Internet Controlled Toy Car

The Modern Entertainment Product

Team 16

Revision	Description
0.8.1	(Calvin Scmidt): Added IOT and serial communications information (David Gietzen): Added power analysis, flow charts, and code snippets (Jared Abel): Provided serial interrupt code, flowcharts (Justin Parson): Provided code snippets, proof-reading
0.6.1	(Calvin Scmidt): Added networking information (David Gietzen): Added menu subsystem
0.5.1	(Jared Abell): Added several code listings (David Gietzen): Information about IR emitter/detector
0.4.1	(Calvin Scmidt): Changes for full H-bridge (Justin Parsons): More detail in sections 3,4 and 6
0.3.1	(Justin Parsons): Added description of motor (David Gietzen): LCD, and chassis assembly
0.1.1	(Jared Abell): Initial design document draft (Calvin Scmidt): Initial sub-sections specified

Team Members

Jared Abell
Calvin Scmidt
Justin Parsons
David Gietzen

Originat Team 16				
Checked: 4/22/2016		Released: 4/23/2016		
Filename: Project 8 Writeup.doc				
Title:	**Internet Controlled Toy Car** **The Modern Entertainment Product**			
Date: **4/23/2016**	Document Number: **0-0000-000-0000-01**	Rev: **0.8.1**	Sheet: **1 of 60**	

Table of Contents

Figures

1. Scope

This product provides the consumer with a high-end remote controlled toy car, innovated with “internet of things” capabilities. The design utilizes a 3D printed chassis and intelligent control board, with all parts selected for easy mass production. This car can be manually controlled over wi-fi connections, as well as automatically guided. Our product supports full movement control, down to individual wheel speed adjustments and precise turning angles. The electronic control systems may be refactored for a multitude of different physical car designs; presenting the manufacturer and consumer with many customization options for increased satisfaction.

2. Abbreviations

ADC	Analog to Digital Converter
CPU	Central Processing Unit
FET	Field Effect Transistor
FRAM	Ferroelectric Random Access Memory
GPIO	General Purpose Input / Output
I/O	Input / Output
IOT	Internet of Things
ISR	Interrupt Service Routine
IR	Infrared
LCD	Liquid Crystal Display.
LED	Light Emitting Diode
MCU	Microcontroller Unit
MSP	Mixed Signal Processor; used to refer to the MSP430FR5739 by Texas Instruments
NFET	Negative Channel Field Effect Transistor
PCB	Printed Circuit Board
PFET	Positive Channel Field Effect Transistor
RC	Remote Controlled.

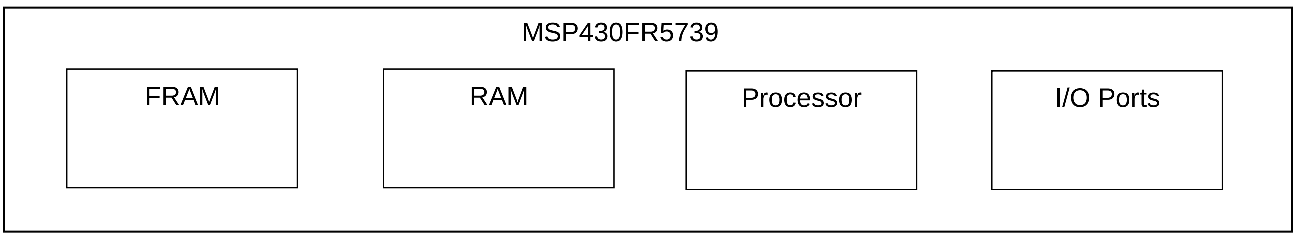
3. Overview

The main component of this toy car is the MSP, which is responsible for running software that determines the car's behavior. To connect the MSP to the other system components, a separate control board is necessary. These other system components include an LCD to provide the user with status information, two small motors to spin the wheels, and a battery pack. It is also necessary to utilize a wi-fi module, so that the car can be controlled over the internet.

3.1. MSP

The MSP controls all processing and computational tasks required by the device. The PCB has FRAM and RAM memory, a CPU, and general purpose ports to communicate with both analog and digital devices. Additionally, the board also has LED's that light up to indicate when the car's motors have been activated as well as other devices.

