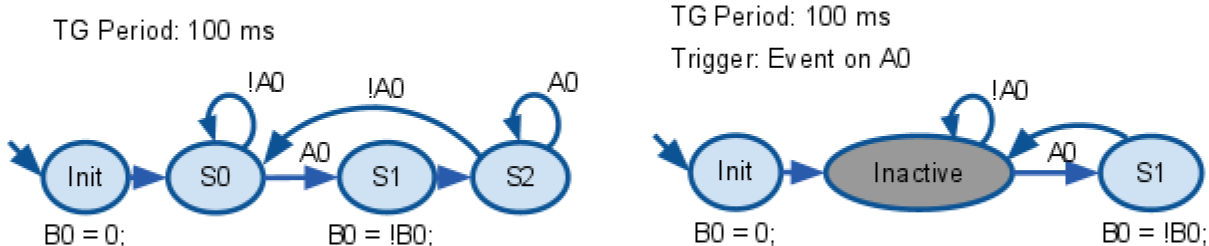
Scheduling approaches can also be divided into preemptive and non-preemptive schedulers. A ***preemptive scheduler*** may temporarily stop (preempt) an executing task if a higher-priority task becomes ready. A ***non-preemptive scheduler*** does not stop an executing task; the task must run to completion before the scheduler considers executing another task. Our earlier simple task scheduler is non-preemptive. A preemptive scheduler is hard to write entirely in C code, and is instead commonly written using at least some assembly code to carry out the low-level saving of registers and program counter values necessary to later resume the preempted task. In the above figure with A, B, and C tasks, a preemptive scheduler at time 100 ms would preempt lower-priority task C, execute A for its 25 ms, and then resume task C at time 125 ms. Preemptive schedulers are commonly used when tasks are written as functions each with an infinite loop (e.g., "while(1)"). In contrast, the approach in this book has been to capture tasks as synchSMs, which naturally divides tasks into distinct "ticks" that each runs to completion. synchSMs represent a form of what is known as "cooperative tasks," which willingly relinquish the microcontroller to allow other tasks to execute, greatly reducing the need for preemption. The synchSM approach in fact makes possible the writing of one's own task scheduler directly in C code; such is not readily possible using infinite-loop tasks. The drawback of synchSMs is that sometimes behavior is easier to capture as an algorithm rather than as a synchSM.

Triggered synchSMs

Sometimes a synchSM tick just samples an input (such as input A0) to detect a change, a process known as ***polling***. To reduce the microcontroller being used for polling, some microcontrollers come with special hardware that detects a change on an input and calls a special ISR in response such as inputChangeISR(). Reducing polling reduces microcontroller utilization, decreases jitter, reduces missed deadlines, and/or enables more tasks to be implemented on the microcontroller.

Availability of such hardware inspires a variation of a synchSM that can make itself inactive by transitioning to a special "Inactive" state. *While inactive, a synchSM does not tick* and thus when implemented will not utilize the microcontroller. The synchSM becomes active again when a specified event occurs, such as an event on A0, after which the synchSM ticks repeatedly at its normal period, until transitioning to the inactive state again. The event *triggers* the synchSM to become active, and thus such a synchSM is called a ***triggered synchSM***.

Consider a system that toggles B0 whenever a button connected to A0 is pressed. One way to capture this behavior uses a synchSM that polls A0 every 100 ms, as shown on the left below.



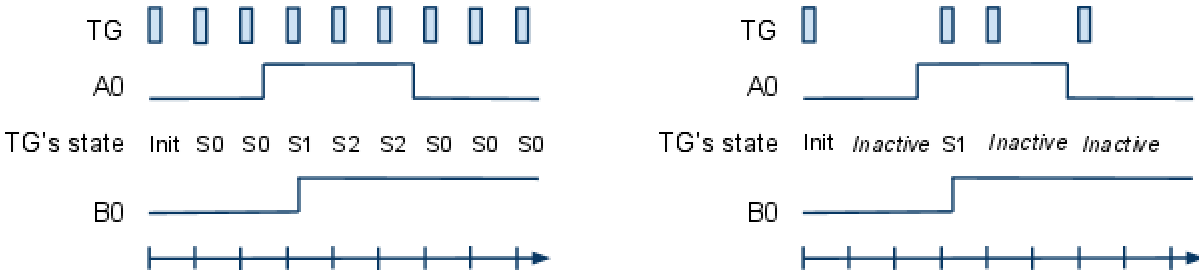
An alternative using a triggered synchSM is shown on the right. After initializing B0, the synchSM becomes inactive. While inactive, the synchSM *does not tick.* When an event occurs on A0, the synchSM becomes active again and checks transitions leaving the Inactive state. A0 being 1 means the event was a change from 0 to 1, so the synchSM goes to state S1, which toggles B0, after which the synchSM transitions back to the Inactive state. Instead, A0 being 0 means the event was a change from 1 to 0, which should not change B0, and thus the transition goes back to the Inactive state. Detecting the event on A0 is not the responsibility of the synchSM; the detection occurs by some other means outside the synchSM.

In scheduling terminology, a task triggered by an event is known as an *aperiodic* task, in contrast to a *periodic* task that ticks at a known rate.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

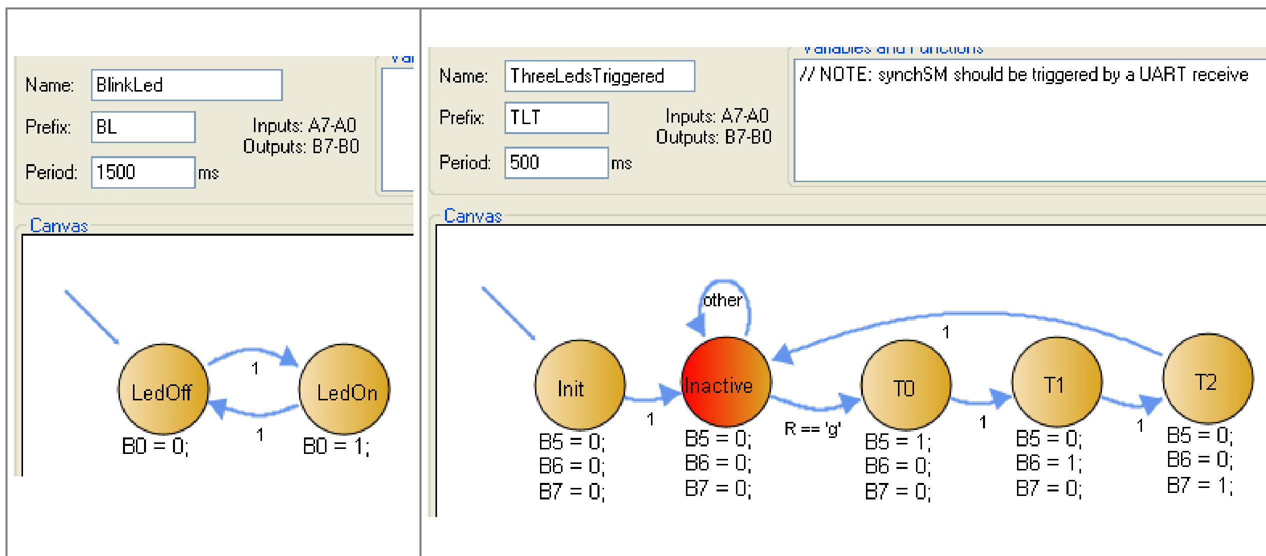
A benefit of a triggered synchSM becomes apparent when examining the utilization of a microcontroller implementing the synchSM, where that microcontroller has special hardware to detect the triggering event.



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 bolded and described in the comments.

```
#include "RIMS.h"

volatile unsigned char TimerFlag=0;
```

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

```
void TimerISR(void) {
   TimerFlag = 1;
}

typedef struct {
   signed   char state;
   unsigned long period;
   unsigned long elapsedTime;
   unsigned char active; // 1: active, 0: inactive
   int (*TickFct)(int);
} task_t;

task_t task_BL, task_TLT; // Global so visible to tick fct and ISR
task_t *tasks[] = { &task_BL, &task_TLT };
const unsigned short numTasks = sizeof(tasks) / sizeof(task_t*);

volatile unsigned char RxFlag=0;
volatile unsigned char RxData=0;

void RxISR(void) {
   task_TLT.active = 1; // TLT task triggered by UART rx, so make active
   task_TLT.elapsedTime = task_TLT.period; // Make ready to tick
   RxData = R; // Read to global var to avoid problems if chars come too fast
   RxFlag = 1;
}

// Task tick functions

enum BL_States { BL_LEDOFF, BL_LEDON };

int BL_Tick(int state) {
   puts("BL_Tick()\n");
   switch(state) { // Transitions
      case -1:
         state = BL_LEDOFF;
         break;
      case BL_LEDOFF:
         state = BL_LEDON;
         break;
      case BL_LEDON:
         state = BL_LEDOFF;
         break;
      default:
         state = -1;
         break;
   }

   switch(state) { // State actions
      case BL_LEDOFF:
         B0 = 0;
         break;
```


Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

```
      case BL_LEDON:
         B0 = 1;
         break;
      default:
         break;
   }
   return state;
}

enum TLT_States { TLT_T0, TLT_T1, TLT_T2, TLT_Init, TLT_Inactive };

int TLT_Tick(int state) {
   puts("TLT_Tick()\n");
   switch(state) { // Transitions
      case -1:
         state = TLT_Init;
         break;
         case TLT_T0:
         if (1) {
            state = TLT_T1;
         }
         break;
      case TLT_T1:
         if (1) {
            state = TLT_T2;
         }
         break;
      case TLT_T2:
         if (1) {
            state = TLT_Inactive;
         }
         break;
      case TLT_Init:
         if (1) {
            state = TLT_Inactive;
         }
         break;
      case TLT_Inactive: // Note: special "Inactive" state's
                         // transitions are same as any other state
         if (RxData == 'g') {
            state = TLT_T0;
         }
         else {
            state = TLT_Inactive;
         }
         break;
      default:
         state = TLT_Init;
   } // Transitions

   switch(state) { // State actions
```

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

```
      case TLT_T0:
         B5 = 1; B6 = 0; B7 = 0;
         break;
      case TLT_T1:
         B5 = 0; B6 = 1; B7 = 0;
         break;
      case TLT_T2:
         B5 = 0; B6 = 0; B7 = 1;
         break;
      case TLT_Init:
         B5 = 0; B6 = 0; B7 = 0;
         break;
      case TLT_Inactive:
         B5 = 0; B6 = 0; B7 = 0;
         task_TLT.active = 0; // Special "Inactive" state
                              // makes task inactive
         break;
      default:
      break;
   } // State actions
   return state;
}

void main()
{
   const unsigned long BL_period = 1500;
   const unsigned long TLT_period = 500;

   const unsigned long GCD = 500;

   unsigned char i; // Index for scheduler's for loop

   task_BL.state       = -1;
   task_BL.period      = BL_period;
   task_BL.elapsedTime = BL_period;
   task_BL.active      = 1; // Task starts as active
   task_BL.TickFct     = &BL_Tick;

   task_TLT.state       = -1;
   task_TLT.period      = TLT_period;
   task_TLT.elapsedTime = TLT_period;
   task_TLT.active      = 1; // Task starts as active
   task_TLT.TickFct     = &TLT_Tick;

   TimerSet(GCD);
   TimerOn();
   UARTOn();

   while(1) {
      for ( i = 0; i < numTasks; ++i ) {
         if (tasks[i]->active) { // Task can only tick if active
            if ( tasks[i]->elapsedTime == tasks[i]->period ) {
```

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

```
            // Task is ready to tick, so call its tick function
            tasks[i]->state =tasks[i]->TickFct(tasks[i]->state);
            tasks[i]->elapsedTime = 0; // Reset the elapsed time
         }
      }
      tasks[i]->elapsedTime += GCD; // Account for below wait
   }
   while(!TimerFlag && !RxFlag); // Wait for next timer tick or UART receive
   TimerFlag = 0; RxFlag = 0;
 }
}
```

The main change is the addition of the "active" field to the task structure, 1 meaning active. The initialization in main sets each task initially active. Transitioning to the special "Inactive" state in task TLT's tick function sets the task to inactive. The scheduler in main skips any inactive tasks when checking if tasks are ready to tick. The UART's ISR "RxISR" re-activates the TLT task, and also sets the task's elapsedTime such that the task is ready to tick. Note that the code moved the task declarations to be global, so that the RxISR and TLT_Tick functions could set the task's active field. Note also at the end of the scheduler code in main that the scheduler's wait can be completed not just by the next timer tick, but also by a UART receive.

Try: Run the above code in RIMS. Each tick function prints its name when it ticks; note that the TLT task stops ticking, and only the BL task ticks. Now enter an 'a' into RIMS' "UART input" text box. Note that the TLT task ticks once, but because the input was not 'g', transitions back to the Inactive state and thus doesn't tick. Now type 'g' and note that TLT ticks several times (and LEDS B5, B6, B7 blink in sequence), and then again stops ticking .

A triggered synchSM can be implemented on a microcontroller that doesn't have special hardware to detect the triggering event. In that case, when translating to C, we introduce a helper synchSM that polls the appropriate input for the triggering event, and communicates with the triggered synchSM by setting a global flag variable to 1, holding the 1 long enough for the original synchSM to detect it. The original synchSM's inactive state needs a new transition that goes back to the inactive state when the flag is 0, while the existing transitions need to check that the flag is 1. Note that such an implementation does not achieve the benefits of reduced polling or faster sampling, but may lead to more intuitive initial synchSMs. Capturing behavior using triggered synchSMs may also lead to more efficient implementations if the synchSMs are ported to another microcontroller that does have the special hardware.

Reducing power consumption using a sleep function

Microcontrollers consume power while running a program, for example one milliwatt. Commonly a microcontroller is executing a while loop waiting for some event to occur, such as "while (!TimerFlag) {}". Executing that loop (or more specifically, the assembly instructions for that loop) is somewhat wasteful because the programmer knows the loop cannot be exited until the TimerISR is called by the hardware. Thus, microcontrollers typically have a special low-power mode, known as a "sleep" mode, in which the microcontrollers stops executing the program but continues to operate other hardware like the timer and the UART. When in sleep mode, the microcontroller consumes substantially less power, for example one microwatt. When such hardware experiences a particular event, like the timer ticking or the UART receiving data, the microcontroller "wakes up", typically executes an ISR in response to the event, and resumes executing the instruction that put the processor to sleep. RIM can be put to sleep by calling a function "Sleep". In general, knowing when to call the sleep function can be challenging to a programmer. Fortunately, the disciplined synchSM approach greatly simplifies the challenge. One can simply call the sleep function just before waiting on the TimerFlag and/or RxFlag (or any other flag set by an ISR). For the above example, we could add the following code:

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

```
puts("Going to sleep...");
Sleep(); // Put processor in low-power mode waiting for wakeup event
// Below instructions won't execute until hw event wakes processor and
//    calls appropriate ISR
puts("Waking up.\n");
while(!TimerFlag && !RxFlag);
```

Try: Modify the above program by adding the above lines as shown. Note from the printed text that the processor repeatedly goes to sleep and wakes up.

The amount of time spent asleep could be further improved by dynamically adjusting the Timer's tick rate to be the GCD of the periods for all *active* tasks, rather than for all tasks.

Because microcontrollers are commonly powered by batteries, sleep modes are extensively used; a microcontroller may be asleep 99% of the time or more, waking up for brief periods, performing a burst of processing, and going back to sleep again.

Disciplined programming

This book described a disciplined approach to time-oriented programming in C. We recommend that programmers think in terms of communicating synchSMs, translating to C using the defined translation methods. After translating to C, some programmers are tempted to introduce statements in places not adhering to the translation methods. Doing so may save time at first, but ultimately may lead to bugs and to code that is harder to maintain. For example, the translation methods use ISRs only to set specific flags, and nothing more. *Programmers should never add additional statements into the ISR, no matter how tempting*; doing so is a common source of tricky bugs. Likewise, adding functionality in main's task scheduler should be avoided. Instead, if functionality is needed, it should almost always be added by expanding the behavior of a synchSM, updating the synchSM's tick function accordingly. An analogy can be made with a car. The driver's interface with the car's hardware is well-defined: Steering wheel, gas pedal, and brakes. A driver who uses engine knowledge to connect extra cables to the engine for some minor acceleration or handling benefit is not considered clever, but more likely dangerous. The disciplined driver, and the disciplined programmer, ultimately understands that respecting the well-defined interface to the lower-level machine yields the best long-term benefit.

