Karen Shay West
9 Shannon Marie Way, North Easton, MA 02356

508-844-9776 http://www.linkedin.com/in/karenshaywest KarenWest15@gmail.com

SUMMARY OF RECENT ONLINE TECHNICAL – Going further into depth with Embedded Systems and Microcontrollers Using C and the Keil Tools – see my Linked In Project Section for details and github links to files:

- **Further Depth in Embedded Systems and Microcontroller Design:**
    - Each lab completed first in simulation and then on the real board. Goal of course was to understand how to do a systems approach to game design, by designing and testing a hand-held video game, which integrates all components from previous labs. Lab work covered:
        - **ADC (Analog to Digital Conversion) and Data Acquisition:** Developed a system that inputs an analog signal with an analog to digital converter, and an autonomous robot that uses two DC motors and two distance sensors, and also a test and position measurement lab, with analog to digital conversion and calibrated output
            - **Analog to Digital Conversion (ADC) and Data Acquisition - ADC software trigger example:** This class example demonstrated the 4 steps of analog to digital conversion: (1) initiate conversion (software trigger), (2) wait for the ADC to finish, (3) read the digital result, (4) clear the completion flag. The most pervasive method for designing the ADC in this microcontroller is successive approximation. A 12-bit successive approximation ADC is clocked 12 times. At each clock another bit is determined starting with the most significant bit. For each clock, the successive approximation hardware issues a guess on Vdac by sitting the bit under test to 1. If Vdac is then higher than the input bit Vin, then the bit under test is cleared. If Vdac is less than Vin then the bit under test remains 1.
            - **Analog to Digital Conversion (ADC) and Data Acquisition - Wall follower class example:** used SysTick interrupts to implement a software PWM to drive two DC motors at variable duty cycles. Two analog IR distance sensors: used to allow the robot to follow a wall. Two IR sensors: are mounted looking forward to the left and forward to the right. Goal: to control power to each wheel so the left and right distances to the walls are equal. A third IR sensor looks directly forward to attempt to avoid a head-on collision. If an object is detected too close to the front of the robot, both wheels are immediately stopped.

```
○     -----------------------------------------wall---------
○                    /
○                   /
○                  /
○                 /
○         -----------
○         |        |
○         | Robot   | ---> direction of motion and third sensor
○         |        |
○         -----------
○                 \
○                  \
```

        - \
        - \
        - --wall---------
    - **Analog to Digital Conversion (ADC) and Data Acquisition Lab: Potentiometer transducer - Measurement of Distance:** 7 objectives of this lab starter code: (1) intro. to sampling analog signals using the ADC interface, (2) development of ADC device driver, (3) learning data conversion and calibration techniques, (4) the use of fixed-point-numbers, which are integers used to represent non-integer values, (5) development of an interrupt-driven real-time sampling device driver, (6) development of a software system using multiple files, (7) learn how to debug one module at a time. Starter files: for designed a distance meter based on the ADC examples done in class. A linear slide potentiometer converts distance into resistance (0 <= R <= 10 kiloOhms). The software uses the 12-bit ADC in microcontroller sampled at 40 Hz using SysTick interrupts. A function converts the ADC sample into distance: units of 0.001 cm. Data stream: will be passed from the ISR into the main program using a mailbox, and the main program will output the data on a display. Microcontroller ADC: input from potentiometer: converts the input voltage to a number from 0 to 4095. The ADC is a successive approximation device with a conversion time on the order of several microseconds. Software: calculates distance with a resolution of 0.001 cm. Position measurements: should be displayed on the Nokia display through UART0. A periodic interrupt can be used for the real-time sampling. Distance resolution: smallest change in distance that the system can reliably detect. Resolution depends on electrical noise, number of ADC bits, and the resolution of the output display software. Accuracy is defined as the absolute difference between the true position and the value measured by the device. The armature is defined as the part that moves, and in this lab we were measuring the position of the armature, which in this lab is the slide potentiometer. Please refer to the lab description documents for further details (in the link above for this lab - starting with: LabC14_ADCandDataAcquisition_DistanceMeter_Page1.pdf)
- **DAC (Digital to Analog Conversion and Sound):**
    - **Digital to Analog Conversion (DAC) and Sound - an R2R DAC 3-bit desgin example:** In this class example, we created a sine wave analog output from digital inputs using an R2R 3bit design for the Digital to Analog (DAC) conversion. We had to generate a 100 Hz sine wave whose table size was 16 entries of integers that when sent to the output port would create the various levels of the sine wave, so we needed a 100Hz*16 = 1.6 kHz interrupt. Each time the SysTick interrupt handler was run it output one of the 16 voltage levels that created the output sine wave. The bus is 80MHz, so SysTick period was 80000kHz/1.6kHz = 50000 for the sine wave. Each time the switch was pressed, it either activated playing this 100Hz sound wave, or stopped it.
    - **Digital Piano Example using Digital to Analog (DAC):** The 3 objectives were to (1) learn about DAC conversion, (2) learn how digital data stored in a computer could be used to represent sounds and music, (3) study how the DAC and interrupts can be used to create sounds. This lab created a 4 note output digital piano. The DAC is 4 bits, but the example we did in class was 3 bits.
- **Interrupts:**
    - **Softsound example with interrupts:** This class example used the SysTick timer to request interrupts at 500Hz. It helped prepare us to do the tuning fork example at