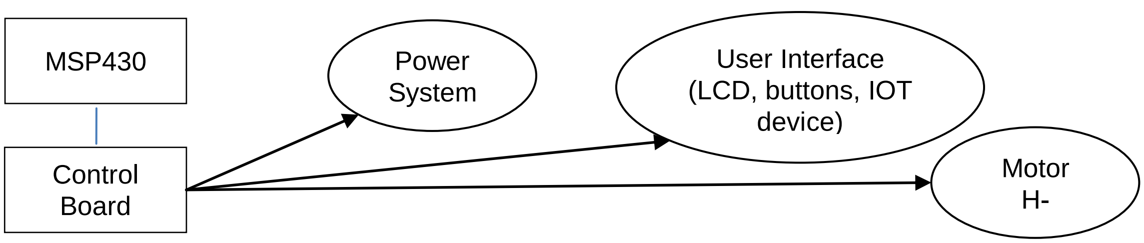
Figure 1 – MSP Overview



3.2. Control Board

The Control Board PCB is installed on top of the MSP, and connects the system to battery power for independent operation. A power switch is mounted on the board to minimize battery drain. Notably, the control board uses a 3.3v booster converter, so that power is supplied steadily and safely. An LCD is also installed on the control board, so that the user can be supplied information such as battery life, operating mode, distance traveled, etc.

Figure 2 – Control Board Overview



3.3. User Interface

The user interface for the car consists of an LCD display, 2 buttons mounted on the MSP, and a control program that the user may access through a standard internet browser. The LCD can display menus and information about the car, such as battery life and mode of operation. These menus may be navigated through the MSP buttons, while the web interface is used to manually move the car in a precise manner.

3.4. Power System

The car is supplied power through 4 AA batteries, with a nominal output voltage of 6v (variation from 5.5v to 6.5v is normal). This voltage is supplied directly to the motors whenever they are switched on, however, this voltage is too high for the other system components. Therefore, a “buck-boost” converter is used to convert the 6v source to 3.3v source for all other components, such as the MSP and LCD.

3.5. Motors

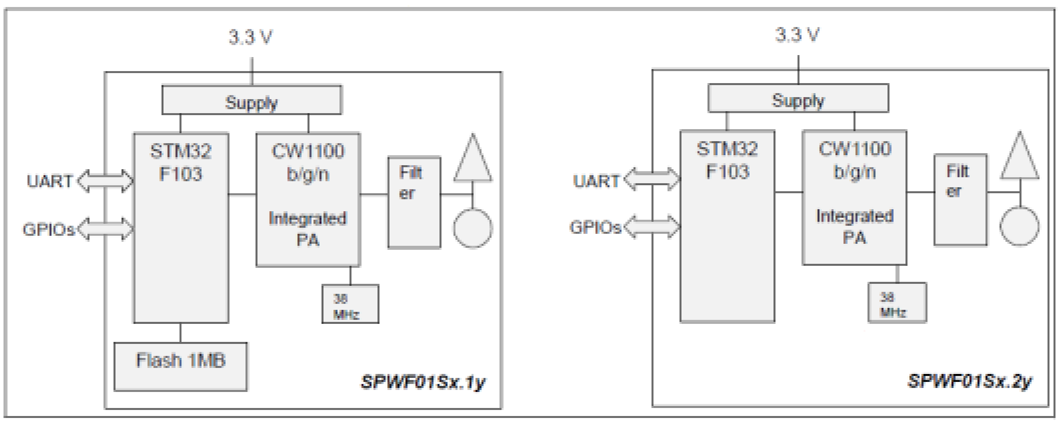
The car uses two small DC motors in a plastic housing to in order turn its wheels. The wheels attached to the motors may either be the two front wheels or two back wheels, depending on the configuration the manufacturer chooses. The motors are connected so that they may either spin clockwise or counterclockwise. To do this safely requires design considerations elaborated in sections 4 and 6.

3.6. IR Emitter/Detector

The car is equipped with an LED that emits infrared light, and a left/right infrared light detector that senses the reflection. These components allow the car to determine the brightness of the surface it is driving over. This allows the car to follow black and white race courses.

3.7. IOT

The IOT device is an intelligent Wi-Fi module used for easy integration of wireless internet connectivity into new or existing products. With low power consumption and small form factor, the modules are ideal for fixed and mobile wireless applications, as well as challenging battery-operated applications.



4. Hardware

This section provides detailed technical information for engineers using the device. When viewing this information, be certain to consider the following. The car requires 4 AA batteries (6v) to supply power, but this voltage must be converted to 3.3v for the MSP, so it must NOT be directly connected to the power supply. When the control board is separated from the MSP, jumpers may be placed on the MSP’s labeled pins to control its behavior while debugging. The MSP has a mini USB port on it, which can supply power to the device. You can simultaneously supply power from both the batteries and the mini USB port, which is useful behavior for debugging.

