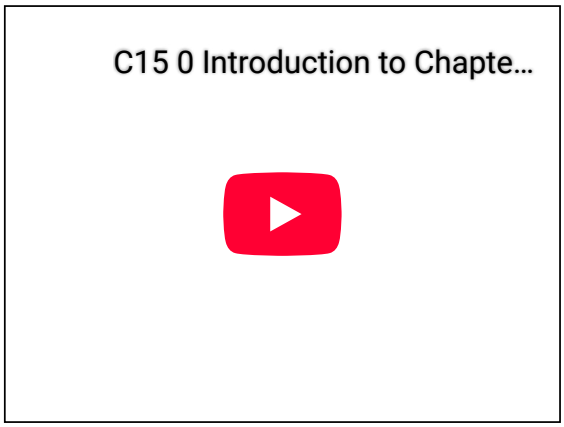
Chapter 9: Embedded System Design

Jonathan Valvano and Ramesh Yerraballi

So far in this course have presented embedded systems from an interfacing or component level. This chapter will introduce systems level design. The chapter begins with a discussion of requirements document and modular design. Next, we will describe data structures used to represent graphics images. We will conclude this course with a project of building a hand-held game.

Table of Contents:



Video 9.0. Introduction to Programming Games

9.1. Requirements Document

A **Requirements Document** states what the system will do. It does not state how the system will do it. The main purpose of a requirements document is to serve as an agreement between you and your clients describing what the system will do. This agreement can become a legally binding contract. We should write the document so that it is easy to read and understand by others. It should be unambiguous, complete, verifiable, and modifiable. In this chapter we will use the framework of the requirements document to describe the hand-held game project.

1. Overview

1.1. Objectives: Why are we doing this project? What is the purpose? The overall objective of this project is to integrate the individual components taught in this class into a single system. More specifically, the objectives of this project are: 1) design, test, and debug a large C program; 2) to review I/O interfacing techniques used in this class; and 3) to design a system that performs a useful task. In particular we will design an 80's-style shoot-em up game like **Space Invaders**.

1.2. Process: How will the project be developed? Similar to the labs, this lab has starter projects: `Lab10_EE319K` for EE319K and `Lab10_C++` for EE319H. The projects include some art and sounds to get you started.

1.3. Roles and Responsibilities: Who will do what? Who are the clients? Students may develop their games using their EE319K teams. The clients for this project will be other classmates and your EE319K professors.

1.4. Interactions with Existing Systems: How will it fit in? The game must be developed in C on the Keil IDE and run on a Tiva LaunchPad. We expect you to combine your solutions to Lab 3 (switches, LED), Lab 6 (interrupts, DAC and sounds), Lab 7 (LCD), Lab 8(slide pot and ADC) into one system. We expect everyone to use the slide pot, two switches, two LEDs, one DAC, and the ST7735R or Nokia5110 LCD screen.

1.5. Terminology: Define terms used in the document. **BMP** is a simple file format to store graphical images. A **sprite** is a virtual entity that is created, moves around the screen, and might disappear. A **public** function is one that can be called by another module. For example if the main program calls **`Sound_Play`**, then **`Sound_Play`** is a public function.

1.6. Security: How will intellectual property be managed? It is scholastic dishonesty to share any Lab 9 software with another student other than your lab partner. TAs will run Lab 9 software through a plagarism checker called **moss**.

2. Function Description

2.1. Functionality: What will the system do precisely? You will design, implement and debug an 80's or 90's-style video game. You are free to simplify the rules but your game should be recognizable and fun. The LCD, LEDs, and sound are the outputs. The slide pot is a simple yet effective means to move your ship. Interrupts must be appropriately used control the input/output, and will make a profound impact on how the user interacts with the game. You could use an edge-triggered interrupt to execute software whenever a button is pressed. You could create two periodic interrupts. Use one fixed-frequency periodic interrupt to output sounds with the DAC. You could decide to move a sprite using a second periodic interrupt, although the actual LCD output should always be performed in the main program.

2.2. Scope: List the phases and what will be delivered in each phase. The first phase is forming a team and defining the exact rules of game play. You next will specify the modules: e.g., the main game engine, a module to input from switches, a module to output to LEDs, a module to draw images on the LCD, and a module that inputs from the slide pot. Next you will design the prototypes for the public functions. At this phase of the project, individual team members can develop and test modules concurrently. The last phase of the project is to combine the modules to create the overall system.

2.3. Prototypes: How will intermediate progress be demonstrated? In a system such as this each module must be individually tested. Your system will have four or more modules. Each module has a separate header and code file. For each module create a header file, a code file and a separate main program to test that particular module.

2.4. Performance: Define the measures and describe how they will be determined. The game should be easy to learn, and fun to play.

2.5. Usability: Describe the interfaces. Be quantitative if possible. The usability of your game will be outlined in your proposal.

2.6. Safety: Explain any safety requirements and how they will be measured. The usual rules about respect and tolerance as defined for the forums apply as well to the output of the video games.

3. Deliverables

3.1. Reports: How will the system be described? Add comments to the top of your C file to explain the purpose and functionality of your game.

3.2. Audits: How will the clients evaluate progress? There will be a discussion forum that will allow you to evaluate the performance (easy to learn, fun to play) of the other games.

3.3. Outcomes: What are the deliverables? How do we know when it is done? You will commit your software to Canvas and demonstrate it to the TA.

Video 9.1.1. Overview of requirements

9.2. Modular Design

The design process involves the conversion of a problem statement into hardware and software components. Successive refinement is the transformation from the general to the specific. In this section, we introduce the concept of modular programming and demonstrate that it is an effective way to organize our software projects. There are four reasons for forming modules. First, functional abstraction allows us to reuse a software module from multiple locations. Second, complexity abstraction allows us to divide a highly complex system into smaller less complicated components. The third reason is portability. If we create modules for the I/O devices, then we can isolate the rest of the system from the hardware details. This approach is sometimes called a hardware abstraction layer. Since all the software components that access an I/O port are grouped together, it will be easier to redesign the embedded system on a machine with different I/O ports. Finally, another reason for forming modules is security. Modular systems by design hide the inner workings from other modules and provide a strict set of mechanisms to access data and I/O ports. Hiding details and restricting access generates a more secure system.

Software must deal with **complexity**. Most real systems have many components, which interact in a complex manner. The size and interactions will make it difficult to conceptualize, abstract, visualize, and document. In this chapter we will present data flow graphs and call graphs as tools to describe interactions between components. Software must deal with **conformity**. All design, including software design, must interface with existing systems and with systems yet to be designed. Interfacing with existing systems creates an additional complexity. Software must deal with **changeability**. Most of the design effort involves change. Creating systems that are easy to change will help manage the rapid growth occurring in the computer industry.