440 Hz.

  - **Tuning Fork Example with interrupts:** The square wave sound is 440 Hz, a standard frequency made by a tuning fork. A positive logic switch was the input, and headphones were used as an output. A resistor placed in series with the output to the headphone speaker controls the loudness (larger resistor, quieter sound, but never go off). Press the switch: the square wave sound begins, press it again, it stops. When the output was oscillating, the period is 1/440 sec = 2.2727ms, so interrupt at 1/880 Hz = 1.13636 ms. The time "t" between touching a switch and the change in output need not be immediate, as long as the toggling begins before the next SysTick interrupt. The resistance of the headphones is 32 ohms, and Power (P) = Volts (V) * Current (I), and the output high voltage is 3.3 volts, and the in series resistor from the output to the headphones is 1500 ohms, the current delivered to the headphones as: I = 3.3 / (1500 + 32) Amps. The power delivered to the headphones should be less than 1 milliwatt, and with P = I**2 * 32 ohms, with R = 1500 ohms, the power delivered = 0.15 mW. Sound: created only when the voltage across the speaker oscillates. The frequency of the oscillation will determine the pitch of the sound. Frequency 440 Hz: note A, above middle C on the piano. To output silence, it requires less energy to leave the output low. An unchanging output results in no sound from the speaker, but it also requires current to flow through the electromagnet in the speaker. The PLL: runs at 80 MHz. With the SysTick interrupt, we found that timing was more accurate the software loop delays, and creating the 440 Hz wave with the 880 Hz interrupt worked better. SysTick interrupt service routine: looked at the state of switch and tracked whether the switch was pressed during the previous ISR service to know whether to toggle or quiet the output. We checked the output wave period as 1000 ms for one toggle cycle (high and low) divided by 2.272 ms = 440 Hz wave form to speaker to make sound.
  - **DC Motors Example with interrupts:** This class example used SysTick interrupts to implement a software pulse-width-modulted output (PWM) to drive a DC motor at a given duty cycle. The built-in button switch1 (SW1) increases the speed, and switch2 (SW2) decreases the speed.
  - **Edge Interrupts Example:** This class example demonstrated how to use the microcontroller board built-in switch button to create a falling edge interrupt.
  - **Periodic SysTick Interrupts:** This class example used the SysTick timer to request interrupts at a particular period.

- **Pulse Width Modulation (PWM):** Developed a system that outputted to a DC motor that uses pulse width modulation, Designed and tested a guitar tuner, producing a 440 Hz tone.
- **Phase Lock Loops (PLL): A software function to change the bus frequency using the PLL:** Being able to dynamically control bus frequency and I/O devices is important for low-power design. The execution speed of a microcontroller is determined by an external crystal. The microcontroller board we used had a 16 MHz crystal. Most microcontrollers include a phase-lock-loop (PLL) that allows the software to adjust the execution speed of the computer. Typically, the choice of frequency involves the tradeoff between software execution speed and electrical power. In other words, slowing down the bus clock will require less power to operate and generate less heat. Speeding up the bus clock obviously allows for more calculations per second, at the cost of requiring more power to operate and generating more heat. The default bus speed for the

microcontroller internal oscillator is 16 MHz ±1%. The internal oscillator is significantly less precise than the crystal, but it requires less power and does not need an external crystal. If we wish to have accurate control of time, we will activate the external crystal (called the main oscillator) to use the PLL to select the desired bus speed. There are two ways to activate the PLL. We could call a library function, or we could access the clock registers directly. To save power, we slow down the bus frequency removing as much of the wasted bus cycles while still performing all of the required tasks. For battery-powered systems the consumed power is a critical factor.