4.1. Control Board

Rather than attaching peripherals directly to the MSP, a separate control PCB is where other components will be attached. The control board will communicate to the MSP using busses that connect to all the available port pins. At this stage of development, the control board is connected to a power system, an LCD display, 2 motors, and mounted on the chassis of the car.

4.2. Power System

The power system uses a buck-boost converter (LT1930, labeled U1) to convert a 6v input to a 3.3v output. A detailed schematic with resistor and capacitor values is given in figure 3, while the power requirements are listed in figure 4. The capacitors should not be replaced by ones of smaller value, otherwise system components may by damaged by unstable voltages.

Figure 3 – Power System

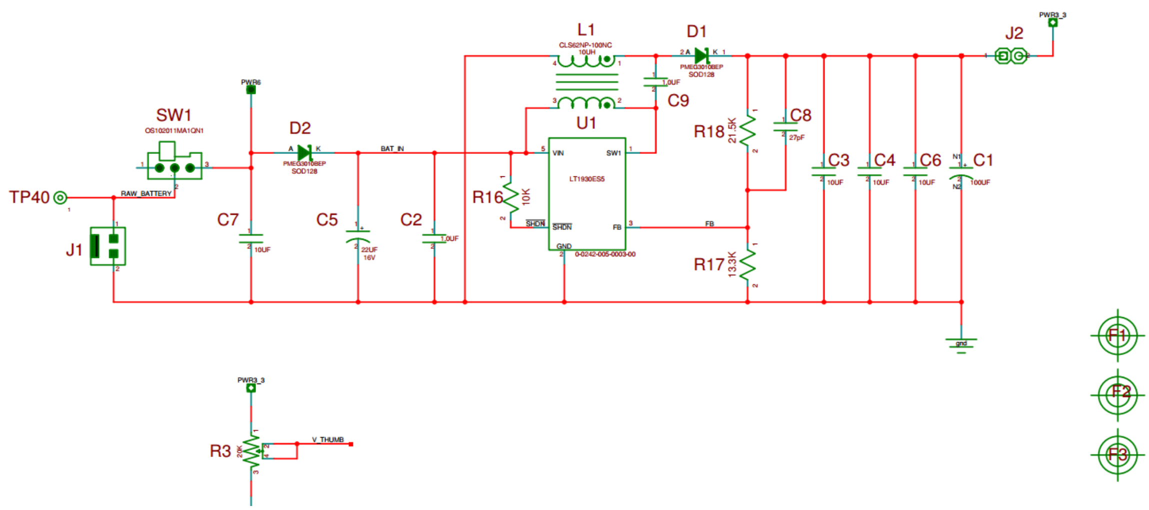


Figure 4 – MSP Power Requirements

Recommended Operating Conditions

Typical values are specified at V_{CC} = 3.3 V and T_A = 25°C (unless otherwise noted)

			MIN	NOM	MAX	UNIT
V_{CC}	Supply voltage during program execution and FRAM programming (AVCC = DVCC) (1)		2.0		3.6	V
V_{SS}	Supply voltage (AVSS = DVSS)			0		V
T_A	Operating free-air temperature	I version	-40		85	°C
T_J	Operating junction temperature	I version	-40		85	°C
C_{VCORE}	Required capacitor at VCORE			470		nF
C_{VCC}/C_{VCORE}	Capacitor ratio of VCC to VCORE		10			
f_{SYSTEM}	Processor frequency (maximum MCLK frequency) (2)	No FRAM wait states (3), 2 V ≤ V_{CC} ≤ 3.6 V	0		8.0	MHz
		With FRAM wait states (3), NACCESS = {2}, NPRECHG = {1}, 2 V ≤ V_{CC} ≤ 3.6 V	0		24.0	

(1) It is recommended to power AVCC and DVCC from the same source. A maximum difference of 0.3 V between AVCC and DVCC can be tolerated during power up and operation.
(2) Modules may have a different maximum input clock specification. See the specification of the respective module in this data sheet.
(3) When using manual wait state control, see the *MSP430FR57xx Family User's Guide* (SLAU272) for recommended settings for common system frequencies.