Towards an RTOS

An operating system is a program that runs on a microprocessor to provide an interface between the user's program and hardware. Microsoft Windows and Unix are well-known and complex operating systems intended for desktop computers. Embedded systems sometimes use a ***real-time operating system*** (***RTOS***) (Wikipedia: RTOS). A key part of an RTOS is a mechanism to allow users to define tasks (sometimes called *threads* or *processes*), including their periods and priorities. Another key part is a task scheduler, such as the scheduler we created above. RTOSes typically support preemptive scheduling, as well as non-preemptive. Numerous microcontroller RTOSes are available, several of them free. (Wikipedia: FreeRTOS).

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

Chapter 10: Programming Issues

Rounding and overflow

Expressions involving division should be treated with extra care due to error that is introduced from rounding. Consider the formula for converting Celsius to Fahreneit: F = (9/5)*C + 32. Suppose a RIMS program is coded as follows:

```
#include "RIMS.h"

unsigned char F2C_uc(unsigned char C) {
    unsigned char F;
    F = (9/5)*C + 32;
    return F;
}

void main() {
    while (1) {
        B = F2C_uc(A);
    }
}
```

The table below shows the actual Fahrenheit values for Celsius values from 0 to 9, followed by those values when rounded to integers, followed by values obtained from the above program:

Celsius	Fahrenheit (actual)	Fahrenheit (integer)	F (from above program)	F (after change below to do division later)
0	32	32	32	32
1	33.8	34	33	33
2	35.6	36	34	35
3	37.4	37	35	37
4	39.2	39	36	39
5	41.0	41	37	41
6	42.8	43	38	42
7	44.8	45	39	44
8	46.4	46	40	46
9	48.2	48	41	48

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

The values obtained from the above program are wrong due to rounding. Because all values in the F2C_uc function's expression are integers, the term "9/5" is computed as an integer; the value is 1.8 but it is rounded to 1 because C rounds by truncating the fraction. Thus, the computation actually being carried out is: F = 1*C + 32.

The rounding problem can be reduced by doing the division as late as possible. For the above program, the line that computes F can be changed to:

```
    F = (9*C)/5 + 32;
```

The values obtained after that change are shown in the last column of the above table. Those values are much closer to the desired values; the differences are due to the C language truncating any fractional part, rather than rounding up if the fractional part is 0.5 or greater.

When computing expressions, care must also be taken to avoid overflow. Embedded systems programs commonly use the smallest possible data types to conserve limited memory and to obtain faster computation on a narrow bitwidth microcontroller (e.g., on an 8-bit microcontroller). For example, if an integer representing a temperature in Celsius has a range of 0 to 255, the integer may be declared as a *char* rather than as a *short* or a *long* (see next section's F2C_uc_array for a clear example of savings obtained by using char). However, use of the smallest possible data types increases overflow situations, where a value is too large for its variable size (e.g., a value of 270 is too large for a char). To be safe, a function might first cast smaller numbers to larger ones, compute, and then explicitly check for overflow before returning a result. For example, the above conversion function might be rewritten as:

```
unsigned char F2C_uc(unsigned char C) {
    unsigned char F;
    unsigned short Csi, Fsi;
    Csi = C;
    Fsi = (9*Csi)/5 + 32;
    if (Fsi<=255) {
        F = (char)Fsi;
    }
    else {
        F = 0; // overflow error
    }
    return F;
}
```

A further improvement might set Csi = 10*C, compute Fsi as above, round Fsi to the 10s place (if the 1s column is 5 or greater, increment the 10s column), and then divide Fsi by 10, before checking for overflow.

C functions versus lookup tables

A function maps possible input values to an output value. A typical C function uses an expression or even an algorithm to perform the mapping, as in the above conversion function that maps an input Celsius value to an output Fahreneit value. However, a faster approach uses a precomputed output value for every possible input value. The values can be precomputed using another C program with a loop that calls a function with every possible input value and prints the output, or by using a spreadsheet, or by other methods. For the above conversion program, good pre-computed values for the first 10 of the possible 255 input values, computed using a spreadsheet, are shown in the third column. The remaining values could be computed similarly, with any output value greater than 255 forced to 0. Then an array could be declared and initialized with 256 constants as follows:

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

```
unsigned char F2C_uc_array[256] = {32, 34, 36, 37, 39, 41, ..., 0, 0, 0};
   // middle values are omitted (...)
```

Such an array of precomputed function output values is called a ***lookup table***. The earlier F2C function could then be rewritten as:

```
inline unsigned char F2C_uc(unsigned char C) {
    return F2C_uc_array[C];
}
```

Whereas the original F2C_uc function may have required dozens of assembly instructions to execute, the lookup-table-based function requires only a few assembly instructions. Declaring the function as inline may even result in the function call and hence the call's associated assembly instructions being eliminated.

The tradeoff present with a lookup table is the larger code size. A function with a single char input parameter would require a 256-item lookup table. A function with a single short int input parameter would require a 65536-item lookup table, which may be tolerable in some cases but in many cases may be too large. A function with two char inputs would use a two-dimensional array (e.g., table_array[256][256]) which also has 65536 items. A function with a single long int parameter would require a lookup table with over 4 billion items. Clearly, use of the lookup table approach is severely limited by the number of possible input values, which determines the table size.

Fixed-point programming

C has built-in support for ***floating-point arithmetic***, wherein variables are declared as *real* numbers and arithmetic/comparison operators (+, -, *, /, <, <=, >, >=, ==, !=) operate on real numbers. (Wikipedia: Floating Point Arithmetic). Examples of real numbers are 25.61 and 1.428. A real number can be declared in C using the float type:

```
float x;
x = 25.61;
```

The *float* type uses 4 bytes. C also provides a *double* type that uses 8 bytes (hence the name "double").

Floating-point refers to the fact that the same number of bytes can represent real numbers with the decimal point in different places, e.g., 25.61 and 1.428 both have four significant digits but have the decimal point in different places. The decimal point can thus move or *float*. The advantage of floating-point implementation is that a real number can be represented over a wide range of magnitudes while maintaining a constant number of significant digits. This advantage becomes especially obvious for very large and very small numbers, e.g., $x = 6.02 \times 10^{23} * 1.50686 \times 10^{-14}$. A float type supports about 7 significant digits, double about 15 significant digits.

Note: RIMS does *not* currently support floating-point.

An example of floating-point arithmetic is a Fahrenheit to Celsius converter using float types:

```
float F2C_fl(float C) {
    float F;
    F = (9.0/5.0)*C + 32;
    return F;
```

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

```
}
```

Note that the earlier-mentioned rounding issue with integers is eliminated due to the use of real numbers.

Another example of floating-point arithmetic is the following C function that calculates the distance between two points on a surface:

```
#include <math.h>

typedef struct {
  double x, y;
} point_t;

double compute_distance(point_t p1, point_t p2) {
  double t1 = p2.x - p1.x;
  double t2 = p2.y - p1.y;
  t1 = t1 * t1;
  t2 = t2 * t2;
  return sqrt(t1 + t2);
}
```

The C standard library *math.h* provides functions for performing common math operations on real numbers such as square root (sqrt) or trigonometric sine (sin).

Unfortunately, floating-point operations typically run much slower than integer operations. Details of the reasons are beyond our scope, but briefly, floating-point involves representing the significand and the exponent separately (a real number is composed as: significand x $10^{exponent}$), so arithmetic operations internally may require shifting to obtain the same exponent, performing separate operations on the significand and the exponents, rounding, and normalizing. Desktop and laptop computers typically have special floating-point hardware to speed up floating-point operations, but such hardware may be large, and floating-point operations may still be slower than integer operations. For example, we ran two simple for-loop examples with repeated floating-point and integer operations, respectively, on an Intel Core Duo laptop machine; the floating-point version ran 3x slower. Embedded processors typically don't have special floating-point hardware and thus, if they support floating-point at all, then they perform floating-point operations using software functions, resulting in slowdowns of 100x or more compared to integer operations.

Fortunately, in many embedded system applications, real numbers may be limited to a narrow range of magnitudes and the number of significant digits required may be far fewer than provided by the built-in floating-point types. In such circumstances, a fixed-point implementation can speed up the computation and reduce the memory required to store the intermediate results. In a ***fixed-point implementation***, integer variables are used to represent real numbers (with limited range and precision). Likewise, integer operations are used in place of floating-point operations, whenever possible. (Wikipedia: Fixed Point Arithmetic).

Consider a very simple fixed-point representation where numbers may range from 0.0 to 20.0. For example, consider a performance rating system where two judges can enter scores between 0.0 and 10.0, and the two scores (x and y, respectively) are added to create a total score (z) which thus can range from 0.0 to 20.0. So a sample computation would be x=9.1, y=9.8, z = x+y (which is 18.9). Using floating-point representation is overkill in this system. Instead, we can declare x, y, and z as small integers (unsigned chars in this case), set x=91 and y=98, and z=x+y (which is 189). In other words, we multiply the real numbers by 10 to obtain an integer representation, and then conduct the necessary integer operations. We know that the answer needs to be divided by 10 to obtain the

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

actual result. (Baseball fans may note that batting averages are always listed as an integer like 352 rather than the actual fraction like 0.352).

The above simple fixed-point representation is intuitive due to being in base 10, but more efficient representations use base 2 to match the computer's underlying representation of numbers. In a fixed-point representation, we need to decide on the number of bits necessary to represent the whole part of the real number (***w***) and the number of bits necessary to represent the fractional part of the real number (***f***). Ideally, when using the C language, the sum of the bits used to represent the whole and fractional parts (***w.f***) should add up to the number of bits of one of the built-in C integer types. For example, if using the *short int* type, we may chose ***w***=12 and ***f***=4. Under this representation, written using the notation "(12.4)", we can represent a real number in the range of (-2048.0000 to 2047.9375). We can convert a floating point number (within the proper range) to the fixed-point representation and the other way around as shown in the pair of functions below.

```
#include <assert.h>

const double MIN_REAL = (short)0x8000 / 16.0;
const double MAX_REAL = (short)0x7fff / 16.0;

short conv_to_fixed_point(double x) {
   if( x < MIN_REAL ) {
      assert( 0 /* underflow */ );
   }
   else if( x > MAX_REAL ) {
      assert( 0 /* overflow */ );
   }
   else {
      return (short)(x * 16.0);
   }
}

double conv_to_floating_point(short x) {
   return x / 16.0;
}
```

For example, 2047.9375 would be converted to (12.4) fixed point representation as (short)(2047.9375 * 16.0) = (short)(32767.00000) = 32767. As another example, 3.14 would be converted as (short)(3.14*16.0) = (short)(50.24) = 50. Note that 50 looks nothing like 3.14, but it indeed is the (12.4) fixed point representation of 3.14.

In the above functions, a fixed-point version of a real number is obtained by multiplying it with 2^{f}. Likewise, a floating-point version of a fixed point number is obtained by dividing by 2^{f}. As a result, the precision of the real number (i.e., the fractional portion) is determined by ***f*** while the range is determined by the choice of ***w***. In the above example, we can represent the following fractions:

0 / 16.0 = 0.000000
1 / 16.0 = 0.062500
2 / 16.0 = 0.125000
3 / 16.0 = 0.187500
4 / 16.0 = 0.250000
5 / 16.0 = 0.312500
6 / 16.0 = 0.375000
7 / 16.0 = 0.437500
8 / 16.0 = 0.500000

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

9 / 16.0 = 0.562500
10 / 16.0 = 0.625000
11 / 16.0 = 0.687500
12 / 16.0 = 0.750000
13 / 16.0 = 0.812500
14 / 16.0 = 0.875000
15 / 16.0 = 0.937500

Once a real number is converted to its fixed-point representation, we can perform the basic add/subtract operations as shown below. Note that all fixed-point numbers must use the same representation, in this case (12.4).

```
inline short fixed_point_add(short x, short y) {
   return x + y;
}

inline short fixed_point_sub(short x, short y) {
   return x - y;
}
```

The above functions do not check for over-/under-flow, but can be modified to do so as shown.

```
#include <assert.h>

const short MIN_FP = 0x8000;
const short MAX_FP = 0x7fff;

inline short fixed_point_check(long x) {
   if( x < MIN_FP ) {
      assert( 0 /* underflow */ );
   }
   else if( x > MAX_FP ) {
      assert( 0 /* overflow */ );
   }
   else {
      return (short)x;
   }
}

inline short fixed_point_add(short x, short y) {
   return fixed_point_check( (long)x + (long)y );
}

inline short fixed_point_sub(short x, short y) {
   return fixed_point_check( (long)x - (long)y );
}
```

Note that, as expected, checking for error requires additional C statements (and execution time overhead). Also, note that we perform the computations using larger integers (type long) initially, check against the bounds, then cast to the shorter return type (short). Certainly, the additional overhead is a small penalty paid for a more robust fixed-point implementation.

Now we consider multiplication and division, as shown.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

```
inline short fixed_point_mul(short x, short y) {
   long r = (long)x * (long)y;
   return fixed_point_check( r >> 4 );
}

inline short fixed_point_div(short x, short y) {
   long r = ((long)x << 4) / (long)y;
   return fixed_point_check( r );
}
```

When multiplying two fixed-point integers, the decimal point shifts left by some number of bits (***f*** in this example). To illustrate, consider multiplication of the decimal numbers 3.1 and 2.7. In the first step, we multiply the two integers 31 and 27 to obtain the integer 837. Then, we place the decimal point two places from the right to obtain 8.37. To maintain our fixed-point format, we need to re-adjust the decimal point by shifting right exactly ***f*** bits. Likewise, when dividing, we need to shift the result left exactly ***f*** bits. However, we do so prior to the integer division, so as to not lose any precision.

Try: Rewrite the above add, sub, mul, and div functions for a (10.6) implementation.

The integer comparison operations work on fixed-point number as well, hence we will not cover those further. In general, you may chose to use a different fixed-point representation for different real-valued variables. In such instances, you need to implement more versatile routines that can perform mixed format operations. For example, you may have to add the variable ***a*** (12.4) with the variable ***b*** (10.6) and write the result into variable ***c*** (11.5). The following segment of code illustrates the necessary computation.

```
inline short mixed_fixed_point_add(short a, short b) {
   short temp1 = a << 2;      // convert from (12.4) to (10.6)
   short temp2 = temp1 + b;   // result is now (10.6)
   short result = temp2 >> 1; // convert from (10.6) to (11.5)
   return result;
}
```

To convert a fixed-point number from (12.4) format to (10.6), we shift left. In other words, shifting left allow us to gain higher precision. To convert a number from (10.6) format to (11.5), we shift right. In other words, shifting right allow us to gain a wider range. The amount to shift is given by the difference between ***f1*** and ***f2.***

Try: Write a fixed-point multiplication function that multiplies a (12.4) format with a (10.6) format to obtain an (24.8) format.
Try: Write a program that calls and checks the above function.

Lookup tables again

In the fixed-point implementation section, we looked at the basic representation and operations on real numbers using integers. Imagine if we are to revisit the first floating-point example given in the earlier section, and shown again below.

```
#include <math.h>

typedef struct {
  double x, y;
```

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

```
} point_t;

double compute_distance(point_t p1, point_t p2) {
  double t1 = p2.x - p1.x;
  double t2 = p2.y - p1.y;
  t1 = t1 * t1;
  t2 = t2 * t2;
  return sqrt(t1 + t2);
}
```

If we were asked to convert the above code to a fixed-point implementation using the (12.4) format, we could do something like the following:

```
#include <math.h>

typedef struct {
  short x, y;  // fixed point format (12.4)
} point_t;

short compute_distance(point_t p1, point_t p2) {
  short t1 = fixed_point_sub(p2.x, p1.x);
  short t2 = fixed_point_sub(p2.y, p1.y);
  short t3;
  double t4;
  t1 = fixed_point_mul(t1, t1);
  t2 = fixed_point_mul(t2, t2);
  t3 = fixed_point_add(t1, t2);
  t4 = conv_to_floating_point(t3);
  return conv_to_fixed_point(t4);
}
```

The above code is actually a good first attempt at tackling the problem. However, we got stuck when we got to the *sqrt* function, and had to make conversion to and back from the double type in order to be able to use the C standard *sqrt* function. This is an acceptable strategy, but what if we wanted to perform all computations strictly in fixed-point format? One option would be to rewrite the *sqrt* function using fixed-point operations. First, you have to recall how square root functions work (Wikipedia: Square Root and Wikipedia: Methods of Computing Square Roots). Your code would look like this.

```
short fixed_point_sqrt(short x) {
   // an algorithm for computing square root goes here ...
}
```

If you take your time (a lot of it) and implement the above function, the result would be very rewarding -- You would really learn a lot about square roots. However, there is another way -- one that might actually be fast (performance wise). First, recognize that the possible real numbers representable by a (12.4) implementation is limited to 65536. Of these, 1/2 are negative, and the *sqrt* function can bypass those. For the remaining 32678, we can pre-compute a table of square root values and store them in an array of (12.4) fixed-point numbers. How do we pre-compute a table? Easy, we write a C program to do the work for us, such as the one shown below.

```
#include <stdio.h>
#include <math.h>
```

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

```
#include <assert.h>

const double MIN_REAL = (short)0x8000 / 16.0;
const double MAX_REAL = (short)0x7fff / 16.0;

short conv_to_fixed_point(double x) {
  if( x < MIN_REAL ) {
    assert( 0 /* underflow */ );
  }
  else if( x > MAX_REAL ) {
    assert( 0 /* overflow */ );
  }
  else {
    return (short)(x * 16.0);
  }
}

double conv_to_floating_point(short x) {
  return x / 16.0;
}

void main() {
  short i;
  printf("short sqrt_loopkup_table[] = {\n");
  for(i=0; i<32678; i++) {
    printf("   %i,\n", conv_to_fixed_point(sqrt(conv_to_floating_point(i))));
  }
  printf("}\n");
  return 0;
}
```

If you compile and run the above C program, you'll end up with a C array, in particular, this fixed-point square root lookup table.

```
short sqrt_loopkup_table[] = {
   0,
   4,
   5,
   6,
   8,
   8,
   9,
   10,
   11,
   12,
   12,
   13,
   13,
   14,
   14,
   15,
   16,
```

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

```
   16,
   16,
   17,
   17,
   18,
   18,
   19,
   19,
   20,
   ...
};
```

Notice that the earlier program was written to generate the C code necessary to initialize our desired lookup table. Programs that generate other programs are common, and are particularly useful when creating large lookup tables.

