Embedded Systems Competency Demonstration

Joshua McCready

August 1^{st}, 2022

1 Summary

Sample 1: CEREBRO and Single Axis Reaction Wheel for Balloon Payload Stabilization, an embedded software I developed while working as a research assistant at MXL.

Sample 2: Walkie Talkie Embedded System Term Project for Electromagnetics Class I did with teammates in 2015.

Sample 3: Low power, interrupt based, code demo for performing a task based on external input created for SkySpecs interview process.

Sample 4: Presentations given to Mathworks Automotive Conference in February '22 to help contextualize Chassis Controls safety software development done at Ford Motor Company.

The samples have been provided in a .zip file.

2 Prompt

Sam DeBruin and I discussed a network of low power embedded systems (spokes) connected by a wireless protocol to a hub device capable of up-linking to a server. The low power embedded systems were to be affixed to different positions inside the turbine blades and capture images of of the interior of the blades. This remote monitoring would limit the need for human inspection inside the blades.

As part of the candidate selection for this role I have been asked to submit a code sample that demonstrates my competence and abilities in writing C code for embedded systems. The submission must identify any other authors and what their contributions were. Several examples were provided. The sample should be submitted by August 1^{st}

3 Response to Prompt

The most relevant projects that I have available samples from are from around 2015. That said, I gained technical depth in my later experiences different roles at Ford Motor Company and as an MSE student that might not be reflected in earlier samples. Since then I have grown in my ability to develop algorithms, plan projects, engineer systems, and create documentation. I have some presentations about this work that were given at the MathWorks Automotive Conference this year by my colleagues and I. I have provided them to help contextualize my recent work.

The older projects I have to share come from my time as a research assistant at the Michigan eXlploration Lab where I wrote software for embedded systems for weather balloons and from a class project for antenna design where we made a set of walkie talkies. There is a pretty good class report for the walkie talkie project, so I will avoid rehashing it too much in this document.

Because these projects are from so long ago, I would like to demonstrate that I am still able to perform the coding responsibilities for this role by writing a demo with low power modes and interrupt based software functions.

4 Sample 1: CEREBRO and Single Axis Reaction Wheel for Balloon Payload Stabilization

This project comes from when I worked in Dr. Cutler's research lab developing embedded software for high altitude balloon payloads that helped to test cube satellite components. This project developed the software for an embedded system called CEREBRO. The software was developed with the Crossworks IDE from Rowley Associates that the lab used and in which the lab's existing codebase was developed. It used the SALVO RTOS on an MSP430 to collect data from environmental sensors. The collected data was logged to an SD card that the MSP430 wrote to. CEREBRO sent the telemetry over UART to a Raspberry Pi that acted as the flight computer. The Raspberry Pi used the telemetry for state estimation and ran controller to command a single axis reaction wheel. The reaction wheel was operated by a motor controller that was being developed for use in cube satellites as a replacement for expensive off the shelf reaction wheels.