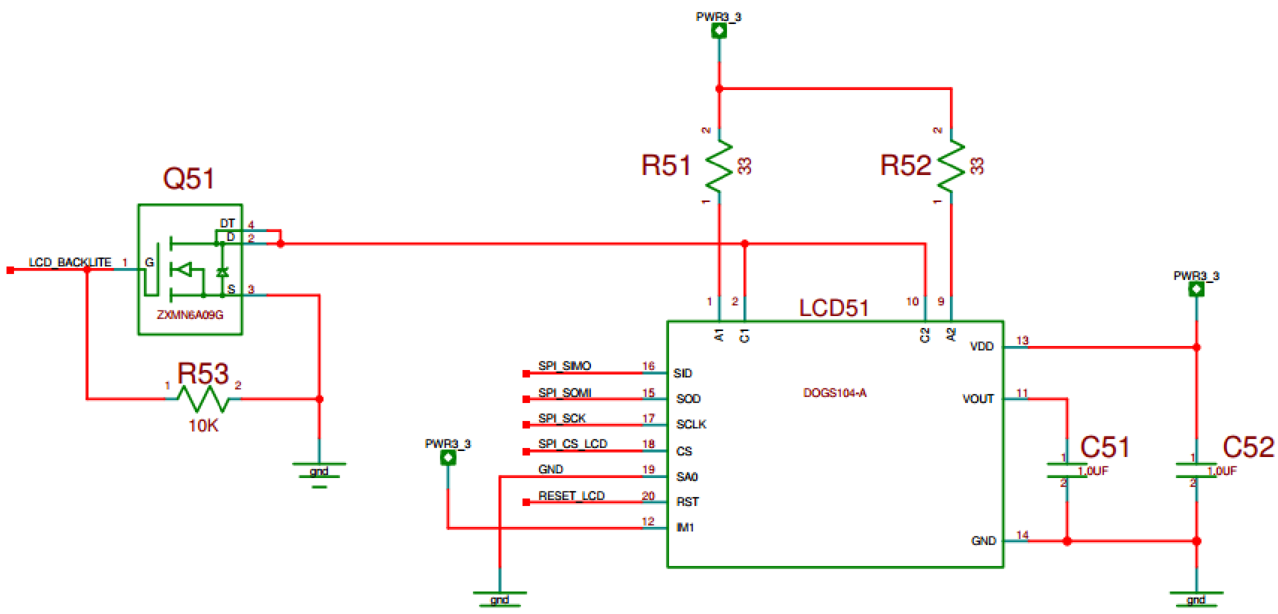
4.3. User Interface Devices

At this stage of development, the user interface consists of an LCD and two buttons mounted directly on the MSP. The buttons are used to navigate through menus displayed on the LCD

The LCD itself is mounted on top of the control board, and is a simple monochrome 4-line 10-character wide display. By using a display of this type, less battery power is consumed than a higher resolution color display. The backlight can be turned on and off using software, further saving power. A schematic for the LCD is given in figure 5.

There is also a thumbwheel placed close to the LCD screen, which the user can rotate to navigate through menus.

Figure 5 – LCD Panel



4.4. Motors

There are 2 motors used for this car, and they are simple DC motors in a plastic housing. The left and right motors are connected to an H-bridge configuration on the control board. The H-bridge configuration allows the system software to set the direction of the motors to forward as well as reverse. The motors should never be set to forward and reverse at the same time, otherwise the H-bridge will create a short from the power supply to ground, which can destroy the product. So, care must be taken when writing software to control the motors. The full H-bridge schematics, for both the left and right motors, are detailed in figures 6 and 7.