Now, we revisit the *sqrt* function.

```
short fixed_point_sqrt(short x) {
   if( x < 0 ) {
      assert( 0 /* can't take sqrt of a negative number */ );
   }
   else {
      return sqrt_loopkup_table[x];
   }
}
```

We can re-write our distance function as follows.

```
typedef struct {
  short x, y;  // fixed point format (12.4)
} point_t;

short compute_distance(point_t p1, point_t p2) {
  short t1 = fixed_point_sub(p2.x, p1.x);
  short t2 = fixed_point_sub(p2.y, p1.y);
  short t3;
  t1 = fixed_point_mul(t1, t1);
  t2 = fixed_point_mul(t2, t2);
  t3 = fixed_point_add(t1, t2);
  return fixed_point_sqrt(t3);
}
```

Generally, lookup tables are values that are pre-computed statically (i.e., during design time) and looked up dynamically (i.e., as the application is running). They are frequently used in fixed-point implementations to compute math functions. However, their use can be applied beyond fixed-point systems. When using lookup tables, a programmer must take into consideration the memory requirements for the table vs. the overhead to compute the values at run-time. In our example, we had to use 64K bytes of memory to store the square root results. This is probably far more than the amount of code necessary to implement the square root function using an algorithm. However, our lookup approach will be much faster than the algorithmic version.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

The fixed-point implementation of the distance function, combined with the lookup approach for calculating square roots will be a much more efficient (time wise) implementation than one that is based on floating point arithmetic and the built-in *sqrt* function.

Do: Implement both a floating-point and a fixed-point version of the distance example. For your fixed-point version, implement one based on lookup-table computation of sqrt and one based on the standard C function. Now write a test program that calls the distance function with 100 different point sets. Measure the time required to run each of these 3 versions. Summarize your results.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

Chapter 11: Implementing SynchSMs on an FPGA

An FPGA is a type of programmable chip that is growing in popularity in embedded systems, commonly found alongside or even instead of a microcontroller. While a microcontroller is programmed with instructions that execute sequentially, an FPGA is programmed with circuits that execute concurrently. A task implemented as a circuit on an FPGA may be able to execute with a smaller period than on a microcontroller, such as with a period of 1 microsecond, versus a period typically no smaller than 1 millisecond on a microcontroller. Furthermore, multiple tasks can be implemented as distinct concurrently-executing circuits on an FPGA, eliminating the need for task scheduling, and eliminating jitter and missed deadlines. The drawback is that FPGAs compared to microcontrollers typically are bigger, consume more power, cost more, and are harder to program.

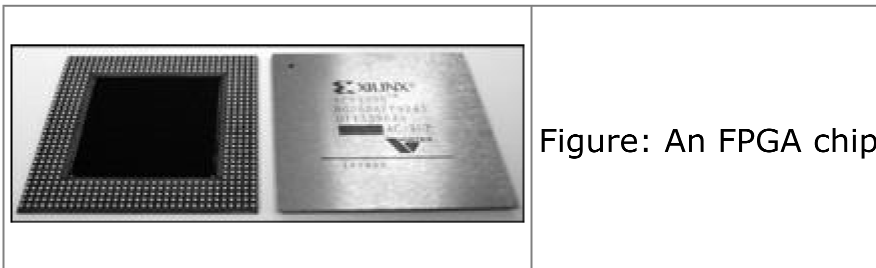


Figure: An FPGA chip

Implementing tasks on FPGAs usually requires extensive learning of languages, tools, and techniques for FPGAs . However, synchSMs can be straightforwardly implemented on an FPGA using a technique similar to implementing synchSMs on a microcontroller. For a microcontroller, we translated to a standard microcontroller language (C), and then we let a tool (a compiler) convert that language description into the machine instructions to be programmed into the microcontroller. Similarly, for an FPGA, we will translate to a standard FPGA language (VHDL), and the we will let a tool (a synthesis tool) convert that language description into the circuits to be programmed into the FPGA.

VHDL is currently one of two common languages for programming FPGAs. The other common language is Verilog. The techniques described can be straightforwardly adapted to Verilog. C is slowly becoming supported by some tools for programming FPGAs, and may become standard enough in coming years that synchSMs can be translated to a form of C, rather than VHDL, for implementation on FPGAs.

Translating a synchSM to VHDL

VHDL is a hardware description language (HDL) intended to describe circuits. This section provides a brief review of parts of VHDL relevant to our goal of implementing synchSMs on FPGAs.

A new system is declared as an *entity* that names the system and that lists the system's inputs and outputs. As before for our RIM microcontroller system, we'll assume the system has eight inputs A and eight outputs B. The system on an FPGA also has a system clock input and a system reset input.

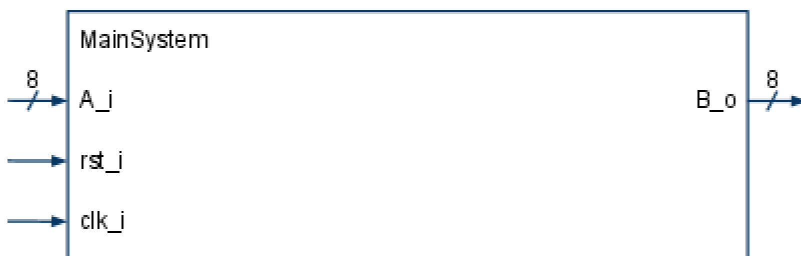
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY MainSystem IS
   PORT(
```

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

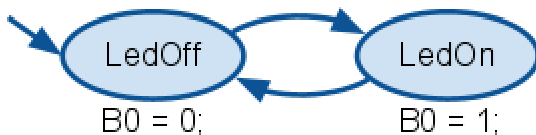
```
      A_i: IN std_logic_vector(7 DOWNTO 0);
      B_o: OUT std_logic_vector(7 DOWNTO 0);
      clk_i: IN std_logic;
      rst_i: IN std_logic
   );
END MainSystem;
```

We named the system "MainSystem." We capitalized VHDL keywords to make them clear, though VHDL is not case sensitive. A "PORT" is an input or output of the entity. We follow the convention of adding "_i" to input names and "_o" to output names. "std_logic" is a single-bit data type defined in the ieee library. "std_logic_vector" is a multi-bit data type. We use "7 DOWNTO 0" rather than "0 TO 7" so that the most significant bit is on the left. The above entity describes a system with the inputs and outputs shown below.



The new entity's internal circuits are described as an *architecture*. VHDL supports various architecture description approaches, but we'll focus on an architecture described as one or more processes. We'll use a process to describe a synchSM that is to be implemented as a circuit on the FPGA. The process consists of sequential statements. Below is an architecture having one process describing the given BlinkLed synchSM, serving as a model for translating synchSMs to VHDL.

BlinkLed
Period: 500 ms;




```
ARCHITECTURE beh OF MainSystem IS
BEGIN

BlinkLed: PROCESS(clk_i)
   TYPE states IS (LedOff, LedOn);
   VARIABLE state : states := LedOff;
   VARIABLE B0_v : std_logic := '0';
BEGIN
   IF (clk_i='1' AND clk_i'EVENT) THEN
      IF (rst_i='1') THEN
         state := LedOff;
         B0_v := '0';
      ELSE  -- SM case stmts go here
         CASE state IS  -- Transitions
            WHEN LedOff =>
               state := LedOn;
```

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

```
            WHEN LedOn =>
               state := LedOff;
            WHEN OTHERS =>
               state := LedOff;
         END CASE;
         CASE state IS -- Actions
            WHEN LedOff =>
               B0_v := '0';
            WHEN LedOn =>
               B0_v := '1';
         END CASE;
      END IF;
      B_o(0) <= B0_v; -- update port signal
   END IF;
END PROCESS;

END beh;
```

The process is named "BlinkLed." The process declaration "PROCESS(clk_i)" makes the process *sensitive* to clk_i, which means that the process executes whenever an event occurs on clk_i. Next come three declarations. The "TYPE" declaration defines possible synchSM states "LedOff" and "LedOn." The "VARIABLE state" declaration creates a variable "state" to maintain the current state, initialized to "LedOff". The "VARIABLE B0_v" declaration will be described shortly.

After "BEGIN" come the process' main statements. Because we only want the process to execute when the clk_i input changes from 0 to 1 (corresponding to a synchSM tick), and not from 1 to 0, the first "IF" statement's condition checks that "clk_i='1' ", meaning the event on clk_i that caused the process to execute must have been a change to 1. (The "clk'EVENT" part of that condition isn't strictly needed, but is included as good programming practice for cases when processes are sensitive to more than one input). The next "IF" statement checks if the rst_i input is 1, meaning the system is being reset, in which case the state is set back to its initial state LedOff and B0_v is set to 0. Otherwise ("ELSE"), the statements that describe the synchSM are executed. Those statements consist of a "CASE" statement for transitions, and another "CASE" statement for actions, just as when translating to C.

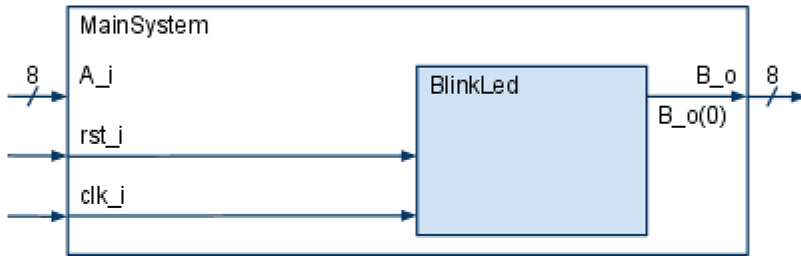
The need for variable B0_v is as follows. In synchSMs, actions can write variables as well as read variables, and system inputs A and outputs B were treated as variables. In VHDL, output ports can only be written, and cannot be read. Thus, the VHDL has a declaration for variable "B0_v" and writes to that variable in the CASE statement, for consistency with the synchSM model. Then, at the end of the process, we updated the output port with the variable's value. We also note that VHDL distinguishes between variables and signals, and ports are signals. The distinction is beyond our scope, but is manifested by the different notations for writing a variable (":=") and for writing a signal ("<=").

"--" starts a comment in VHDL; any text appearing on the rest of the line is ignored.

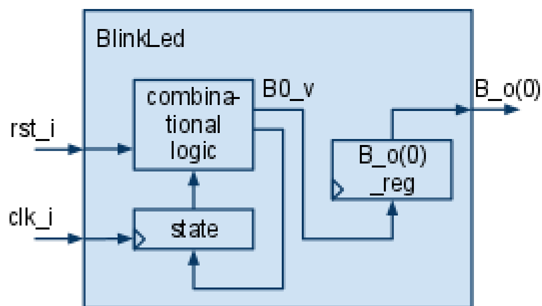
The above VHDL code describes the following architecture for the FPGA system. The BlinkLed process writes to just B_o(0) of the eight-bit port B_o.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu



The VHDL description can be provided to a synthesis tool, which will convert the BlinkLed process into a custom processor circuit. For the interested reader, the custom processor might have the following circuit structure.



The state variable becomes a register, because its value must be stored to be read by the next process execution. Combinational logic converts the current state register value (and the rst_i input) into a next state value, and a value for B_o(0). B0_v is not read across process executions, so becomes a wire rather than a register, but the B_o(0) signal is only written on rising clock edges and thus must retain its value at other times, necessitating a register.

Achieving the proper synchSM tick rate

The above VHDL assumes that clk_i ticks at 500 ms, as required by the BlinkLed synchSM. However, an FPGA's system clock input typically ticks at a much faster rate like 1 MHz or 100 MHz. Thus, another process is needed to convert the faster FPGA system clock to the desired 500 ms clock for the synchSM.

```
ARCHITECTURE beh OF MainSystem IS
   SIGNAL BLclk_s: std_logic;
BEGIN

BlinkLed: PROCESS(BLclk_s)
   ...
   IF (BLclk_s='1' AND BLclk_s'EVENT) THEN
   ...
END PROCESS;

BlinkLedClk: PROCESS(clk_i) -- Suppose 1 MHz
   VARIABLE cnt: INTEGER := 0;
BEGIN
   -- 1 MHz means 1 microsecond(us) per tick
   -- 500 ms * 1000us/ms = 500,000us
   IF (clk_i='1' AND clk_i'EVENT) THEN
```

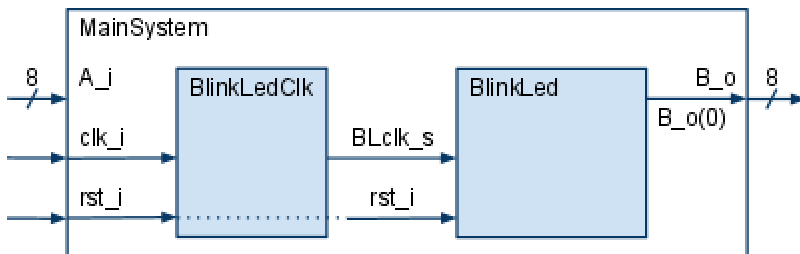
Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

```
      IF (rst_i='1') THEN
         cnt := 0;
      ELSE
         cnt := cnt + 1;
         IF (cnt = 500000) THEN
            BLclk_s <= '1';
            cnt := 0;
         ELSE
            BLclk_s <= '0';
         END IF;
      END IF;
   END IF;
END PROCESS;
-- Note: above process is shorthand for 1-state synchSM