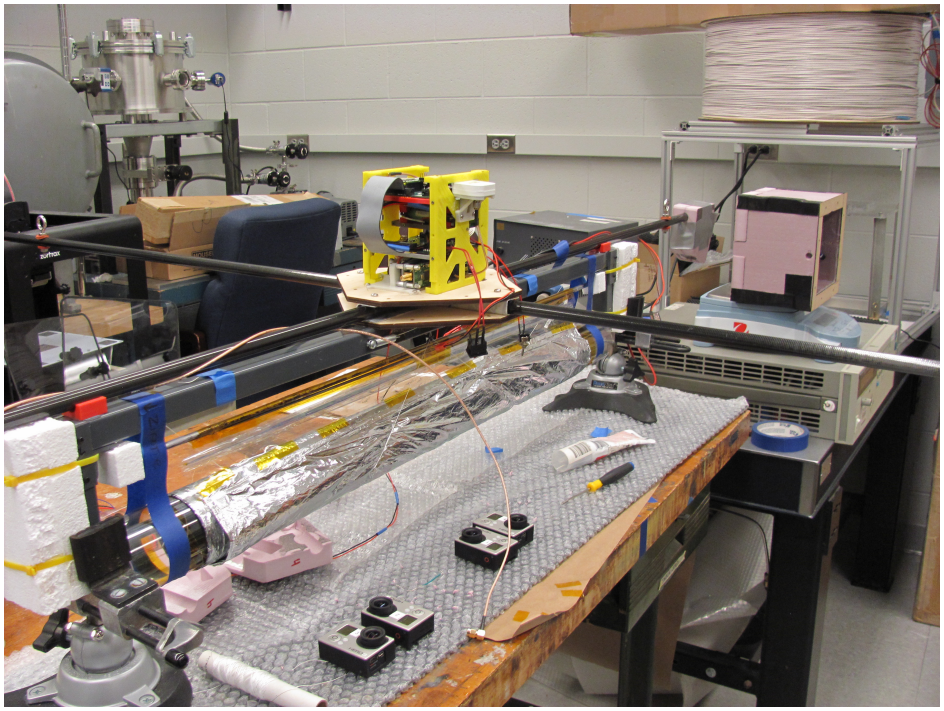

Figure 1: Balloon flight preparation photo from October 2015 featuring the CEREBRO embedded system, a Raspberry Pi as the flight computer, and stabilization booms to help steady the payload during flight. It is missing some of the photodiodes that mounted on the four vertical faces of the payload.

4.1 Implementation and Authorship

The CEREBRO software map (*EmbeddedDesignEngineerDemo\Sample 1\CEREBRO software map.pdf*) documents the basic software functions and the devices that were queried through I2C and UART. Most device drivers were written by me, but some were adapted from open source code, the existing Attitude Control software MXL, or the SW development kit from Salvo/Pumpkin corporation. A later software collaborator on the project was Andrew Wagenmaker who I worked closely with towards the end of my involvement with the project. The author blocks are

largely accurate. I worked closely with Andrew Taylor, another research assistant, to develop what components would be used, he did the the board layout and was responsible for the hardware and for the algorithm development on the state estimation and control law on the flight computer.

The *src* folder contains definitions for the scheduled tasks (*EmbeddedDesignEngineerDemo\Sample 1\proj_cerebro_dev \src)*, which are outlined in the software map and are scheduled in *main.c*. The setup for priority for the tasks are defined in *EmbeddedDesignEngineerDemo\Sample 1\proj_cerebro_dev \main.c* as are some semaphores for the various system events that instruct the tasks to yield, stop, or continue running. The tasks rely on device and bus drivers. There were many peripheral sensor devices which communicated through either I2C or UART. Because there were so many and there were enough I2C and UART channels available on the MSP430, the system used a switch device to connect the MSP430 to the different peripherals as needed.

UART was used to communicate to the GPS and flight computer, I2C was used to communicate with the environmental sensors and peripheral switching device. There are a few copies of the I2C and UART files floating around the file structure, they should have been named differently to avoid confusion. The device drivers are located *EmbeddedDesignEngineerDemo \Sample 1\drivers_i2c_devices* and *EmbeddedDesignEngineerDemo \Sample 1\drivers_uart_devices*. There is another sprinkling of drivers in the *EmbeddedDesignEngineerDemo \Sample 1\drivers_MSP_devices*, but most of those are adapted from other's work.

The Interrupt Service Routines for this project are mostly in the *EmbeddedDesignEngineerDemo\Sample 1\proj_cerebro_dev* folder in the *I2C.c*, *UART.c* and *init.c* files. There could be some others squirreled away that I have forgotten about.