Figure 6 – Left Motor H-Bridge

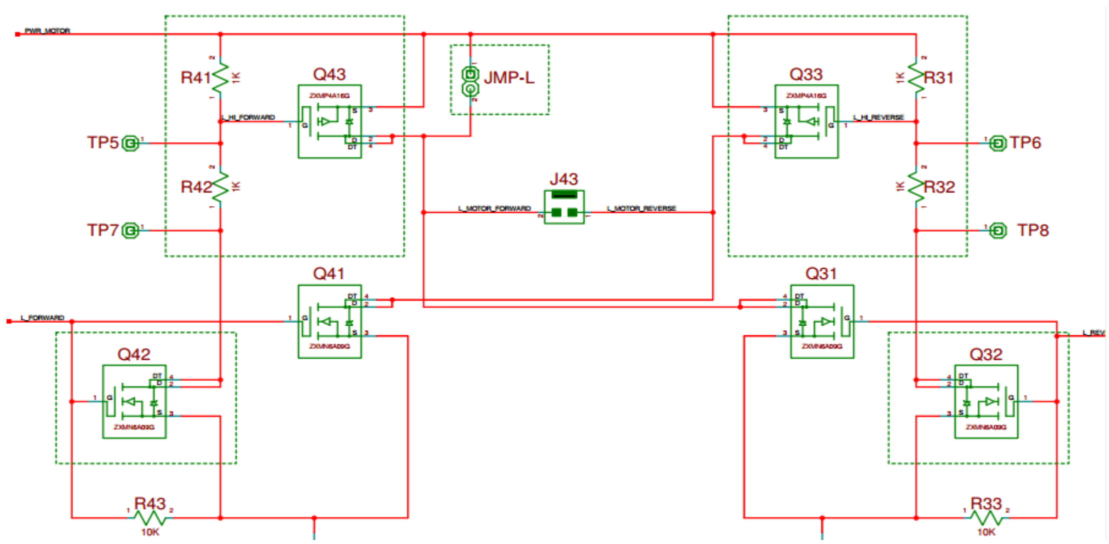
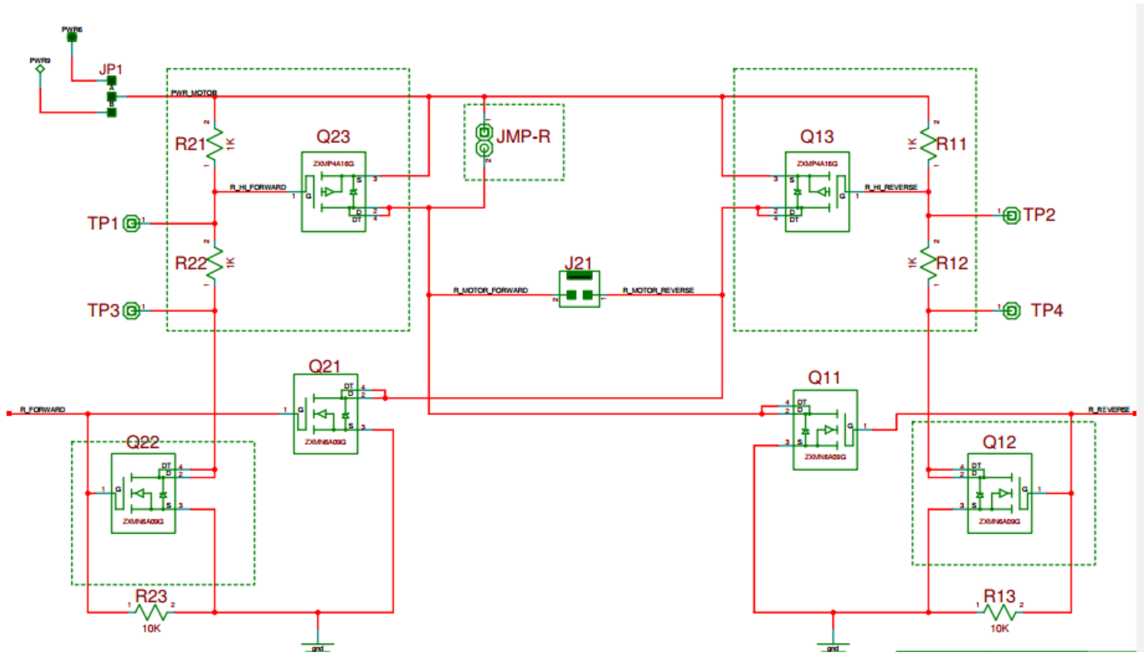