The basic goals of modular design is to maximize the number of modules and minimize the interdependence. There are many ways modules interact with each other. Three of the ways modules interact are

- **Invocation coupling**: one module calls another module,
- **Bandwidth coupling**: one module sends data to another,
- **Control coupling**: shared globals in one module affects behavior in another

We specify invocation coupling when we draw the **call graph**. We specify bandwidth coupling when we draw the **data flow graph**. Control coupling occurs when we have shared global variables, and should be avoided. However, I/O registers are essentially shared global objects. So one module writing to an I/O register may affect behavior in another module using the sample I/O. Minimizing control coupling was the motivation behind writing friendly code.

A software module has three files:

- **Header file**: comments that explain what the module does, prototypes for public functions, shared #define, shared structures, shared typedef, shared enum
- **Code file**: comments that explain how the module works, implementation of all functions, private variables (statics), helper functions, comments to explain how to change the module
- **Test main file**: comments that explain how the module is tested, test cases, examples of module usage

The key to completing any complex task is to break it down into manageable subtasks. Modular programming is a style of software development that divides the software problem into distinct well-defined modules. The parts are as small as possible, yet relatively independent. Complex systems designed in a modular fashion are easier to debug because each module can be tested separately. Industry experts estimate that 50 to 90% of software development cost is spent in maintenance. All five aspects of software maintenance

- Correcting mistakes,
- Adding new features,
- Optimizing for execution speed or program size,
- Porting to new computers or operating systems, and
- Reconfiguring the software to solve a similar related program

are simplified by organizing the software system into modules. The approach is particularly useful when a task is large enough to require several programmers.

A **program module** is a self-contained software task with clear entry and exit points. There is a distinct difference between a module and a C language function. A module is usually a collection of functions that in its entirety performs a well-defined set of tasks. A collection of 32-bit trigonometry functions is an example of a module. A device driver is a software module that facilitates the use of I/O. In particular it is collection of software functions for a particular I/O device. Modular programming involves both the specification of the individual modules and the connection scheme whereby the modules are interfaced together to form the

software system. While the module may be called from many locations throughout the software, there should be well-defined **entry points**. In C, the entry point of a module is defined in the header file and is specified by a list of function prototypes for the public functions.

> **Common Error:** In many situations the input parameters have a restricted range. It would be inefficient for the module and the calling routine to both check for valid input. On the other hand, an error may occur if neither checks for valid input.

An **exit point** is the ending point of a program module. The exit point of a function is used to return to the calling routine. We need to be careful about exit points. Similarly, if the function returns parameters, then all exit points should return parameters in an acceptable format. If the main program has an exit point it either stops the program or returns to the debugger. In most embedded systems, the main program does not exit.

In this section, an object refers to either a function or a data element. A **public** object is one that is shared by multiple modules. This means a public object can be accessed by other modules. Typically, we make the most general functions of a module public, so the functions can be called from other modules. For a module performing I/O, typical public functions include initialization, input, and output. A **private** object is one that is not shared. I.e., a private object can be accessed by only one module. Typically, we make the internal workings of a module private, so we hide how a private function works from user of the module. In an object-oriented language like C++ or Java, the programmer clearly defines a function or data object as public or private. The software in this course uses the naming convention of using the module name followed by an underline to identify the public functions of a module. For example if the module is ADC, then **`ADC_Init`** and **`ADC_Input`** are public functions. Functions without the underline in its name are private. In this manner we can easily identify whether a function or data object as public or private.

At a first glance, I/O devices seem to be public. For example, the I/O register **GPIOB_DIN31_0** resides permanently at the fixed address of 0x400A3380, and the programmer of every module knows that. In other words, from a syntactic viewpoint, any module has access to any I/O device. However, in order to reduce the complexity of the system, we will restrict the number of modules that actually do access the I/O device. From a "what do we actually do" perspective, however, we will write software that considers I/O devices as private, meaning an *I/O device should be accessed by only one module*. In general, it will be important to clarify which modules have access to I/O devices and when they are allowed to access them. When more than one module accesses an I/O device, then it is important to develop ways to arbitrate or synchronize. If two or more want to access the device simultaneously arbitration determines which module goes first. Sometimes the order of access matters, so we use synchronization to force a second module to wait until the first module is finished. Most microcontrollers do not have architectural features that restrict access to I/O ports, because it is assumed that all software burned into its ROM was designed for a common goal, meaning from a security standpoint one can assume there are no malicious components. However, as embedded systems become connected to the Internet, providing the power and flexibility, security will become important issue.

> Checkpoint 9.2.1 **:** Multiple modules may use Port A, where each module has an initialization. What conflict could arise around the initialization of a port?

Information hiding is similar to minimizing coupling. It is better to separate the mechanisms of software from its policies. We should separate "what the function does" from "how the function works". What a function does is defined by the relationship between its inputs and outputs. It is good to hide certain inner workings of a module and simply interface with the other modules through the well-defined input/output parameters. For example we could implement a variable size buffer by maintaining the current byte count in a global variable, **`Count`**. A good module will hide how **`Count`** is implemented from its users. If the user wants to know how many bytes are in the buffer, it calls a function that returns the count. A badly written module will not hide **`Count`** from its users. The user simply accesses the global variable **`Count`**. If we update the buffer routines, making them faster or better, we might have to update all the programs that access **`Count`** too. Allowing all software to access **`Count`** creates a security risk, making the system vulnerable to malicious or incompetent software. The object-oriented programming environments provide well-defined mechanisms to support information hiding. This separation of policies from mechanisms is discussed further in the section on layered software.

> **Maintenance Tip:** It is good practice to make all permanently-allocated data and all I/O devices private. Information is transferred from one module to another through well-defined function calls.

The **Keep It Simple Stupid** approach tries to generalize the problem so that the solution uses an abstract model. Unfortunately, the person who defines the software specifications may not understand the implications and alternatives. As a software developer, we always ask ourselves these questions:

> "How important is this feature?"
> "What if it worked this different way?"

Sometimes we can restate the problem to allow for a simpler and possibly more powerful solution. We begin the design of the game by listing possible modules for our system.

ADC	The interface to the joystick

Switch	User interaction with LEDs and switches
Sound	Sound output using the DAC
ST7735	Images displayed on the LCD
Game engine	The central controller that implements the game

Figure 9.2.1 shows a possible **call graph** for the game. An arrow in a call graph means software in one module can call functions in another module. This is a very simple organization with one master module and four slave modules. Notice the slave modules do not call each other. This configuration is an example of good modularization because there are 5 modules but only 4 arrows.