- **SysTick Timers and their accuracy:** The accuracy of SysTick depends on the accuracy of the clock. We use the PLL to derive a bus clock based on the 16 MHz crystal, the time measured or generated using SysTick will be very accurate. More specifically, the accuracy of the NX5032GA crystal on the LaunchPad board is ±50 parts per million (PPM), which translates to 0.005%, which is about ±5 seconds per day. One could spend more money on the crystal and improve the accuracy by a factor of 10. Not only are crystals accurate, they are stable. The NX5032GA crystal will vary only ±150 PPM as temperature varies from -40 to +150oC. Crystals are more stable than they are accurate, typically varying by less than 5 PPM per year. We learned how to implement time delays based on SysTick. The RELOAD register is set to the number of bus cycles one wishes to wait. If the PLL function has been executed, then the units of this delay will be 12.5 ns. Writing to CURRENT will clear the counter and will clear the count flag (bit 16) of the CTRL register. After SysTick has been decremented delay times, the count flag will be set and the while loop will terminate. Since SysTick is only 24 bits, the maximum time one can wait with SysTick_Wait is 224*12.5ns, which is about 200 ms. To provide for longer delays, the function SysTick_Wait10ms calls the function SysTick_Wait repeatedly. Notice that 800,000*12.5ns is 10ms. What is the longest time one could wait using SysTick_Wait10ms? The maximum delay is 232 * 10ms, which is about 500 days.
- **Finite State Machines (Mealy and Moore):**
  - **Odd Ones Detector:** We learned to use the static timer to get precise delays. Precision like this is required in real time systems where not meeting delays can have consequences, ranging from minor inconveniences to serious catastrophes (air traffic control, nuclear power plants, anti-lock brakes in a car, etc). We learned to build an abstraction called a finite state machine which makes as a design, complex systems using a simple abstraction that lends itself to probability of function. FSMs are a design process that is easy to verify and has a set of theory behind it. The same engine we build for one FSM can be used no matter how complex the system is. An FSM is a 5-tuple: sets of: inputs, outputs, states, transitions (STG = state transition graph, and a matrix can be used too), output determination. The Moore FSM is the focus for this class, where the next state is a function of current state and input, and the output is a function of the current state. Mealy machines are not covered in this class, but their output is a function of the current state and the input. For the odd ones detector shown in this code link above, the input is a bit, either one or zero, as is the output. There are 2 states, even and odd. When in even state and input = 1, you go to odd state where output = 1. When in even state and input = 0, we stay in even state and output = 0. When in odd state and input = 1, you go to even state and output = 0. When in odd state and input = 0, you stay in the odd state and output = 1. This is a very, very simple Moore FSM to demonstrate the concept. The time delay is 1 sec. spent in each state.