Figure 7 – Right Motor H-Bridge



4.5. IR Emitter/Detector

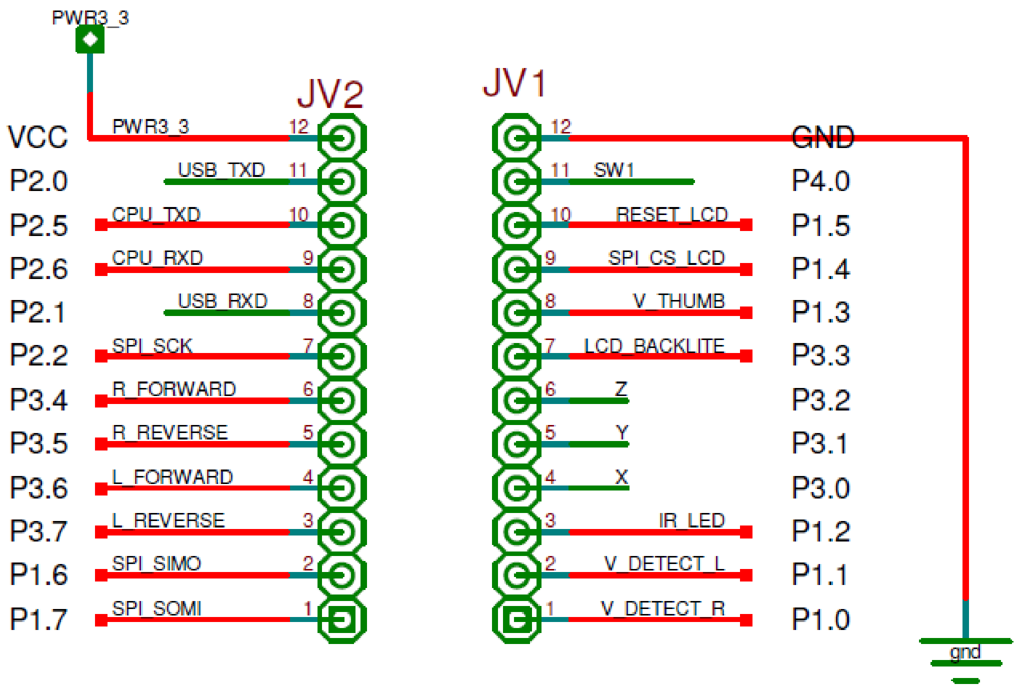
The car has an LED that emits infrared (IR) light, as well as left/right IR detectors. These components are connected to port 1 as the GPIO pins “IR_LED”, “V_DETECT_L”, and “V_DETECT_R”. The detectors must also provide a voltage to the MSP’s ADC to generate a reading. An example of how to configure the ADC interrupt to perform this reading can be found in section 9.

By shining infrared light onto the ground, the IR detectors can determine the brightness of the area the car is moving over. However, there is no ADC value from the IR detectors that is universally recognized as “white” or “black”. This is due to manufacturing variation between individual detectors, as well as the ambient light of the room affecting the reading.

Therefore some sort of calibration mode must be implemented if one chooses to use the IR emitter and detectors. This calibration mode can available to the end user, so that a faulty calibration does not necessitate a product replacement.

There can also be variation between multiple readings from the same detectors, which can fluctuate by about 10%. So rather than take a single reading, it is better to perform multiple readings at a time and average the result.

Figure 8 – Port Pins



4.6. MSP

The MSP430FR5739 is an MCU developed by Texas Instruments, which makes use of FRAM to store programs. This allows the program to be saved even when the system is powered off, yet it maintains a high write/read rate. As a result, newly written code can be tested quickly and efficiently. To write executable code to this device, one must use the IAR Embedded Workbench from IAR Systems.

The MSP has a 16-bit RISC CPU. All of the CPU's operations may be performed using only the CPU's built-in registers, excluding program-flow instructions. The CPU has 16 of these registers, which perform much faster than the system RAM. As a result, a register-to-register operation only requires one clock cycle. The following registers have special meanings:

FB	Frame Base
INTB	Interrupt Table Pointer
ISP	Interrupt Stack Pointer
R0	Program Counter (PC)
R1	Stack Pointer

R2	Status Register
R3	Constant Generator
SB	Static Base
USP	User Stack Pointer

There are 51 available CPU instructions, all with seven addressing modes for the source operand and 4 addressing modes for the destination operand. Each of these instructions may operate on word data, or byte data.

Using this board allows any manufacturer of this car to customize its behavior, from how it performs driving maneuvers to how it can be controlled through network devices. This can potentially be used to load firmware updates as well, provided that the manufacturer configures it to access some update server.