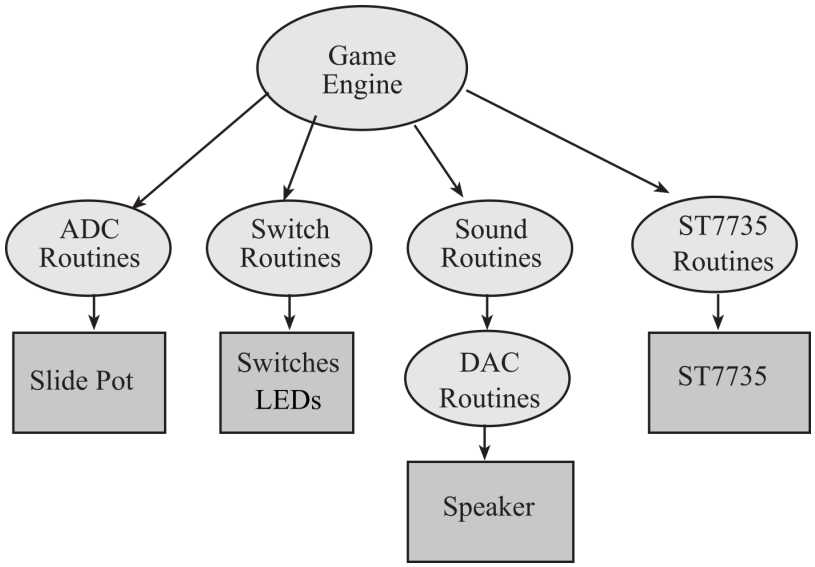


Figure 9.2.1. Possible call graph for the game.

Figure 9.2.2 shows on possible data flow graph for the game. Recall that arrows in a data flow graph represent data passing from one module to another. Notice the high bandwidth communication occurs between the sound module and its hardware, and between the LCD module and its hardware. We will design the system such that software modules do not need to pass a lot of data to other software modules.

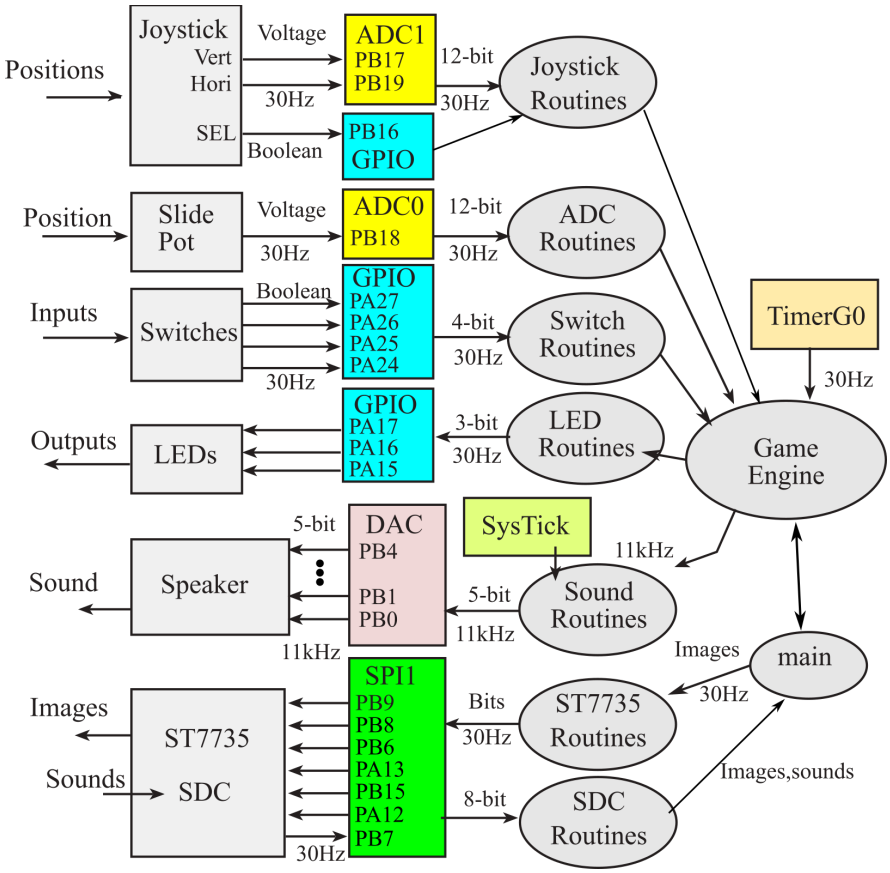


Figure 9.2.2. Possible data flow graph for the game. You can add LEDs if you wish.

The SysTick ISR will output a sequence of numbers to the DAC to create sound. Let **`explosion`** be an array of 2000 5-bit numbers, representing a sound sampled at 11 kHz. If the game engine wishes to make the explosion sound, it calls **`Sound_Play(explosion,2000);`** This function call simply passes a pointer to the **`explosion`** sound array into the sound module. The SysTick ISR will output one 5-bit number to the DAC for the next 2000 interrupts. Notice the data flow from the game engine to the sound module is only two parameters (pointer and count), causing 2000 5-bit numbers to flow from the sound module to the DAC.

The LCD module needs to send images to the LCD. The screen should be updated 30 times/sec so changes in the image looks smooth to the eye.

Figure 9.2.3 shows on possible flow chart for the game engine. It is important to perform the actual LCD output in the foreground. In this design there are three threads: the main program and two interrupts. Multithreading allows the processor to execute multiple tasks. The main loop performs the game engine and updates the image on the screen. At 30 Hz, which is fast enough to look continuous, the TimerG0 ISR will sample the ADC and switch inputs. Based on user input and the game function, the ISR will decide what actions to take and signal the main program. To play a sound, we send the Sound module an array of data and arm SysTick. Each SysTick interrupt outputs one value to the DAC. When the sound is over we disarm SysTick.