- **Traffic Light Controller FSM – first example:** Design a traffic light controller for the intersection of two equally busy one-way streets. Solution: The intersection has two one-ways roads with the same amount of traffic: North and East. We begin the design defining what constitutes a state. In this system, a state describes which road has authority to cross the intersection. The basic idea, of course, is to prevent southbound cars to enter the intersection at the same time as westbound cars. In this system, the light pattern defines which road has right of way over the other. Since an output pattern to the lights is necessary to remain in a state, we will solve this system with a Moore FSM. It will have two inputs (car sensors on North and East roads) and six outputs (one for each light in the traffic signal.) The six traffic lights are interfaced to Port B bits 5–0, and the two sensors are connected to Port E bits 1–0, PE1=0, PE0=0 means no cars exist on either road, PE1=0, PE0=1 means there are cars on the East road, PE1=1, PE0=0 means there are cars on the North road, PE1=1, PE0=1 means there are cars on both roads. The next step in designing the FSM is to create some states. We create decision rules defining what to do for each possible input and for each state. For this design we can list heuristics describing how the traffic light is to operate: If no cars are coming, stay in a green state, but which one doesn't matter. To change from green to red, implement a yellow light of exactly 5 seconds. Green lights will last at least 30 seconds. If cars are only coming in one direction, move to and stay green in that direction. If cars are coming in both directions, cycle through all four states.
- **Vending Machine FSM:** This section uses a function variable. For example we could define a function variable as such void (*functionPt)(void); Design: vending machine with two outputs (soda, change) and two inputs (dime, nickel). Solution: This vending machine example illustrates additional flexibility that we can build into our FSM implementations. Rather than simple digital inputs, we will create an input function that returns the current values of the inputs. Similarly, rather than simple digital outputs, we will implement general functions for each state. This approach provides an alternative mechanism when the input and/or output operations become complex. Our simple vending machine has two coin sensors, one for dimes and one for nickels. When a coin falls through a slot in the front of the machine, light from the sensor reflects off the coin and is recognized back at the sensor. An op amp creates a digital high at the Port B input whenever a coin is reflecting light. So as the coin passes the sensor, a pulse (V2) is created. The two coin sensors will be inputs to the FSM. If the digital input is high (1), this means there is a coin currently falling through the slot. When a coin is inserted into the machine, the sensor goes high, then low. Because of the nature of vending machines we will assume there cannot be both a nickel and a dime at the same time. This means the FSM input can be 0, 1, or 2. To implement the soda and change dispensers, we will interface two solenoids to Port E. If the software makes PE0 high, waits 10ms, then makes PE0 low, one soda will be dispensed. If the software makes PE1 high, waits 10ms, then makes PE1 low, one nickel will be returned.
- **Engine Control Mealy Type FSM:** It uses a pointer implementation of a Mealy finite state machine to operate an engine with a control input, gas and brake outputs, and two states.
- **FSM Used to Control a Stepper Motor to a Robot:** A motor can be evaluated in terms of its maximum speed (RPM), its torque (N-m), and the efficiency in which it translates electrical power into mechanical power. Sometimes however, we wish to

use a motor to control the rotational position (θ=motor shaft angle) rather than to control the rotational speed (ω=dθ/dt). Stepper motors are used in applications where precise positioning is more important than high RPM, high torque, or high efficiency. Stepper motors are very popular for microcontroller-based embedded systems because of their inherent digital interface. This next video shows a stepper motor controlled by a FSM. The first button makes it spin one way, the second button makes it spin the other way, and the third button makes it step just once. If both the first two buttons are pressed it wiggles back and forth.

  - **Traffic Light FSM – second example:** This lab demonstrated how to design a Moore FSM based on lecture examples to control a traffic light. The microcontroller had 3 input sensors, one each indicating cars on the east-west and north-south roads, and one for a pedestrian. There were 8 output LEDs, 3 each for the traffic lights for the east-west and north-south roads, and one each for the walk - don't walk signals. Nine states with each output wait time 0.5 sec. Cars were not allowed to crash, and when the don't walk signal flashed on and off, the pedestrian knew to hurry up and cross. When don't walk did not flash pedestrians knew not to cross. Pedestrians were not allowed to cross when cars were present on either road. If all 3 sensors were on, an output pattern was output to the LEDs for 20 sec. The system had input independence since each state had 8 arrows to the next state, depending on the input and current state. A Moore machine was used since the LED output pattern was based on what state you were currently in.

- **Arrays and Functional Debugging:** Developed a system that debugs by dumping data into an array.
- **UART (Universal Asynchronous Receiver Transmitter),The Serial Interface, I/O Synchronization:**
  - **UART Network Tool Example:** We developed a communication network between 2 LaunchPad boards over the UART channels. The switches were inputs, LEDs were outputs, and UARTs used to communicate. Information was encoded as colors on the LEDs. There were 5 questions with each question associated with one of the 5 colors. There were 3 answers to these questions associated with 3 other LED colors. A communication network includes both the physical channel hardware and logical procedures / software that allow users to communicate with each other. The network provides the transfer of information as well as the mechanism for process synchronization. When faced with a complex problem, one could develop a solution on one powerful and centralized computer system. Alternatively, a distributed solution could be employed using multiple computers connected by a network. The processing elements of a distributed system may be a powerful computer, microcontroller, an application specific integrated circuit (ASIC) or a smart sensor / actuator. Distributed processing places input, output and processing at multiple locations connected together with a network. There are many reasons to consider a distributed system network over a centralized solution. Often multiple simple microcontrollers can provide a higher performance at a lower cost compared to one computer powerful enough to run the entire system. Some embedded applications require input/output activities that are physically distributed. For real-time operation, there may be not enough time to allow communication for a remote sensor and a central computer. A distributed system can often be deployed and scaled. If complexity increases, you can add a network node, and vice versa.
  - **Nokia 5110 Display Class Example Using UARTS:** This code writes a series of