Figure 9 – Functional Block Diagram of MSP430FR5739IRHA

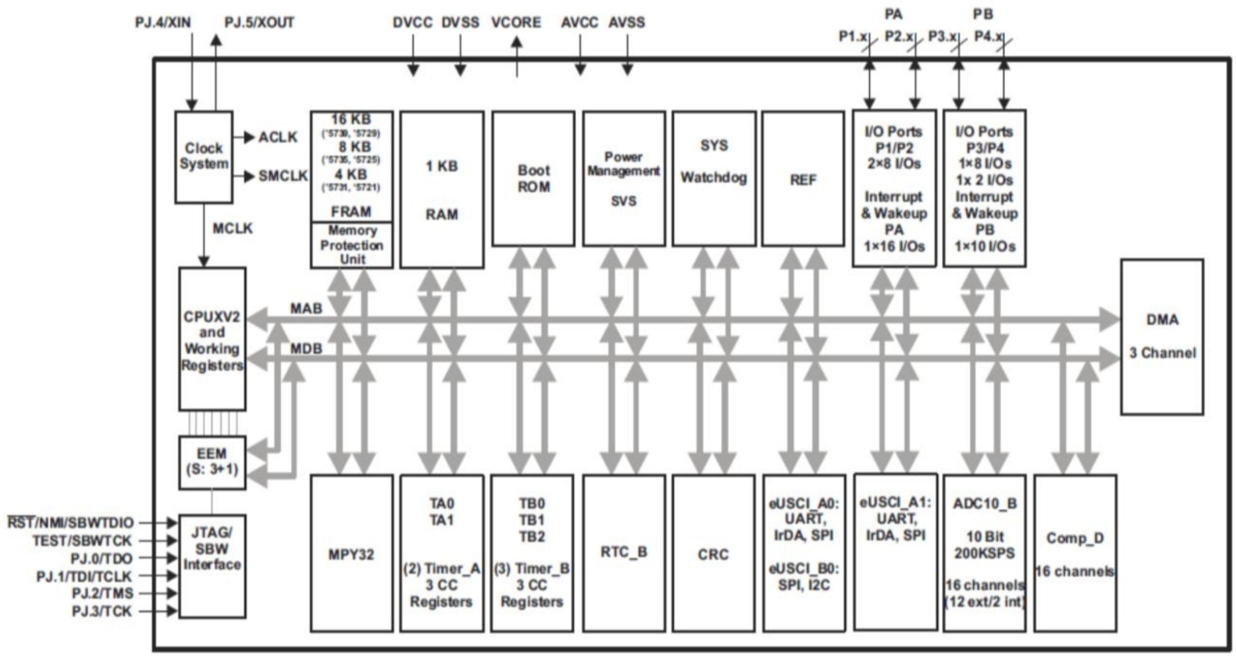


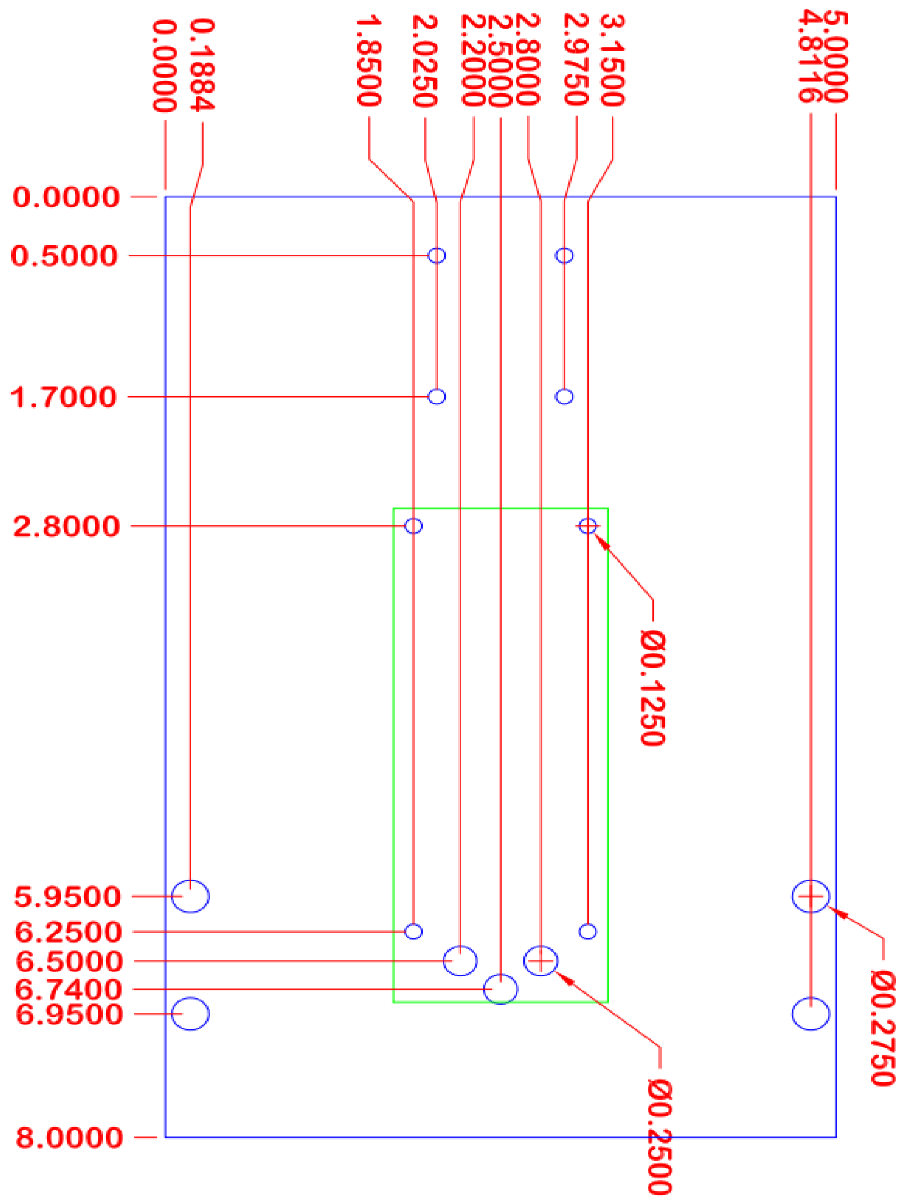
Figure 10 – MSP430FR5739 Memory Map

Address		Size	Description	Type
Begin	**End**			
0x000000	0x000FFF	4 KB	Peripherals	Registers
0X001000	0X0011FF	512 B	Boot Strap Loader 0	ROM
0X001200	0X0013FF	512 B	Boot Strap Loader 1	ROM
0X001400	0X0015FF	512 B	Boot Strap Loader 2	ROM
0X001600	0X0017FF	512 B	Boot Strap Loader 3	ROM
0X001800	0X00187F	128 B	Info B	FRAM
0X001880	0X0018FF	128 B	Info A	FRAM
0X001900	0X00197F	128 B	Mirrored to Info B	FRAM
0X001980	0X0019FF	128 B	Mirrored to Info A	FRAM
0X001A00	0X001A7F	128 B	Device Descriptor Info	TLV FRAM
0X001C00	0X001FFF	1 KB	RAM	RAM
0X002000	0X00C1FF	40 KB	Not Used	
0X00C200	0x00FF7F	15 KB	Main: Code Memory	FRAM
0x00FF80	0x00FFFF	128 B	Main: Interrupt Vectors	FRAM

4.7. Reference Car Chassis

The reference car chassis is made out of a 1/8 inch thick, 8” x 5” piece of acrylic plastic. A diagram of exact measurements is featured in figure 10. It includes mounting holes for a caster wheel, 2 motors, 3 holes for an optional black-line detector, and 4 holes for the control board. The control board should be placed on stand-offs, so that the MSP board can hang underneath it by its busses. The motors mounted on the chassis may use any light wheel that can fit on the axle (in our testing we used two Lego wheels (normal wide Ø43.2 X 22 with rim wide w/.cross 30/20)). The battery pack may be mounted on the underside of the chassis or near the corners opposite to the motors. For even weight