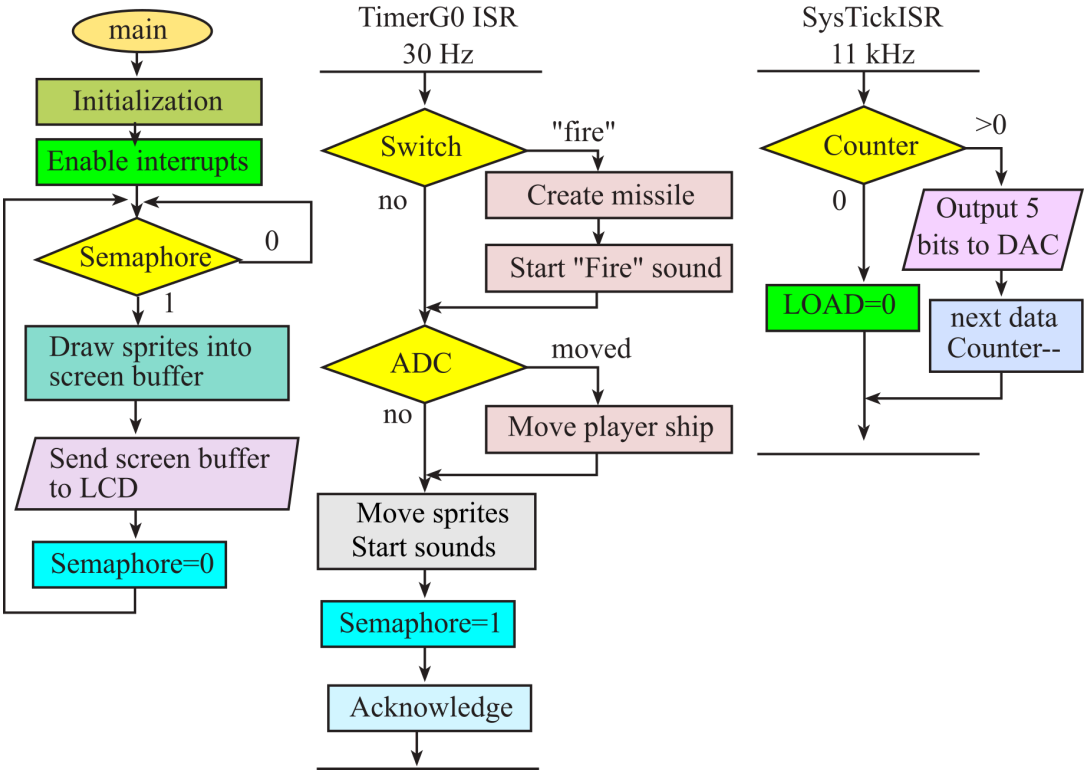
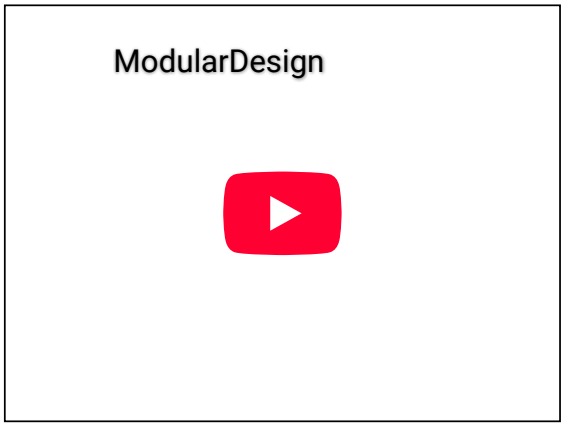


Figure 9.2.3. Possible flowchart for the game.

For example, if the ADC notices a motion to the left, the TimerG0 ISR can tell the main program to move the player ship to the left. Similarly, if the TimerG0 ISR notices the fire button has been pushed, it can create a missile object, and for the next 100 or so interrupts the TimerG0 ISR will move the missile until it goes off screen or hits something. In this way the missile moves a pixel or two every 33.3ms, causing its motion to look continuous. In summary, the ISR responds to input and time, but the main loop performs the actual output to the LCD.

Checkpoint 9.2.2 : Notice the algorithm in Figure 9.2.3 samples the ADC and the fire button at 30 Hz. How times/sec can we fire a missile or wiggle the slide pot? Hint: think Nyquist Theorem.

Checkpoint 9.2.3 : Similarly, in Figure 9.2.3, what frequency components are in the sound output?



Video 9.2.1. Modular design

9.3. Introduction to Graphics

A **matrix** is a two-dimensional data structure accessed by row and column. Each element of a matrix is the same type and precision. In C, we create matrices using two sets of brackets. Figure 9.3.1 shows this byte matrix with six 8-bit elements. The figure also shows two possible ways to map the two-dimensional data structure into the linear address space of memory.

```
unsigned char M[2][3]; // byte matrix with 2 rows and 3 columns
```

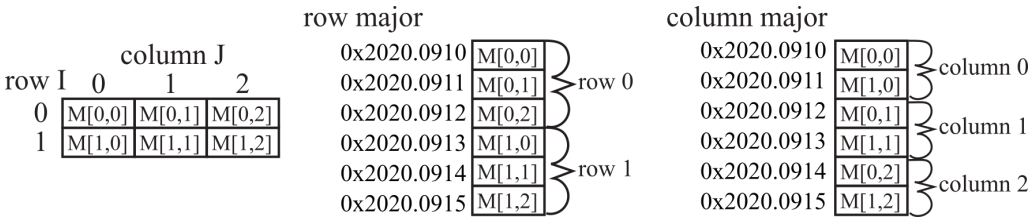


Figure 9.3.1. A byte matrix with 2 rows and 3 columns.

With row-major allocation, the elements of each row are stored together. Let **i** be the row index, **j** be the column index, **n** be the number of bytes in each row (equal to the number of columns), and **Base** is the base address of the byte matrix, then the address of the element at **i,j** is

```
Base+n*i+j
```

With a halfword matrix, each element requires two bytes of storage. Let **i** be the row index, **j** be the column index, **n** be the number of halfwords in each row (equal to the number of columns), and **Base** is the base address of the word matrix, then the address of the element at **i,j** is

```
Base+2*(n*i+j)
```

With a word matrix, each element requires four bytes of storage. Let **i** be the row index, **j** be the column index, **n** be the number of words in each row (equal to the number of columns), and **Base** is the base address of the word matrix, then the address of the element at **i,j** is

```
Base+4*(n*i+j)
```

In the game industry an entity that moves around the screen is called a *sprite*. You will find lots of sprites in the Lab9 starter project. You can create additional sprites as needed using a drawing program like Paint. Many students will be able to complete the project using only the existing sprites in the starter package. Having a 2-pixel black border on the left and right of the image will simplify moving the sprite 2 pixels to the left and right without needing to erase it. Similarly having a 1-pixel black border on the top and bottom of the image will simplify moving the sprite 1 pixel up or down without needing to erase it. You can create your own sprites using Paint by saving the images as 24-bit BMP images. Figure 9.3.2 is an example BMP image. Because of the black border, this image can be moved left/right 2 pixels, or up/down 1 pixel. Within this project there is a Windows application called **BmpConvert16.exe** that will convert the bmp file to a text file, which can be copy-pasted into your C project. The directions on how to use this converter can be found in the file **BmpConvert16Readme.txt**. To build an interactive game, you will need to write programs for drawing and animating your sprites.

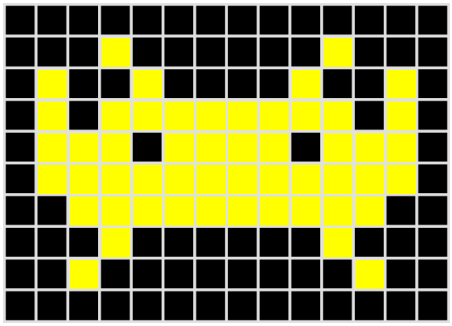


Figure 9.3.2. Example image, 16-bit color, 16 pixels wide by 10 pixels high.