END beh;
```

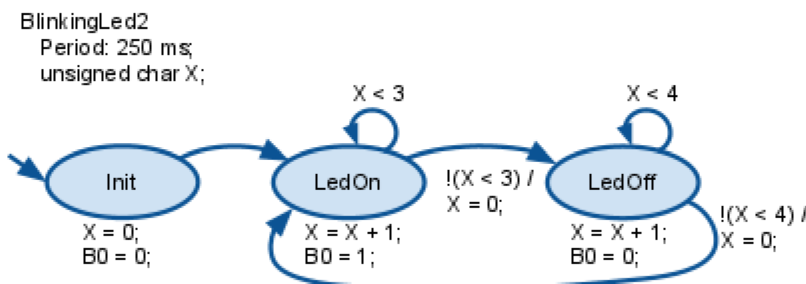
The processes work together as follows. Suppose the system clock clk_i runs at 1 MHz, meaning 1 microsecond per tick, so 500,000 such ticks would equal the desired 500 ms. The BlinkLedClk process is sensitive clk_i, and each time clk_i ticks (i.e., changes from 0 to 1), the process increments a counter variable "cnt". When cnt reaches 500,000, the process sets a signal BLclk_s to 1 and resets variable cnt to 0, else the process sets BLclk_s to 0. BLclk_s is declared as a global signal in the architecture (shown in bold above), so both processes can access that signal. The BlinkLed process is changed to be sensitive to BLclk_s (shown in bold above), so the BlinkLed process will now tick once every 500 ms, as desired.

The above VHDL code describes the following architecture for the FPGA system.



A synthesis tool would convert the two processes into two concurrently-executing custom processor circuits, connected as above. For the interested reader, the custom processor for BlinkLedClk will have a register for cnt, and combinational logic that can add 1 to cnt, store 0 into cnt, and compare cnt with 500,000, setting BLclk_s to 1 in that case.

Below is a synchSM that reads input A_i, has transition conditions, and has a local variable.



Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

The synchSM would again be described with two processes. One process would generate the 250 ms clock. The other process would look as follows.

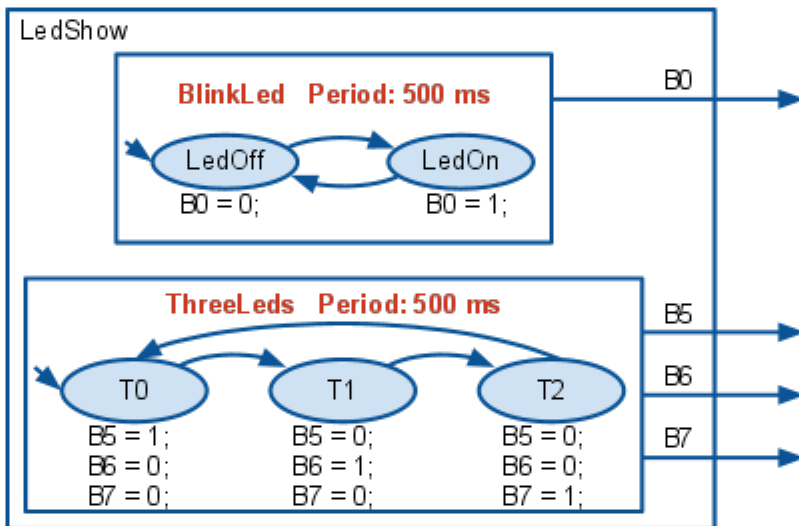
```
ARCHITECTURE beh OF MainSystem IS
   SIGNAL BL2clk_s: std_logic;
BEGIN

BlinkingLed2: PROCESS(BL2clk_s)
   TYPE states IS (Init, LedOn, LedOff);
   VARIABLE state : states := Init;
   VARIABLE X_v : std_logic_vector(7 DOWNTO 0);
   VARIABLE B0_v : std_logic;
BEGIN
   IF (BL2clk_s='1' AND BL2clk_s'EVENT) THEN
      IF (rst_i='1') THEN
         state := Init;
      ELSE
         CASE state IS  -- Transitions
            WHEN Init=>
               state := LedOn;
            WHEN LedOn =>
               IF (X_v < "00000011") THEN
                  state := LedOn;
               ELSIF (NOT(X_v < "00000011")) THEN
                  state := LedOff;
               END IF;
            WHEN LedOff => ...
            WHEN OTHERS =>
               state := Init;
         END CASE;
         CASE state IS -- Actions
            WHEN Init =>
               X_v := "00000000";
               B0_v := '0';
            WHEN LedOn =>
               X_v := X_v + "00000001";
               B0_v := '1';
            ...
         END CASE;
      END IF;
      B_o(0) <= B0_v;
   END IF;
END PROCESS;
```

Multiple synchSMs

A system containing multiple synchSMs is described as above with two processes per synchSM, one describing the states, the other to convert the system clock to a clock for the synchSM. Consider the LedShow system from an earlier chapter:

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu



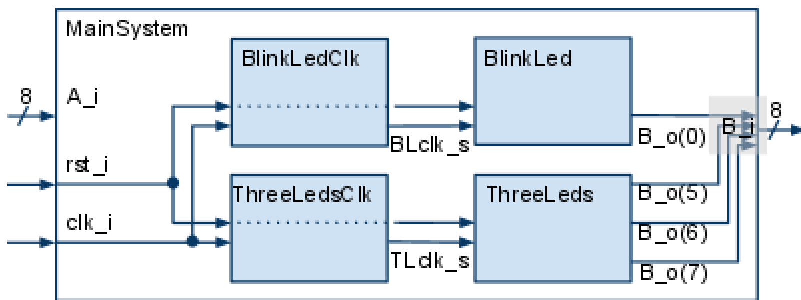
The system would be captured with two processes per synchSM, as shown:

```
ARCHITECTURE beh OF MainSystem IS
   SIGNAL BLclk_s, TLclk_s: std_logic;
BEGIN

-----

BlinkLed: PROCESS(BLclk_s)
   ...
END PROCESS;

BlinkLedClk: PROCESS(clk_i) -- gen 300 ms BLclk_s
   ...
END PROCESS;

------

ThreeLeds: PROCESS(TLclk_s)
   ...
END PROCESS;

ThreeLedsClk: PROCESS(clk_i) -- gen 200 ms TLclk_s
   ...
END PROCESS;

END beh;
```

The above code describes the following architecture for the FPGA:

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu



Suppose instead that the BlinkLed and ThreeLeds synchSMs had the same period and thus could share a clock signal. Good practice still uses distinct clock signal processes, so that later modifications (e.g., changing BlinkLed's period to 300 ms) are straightforward.

Some synchSMs communicate using a shared variable. VHDL supports shared variables, so translating from such synchSMs to VHDL is straightforward. Recall the MotionTriggeredLamp system from an earlier chapter, in which synchSM DetectMotion writes to a shared variable "mtn," which is read by a synchSM IlluminateLamp. The VHDL architecture would be as follows.

```
ARCHITECTURE beh OF MainSystem IS
   SIGNAL DMclk_s, ILclk_s: std_logic;
   SHARED VARIABLE mtn_sv: std_logic;
BEGIN

-----

DetectMotion: PROCESS(DMclk_s)
   ...
         mtn_sv := '1';
   ...
END PROCESS;

DetectMotionClk: PROCESS(clk_i) --gen 200 ms DMclk_s
   ...
END PROCESS;

------

IlluminateLamp: PROCESS(TLclk_s)
   ...
         IF (mtn_sv = '1') THEN
   ...
END PROCESS;

IlluminateLampClk: PROCESS(clk_i) --gen 200 ms ILclk_s
   ...
END PROCESS;

END beh;
```

Note that the shared variable is declared global to the architecture, in contrast to other variables that are declared within a process. The shared variable synthesizes to a register component that exists in addition to the four custom processors for the four processes.

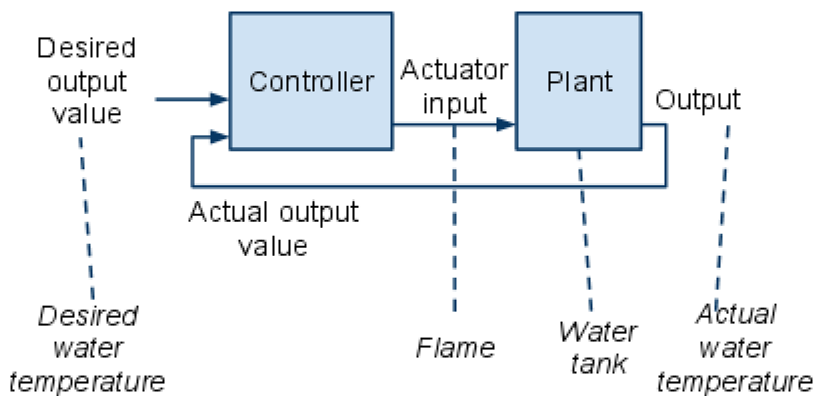
Copyright © Frank Vahid and Tony Givargis 2011

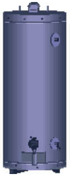
Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

Chapter 12: Basic Control Systems

A ***control system*** is a common type of embedded system that regulates the behavior of a physical device. A common example is a car's cruise control system, which regulates a car's speed. Another common example is a water heater, which regulates the temperature of water in a tank. The device being controlled is called the ***plant*** (such as the car or the water tank). In a simple control system, the plant has an ***output*** whose actual value is to be regulated (such as the car's speed or the water's temperature), and has an ***actuator input*** that affects the plant's output behavior (such as the car's accelerator or the water tank's flame). A ***desired output value*** is an input to the system (such as the desired car speed or the desired water temperature). The difference between the desired output value and the actual output value (desired - actual) is known as the ***error***. A ***controller*** strives to reduce the error to zero by changing the value of the actuator input in response to positive or negative error values.





Consider designing a controller for a water heater. The controller will be implemented on the RIM microcontroller. Suppose the flame (the actuator, which is actually a gas valve) input value can range from 0 (meaning no flame) to 200 (meaning maximum flame), and is output from RIM as an 8-bit unsigned number on B. Suppose that the desired water temperature can range from 0 to 100, but due to limited number of input pins, is divided by 10 and then input to RIM as a 4-bit number (ranging from 0 to 10) on A3A2A1A0. The output water temperature similarly ranges from 0 to 10 and is input to RIM on A7A6A5A4. The controller should sample the actual output value and compute a new actuator input value at least once every 5 seconds.

Try: Complete the following synchSM describing a controller that sets the flame value maximally (i.e., to 200) whenever the desired temperature is greater than the output temperature.

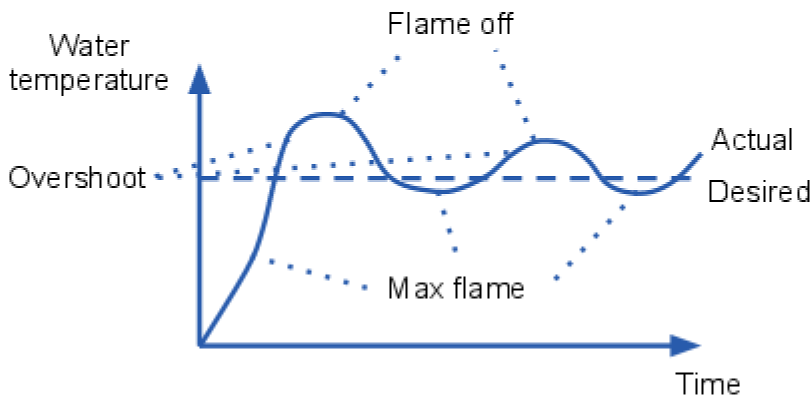
Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

```
OnOff_Ctrl
  Period: 5 seconds;

  #define Actuator B
  unsigned char Desired(){
    return (A & 0x0F);
  }
  unsigned char Actual() {
    return ( (A & 0x0F) >> 4);
  }
  char Error;
```

```
  Actuator = 0;        Error = Desired() - Actual();
                       FINISH...
```

The above controller approach, called ***on-off control***, is simple but will likely have the undesirable feature of overshooting the desired water temperature. Turning on the flame maximally may provide too much heat to the water; when the flame is then is turned off, the latency of the rising water temperature causes the temperature to rise above the desired temperature, a problem known as ***overshoot***. Furthermore, a second problem is the ***oscillation*** that may occur as the actual temperature rises above and falls below the desired temperature. Trying to solve these problems by decreasing the "on" value from 7 to a smaller value like 3 will decrease overshoot and oscillation, but with the problem of a slower ***rise time***, which is the time required to bring a lower actual value up to the desired value. For a water heater used for a shower, the problem of overshoot may result in a scalding hot shower, oscillation may result in a shower that annoyingly keeps changing temperature, and slow rise time may result in a shower that is too cold for a long time after someone else has taken a shower.



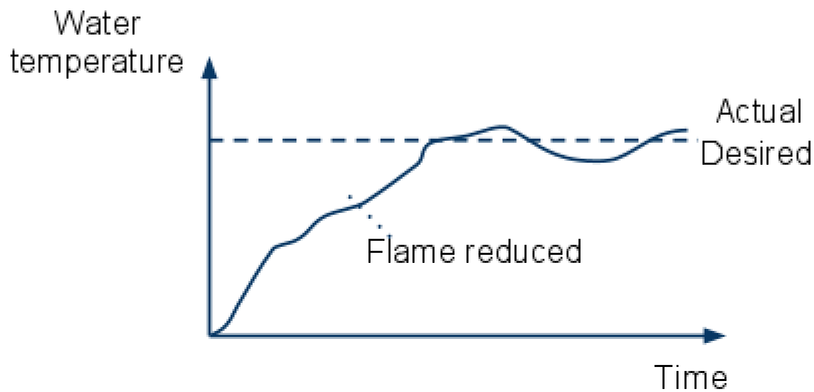
Oscillation whose amplitude decreases over time is a problem; even worse is oscillation in which the amplitude *increases* over time. A system with such oscillation is said to be **unstable**, resulting in the system going "out of control." Instability should be avoided entirely.

Proportional control

A partial solution to the above problems is to set the flame amount *proportional* to the error. When the error is very large, the flame amount should be large. When the error is smaller, the flame should be smaller. Thus, if the water is very cold, the flame will be maximum, decreasing rise time. As the actual temperature approaches the desired temperature, the flame is reduced, resulting in less overshoot when the flame is turned off upon the actual temperature reaching the desired temperature. Oscillation may also be reduced.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

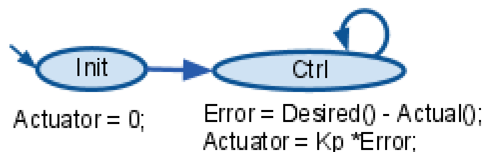


A ***proportional controller*** sets the actuator input value equal to the error times a constant:

Actuator = Kp*Error

Kp is a constant. The best value of Kp may be determined through experimentation with a prototype water heater. Alternatively, the value of Kp may be determined by first creating a water tank simulator (perhaps written in C and running on a PC), which includes modeling the tank of water using mathematical equations based on the physics of water and heat transfer, and then experimenting to find the best Kp. Engineers trained in control systems may instead determine Kp via analysis of the mathematical equations.

```
P_Ctrl

  Period: 5 seconds;

  // Actuator, Desired, and Actual defined as earlier

  char Error;
  const char Kp = 5; // Assume found to be the best
```

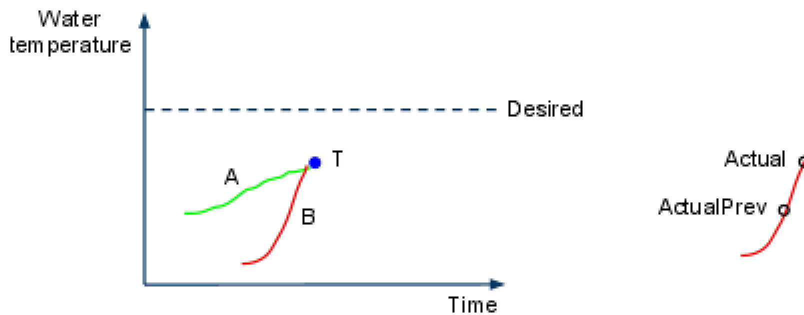


Proportional-derivative (PD) control

The problems of overshoot, oscillation, and slow rise time can be further reduced by having the controller consider the *rate of change* of the actual output value as it approaches the desired value. Consider the given point in time T for two different situations A and B of the actual output value:

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu



Both situation A and situation B have the same error at time T. However, situation A has a slow rate of change just before time T, whereas situation B has a fast rate of change. Situation B clearly requires a reduction of the actuator input value, lest the desired value be overshot. On the other hand, situation A could continue with the same actuator input value. In a water heater, scenario A might occur in the winter when the air around a poorly-insulated water heater (located in a a garage perhaps) is very cold; scenario B may correspond to a hot summer day. In a car cruise controller, scenario A may occur when a car is driving into the wind, uphill, and/or is carrying a heavy load; scenario B when going with the wind, downhill, and/or when carrying a light load.

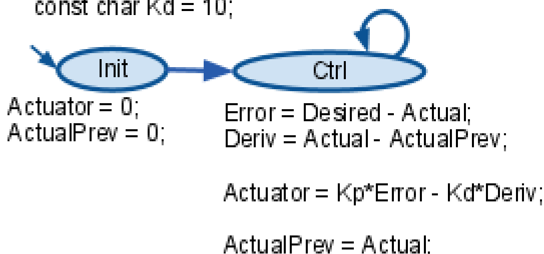
The rate of change of the actual output plot can be determined by computing the slope of the plot at point T. Recall from calculus that the derivative of a function yields the slope. A simple way to approximate the derivative at a given point is to compute the difference between the current actual output value and the previously-sampled actual output value: Deriv = Actual - ActualPrev. This derivative can be multiplied by a constant Kd, and the resulting derivative term can then be added to the proportional term:

Actuator = Kp*Error - Kd*Deriv

Kd is typically larger than Kp. The derivative term is subtracted from the proportional term rather than added, because if the error is positive and the slope is very large (as in situation B above), we want to *reduce* the actuator value.

A synchSM can then be defined as follows:

```
PD_Ctrl
  Period: 5 seconds;
  // Actuator, Desired, and
  Actual defined as earlier

  char Error;
  char Deriv;
  unsigned char ActualPrev;
  const char Kp = 5;
  const char Kd = 10;
```

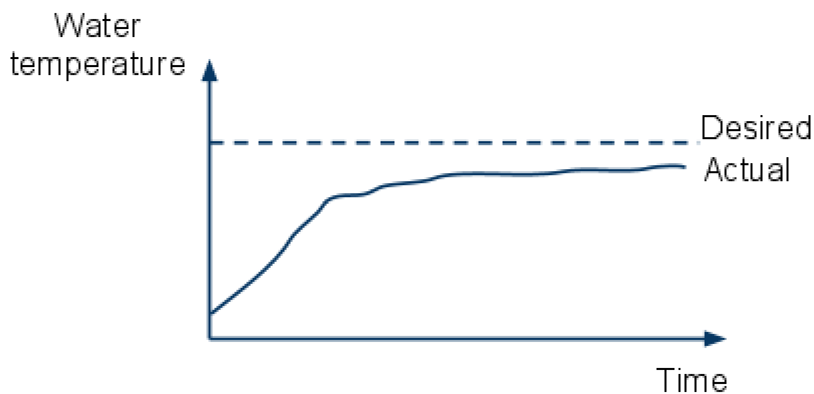


Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

Proportional-integral (PI) control

An undesirable feature in a control system is ***steady-state error***, wherein the actual output value never actually reaches the desired output value:



In a water heater, steady-state error could occur for example if the proportional constant Kp was determined with the surrounding air at a normal room temperature, but then the water heater is installed into a very cold garage, such that the constant Kp does not create a flame strong enough to overcome the loss of heat from the water tank to the surrounding air of the garage.

Steady state error can be approximated by summing the error values computed for each sample. Recall from calculus that the sum of the points of a plot can be computed as the area under the curve, which is the plot's integral, so this term is called an integral term. To prevent this term from growing too much, the term is typically constrained to be within a max and min value, as follows:

```
Integ = Integ + Error;
if (Integ > IntegMax) { Integ = IntegMax; }
else if (Integ < IntegMin) { Integ = IntegMin; }
```

This integral can be multiplied by a constant Ki, and added to the proportional and derivative terms:

Actuator = Kp*Error + Ki*Integ - Kd*Deriv

Ki is typically much smaller than Kp.

A controller using the above equation is known as a PID controller (each letter indicating inclusion of the proportional, integral, and derivative term, respectively).

Additional issues

The Actuator calculation could result in a value outside the allowed Actuator value range. For the water heater example, only values between 0 and 200 are allowed. A controller may therefore introduce a temporary variable, compute the actuator value as above, and then set that value to 200 if it is greater, or set it to 0 if it is less, before finally setting Actuator to that temporary value.

Sampling rate impacts the calculation of the derivative. If a very fast sampling rate is being used, the derivative might be computed as the average of the past several computed slopes, to avoid very rapid changes.

PID control is commonly done using floating point numbers rather than integers.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

PID tuning

The best values of constants Kp, Kd, and Ki depends on the plant being controlled. Example constant values for a water heater might be Kp=5, Kd=10, and Ki=1, but these values can vary. Several trial-and-error methods have been defined to select good constant values, assuming the plant can be experimented with or a simulation system exists. A popular method is called the Ziegler-Nichols method:

- Set Ki=0 and Kd=0 initially, and set Kp to some small value initially (e.g., 1).
- Change the desired output value and observe the actual output value. Increase Kp until the actual output value oscillates with a constant amplitude (the oscillation amplitude is not increasing over time, nor is it decreasing over time -- it's a steady oscillation).
- Record the value of Kp, calling it Ku. Also record the oscillation period (in seconds), calling it Pu.
- Set Kp = Ku / 1.7, Ki = (Kp * 2) / Pu, Kd = (Kp * Pu) / 8,

The above is suitable for a rough tuning. More sophisticated methods and tools exist. Control system design is a thoroughly developed field with entire textbooks devoted to its study. For any safety-critical or other system where excellent response is required and/or instability is dangerous, qualified control engineers should be sought.

Online simulators can be found to simulate PID controllers and plants, such as the simulator at:

http://www.chem.mtu.edu/~tbco/cm416/newpida.html (To use, press Pause, change Manual to PID, select Kc (same as our Kp), and resume. Reload page and repeat until oscillation found, then related/pause and set Kc, Ti, and Td values and resume.
http://controlcan.homestead.com/files/Controller/controlcontroller3p_pid.htm

In using such simulators, be aware that many PID introductions and simulators use a different convention for constants in the PID equation:

Actuator = Kp * (Error + (1/Ti)*Integ - Td*Deriv)

Ti and Td are constants that adjust the Kp constant. Using that convention for the constants, the Ziegler-Nichols method's first step sets Ti=a large number and Td=0, and the last step sets the constants as follows: Kp = Ku / 1.7, Ti = Pu/2, Td = Pu/8.

Sources for the above information include:

- PID Tuning -- Classical: http://controls.engin.umich.edu/wiki/index.php/PIDTuningClassical#Ziegler-Nichols_Method
- PID Without a PhD -- http://www.embedded.com/2000/0010/0010feat3.htm

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

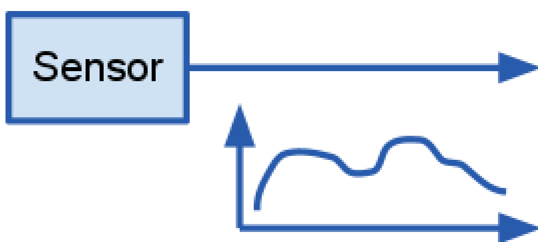
Chapter 13: Basic Digital Signal Processing

A ***signal*** is a value that is continuously changing over time. Often, signals represent some physical phenomena, for instance temperature, sound, light, pressure, or force. Signals may also represent things other than physical phenomena, for example, the value of a company stock, or the number of hits on a web site per second. It is often necessary to ***process*** signals in a number of ways to obtain usable results for a variety of applications. Some of these applications are very simple in nature, for example, a ***thermostat*** continuously monitoring the temperature in a room and enabling/disabling the heating/cooling system when certain high/low thresholds are exceeded. Other applications may be more complicated, for example, a ***radio*** continuously receiving electromagnetic waves and extracting an audio broadcast that is played back through the radio's speaker. With the increased use of digital components in system design (in particular the microprocessor) signal processing is frequently referred to as **digital signal processing (DSP)**. Regardless of the application, DSP is an essential engineering domain that follows well established design principles. This chapter is a light introduction to the field of DSP. We attempt to introduce the basic concepts in an informal manner. Interested students are encouraged to seek a dedicated text book or course on the topic of digital signal processing.

Throughout this chapter, we will present ***concepts*** and ***implementations*** as two separate entities, with an emphasis on concepts. Concepts help us gain a strong understanding of things in an abstract way. Implementations help us understand design considerations and tradeoffs in a tangible way. Remember that a good engineer must be able to distinguish between the two and be ready to reason and conduct work in one, the other, or combined fashion.

Sensors

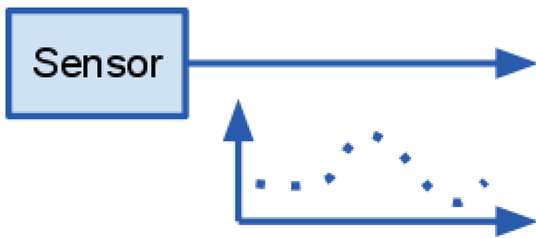
We begin by considering a ***sensor***. Conceptually, a sensor is a component with a single output as shown below. A sensor outputs a signal that varies over time. When dealing with signals that represent physical phenomena, we can think of a sensor as a device that converts energy from one form (heat, light, pressure, etc.) to another form (typically an electrical voltage).



Most sensors output an ***analog signal***. An analog signal is one that is defined for all instances of time, in other words, an analog signal is uninterrupted. For example, a temperature sensor that outputs an analog voltage in the range of 2V to 3V, corresponding to a temperature in the range of 0 to 100 degree Celsius. We can just as easily imagine a sensor that outputs a ***digital signal***. A digital signal is one that is defined for discrete instances of time. For example, a digital temperature sensor might output an 8-bit binary value, in the range of 0 to 100 degree Celsius, every 0.1 second. Viewed from the outside, such a sensor would be generating a stream of ***samples*** (say Temp0, Temp1, Temp2, ...) that represent the temperature at time 0 second, 0.1 seconds, 0.2 seconds and so on. To be more precise, a sensor with a digital output should be shown as follows.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

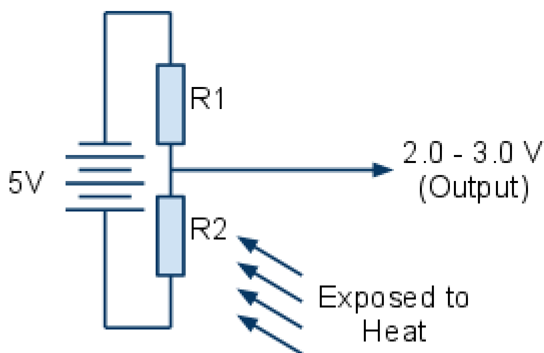


However, we will draw a signal as a continuous curve, with the understanding that if referring to a digital signal, it should be interpreted as a sequence of samples and if referring to an analog signal it should be interpreted as a continuous value.

When selecting a sensor for an application, we must be aware of the following characteristics:

- Sensors are limited in their sensing capabilities. Our analog temperature sensor is able to sense temperatures in the ***range*** of 0-100 degree Celsius.
- Sensors have accuracy limitations. Our digital temperature sensor can tell us that the temperature is 15 or 16 degrees, but not 15.3 degrees. In technical terms, our temperature sensor has a +/- 1 degree ***resolution***.
- Sensors output values that are ***raw*** and would require further processing to be of much use. Our analog temperature sensor outputs a voltage in the range of 2V to 3V that would require conversion before it can be displayed in Celsius.

As expected, the implementation of a sensor varies depending on what it is designed to sense, how accurate it needs to be, and what conditions it must operate under. We can, for instance, design a simple analog temperature sensor as shown below. Such a temperature sensor will surely not win any kind of a design award, but it might help with demystifying the art of sensor design.



Using Ohm's law and basic circuit analysis, we can see how such a sensor might work. First, note that the output is the voltage drop across R2. The voltage drop across R2 is the current through R2 multiplied by R2's resistance. The current through R2 is the supply voltage (5V) divided by the total resistance in the system, or, I = 5 / (R1 + R2). Combining these, we obtain the voltage at the output of the sensor as V = R2 * (5 / (R1 + R2)).

Assume that R1 is a very accurate resistor that is shielded against heat and manufactured using materials that have excellent tolerance to temperature. In other words, R1's resistance, say 100 Ohms, is not expected to change by much as the temperature around the sensor rises or falls. R2, on the other hand, is fully exposed to ambient temperature and is manufactured using material that are very sensitive to temperature. In other words, R2's resistance will increase or decrease as the temperature around it rises and falls. At an ambient temperature of 50 degree Celsius, R2's resistance is expected to be equal to that of R1, i.e., R1 = R2 = 100 Ohms. Thus the output will be V = 100 * (5 / (100 + 100)) = 2.5V, as expected. As the temperature rises, R2's resistance will

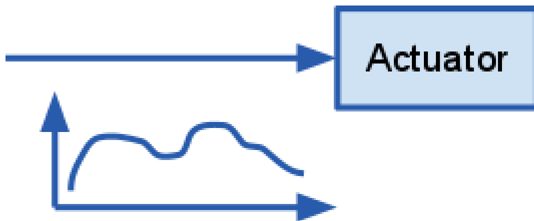
Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

increase and, at 100 degree Celsius, R2's resistance might be 150 Ohms. Thus, the output will be V = 150 * (5 / (100 + 150)) = 3V, as expected. At a temperature of 0 degree Celsius, R2 might be 67 Ohms. Thus, the output will be V = 67 * (5 / (100 + 67)) = 2V.

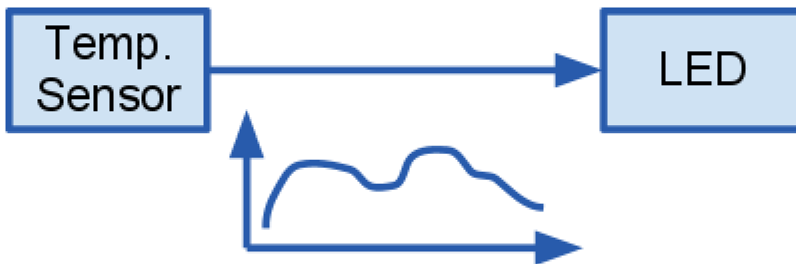
Actuators

We now consider an ***actuator***. Much like a sensor, conceptually, an actuator is a component with a single input as shown below. Often times, an actuator will convert its analog or digital input signal to some form of energy. Examples of actuators are speakers (mechanical energy), LEDs (light energy), heaters (heat energy), and so on.



As is the case with sensors, we must be aware of the range and accuracy limitations of actuators and carefully consider their input signal specification. As expected, the implementation of actuators is highly application specific. Luckily, an embedded system designer can easily obtain sensors and actuators of all kinds from various vendors that specialize in the design of these devices. The real challenge is selecting appropriate devices that best fit the application requirements.

With what we have learned, we can attempt to build our first system, a so called "Hello World" example of signal processing. Let us do so as shown below.

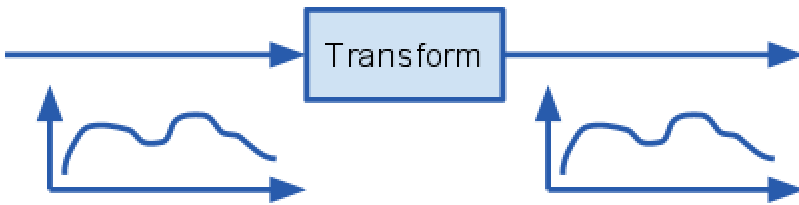


In the above example, we attempt to build a system where the LED's brightness increases or decreases as the room temperature rises and falls. For now, we assume that the output/input of these components are compatible. Provided that this is the case, we might have a system that is functioning as expected. However, our "Hello World" example is too limited and utterly boring!

What we need is to add "processing" to the mix so that our "Hello World" example can become an authentic signal processing application. For that to happen, we introduce a very important building block, namely a ***transform*** component, as shown below.

Copyright © Frank Vahid and Tony Givargis 2011

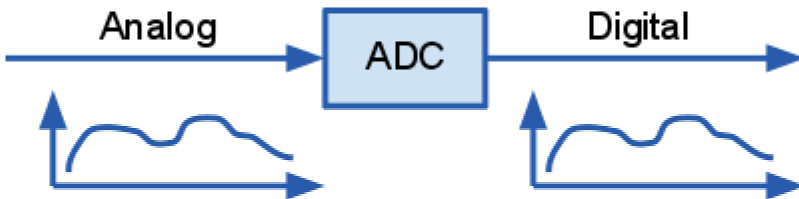
Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu



A transform component is one that takes its input signal, does some transformation or processing, and generates a corresponding output signal. Some transform components are very simple, others are very intricate. We'll introduce a number of these transforms with the aim to eventually establish a generic DSP architecture.

Analog to Digital Converter

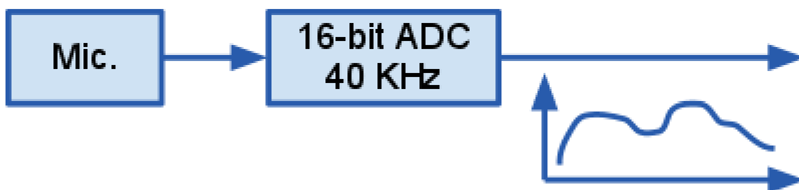
An important transform component is the ***analog to digital converter*** (ADC). An ADC is a ***mixed-signal*** transform. A mixed-signal device is one that has a mix of both analog and digital I/Os. An ADC, for example, has an analog input and a digital output.



ADCs have the following key characteristics:

- Input ***range***, measured in Volts, is the range of voltages that can be applied at the input of the ADC.
- Output range, or ***quantization***, measured in bits, is the number of bits in each output sample (typical values are 8, 12, 16, 24, and 32). It is common to say an "8-bit ADC", meaning an ADC with output samples that are 8-bit wide.
- ***Sampling rate***, measured in Hz, is the rate at which the ADC generates its output samples.

Let us consider a 16-bit ADC with an input range of -1V to +1V and sampling rate of 40 KHz. We will connect this ADC to the output of a microphone in a naive attempt at converting sound to a stream of samples for further processing.



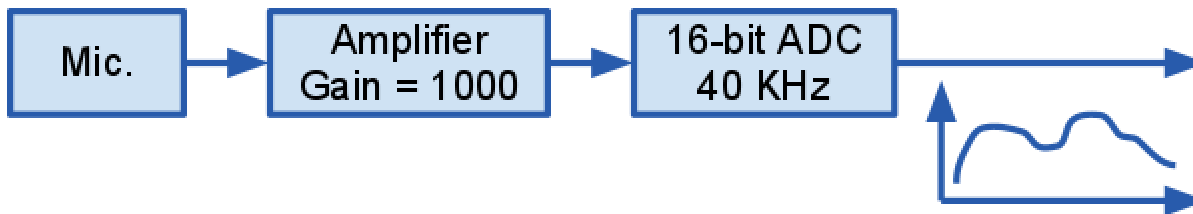
Amplifier

In our attempt above, we will be faced with a number of issues. First, the microphone's output is likely to be in the range of +/- 1mV. The ADC's input is in the range of +/- 1V or 1000mV. We can be sure that under this arrangement we are unlikely to make a good use of our ADC capabilities,

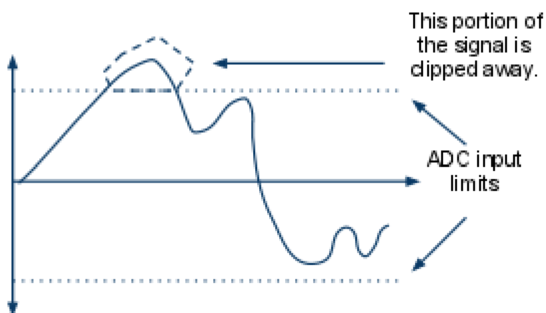
Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

essentially under utilizing our ADC. An electrical engineer might use the term ***signal under-loading*** to describe this design flaw. We solve this problem by adding an ***amplifier*** trasform. An amplifier is an analog device that takes its input signal and multiplies it by a number, called ***gain***, to obtain its output. Most commonly, the gain of an amplifier is larger than 1, for instance 1000. However, it is possible to design an amplifier with a gain of less than 1. Such a device is often referred to as an ***attenuator***. An amplifier with a gain of 1 is referred to as a ***buffer***. Continuing with our design, we obtain the following system.



The microphone's output, after amplification, is expected to be in the range of +/- 1V. Our attempt at fixing the problem of under-loading may introduce a new problem, commonly known as ***signal overloading***. Signal overloading occurs when the input range of a transform is exceeded. The result of such a condition is ***clipping***, namely the loss of signal values that are beyond the input range, as shown below. Consider the case that our microphone, in the presence of a loud sound, will output a voltage that is within +/- 1.2mV. After amplification, the resulting signal, at +/- 1.2V, will exceed the ADC's input specification. Clearly, there is an inherent tradeoff between maximizing conversion accuracy by amplifying the signal to fit the input of the ADC while not exceeding the limits and cause clipping.

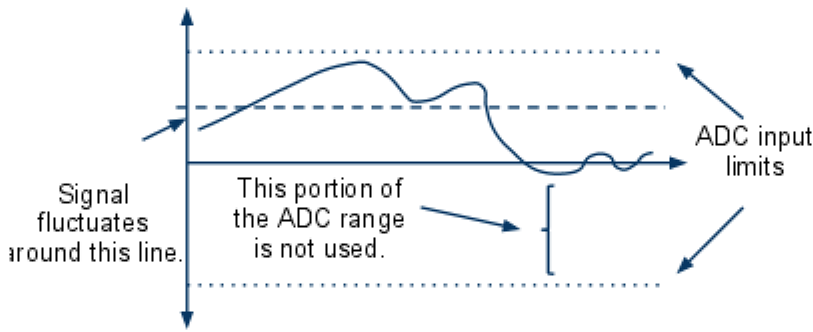


Bias Correction

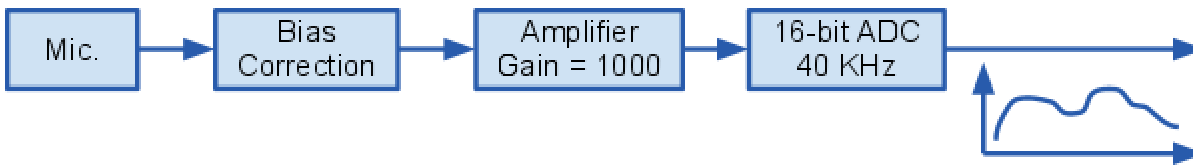
Another problem that needs attention is that of ***bias*** correction. Many signals (including audio, video, and electromagnetic waves) are centered around 0. Specifically, a signal is centered around 0 if it can be viewed as a curve that fluctuates (or alternates) above and below the horizontal axis (e.g., the figure above). Mathematically, the integral (sum of values) of such a signal yields 0. Sometimes, a sensor might generate a signal that fluctuates above and below a non-zero value. Put another way, it has a bias built into its output. (This is often due to manufacturing flaws.) Engineers sometimes refer to a signal that has an alternating nature to it as an AC signal (alternating current). Likewise, a bias is referred to as a DC (direct current) signal. Thus, it is common to speak of the bias correction problem in terms of "eliminating the DC component." Regardless of what it is called, a bias will lead to a waste in the ADC conversion precision. Can you see how? The answer is give in the following figure.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu



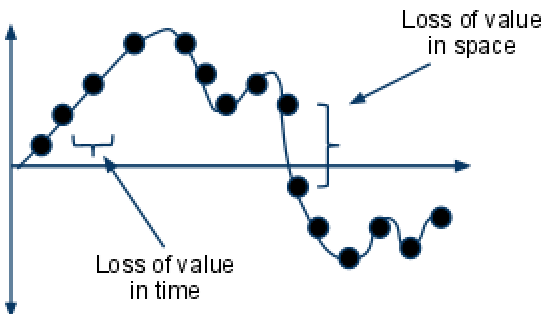
Note that amplification can not eliminate the waste. In fact, bias correction must be performed first to allow for effective amplification. To eliminate the bias, we introduce a new transform component and revise our on going example.



A bias correction device is one that maintains the integral of its input signal at all times (i.e., the bias) and outputs a value that is equivalent to the input signal minus this bias. A simple implementation of a bias correction device uses a capacitor as a way to block the DC (bias) but let the AC (signal) through.

Sampling Rate & Quantization

So far, we have made an effort to fully utilize our ADC input range by performing ***signal conditioning*** activities along the path from the sensor to the ADC. An important question that one should be asking is just how precise does the analog to digital conversion need to be? As with most answers, we take a look at this question in terms of space and time. The conversion process from analog to digital breaks our signal value into little chunks (space). The conversion also breaks our signal value into samples (time). Both of these processes remove information from the original signal as shown below.



What governs the amount of loss of signal along the time axis has to do with the sampling rate of the ADC. In other words, the higher the sampling rate, the smaller the loss of value along the time axis. What governs the amount of loss of signal in the space axis has to do with the quantization or bit-width of the ADC (sometimes referred to as the ***resolution*** of the ADC). In other words, the

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

higher the number of bits in each sample, the smaller the loss of value along the space axis. We'll take a look at how to determine these two parameters in a bit more detail.