    images and text to the Nokia 5110 84x48 graphic LCD. Do not drive the backlight with 5V. It will smoke. However, the backlight on the LCD seems to be happy with direct drive from the 3.3V regulator. Although the PCD8544 controller datasheet recommends 3.3V, the graphic Nokia 5110 LCD can run at 3.3V or 5V. No resistors needed on the signal lines. You will need 5 signal lines to connect to the LCD, 3.3 or 5V for power, 3.3V for LED backlight, and 1 for ground.
  - **UARTs Lab** - Universal Asynchronous Receiver Transmitters - by reading in a number, converting it to ASCII, and sending it to a display connected to the UART.
  - **Serial Interface and UARTs Lab Functions:** UART Universal Asynchronous Receiver Transmitter serial interface. A UART input function accepts ASCII characters from the serial port and adds them to a string until <enter> is typed or until max length of the string is reached. It echoes each character as it is inputted. If a backspace is inputted, the string is modified and the backspace is echoed. It terminates the string with a null character. It uses busy-waiting synchronization on RDRF. A UART output function outputs a 32-bit number in unsigned hexadecimal format. Variable format 1 to 8 digits with no space before or after. A UART hex-input function accepts ASCII input in unsigned hexadecimal (base 16) format. No '$' or '0x' need be entered, just the 1 to 8 hex digits. It will convert lower case a-f to uppercase A-F and converts to a 16 bit unsigned number value range is 0 to FFFFFFFF. If you enter a number above FFFFFFFF, it will return an incorrect value. Backspace will remove last digit typed. UART output function for decimal numbers output a 32-bit number in unsigned decimal format. Variable format 1-10 digits with no space before or after. UART_InUDec: Input function for decimal numbers accepts ASCII input in unsigned decimal format and converts to a 32-bit unsigned number valid range is 0 to 4294967295 (2^32-1). If you enter a number above 4294967295, it will return an incorrect value. Backspace will remove last digit typed. The UART output string function Outputted a null terminated string. The UART character output function outputted an 8-bit character to the serial port. UART input character function waits for a new serial port input. UART input function waits for a new serial port input. UART Init initialized the UART for 115,200 baud rate (assuming 50 MHz UART clock), 8 bit word length, no parity bits, one stop bit, FIFOs enabled.
- **Goal of course – to put together all that we learned in an old fashioned video game from the 1970's and 1980's, Space Invaders, which was optional and I did not do, but may do someday:**
  - **Hardware:** ARM based microcontroller board (LaunchPad) and a simulator (from professors). Game controller hardware involved was a slide potentiometer, 2 switches, 2 LEDs, resistors to interface to the switches and LEDs, and a mini-display. Images of the space invaders, space ships, lasers, bunkers, etc, were 2 dimensional arrays. Microcontroller timers were used for periodic interrupts. The DAC (Digital to Analog Converter) was used to make sound. Graphics were displayed on the mini-display LCD.
  - **Programming involved:** timers, interrupt service routines, the Analog to Digital (ADC) Converter, and device drivers for the DAC and LCD. Professors provided code: interfacing the graphics to the mini-display.
  - **Game controller:** 2 buttons, 2 LED's and a slide potentiometer which acts as a joystick to move yourself to be protected by a bunker from the space invader missiles, or to move beyond the bunker to fire at them.

- **Audio files:** were provided and a DAC was used for making sounds to a speaker/headphones connected by an audio jack.
- **Main game engine:** runs in an infinite loop - interrupted by 2 interrupt service routines (ISRs). SysTick ISR runs 30 times per second (30 Hz) and handles the player ship fires (button press), the player ship being moved (slide potentiometer ADC input), but the sound made for a button press fire is scheduled by SysTick ISR, so that when the other faster ISR, Timer2A ISR runs, it can output more quickly (11 kHz, runs 11,000 times per second), the explosion sound array to the speaker/headphone.
- **Graphics (space invader enemies, player ship, etc):** movements are put into a buffer, but the main game engine outputs them to the LCD display, (time consuming).