4.2 Demonstrated Competencies

This project was a foundational one for me. Up to that point, I had some programming experience, but not a ton, so it was learn by doing. Especially in the setting of a reseach lab, it was important to to work from the available documentation rather than the PI to answer questions. What I learned about simple operating systems, interrupts, device drivers, communication protocols, peripherals, board layout, IC selection, working with debuggers and IDEs, etc. from this project has been invaluable too. It has added a great depth to my work at Ford and elsewhere. The tools change but the common threads are real time c applications that sense, actuate, and transmit information to between systems. Even though this work is from long ago, I felt it valuable to share because it taught me so much and because it touched on many of the example competencies that SkySpecs is seeking.

5 Walkie Talkie Embedded System Term Project for Electromagnetics Class

I won't offer too much in the way of an explain-er here. For that please see the *EmbeddedDesignEngineerDemo\Sample 2 \Final Report.pdf* document and *EECS 330 Project Presentation.pdf* in that same folder.

5.1 Implementation and Authorship

I did the software for the most part, but by and large it was a fairly collaborative project as I recall and the different team members all helped out for other aspects of the project, especially the testing of the end product.

5.2 Demonstrated Competencies

This was a project that came together part time in 6 weeks. I wanted to include it because it showed that together with my peers I could produce an embedded system demo over weeks not months with an existing codebase and having development top of mind. In this project I helped to layout a PCB. I say helped because my teammate mainly drove Altium while the others and I watched. I remember the project being a lot of fun.

6 Sample 3: Developed from Prompt

6.1 Development of Code Sample from Prompt

Because the code samples I have to furnish are from years ago, to demonstrate what SkySpecs is looking for I decided to get out an old TI Launchpad with an MSP430G2553 and download Code Composer to do a demonstration. It touches on the concepts needed for a low power solution for monitoring the inside a wind turbine with an image processing device. Given that scenario, I came up with some example requirements for my demo

1. The total power consumption for the device must be minimized.
2. The device must wake when prompted by an external event such as a button press to perform its task
3. The task the device must perform is to blink its LED 3 times

In order to minimize the amount of power consumed by the device, the MSP430X2XXX Product guide (2.4) states:

> "The most important factor for reducing power consumption is using the MSP430 clock system to maximize the time in LPM3"

LPM3 allows for interrupts to be active and ACLK to oscillate while consuming only 2 μA. This mode will enable requirement 1 and 2 to be met. However, if requirement 3. is done improperly, requirement 1. will be violated. It would be very easy to write a software driven blink waiting a long time in the function after the button press to toggle a GPIO pin using poling. But, it would be better to handle the blinking function another way; one that allows LPM3 to resume when LED is not being switched on or off for the blink sequence.

To further develop this concept, I have created a state machine diagram, state tables, and transition tables to aid in the implementation of code for Sample 3:

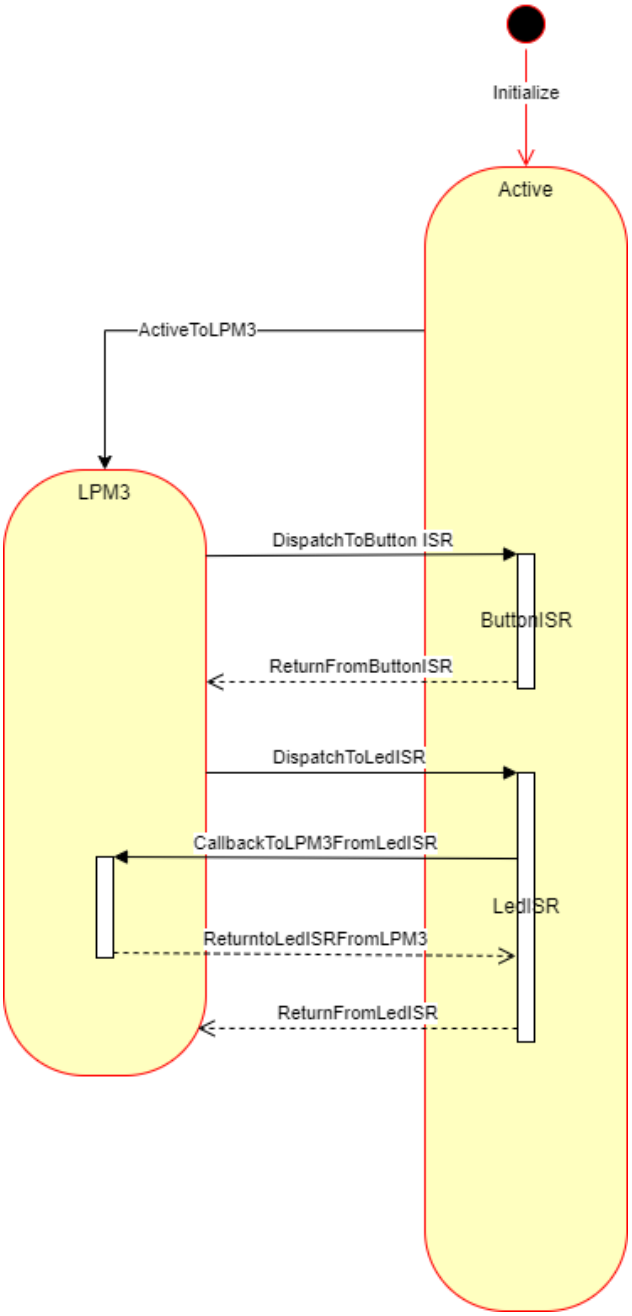

Figure 2: Application State Machine for requirements developed from prompt, that showing the application states and outlining the procedures required for the function.

Active State Table			
State	*On Entry*	*During*	*On Exit*
Active	Set Status Registers to Active Mode: SCG1 = 0 SCG0 = 0 OSCOFF = 0 CPUOFF = 0	Perform system functions	No Action

LPM3 State Table			
State	*On Entry*	*During*	*On Exit*
Active	Set Status Registers to LPM3 Mode: SCG1 = 1 SCG0 = 1 OSCOFF = 0 CPUOFF = 1	Conserve power, allow interrupts	No Action

With the states defined, I can define the events and procedures that will form the behaviors outlined in the requirements.

ButtonISR			
Active:: ButtonISR	*Dispatch To Button ISR*	*During*	*Return From Button ISR*
ButtonISR	1. GPIO pin connected to push button is pressed. 2. Active Blink variable is False	1. Set GPIO pin connected to LED1 to high 2. Set LED duration timer variable to 0. 3. Set LED blink count variable to 0 Set Active Blink variable to True	Return to PC

LedISR			
Active:: LedISR	*Dispatch To Led ISR*	*During*	*Return From Led ISR*
LedISr	All of the following 1. Blink Timer exceeds theshhold 2. Led Blink count variable is less than blink count threshold	1. Toggle LED GPIO pin 2. Increment Blink Count variable 3. Set Blink Timer Variable to 0 4. If Blink Count variable is equal to the blink count threshold, Then set Active Blink Count variable to False	ANY of the following 1. Callback to LPM3 from Led ISR 2. Set Active Blink variable to False, return to PC

6.2 Implementation

To implement the proposed demo, I downloaded Code Composer Studio and created the *MSP430G2553_Demo* project located in *EmbeddedDesignEngineerDemo\Sample 3\MSP430G2553_Demo*. I did not use the Grace tool, because I forgot about it. The demo has a *main.c*, *ISR.c*, and *config.c*. Nothing happens in the main but calling the *config()* function to setup the Basic Clock Module, Timer A, and GPIO pins used for the application. There are three ISRs, one for the button press event, and two for timer compare events, similar to what was shown in the Application State Machine. And like the diagram, when those ISRs are not active, the system is in a low power mode.

There are two pins setup as outputs for LEDs, one of which I just turned on permanently, the other is triggered by the button press and toggles three times. There is one pin set up as an input for the push button. This setup is shown in the *init_GPIO()* function of *config.c*.