We begin by introducing the concept of ***dynamic range***. The dynamic range of a device (e.g., an ADC) is a measure of the precision of a device with respect to the range of values that it can convert. The ADC we've used in our example can convert an input voltage in the range of -1V to +1V, thus we might reason that the conversion ***range*** of the ADC is 2 Volts. But, do not confuse the conversion range with dynamic range. The conversion range ignores the voltage ***gap*** between two consecutive samples. For example, a 16-bit ADC designed to convert an input signal in the range of -10V to +10V will have a range that is 20 Volts (better than the ADC from our example). But, it would be unfair to say that this ADC has a better accuracy than the one from our example. Can you see why? The answer is in the fact that both ADCs are bound to use the same 2^{16} distinct binary values to represent their inputs. One has a wider input range, but it has a bigger gap between two consecutive voltages. A better method for computing the dynamic range, thus, would be to combine the range and gap in a single formula, as follows:

$$range = V_{max} - V_{min}$$
$$gap = range / 2^N$$
$$dynamic\ range = range / gap = (V_{max} - V_{min}) / ((V_{max} - V_{min}) / 2^N) = (V_{max} - V_{min}) / (V_{max} - V_{min}) * 2^N = 2^N$$

Interestingly, all N-bit ADCs have the same dynamic range regardless of their input voltage specification. Engineers often express dynamic range using the decibel logarithmic scale as shown below:

$$dynamic\ range\ (dB) = 10 * \log(range^2 / gap^2) = 10 * \log(range/gap)^2 = 20 * \log(range/gap) = 20 * \log(2)^N = N * 20 * \log(2) = 6.02 * N$$

You may wonder why range and gap are squared. This has to do with the fact that range and gap are measured in Volts. However, dynamic range is often used in the context of signals that represent physical phenomenas, i.e., energies. You may recall, from your physics courses, that energy is proportional to V^2. Per the above discussion, the ADC used in our example has a dynamic range of about 96 dB.

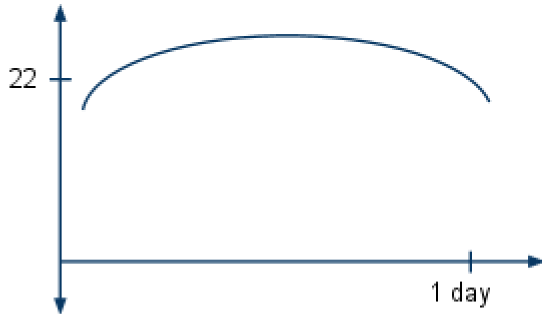
In the context of DSP, dynamic range sums up, as a single number, the range and precision of a system. Clearly, the desired dynamic range is an application specific parameter. Let us imagine the human ear as a sensor (a reasonable assumption). As such, it has a precision or sensitivity to the loudness of sound that is present in its surroundings. For example, it might tell that sound A is louder than sound B as long as the difference between A's and B's loudness level is greater than *gap* -- it is certainly the case that the human ear does not have infinite precision. The human ear is also limited to how loud of a sound it can hear. It might, for instance be able to hear anything from a complete silence (loudness 0) to a really loud sound (loudness *range*) -- beyond this level, the human ear would suffer permanent damage. Just like our ADC, we can compute the dynamic range of the human ear using *gap* and *range*. Scientists, through experimentation have established that, on the average, the human ear has a dynamic range of 130 dB. If we were to design an audio system that fully engage our human subject, we would consider using an ADC that could deliver about 130 dB. Similar dynamic range measurements have been performed on human eye, sensitivity to heat through our skins, etc.

Try: What kind ADC resolution is necessary to achieve 130 dB?

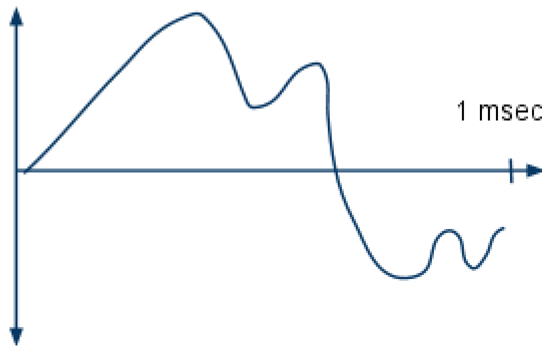
We now turn our attention to ***sampling rate***, or the rate at which our input signal is converted to a stream of samples. We'll need to come up with a framework for determining an appropriate sampling rate. To do so, we'll consider a couple of different scenarios. First, we'll look at a very boring signal, one that changes very slowly. One such signal might be the temperature of a room, as shown below.

Copyright © Frank Vahid and Tony Givargis 2011

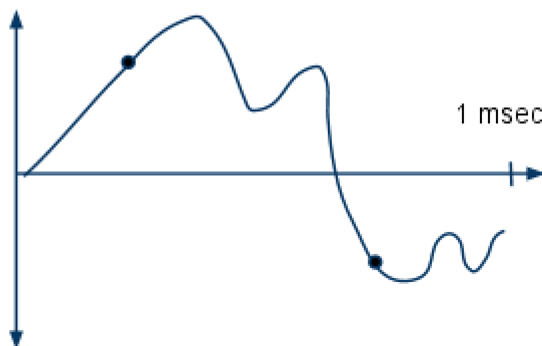
Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu



The temperature of the room, per the above plot, changes very slowly. For instance, during an entire day, it might slowly fluctuate between 20 and 24 degree Celsius. Would it make sense to sample the temperature of the room at a rate of 40 KHz, or 40,000 samples per second? Clearly, this would be a major waste of resources. If the stream of samples is examined, one would notice that most are identical to each other. There is very little "interesting" information captured. Now, let us go to the other extreme. Consider our running example. Our sound system, at some point in time, might be exposed to a short music clip that is 1 msec in duration, as shown below.



Would it make sense to sample this sound clip at the rate of 2 KHz? At this rate, we would obtain exactly 2 samples, as shown below.



Would it be possible to reconstruct, or determine the nature of the original signal, if all we had were the two samples? The answer is a clear no. We would be much better off sampling at a higher rate to be able to obtain a sequence of samples that resemble the shape of the original signal. You must start to see that the sampling rate must have something to do with how busy our input signal is. Put another way, the sampling rate is related to the rate of change of the input signal. According to the ***sampling theorem***, an analog signal can be reconstructed from its samples if the sampling rate is at least twice the highest frequency present in the signal. This means, if the signal contains

Copyright © Frank Vahid and Tony Givargis 2011

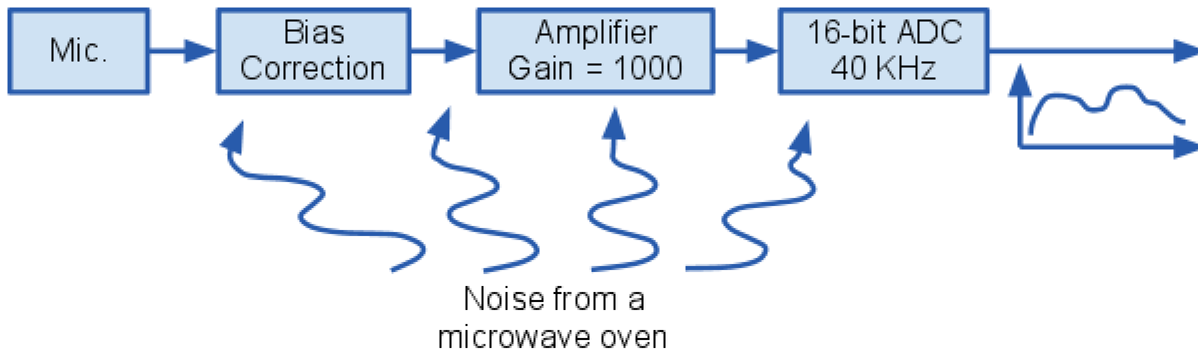
Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

the highest frequency component of *f* Hz, the sampling rate must be at least 2 * *f* Hz. This sampling rate is also known as the ***Nyquist*** rate.

The highest frequency present in a signal is very application dependent. Our microphone, for instance, will sense sound, which is known to be less than 20 KHz. Thus, our ADC, with a sampling rate of 40 KHz is sufficient to faithfully convert the sound to a digital stream. It should not come as a surprise to you that the audio CD format has a sampling rate of 44.1 KHz.

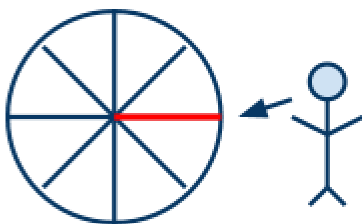
Aliasing

All systems are subject to ***noise***. Noise is any unwanted signal, present in the surroundings of a DSP system, that makes its way into the signal path. As illustrated below, a microwave owen might be radiating small amounts of electromagnetic energies that somehow penetrate our system and combine with our sound signal. Engineers always talk about signal to noise ratio. Obviously, it is desirable to have a very small noise relative to the signal, thus the ***signal-to-noise ratio (SNR)***. Higher SNR is better.



Noise reduction is often an implementation concern. Each of the components described earlier can be designed with sufficient shielding to eliminate noise. If you take your cell phone apart, you'll see that some of the electronic components are enclosed in metal cases. Likewise, if you take a look at the coax cable that connects your DVD player to your TV, you'll notice that the cable is heavily shielded. If you look for them, you'll see noise reduction shields everywhere.

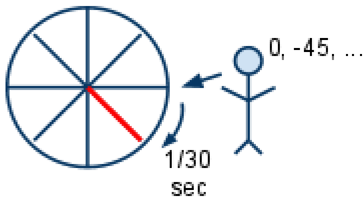
Despite all implementation efforts to shield a DSP system, some noise will find its way into the signal path. The question is if we need to be concerned with *all* noise? Consider our example. Since we are interested in sound, we might decide that noise from the microwave owen, being in the GHz range, can be tolerated. In other words, such high frequencies will not be audible to a human and, even if they are present in our signal path. They'll make their way through our DSP system and come out the other end (say a speaker) without being noticeable. This is a reasonable conjecture. But, as it turns out, due to ***aliasing***, it is flawed. Aliasing is the presence of a false or unexpected signal in our signal path. Let us illustrate the concept with a simple example. Imagine a bicycle wheel that is stationary. Now, imagine a person staring at one of the spokes of the wheel, shown in red below.



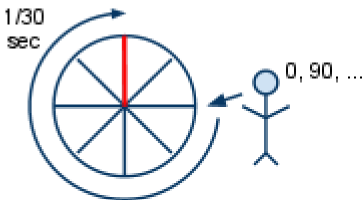
Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

Now, let us turn the wheel very slowly. In fact, let us turn it so that in exactly 1/30th of a second the wheel appears as follows.



Let us assume that the human eye is capable of sampling at a rate of 30 Hz (this is probably an over estimate, the real number is closer to 25 Hz -- coincidently, birds have a much higher sampling rate, therefore, if you make your bird watch TV with you, your are making it watch a slide show). Our human subject will take 2 samples, one when the wheel was stationary (0 degrees) and the other when the wheel is at the new location after 1/30th of a second (-45 degrees). The two samples from the eye will be transmitted to the human's brain. The human brain, being logical (most of the time), will conclude that the wheel must be turning clockwise. This is a "true" interpretation of the signal (the two samples). Now, imagine that the wheel is turning a bit faster. In fact, in exactly 1/30th of a second, the wheel would go from the resting position (0 degrees) to a new position at (90 degrees), as shown below.



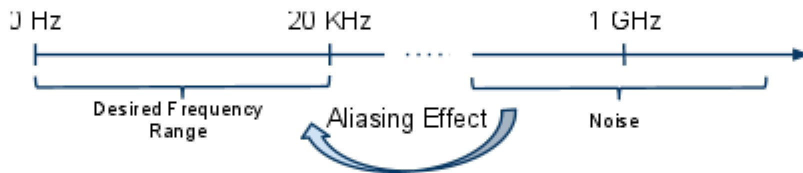
Our human subject will generate two samples 0 and 90 and transmit those to the brain. The brain, applying logic, will make the determination that the wheel is turning counter clockwise -- the most logical way to go from point A to point B, is the shortest distance. Clearly, our brain's interpretation of the data is false. What has occurred is aliasing. Our sampled data at 0 and 90 contains misinformation because of the slow sampling rate of the eye. The only way to solve this problem is to reengineer the eye to take more frequent samples. You must have experienced something like our example many times when staring at a turning wheel. This example should also give you some insight into the sampling theorem and why one must sample at 2x the highest frequency.

Try: What would the human subject conclude if the wheel was turning such that in exactly 1/30th of a second, it completed a single revolution?
Try: What would the human subject conclude if the wheel was turning such that in exactly 1/30th of a second, it completed one revolution + an additional 46 degrees?

It seems like we can avoid aliasing if we sample fast enough (i.e., 2x the highest frequency). However, recall that we are trying to avoid unwanted noise in the high frequency range. We had reasoned that frequencies that are beyond those of interest can be ignored. However, due to aliasing, this noise (e.g, the wheel turning really fast) would present itself as a valid signal (e.g., the wheel perceived to be turning really slow). This is illustrated in the figure below.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu



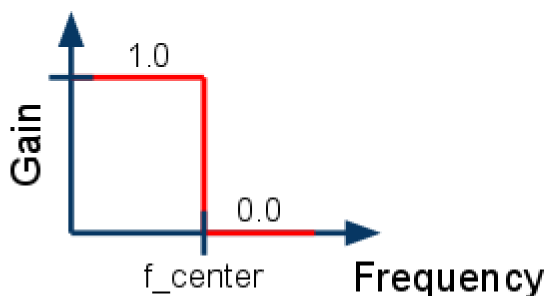
In our running example, we are sampling fast enough to capture all the valid sound information. However, due to aliasing of the noise from the microwave owen (in the GHz range), we will be faced with an unwanted signal that is in the audible range. Our recording, for instance, would produce an ugly sound that is directly from the microwave own, even though the microwave own is not generating any sounds.

The solution, you might say, is to sample at a very high rate, say 10 GHz, so that we can capture the high frequency data from the microwave own and digitally eliminate it (a topic that we'll discuss shortly). This is a valid solution. In fact, one way to avoid aliasing is by over sampling. However, it comes at a big cost. The higher the sampling rate, the more expensive it becomes to transmit, store, and process signals.

Try: Assuming a CD can hold 700 MB of data, at 16 bits per sample, 2 channels, and 44.1 KHz sampling rate, how many minutes of music can we store?
Try: Same assumptions as above, but sampling at 10 GHz, how many minutes of music can we store?

Low Pass Filter

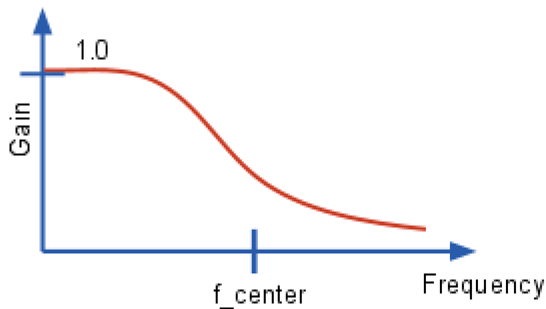
A better solution is to eliminate the higher frequencies from our signal path using more dedicated analog components. This can be accomplished using a ***low pass filter***. A low pass filter will eliminate the higher (unwanted) frequencies prior to the conversion process, eliminating the problem of aliasing. In fact, a low pass filter is sometimes referred to as an ***anti-aliasing filter***. Low pass filters are design to let all frequencies below a specific value, called the ***center frequency***, through while eliminating all frequencies above the center frequency. Conceptually, a perfect low pass filter would be like an amplifier that has 2 gain settings, gain=1.0 or gain=0.0, determined by the input frequency.



In reality, it is impossible to design such a perfect low pass filter. A more realistic design might have the following characteristics.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

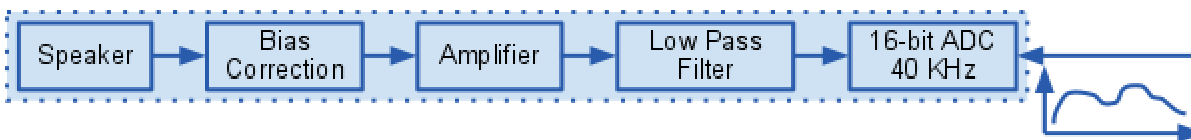


Let us revise our DSP example to include a low pass filter.

Our example is now complete. We have managed to convert the raw sound, picked up by the microphone (sensor), and convert it to a stream of 16-bit samples at 40KHz. Along the way, we have addressed a number of problems, namely, under-loading, overloading, bias correction, anti-aliasing, quantization, and sampling. In essence, we have composed a single component made of a number of sub components. In fact, we can view our example, thus far, as a digital microphone, i.e., a sensor that outputs a digital signal. Most sensors that have a digital output are designed in a similar manner.

Playback Path

We can consider what we have studied so far as being along the receiving path. In other words, we have been concerned with the signal traveling from the source (sensor) to the point where it is digitized. We can, use the same design principles in reverse to build the playback path, or the path from the digitized stream to an actuator. This is illustrated in the following figure.



Most of the components should be familiar and their function clear to you. The speaker is the actuator. The amplifier must have a gain that is sufficiently high to ***drive***, or operate, the speaker. To determine the gain, one must study the electrical requirements of the speaker. The bias correction is placed to eliminate any unwanted bias that has been introduced along the return path. The low pass filter is placed ahead of the amplifier so that unecessary noise, that has entered the system, is eliminated prior to amplification.

Digital to Analog Converter

Another important transform component is the ***digital to analog converter*** (DAC). Similar to an ADC, a DAC is a mixed-signal transform with a digital input and an analog output. A DAC converts the digital value on its input to a proportional analog value on its output. DACs have the following key characteristics:

Copyright © Frank Vahid and Tony Givargis 2011

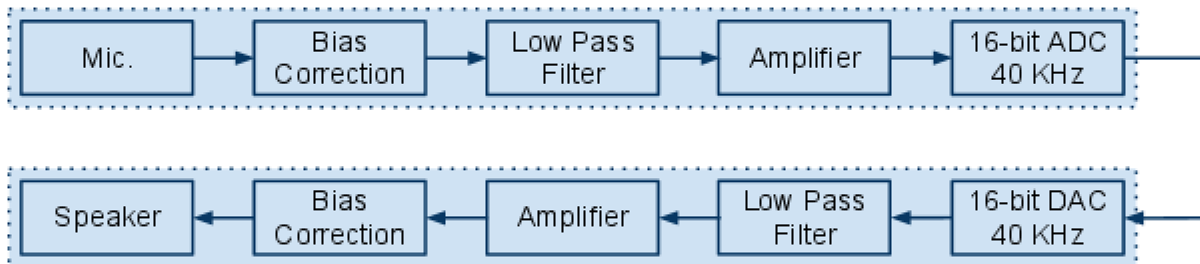
Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

- Input ***range***, or quantization, measured in bits, is the number of bits in each input sample (typical values are 8, 12, 16, 24, and 32).
- Output range, measured in Volts, is the range of voltages that are generated at the output of the DAC.
- ***Sampling rate***, measured in Hz, is the rate at which the DAC consumes its input samples.

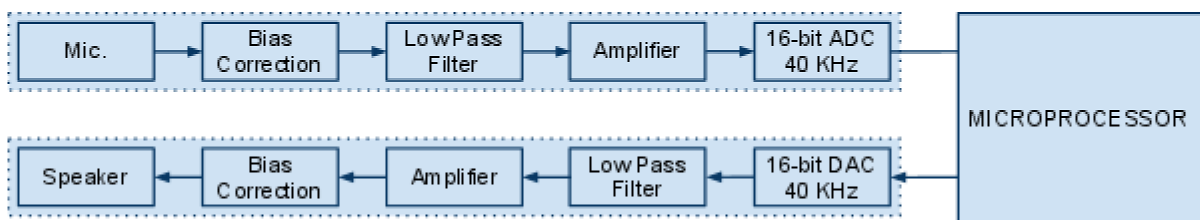
As you can see, a DAC is simply an ADC in reverse. It has the same specifications and, in most DSP applications, the DAC parameters must be matched to those of the ADC parameters. Specifically, DAC parameters must be selected according to the familiar concepts of dynamic range, sampling theorem, and anti aliasing requirements.

A Complete Example

We can construct a complete DSP system by combining our receive and playback components, as shown below.



Any sound picked up by the microphone will playback through the speaker. Along the way, the signal will be digitized. Our design will ensure that the signal quality is high, noise is low, and all sound frequencies of interest are present in the speaker's output. This is already a significant design achievement. However, the system is still limited. For example, we are unable to do even the basic task of controlling the sound volume. To do so, we need to introduce a final transform component, namely the microprocessor. A microprocessor will allow us to process the digitized signal in a number of ways. For instance, we can store it and play back later. We can stream it over the Internet to our friends. We can compress it, and so on. But for now, we'll be looking at the microprocessor to do activities that manipulate the signal itself, i.e., perform digital signal processing. The new architecture might look as follows.



The advantage of using a microprocessor is that we can implement concepts using software. Unlike the previous components, which were analog or mixed-signal in nature, our microprocessor is a familiar device. In many DSP designs, the choice of this microprocessor is very critical. The microprocessor must perform all necessary computations in real time. Often times, engineers select specially designed microprocessors, called ***digital signal processors*** (DSPs) to ensure that all processing can be done quickly and with little power consumption. However, without loss

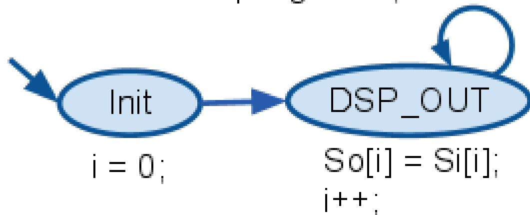
Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

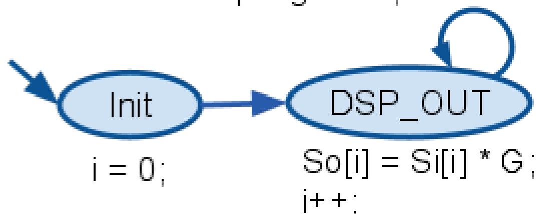
of generality, we'll maintain a programming perspective that is independent from the choice of microprocessor.

Digital Processing

At this stage, our microprocessor receives a stream of digital samples, Si[0], Si[1], Si[2], ..., and so on. If playback in real-time is desired, the microprocessor generates a stream of corresponding digital samples So[0], So[1], So[2], ..., and so on. The simplest transform is one that sends the input to the output, verbatim.

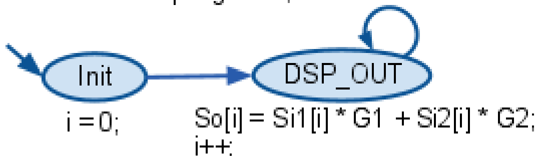


Let us make things more interesting by introducing a transform called ***scaling***.



