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 TM4C123 was our microcontroller, which is based on the ARM Cortex M4 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 10) and again in the game (Lab 15) 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 occuring 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 SSI. 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 **SSI** to communicate between the microcontroller and the Nokia display. The SSI 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 ½ 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 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 2*f*. Parameters of the ADC also included **precision**, **resolution**, **range** and **speed**.

Help

One of the factors that make embedded systems so pervasive is their ability to measure, control and manipulate **time**. Our TM4C123 had a timer called SysTick. We used SysTick three ways in this class. First, we used SysTick to measure elapsed time by reading the counter before and after a task. Second, we used SysTick to control how often software was executed. In Lab 10, we used it to create accurate time delays, and then in Labs 12-15, we used SysTick to create **periodic interrupts**. Interrupts allowed software tasks could be executed at a regular rate. Lastly, we used SysTick 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 SOS output in Lab 7 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 UART I/O in Labs 5 and 11, and the ADC input in Lab 14 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 12, 13, and 14 we used SysTick interrupts to execute a software task at a regular rate. In Chapter 12, we saw that interrupts could be triggered on rising or falling edges of digital inputs. In this chapter we added more periodic interrupts using the timers. Embedded Systems

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 tasks 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, SSI 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 9 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 10 grader would complain if you moved a light from green to red without first moving through yellow. *Question*: How does the Lab 10 grader work? *Answer*: It first sets the input parameter, then it dumps your I/O data into a buffer just like Lab 9, and then looks to see if your I/O data makes sense. The Lab 13 grader tested both the shape and frequency of your sound outputs. Lab 14 tested the **accuracy** and **linearity** of your distance measurement system.

Furthermore, you had the opportunity to use test equipment such as a **voltmeter** (PD3+TExaS), **logic analyzer** (Keil simulation), and **oscilloscope** (PD3+TExaSdisplay). 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**.

Help

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. Even though Lab 15 is not graded, we hope you practice the skills you learned in this class to design a fun game to share with friends and classmates.


(https://plus.google.com
/108235383044095082735/posts)

(http://youtube.com/user/edxonline)

Help