Let's begin with an overview of the two graphics functions needed to implement the game. The first function is **ST7735_FillScreen**, which will set the entire screen to a single color. The videos implement a black background, but any solid color would suffice. The second function is **ST7735_DrawBitmap**, which will draw a BMP image on the screen. Its prototype is

Maintenance Tip: In order to eliminate flicker, we will execute **ST7735_FillScreen** once at the beginning.

```
//------------ST7735_DrawBitmap------------
// Displays a 16-bit color BMP image.
// (x,y) is the screen location of the lower left corner of BMP image
// Requires (11 + 2*w*h) bytes of transmission (assuming image fully on screen)
// Input: x     horizontal position of the bottom left corner of the image, columns
from the left edge
//        y     vertical position of the bottom left corner of the image, rows from the
top edge
//        image pointer to a 16-bit color BMP image
//        w     number of pixels wide
//        h     number of pixels tall
// Output: none
// Must be less than or equal to 128 pixels wide by 160 pixels high
void ST7735_DrawBitmap(int16_t x, int16_t y, const uint16_t *image, int16_t w, int16_t
h);
```

Video 9.3.1. Custom image creation for the ST7735R

The size of the ST7735R LCD is too large to implement **buffered graphics**. Each pixel has a 16-bit color, requiring 2 bytes. The LCD is 160 by 128. Therefore the image would require 160*128*2=40960 bytes, which is more than the 32,768 bytes of available RAM. Furthermore, even if you had more RAM, it would take too long to send an entire image to the LCD and it would flicker. In this section, we discus an approach called **demand-based** that does not require you to place an entire image in RAM. We create small static images and place them in ROM as described by the first video. The basic approach to demand based graphics to only call the function **ST7735_FillScreen** once at the beginning, or when transistioning from one level of the game to another. In order to eliminate flicker during game play, we will not attempt to draw large images on the screen. Our images will be small objects called sprites, where small usually means less than 20 by 20 pixels. There are two approaches to drawing sprites on the screen. Assume for now your game uses a black background. A simple approach is to make a 2-pixel black border around each sprite. If we limit the movement of each sprite to -2,-1,0,1,2 change from one frame to the next in both x and y dimensions, we can simple move the sprite by drawing it on the screen. It will automatically cover up its position from the previous frame. If we update the screen at 30 Hz, this means our maximum sprite speed is 60 pixels per second, which should be fast enough for most games If the sprite has moved since the last frame we redraw it with one call:

1) **ST7735_DrawBitmap(x,y,image,w,h);**

A more flexible approach allows you to move a sprite anywhere on the screen between frames. Again assume your game uses a black background. With this approach, we make a second image the same size as the regular image, but color it all black. In this approach we explicitly cover up the sprite from the last frame by drawing a blank image in the old location. If the sprite has moved since the last frame we redraw it with two calls:

1) **ST7735_DrawBitmap(oldx,oldy,black,w,h);**
2) **ST7735_DrawBitmap(newx,newy,image,w,h);**

This second approach is more flexible, but runs twice as slow and may cause weird images if sprites overlap in space.

Maintenance Tip: It is important not to erase the screen during normal game play. Rather, we perform graphics commands to only output the pixels that need changing.

9.4. Creating and playing audio

Say you want to convert a wav file **(blah.wav**) you want to use in your game. Here is a Matlab (or a free alternative to Matlab from GNU called Octave – http://octave.org) script file (WC.m) you can use to convert the wav file into a C array declaration that can be used in your code. Run the script by passing it the file as input: **WavConv('blah')**. That's it, you should have a file (called **blah.txt**) with a declaration you can cut and paste in your code. Note that the samples are 4-bit samples to be played at 11.025kHz.

Video 9.4.1. Converting WAV files to C code

The following video shows a simple approach to sound. SysTick periodic interrupts are used to output sounds to the DAC.

*Video 9.4.2. Playing simple sound on the DAC (**needs recording**)*

```
const uint8_t shoot[4080] = {8,6,6,10,12,8,2,6,12,7,3,5,9,13,9, ...
};
static uint32_t I=0;
void Sound_Off(void){
  SysTick->LOAD = 0; // stop sound
}
void Sound_Start(void){
  I = 0;
  SysTick->LOAD = 7256; //start sound
}
int main(void){
  __disable_irq();
  DAC5_Init();
  Clock_Init80MHz(0); // 80M/7256 = 11.025kHz
  SysTick_IntArm(7256,1,0); // high priority
  Sound_Off();
  __enable_irq();
  while(1){
    Sound_Start(); Clock_Delay(80000000);
  }
}
void SysTick_Handler(void){
  DAC5_Out(shoot[I]); // output one
  I = I+1;
  if(I >= 4080) Sound_Off();
}
```

Program 9.4.1. Each invocation of the periodic interrupt outputs one value to the DAC

9.5. Using Structures to Organizing Data

When defining the variables used to store the state of the game, we collect the attributes of a virtual object and store/group them together. In C, the **`struct`** allows us to build new data types. In the following example we define a new data type called

Sprite_t that we will use to define sprites. The enumerated type defines the status of the sprite. Notice, we use signed integers so positions that end up negative are considered off the screen at the correct orientation.

```
typedef enum {dead,alive} status_t;
struct sprite {
  int32_t x;      // x coordinate
  int32_t y;      // y coordinate
  int32_t vx,vy;  // pixels/30Hz
  const unsigned short *image; // ptr->image
  const unsigned short *black;
  status_t life;        // dead/alive
  int32_t w; // width
  int32_t h; // height
  uint32_t needDraw; // true if need to draw
};
typedef struct sprite sprite_t;
```

Program 9.5.1. Example use of structures.

*Video 9.5.1. Demand-based Graphics on the ST7735R (**needs recording**)*

In C++, the **class** allows us to build new data types, and provide methods. In the following example we define a new class called **Sprite** that we will use to define sprites. The enumerated type defines the status of the sprite. Notice, we use signed integers so positions that end up negative are considered off the screen at the correct orientation.

```
typedef enum {dead,alive} status_t;
class Sprite {
private:
  int32_t x;      // x coordinate
  int32_t y;      // y coordinate
  int32_t vx,vy;  // pixels/30Hz
  const unsigned short *image; // ptr->image
  const unsigned short *black;
  status_t life;        // dead/alive
  int32_t w; // width
  int32_t h; // height
  uint32_t needDraw; // true if need to draw
public:
// constructor, Init, Move, Draw };
```

Program 9.5.2. C++ definition of a sprite object.

*Video 9.5.2. Introduction to C++ in Lab 9, starter code (**needs recording**)*

*Video 9.5.3. Creating a simple sprite class for Lab 9 (**needs recording**)*

9.6. Edge-Triggered Interrupts

*Video 9.6.1. Edge-Triggered Interrupt Configuration (**needs recording**)*

Synchronizing software to hardware events requires the software to recognize when the hardware changes states from busy to done. Many times the busy to done state transition is signified by a rising (or falling) edge on a status signal in the hardware. For these situations, we connect this status signal to an input of the microcontroller, and we use edge-triggered interfacing to configure the interface to set a flag on the rising (or falling) edge of the input. Using edge-triggered interfacing allows the software to respond quickly to changes in the external world. If we are using busy-wait synchronization, the software waits for the flag. If we are using interrupt synchronization, we configure the flag to request an interrupt when set. Each of the digital I/O pins on the MSPM0 family can be configured for edge triggering.