The clock system is not setup in any particular way except that the ACLK is sourced from the Internal Very-Low-Power Low-Frequency Oscillator (VLO). I was going to use an external watch crystal, but after some testing it seems that it generated a fault bit when I tried to used it as the source of ACLK. Anyway, ACLK is used as the clock for Timer A. Timer A is setup in Up mode where it counts up to the values I set in the compare registers TA0CCRx. The compare registers of Timer A are used to set a period for the LED to be toggled. I divided the ACLK both using Timer A's DIVAx bits and the Basic Clock System's DIVAx bits, for a total divisor of 16. This was done to reduce the number of ticks in the period I wanted. Doing so, the interrupts would trigger less, reducing time out of low power mode. These divisors and the VLO's nominal 12 kHz frequency allowed me to calculate the values required for the compare registers for a approximately 0.5 second blink. All this is shown the *init_BCS()* and *init_Timer_A()* functions of *config.c*.

The *isr_port1()* works as described in the ButtonISR state table. It has a boolean signal that acts as semaphore to ensure that once a blink sequence starts it cannot be interrupted by a new one until the old one is finished. It also sets up a counter so that the Timer ISRs toggle the LED a fixed number of times. It is also important that when the button is pressed, the timer's value is reset to ensure that the total time in the blink sequence is constant.

The two timer ISRs *isr_timerA0(void)* and *isr_timerA1(void)* accomplish the TimerISR state table. They are triggered when the timer reaches the one of values in their respective compare register (rising edge only). This enables them to carry out the work of checking for an active semaphore and how many times they have toggled the LED. Once they have collectively toggled it enough times, they change the semaphore's value so a new blink sequence can be requested by a push button press. I have added a video of the working demo to the Sample 3 folder.

There is one oversight with these three ISRs, which is that when not in use TIMERA is still active. The application could use less power if after the Semaphore was reset, the timer ISR also disabled Compare register ISR flags. This would stop the timer ISRs from executing when there was not active request. The button ISR would have to re-enable the interrupts in this case.

6.3 Demonstrated Competencies

This demo shows that I am able to write embedded C for TI products with only minor struggle. The minor struggles I encountered was not remembering that you have clear interrupt flags at the

end of the interrupt, finding out my watch crystal had broken, and having the wrong vectors in my interrupts for a while. All of these issues slowed me down, but with the use of the debugger to inspect the registers, some example code from TI, and reading the documentation, I was able to resolve everything. Hopefully the demo was not too Micky Mouse, and showed that I understand the concepts of operation required for making low power embedded systems, that I can use interrupts, and that I can write code that others can understand.

7 Sample 4: Software Design for Automotive Functional Safety

My work at Ford Motor Company lately has been developing software and a software process for Ford developed software libraries that integrate into the redundant Brake Controls Modules for autonomous vehicles. This development differs somewhat from normal embedded development. The software libraries that Ford develops are largely at an application layer where the interfaces to the supplier software environment need to be designed well in advance with corresponding system level requirements that are shared across the software system. These system level requirements must be validated and decomposed, and likewise those software requirements validated.

As a result, the safety software libraries Ford develops must be as simple as possible both to simplify software integration, but also so to enable full validation that the software is functionally safe. The two presentations I have provided represent the byproducts of a deep engagement project that I led with my group and Mathworks to enable our software development to achieve the standard. Part 1 was done in April by me with some help from Mathworks and my colleague Hans, Part 2 was done later by my colleague Eric who more or less took the ball and ran with it. Since I cannot provide a code sample from the last two years of working at Ford because it is confidential, these presentations represent a software process that approaches compliance to the functional safety standards required of such software.

They demonstrate good knowledge of the software life-cycle, its connection to customer requirements, and complete requirements based test coverage. I believe it demonstrates the skills I have gained as a system engineer. Due to some delays caused by the pandemic, I have been able to exercise this software process more or less from beginning to end without having to skip steps along the way. In the process I championed new-to-Ford tools, project management styles and helped Mathworks develop features for Matlab and Simulink that others could use to further their compliance to functional safety standards.