G is the scaling factor. The output can now be attenuated or amplified. For example, this transform will allow us to add volume control to our sound system. We would have a slider in our GUI that allows the user to set G to a range of 0.0 to 1.0. This is what happens when you change the volume control on your iPod.

Another transform might be one that ***mixes*** the two signals Si1 and Si2.

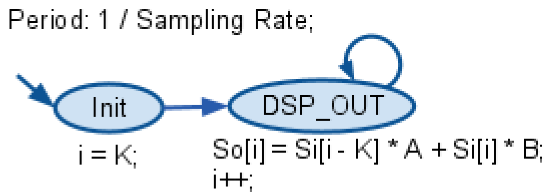


For example, we might add a second sound signal to our example and use the microprocessor to perform digital mixing of the two audio streams. The two gains G1 and G2 will allow us to determine how loud one sound source is relative to the other. A sound mixing equipment that is used in music studios has many sound channels and associated sliders (i.e., gain), but its core processing functionality is similar to the mixing example above.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

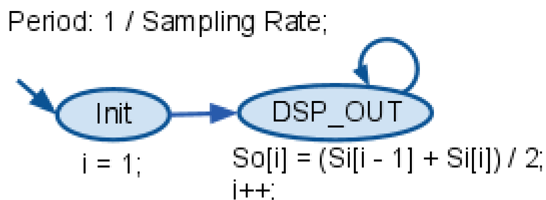
Another transform might be one that adds an ***echo*** to the stream.



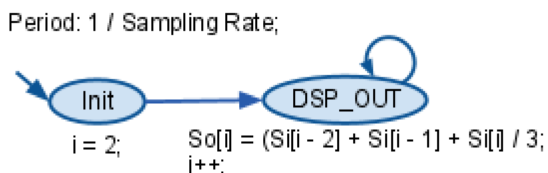
Here, K, A, and B are scalar constants. The idea is that the sound being generated *now* is a mix of sound from the past (S at K samples ago) and the sound from present (S_i). One would expect that A < B. In sound, for example, an echo is sound that reflects off of far away objects. This sound is delayed (molded by K) and is weaker relative to the sound from present. Of course, one can experiments with all sorts of settings for K, A, and B to obtain interesting sound manipulations. Also, one can apply this transform to any kind of signal, not just sound. For instance, applied to video, an echo would appear as a shadow effect.

In the above examples, we have assumed that the input and output samples are stored in arrays that appear to be unbounded. Clearly, this scheme is not practically. No processor has infinite memory. Depending on the application, these arrays may be implemented as ***bounded buffers***. In the case of bounded buffers, the arrays have to be large enough to accommodate the needs of the application. For instance, in the echo example, the buffers can be bounded to K samples long.

Our last transformation is a ***digital low pass filter***.



The above transform generates an output sample by taking the average of two consecutive input samples. In essence, we are performing signal smoothing, or eliminating sudden fluctuations (i.e., higher frequencies) in the input stream. We can control the filtering effect by increasing the number of samples that are averaged. For instance, the following example is a low pass filter with a slightly lower center frequency.



Taking the example to an extreme, we can design software that allows the user to move a slider, which in turn will increase/decrease the averaging window size. Such a slider would allow the user to adjust the low pass filter setting. In sound, this is the function performed by an ***audio equalizer***. An audio equalizer has many sliders that allow us to adjust the level of a number of different frequencies for very fine tuning of the sound quality. To see how such a device might be implemented

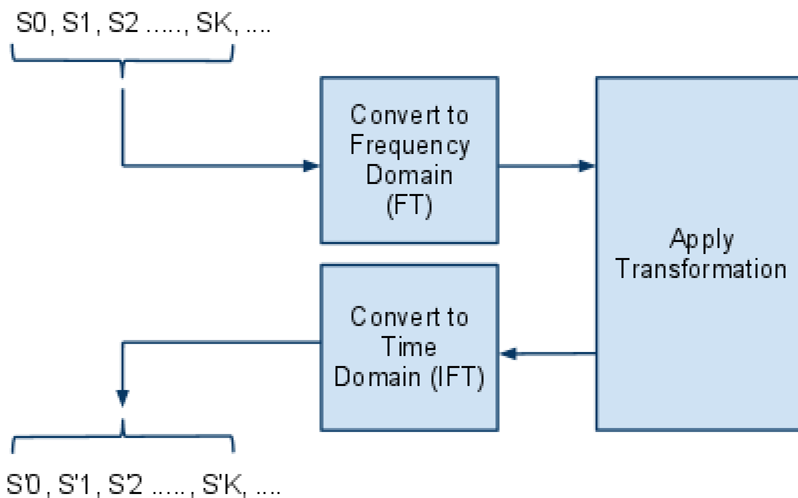
Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

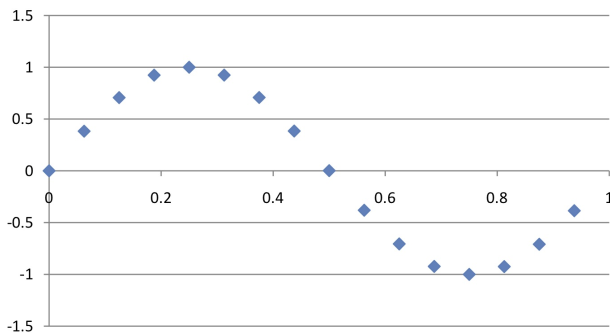
we'll need to make a distinction between ***time-domain*** transformations vs. ***frequency-domain*** transformations.

This is a good time to remind ourselves that sound is not the only signal. All the concepts we have discussed thus far, including the transformations, are equally applicable to signals of different nature. We like to use sound because it is familiar and can be detected by our ear.

Time-domain transformations are transformations, such as those just introduced, where the processing is done along the time axis and directly on the samples as they arrive. In contrast, frequency-domain transformations are those that first convert a K-sample long stretch of the input to a frequency representation, perform the transform on this frequency representation, and regenerate, from the frequency representation, a K-sample long stretch of output. In essence, frequency-domain transformations follow the following schematic.



The french mathematician Joseph Fourier discovered that certain signals (such as those we've been looking at) can be decomposed into sine and cosine functions and, if needed, these sine and cosine functions can be algebraically summed to obtain the original signal. These transformations, known as the ***Fourier transform*** (FT) and ***inverse Fourier transform*** (IFT), are the cornerstone of most signal processing activities. Before becoming consumed by the details, let us gain some insight into how these transforms work. Consider the following input signal, it happens to be a sine function with frequency of 1 Hz.

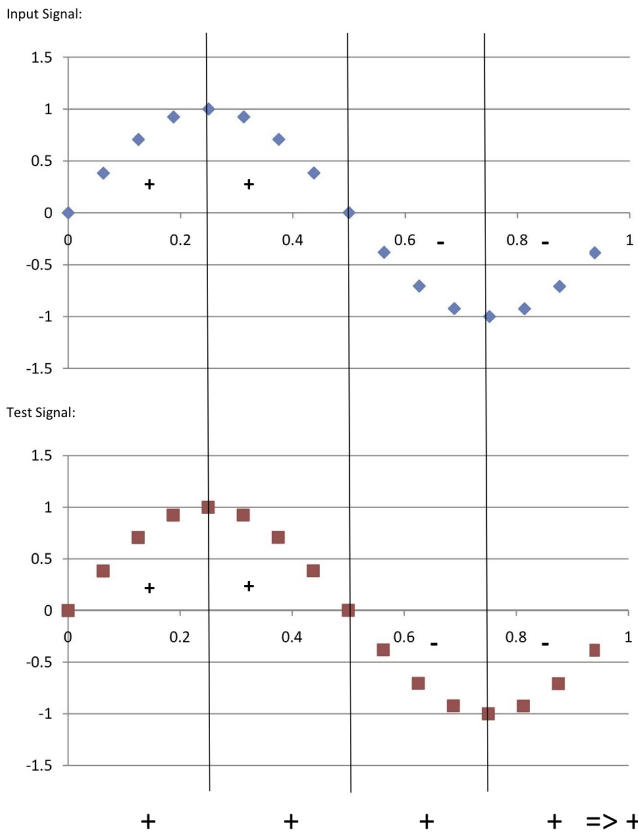


The FT process is one of a guessing game. We guess that some frequency, say 1 Hz, is present in the input signal. Then, we do some math, and the result will validate our guess. The math is a sample by sample multiplication of the input by the guess. These products are then summed and the final

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

sum is a measure of correctness of our guess. In other words, if our guess is correct, the sum will be maximum. If our guess is wrong, the sum will be 0. The following figure demonstrates this process.



Rather than do all the math, we can observe that the first 8 samples are positive (+) in our input signal as well as the test signal. Therefore, we can conclude that the running sum, thus far, will be some positive (nonzero) number. The remaining 8 samples are negative (-) in our input signal as well as the test signal. The product of two negative values yields a positive value. Thus, the running sum of the second half will also be some positive (nonzero) number. In fact, the total sum of our multiply-accumulate exercise will be a positive number, hinting that the input signal has a component of 1 Hz in it.

Try: Use anExcel spreadsheet to generate the two sine functions (1 Hz and 1 Hz) and then program your Excel spreadsheet to perform the addition. What is the sum? Hint, to generate a sine function, you can use the following excel strategy:

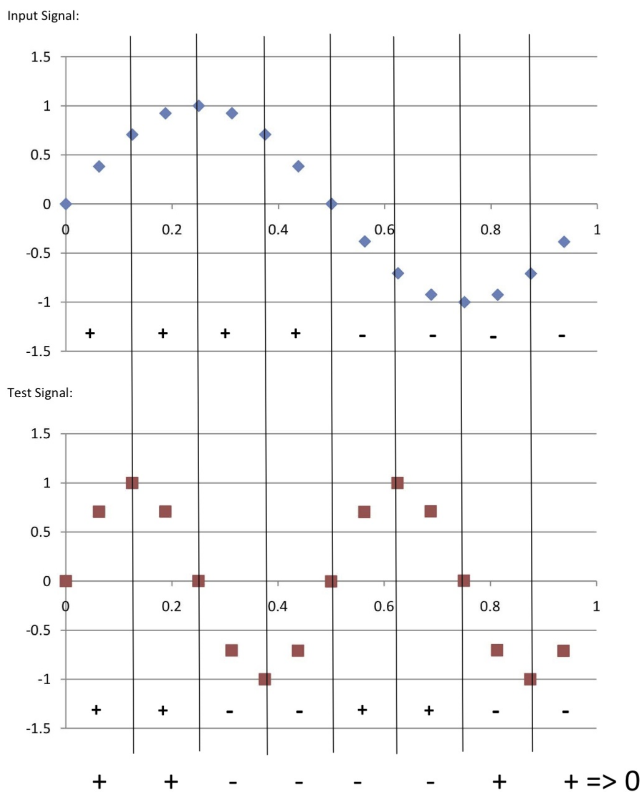
Sample	Time (x axis)	Signal (y axis)
0	=0/16	=sin(2*3.14*Frequency*Time)=sin(0)
1	=1/16	=sin(2*3.14*Frequency*Time)=sin(6.28/16)

...

We can continue our test by checking the input signal against a different frequency. Let us now try a test signal of 2 Hz.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu



The result, in this case, will be 0. Can you see why? Hint, use the +/- annotations.

Try: Use an Excel spreadsheet to generate the two sine functions (1 Hz and 2 Hz) and then program your Excel spreadsheet to perform the addition. What is the sum?

Of course, we are far from decomposing an arbitrary signal to its frequency domain representation. For starters, you might be wondering just how many frequencies to test? The answer is simple when dealing with digitized signals. The number of frequencies to test should be equal to the sampling rate of the DSP system multiplied by the window of time being converted. The frequencies should be equally spread between the lowest and the highest frequencies know to be present in the input signal. For instance, in our sound example, we know that the input frequency is in the range of 0-20 KHz. We also know that the sampling rate is 40 KHz. Assuming that our system continuously converts a 0.1 second window of sound to frequency domain, does some processing, and then outputs the result, we would need to test 0.1 seconds * 40,000 samples/second = 4000 frequencies in the range of 0 to 20,000 Hz. As you can see, there are a number of tradeoffs here. If we like our frequency domain conversion to be very detailed, i.e, have the ability to decompose the input into many frequencies, we need to processes a larger window of samples at a time. This would add some latency in our signal flow. The decision, as usual is very application dependent.

A second issue that might come up is that, if the input signal is slightly shifted (for instance, it has a ***phase shift*** of 45 degrees), our test fails to work. The solution is to simultaneously test the input signal against two test signals of 1 Hz, one being a sine and the other a cosine function. As you know sine and cosine are phase shifted by 90 degrees, but otherwise identical. If the test frequency matches that of the input frequency, one or the other of the sine or cosine tests will yield a big positive number. Therefore, when combined, the two tests can tell us if (and to what degree) the test signal is present in the input *sqrt(sun_sin^2 + sun_cos^2)*, and by how much it is phase shifted *atan(sum_sin / sum_cos)*.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

Try: Use an Excel spreadsheet to generate an input signal with a particular frequency and phase. Then, generate the sine and cosine functions test frequencies. Perform the transform computations and try to determine the phase shift of the input signal according to the above.

As you see, the conversion to frequency domain would require a large number of computations, in particular multiplication and addition. Many digital signal processors have special ***multiply-accumulate*** instructions to perform these computations a little more efficiently. Computer scientists have studied the FT procedure from an algorithmic point of view and have developed a version that is highly optimized, the so called ***fast Fourier transform*** (FFT). The inverse FT is a bit simpler to implement. To recreate the signal in time domain, one needs to generate each and every one of the individual sine functions (with the appropriate amplitude and phase) and algebraically add these together.

We now return to our sound example. Once our sound stream is converted to a set of frequencies (amplitude/phase pairs), we can manipulate the sound by increasing or decreasing a particular frequency as desired.

Final Remarks

As you can see, digital signal processing is a very interesting field of study. It covers issues from a number of domains, such as mathematics, physics, electrical engineering, computer science, logic design, computer architecture, embedded systems and real time software engineering. Our intent so far has been to introduce a number of concepts that are key. We have, for the most part, left the implementation details out of the discussion. Interested students are encouraged to seek a dedicated text book or course on the topic of digital signal processing.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

Book and Author Info

This book, Programming Embedded Systems (PES), introduces a disciplined approach to programming embedded systems. The approach involves definition of a computation model (synchronous state machines, or synchSMs) appropriate for capturing behavior of time-oriented systems, with time-orientation being perhaps the most unique and important distinguishing feature of embedded systems. SynchSMs can be easily converted to structured and maintainable C code running on a microcontroller. PES shows how to capture behavior with multiple synchSMs, and convert those to a single C program. It shows how to write a simple task scheduler, thus providing a solid understanding of a key part of what is "under-the-hood" of real-time operating systems, useful not only for programming without an RTOS, but also for making better use of an RTOS. PES also covers various important embedded programming issues, such as bit-level manipulation in C, coding issues like rounding and fixed-point programming, basic control systems, and basic digital signal processing systems. PES also shows how to convert synchSMs for implementation on FPGAs, which is also straightforward.

PES is intended to elevate embedded systems programming to a discipline. Current university courses commonly introduce details of a microcontroller and its peripherals, with little guidance on how to write programs, leading to a huge variety of ad hoc programming styles that lead to programs that suffer from timing problems, that are hard to maintain, and that do not scale. In contrast, PES introduces concepts that are independent of any particular microcontroller, where those concepts lead to highly-structured, readable, maintainable, scalable, and analyzable code.

PES represents a modern approach to creating learning content. Its conciseness enables a complete read of key concepts, with reference information today available online. PES is intended for electronic publication, being read on various devices, or being printed by the end user without excessive paper use. As such, PES passes on highly polished formatting and figures, in favor of content that is viewable on various devices, and is amenable to update and revision on a faster cycle than traditional book "editions," leading to a book that is more modern, and has a far lower purchase price.

PES also uses a modern approach by stressing active learning, coming with tools that enable a reader to put learned concepts into practice. The Riverside-Irvine Microcontroller Simulator (RIMS) includes complete C capture, compilation, simulation, and debug for microcontroller programming, all in a single extremely easy-to-use graphical interface. RIMS alone is sufficient to support the learning of embedded programming. In addition, the Riverside-Irvine Builder of State machines (RIBS) supports graphical capture of synchSMs and automatic conversion to C, along with visualization during RIMS simulation. The Riverside-Irvine Timing-diagram Solution (RITS) is useful for display and printing of the simulation output of RIMS. A course based on PES may have a lab that uses a particular microcontroller or embedded processor, but because setting up and maintaining such labs can be difficult, and because such labs are not available for online courses, PES and its tools can be used to learn a majority of the key concepts and skills necessary to write time-oriented embedded systems programs. PES is thus well-suited for traditional in-person course as well as for online courses. PES tools presently run on Microsoft Windows platforms.

Chapters 1-10 form the core of PES. They can typically be covered in 9-10 weeks of a university course meeting 3 hours per week. After that, a course may teach programming with an RTOS, programming of FPGAs, control systems, digital signal processing, microcontroller peripherals, or any of various other directions. PES can be used to revise the lecture content of existing microcontroller courses, while keeping labs intact (sometimes involving moving some lab training content from lecture into lab time).

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

PES and its tools are based upon work supported by the U.S. National Science Foundation's CCLI (Course Curriculum and Laboratory Improvement) program under grant number DUE-0836905. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Frank Vahid is a Professor of Computer Science and Engineering at the University of California, Riverside, and Associate Director of the Center for Embedded Systems at UC Irvine. He received his bachelor's degree in electrical engineering from the University of Illinois at Urbana-Champaign, and his master's and doctoral degrees in computer science from the University of California, Irvine. He has worked for Hewlett Packard and AMCC, and has consulted for Motorola, NEC, Atmel, Cardinal Health, and several other engineering firms. He is the inventor on three U.S. patents and has published over 150 research papers. He is author of "Digital Design, with RTL Design, VHDL, and Verilog" (J. Wiley and Sons, 2011, 2nd ed), books on VHDL and Verilog, and co-author of "Embedded Systems Design: A Unified Hardware/Software Introduction" (J. Wiley and Sons, 2001), and "Specification and Design of Embedded Systems" (Prentice-Hall, 1994 -- perhaps the first published book on embedded systems). He helped establish the Embedded Systems Week conference and has chaired the CODES and ISSS symposia. He established UCR's Computer Engineering program, and has received several UCR teaching awards. His current research focuses on embedded systems design and programming for medical devices and for home/office monitoring and control. http://www.cs.ucr.edu/~vahid.

Tony Givargis is a Professor of Computer Science at the University of California, Irvine, and a member of the Center for Embedded Computer Systems at UC Irvine. He received his B.S. and Ph.D. in Computer Science from the University of California, Riverside in 1997 and 2001 respectively. His research focuses on Realtime Operating System (RTOS) synthesis, high-confidence embedded software, serializing compilers, and code transformation techniques for efficient software to hardware migration. He is coauthor of a textbook entitled "Embedded System Design: A Unified Hardware/Software Introduction," has published over 70 peer reviewed articles in the general areas of embedded system research, and is co-inventor on 8 patents. His passion for teaching has earned him the UC Irvine Information & Computer Science Dean's Award for the Excellence in Undergraduate Teaching in 2010, the UC irvine Excellence in Teaching Award in 2003, and the UC Irvine Chancellor's Award for Excellence in Fostering Undergraduate Research, in 2005. http://www.ics.uci.edu/~givargis. He received the 2011 Terman award from ASEE for outstanding contributions to engineering education including authoring of a high-impact textbook.

Please see www.programmingembeddedsystems.com for further information, to obtain the most recent version of PES, or to obtain the RIMS/RIBS/RITS tools. Send comments, questions, and corrections to info@programmingembeddedsystems.com.

This book was created using Google Docs.

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu

Copyright © Frank Vahid and Tony Givargis 2011

Licensed to: Tony Givargis - 24 Murasaki Irvine, CA 92617 - 949-232-7909 - givargis@uci.edu