Program 9.6.1 configures PB21 to interrupt on the falling edge. See Figure 2.3.2. shows the negative logic switch interface to PB21. Because it is negative logic, the interrupt will occur when the operator touches the switch.

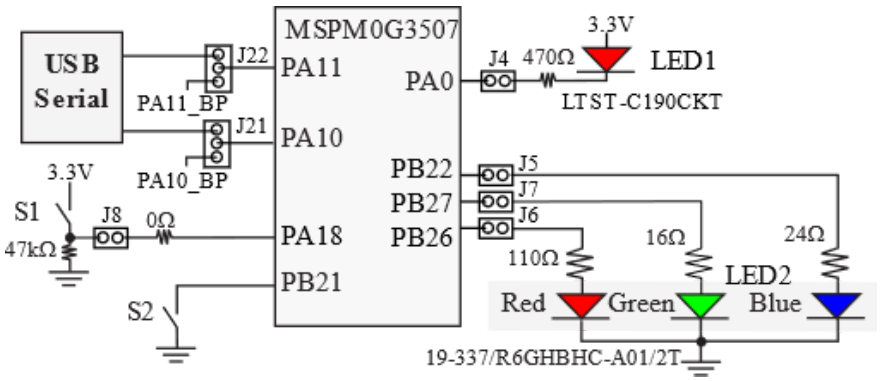


Figure 2.3.2. Switch and LED interfaces on the MSPM0G3507 LaunchPad Evaluation Board.

```
// Arm interrupts on fall of PB21
// interrupts will be enabled in main after all initialization
void EdgeTriggered_Init(void){
  GPIOB->POLARITY31_16 = 0x00000800; // falling
  GPIOB->CPU_INT.ICLR = 0x00200000; // clear bit 21
  GPIOB->CPU_INT.IMASK = 0x00200000; // arm PB21
  NVIC->IP[0] = (NVIC->IP[0]&(~0x0000FF00))|2<<14; // set priority (bits 15,14) IRQ 1
  NVIC->ISER[0] = 1 << 1; // Group1 interrupt
}
uint32_t Count; // number of times switch is pressed
int main(void){
  __disable_irq();
  LaunchPad_Init(); // PB21 is input with internal pull up resistor
  EdgeTriggered_Init();
  Count = 0;
  __enable_irq();
  while(1){
    GPIOB->DOUTTGL31_0 = GREEN; // toggle PB27
  }
}
// do not need to use IIDX when there is a single interrupt source
void GROUP1_IRQHandler(void){
  Count++; // number of touches
  GPIOB->DOUTTGL31_0 = RED; // toggle PB26
  GPIOB->CPU_INT.ICLR = 0x00200000; // clear bit 21
}
```

Program 9.6.1. Using interrupts to count switch touches.

9.7. Switch Debouncing

The PB2 interface in Figure 9.7.1a) implements negative logic switch input, and the PB3 interface in Figure 9.7.1b) implements positive logic switch input. Most inexpensive switches will **bounce** when touched and released. This means the digital signal will have multiple transitions lasting about 1ms on touch and on release. Notice that 10k*0.22uF equals 2.2ms, which is longer than the switch bounce time. When the switch is released, charge is added to the capacitor with the 2.2ms time constant, removing any switch bounce. When the switch is touched, the charge in the capacitor is discharged across the switch. We add 100 ohm resistor in series with the capacitor to prevent sparks across the switch when the switch is touched. The 100/10k ratio guarantees the high voltage will be close to 3.3V and the low voltage will be close to 0V.

Checkpoint 9.7.1 : What do negative logic and positive logic mean in this context?

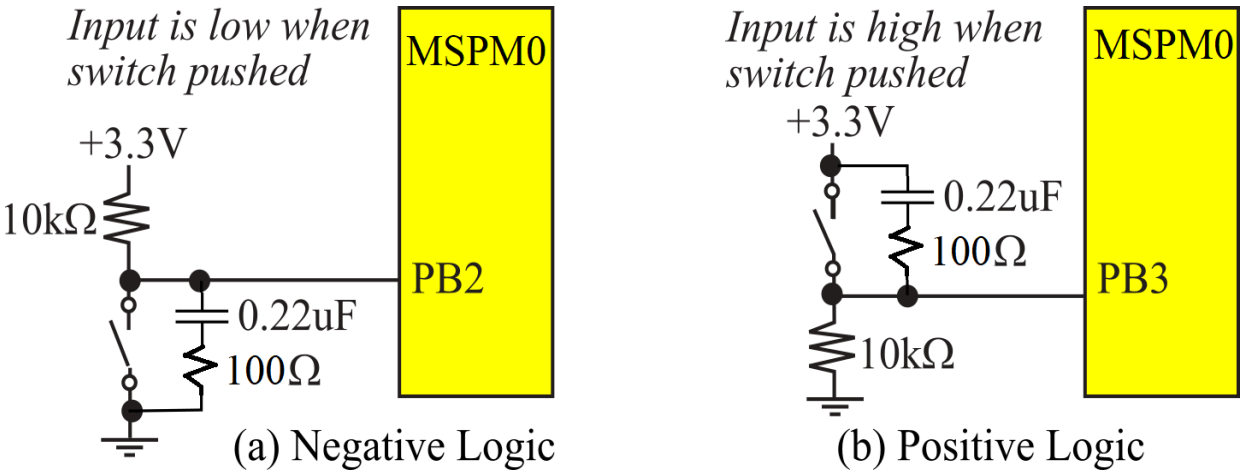


Figure 9.7.1. A capacitor can be placed across the switch to debounce it.

Checkpoint 9.7.2: What would happen if we left off the capacitor in Figure 9.7.1, and we used edge-triggered interrupts as shown in Section 9.6?

The simplest way to debounce a switch is to read the switch value in a periodic interrupt. Let $\mathbf{T_{touch}}$ be the shortest the time between touching and releasing. Let $\mathbf{T_{release}}$ be shortest time between releasing and touching. These $\mathbf{T_{touch}}$ and $\mathbf{T_{release}}$ are determined by how fast a human can touch/release the switch. Let $\mathbf{T_{bounce}}$ be the longest bounce time of the switch. $\mathbf{T_{bounce}}$ is determined by the mass, spring, and friction in the switch. We choose the interrupt period to be longer than $\mathbf{T_{bounce}}$, but shorter than $\mathbf{T_{touch}}$ and $\mathbf{T_{release}}$. Program 9.7.1 uses a periodic SysTick interrupt at 30 Hz to count the number of touches similar to Program 9.6.1.

```
// Arm periodic SysTick interrupts
void SysTick_IntArm(uint32_t period, uint32_t priority){
  SysTick->CTRL = 0x00; // disable during initialization
  SysTick->LOAD = period-1; // set reload register
  SCB->SHP[1]   = (SCB->SHP[1]&(~0xC0000000))|priority<<30; // set priority (bits
31,30)
  SysTick->VAL = 0; // clear count, cause reload
  SysTick->CTRL = 0x07; // Enable SysTick IRQ and SysTick Timer
}
uint32_t Count; // number of times switch is pressed
uint32_t Last; // Last value of PB21
int main(void){
  __disable_irq();
  Clock_Init80MHz(0);
  LaunchPad_Init(); // PB21 is input with internal pull up resistor
  Last = GPIOB->DIN31_0&(1<<21); // PB21
  SysTick_IntArm(80000000/30,1); // 30 Hz
  Count = 0;
  __enable_irq();
  while(1){
    GPIOB->DOUTTGL31_0 = GREEN; // toggle PB27
  }
}
void SysTick_Handler(void){uint32_t now;
  now = GPIOB->DIN31_0&(1<<21); // PB21
  if((now==0)&&(Last!=0)){
    Count++; // number of touches
    GPIOB->DOUTTGL31_0 = RED; // toggle PB26
  }
  Last = now; // for next time
}
```

Program 9.7.1. Using periodic SysTick interrupts to count switch touches.

*Video 9.7.1. Using periodic interrupts to debounce a switch (**needs recording**)*

Checkpoint 9.7.3 : Why did we implement (now==0) to mean touch?

Checkpoint 9.7.4 : What would happen if we left off (Last!=0) in Program 9.7.1?

9.8. Random Number Generator

The starter project includes a random number generator. To learn more about this simple method for creating random numbers, do a web search for **linear congruential multiplier**. The random number generator in Program 9.8.1 seeds the number with a constant (**M**=1); this means you get exactly the same random numbers each time you run the program. The constants **1664525** and **1013904223** were chosen so **M** will go through all 2^{32} possible integers before cycling back to **M**=1.

```
uint32_t M=1;
uint32_t Random32(void){
  M = 1664525*M+1013904223;
  return M;
}
```

Program 9.8.1. Linear congruential multiplier implementation to generate random numbers.

To make your game more random, you could seed the random number sequence using the SysTick counter that exists at the time the user first pushes a button (copy the value from **SysTick->VAL** into the private variable M). The problem with LCG functions is the least significant bits go through very short cycles. For example

Bit 0 of **M** has a cycle length of 2, repeating the pattern 0,1,....
Bit 1 of **M** has a cycle length of 4, repeating the pattern 0,0,1,1,....
Bit 2 of **M** has a cycle length of 8, repeating the pattern 0,1,0,0,1,0,1,1,....

Therefore, using the lower order bits is not recommended. For example

```
  n = Random32()&0x03;   // has the short repeating pattern 1 0 3 2
  m = Random32()&0x07;   // has the short repeating pattern 0 7 2 1 4 3 6 5
```

You will need to extend this random number module to provide random numbers as needed for your game. For example, if you wish to generate a random number between 1 and 5, you could define this function

```
uint32_t Random5(void){
  return ((Random32()>>24)%5)+1;  // returns 1, 2, 3, 4, or 5
}
```

Using bits 31-24 of the number will produce a random number sequence with a cycle length of 2^{24}. Seeding it with 1 will create the exact same sequence each execution. If you wish different results each time, seed it once after a button has been pressed for the first time, assuming SysTick is running

```
   M = SysTick->VAL;
```

9.9. Summary and Best Practices

As we bring this class to a close, we thought we'd review some of the important topics and end with a list of best practices. Most important topics, of course, became labs. So, let's review what we learned.

Embedded Systems encapsulate physical, electrical and software components to create a device with a dedicated purpose. In this class, we assumed the device was controlled by a single chip computer hidden inside. A single chip computer includes a processor, memory, and I/O and is called a **microcontroller**. The MSPM0 was our microcontroller, which is based on the ARM Cortex M0+ processor.

Systems are constructed by **components**, connected together with **interfaces**. Therefore all engineering design involves either a component or an interface. The focus of this class has been the interface, which includes hardware and software so information can flow into or out of the computer. A second focus of this class has been **time**. In embedded system it was not only important to get the right answer, but important to get it at the correct time. Consequently, we saw a rich set of features to measure time and control

the time events occurred.

We learned the tasks performed by a **computer**: collect inputs, perform calculations, make decisions, store data, and affect outputs. The microcontroller used **ROM** to store programs and constants, and **RAM** to store data. ROM is **nonvolatile**, so it retains its information when power is removed and then restored. RAM is **volatile**, meaning its data is lost when power is removed.

We wrote our software in **C**, which is a structured language meaning there are just a few simple building blocks with which we create software: sequence, if-then and while-loop. First, we organized software into functions, and then we collected functions and organized them in modules. Although programming itself was not the focus of this class, you were asked to write and debug a lot of software. We saw four mechanisms to represent data in the computer. A **variable** was a simple construct to hold one number. We grouped multiple data of the same type into an **array**. We stored variable-length ASCII characters in a **string**, which had a null-termination. During the FSM (Lab 4) and again in the game (Lab 9) we used **structs** to group multiple elements of different types into one data object. In this chapter, we introduced **two-dimensional arrays** as a means to represent graphical images.

The focus of this class was on the **input/output** performed by the microcontroller. We learned that parallel ports allowed multiple bits to be input or output at the same time. Digital input signals came from sensors like switches and keyboards. The software performed input by reading from input registers, allowing the software to sense conditions occurring outside of the computer. For example, the software could detect whether or not a switch is pressed. Digital outputs went to lights and motors. We could toggle the outputs to flash LEDs, make sound or control motors. When performing port input/output the software reads from and writes to I/O registers. In addition to the registers used to input/output most ports have multiple registers that we use to configure the port. For example, we used **direction registers** to specify whether a pin was an input or output.

We saw two types of **serial input/output**, UART and SPI. Serial I/O means we transmit and receive one bit at a time. There are two reasons serial communication is important. First, serial communication has fewer wires so it is less expensive and occupies less space than parallel communication. Second, it turns out, if distance is involved, serial communication is faster and more reliable. Parallel communication protocols are all but extinct: parallel printer, SCSI, IEEE488, and parallel ATA are examples of obsolete parallel protocols, where 8 to 32 bits are transmitted at the same time. However, two examples of parallel communication persist: memory to processor interfaces, and the PCI graphics card interface. In this class, we used the **UART** to communicate between computers. The UART protocol is classified as **asynchronous** because the cable did not include the clock. We used the **SPI** to communicate between the microcontroller and the LCD display. The SPI protocol is classified as **synchronous** because the clock was included in the cable. Although this course touched on two of the simplest protocols, serial communication is ubiquitous in the computer field, including Ethernet, CAN, SATA, FireWire, Thunderbolt, HDMI, and wireless.

While we are listing I/O types, let's include two more: analog and time. The essence of **sampling** is to represent continuous signals in the computer as discrete digital numbers sampled at finite time intervals. The **Nyquist Theorem** states that if we sample data at frequency f_s, then the data can faithfully represent information with frequency components 0 to $< f_s$. We built and used the **DAC** to convert digital numbers into analog voltages. By outputting a sequence of values to the DAC we created waveform outputs. When we connected the DAC output to a speaker or headphones, the system was able to create **sounds**. Parameters of the DAC included **precision**, **resolution**, **range** and **speed**. We used the ADC to convert analog signals into digital form. Just like the DAC, we used the Nyquist Theorem to choose the ADC sampling rate. If we were interested in processing a signal that could oscillate up to f times per second, then we must choose a sampling rate greater than $2f$. Parameters of the ADC also included **precision**, **resolution**, **range** and **speed**.

One of the factors that make embedded systems so pervasive is their ability to measure, control and manipulate **time**. Our MSPM0 had many timers: some of which are SysTick, TimerA0, TimerG0, TimerG12. We used the timer three ways in this class. First, we used timer to measure elapsed time by reading the counter before and after a task. Second, we used the timer to control how often software was executed. In Lab 4, we used it to create accurate time delays in the FSM, and then in Labs 5-9, we used timers to create **periodic interrupts**. Interrupts allowed software tasks could be executed at a regular rate. Lastly, we used timers to create **pulse width modulated (PWM)** signals. The PWM outputs gave our software the ability to adjust power delivered to the DC motors.

In general, **interrupts** allow the software to operate on multiple tasks concurrently. For example, in your game you could use one periodic interrupt to move the sprites, a second periodic interrupt to play sounds, and edge-triggered interrupts to respond to the buttons. A fourth task is the main program, which outputs graphics to the LCD display.

One of the pervasive themes of this class was how the software interacted with the hardware. In particular, we developed three ways to **synchronize** quickly executing software with slowly reacting hardware device. The first technique was called blind. With **blind synchronization** the software executed a task, blindly waited a fixed amount of time, and then executed another tasks. The LED output in Lab 2 was an example of blind synchronization. The second technique was called busy wait. With **busy-wait** synchronization, there was a status bit in the hardware that the software could poll. In this way the software could perform an operation and wait for the hardware to complete. The SPI output and the ADC input were examples of busy-wait synchronization. The third method was interrupts. With **interrupt synchronization**, there is a hardware status flag, but we arm the flag to cause an

interrupt. In this way, the interrupt is triggered whenever the software has a task to perform. In Labs 5, 7, 8 and 9 we used timer interrupts to execute a software task at a regular rate. In Lab 9, we saw that interrupts could be triggered on rising or falling edges of digital inputs. Embedded systems must respond to external events. **Latency** is defined as the elapsed time from a request to its service. A **real-time system**, one using interrupts, guarantees the latency to be small and bounded. By the way, there is a fourth synchronization technique not discussed in this class called **direct memory access** (DMA). With DMA synchronization, data flows directly from an input device into memory or from memory to an output device without having to wait on or trigger software.

When synchronizing one software task with another software task we used semaphores, mailboxes, and FIFO queues. Global memory was required to pass data or status between interrupt service routines and the main program. A **semaphore** is a global flag that is set by one software task and read by another. When we added a data variable to the flag, it became a **mailbox**. The **FIFO queue** is an order-preserving data structure used to stream data in a continuous fashion from one software task to another. You should have noticed that most of the I/O devices on the microcontroller also use FIFO queues to stream data: the UART, SPI and ADC also employ hardware FIFO queues in the data stream.

Another pervasive theme of this class was **debugging** or testing. The entire objective of Lab 3 was for you to learn debugging techniques. However, each of the labs had a debugging component. A benefit of you interacting with the automatic graders in the class was that it allowed us to demonstrate to you how we would test lab assignments. For example, the Lab 4 grader would complain if you moved a light from green to red without first moving through yellow. QUESTION: How does the automatic graders in the Exam2 projects work? ANSWER: It first sets the input parameter, then it dumps your I/O data into a buffer, and then looks to see if your I/O data makes sense.

Furthermore, you had the opportunity to use test equipment such as a **voltmeter**, **logic analyzer**, and **oscilloscope**. Other debugging tools you used included **heartbeats**, **dumps**, **breakpoints**, and **single stepping**. **Intrusiveness** is the level at which the debugging itself modifies the system you are testing. One of the most powerful debugging skills you have learned is to connect unused output pins to a scope or logic analyzer so that you could profile your real-time system. A **profile** describes when and where our software is executing. Debugging is not a process we perform after a system is built; rather it is a way of thinking we consider at all phases of a design. Debugging is like solving a mystery, where you have to ask the right questions and interpret the responses. Remember the two keys to good debugging: **control and observability**.

Although this was just an introductory class, we hope you gained some insight into the **design process**. The **requirements document** defines the scope, purpose, and expected outcomes. We hope you practice the skills you learned in this class to design a fun game to share with friends and classmates.

Our parting thoughts about best practices (in no particular order of importance):

Here are thoughts about things to remember when designing or building embedded systems, in no particular order of importance:

- Consider debugging when defining, designing, implementing, building and deploying.
- Careful thought during design can save lots of time during implementation and debugging.
- Choose good variable names so the software is easier to understand.
- Divide large projects into modules and test each module separately.
- Separate hardware from software bugs by first testing the software on a simulator.
- When designing modules start with the interfaces, e.g., the header files.
- The second step when designing modules is pseudo code typed in as comments.
- Make the time to service an interrupt short compared to the time between interrupts.
- When developing a modular system, try not to change the header files.
- Use a consistent coding style so all your software is easy to read, change, and debug.
- Most of your time is spent changing or fixing existing code called **maintenance**.
- So, when designing code plan for testing and make it easy to change.
- Writing friendly code makes it easier to combine components into systems.
- Use quality connectors, because faulty connectors can be a difficult flaw to detect.
- It is your responsibility to debug your hardware and software.

- It is also your responsibility to debug other hardware/software you put into your system.
- A simple solution is often more powerful than a complex solution.
- Listen carefully to your customer so you can understand their needs.
- Draw wiring diagrams of electrical circuits before building.
- Double-check all the wiring before turning on the power.
- Double-check all signals in cables, donâ€™t assume red is power and black is ground.
- Be courageous enough to show your work to others.
- Be humble enough to allow others to show you how your system could be better.

This material was created to teach ECE319K at the University of Texas at Austin

Reprinted with approval from Introduction to Embedded Systems Using the MSPM0+, ISBN: 979-8852536594

.