distribution, it is recommended to mount the battery pack on the underside of chassis directly underneath the control board. This can be done with adhesive, or additional holes may be drilled for this purpose (granted that the screws do not make contact with the MSP board).

Figure 11 - Reference Car Chassis Diagram



4.8. IOT Module

The MSP is equipped with a SPWF01SA Serial-to-Wi-Fi b/g/n intelligent module allowing the car to communicate with 802.11 b/g/n. This device allows the car to wirelessly transmit data at 9600 baud. Configured around a single-chip 802.11 transceiver with integrated PA, and an STM32 microcontroller with an extensive GPIO suite, the modules also incorporate timing clocks and voltage regulators. With low power consumption and small form factor, the module is ideal for our application, even though it would not be considered “high performance”.

The SPWF01Sx.y1 parts are released with an integrated TCP/IP protocol stack, a web server, and additional application service capabilities. The SW package also includes an AT command layer interface for user-friendly access to the stack functionalities via the UART serial port. The SPWF01SA and SPWF01SC are surface mount modules with a 6-layer PCB.

5. Power Analysis

Determined experimentally, the current through the battery pack when the car is on is 155 mA. If the motors are turned on, the current through the battery pack increases to 320 mA.

For a Duracell brand battery discharging in this range of current, it can provide about 1.3 amp-hours. Note that the current through all of the AA batteries in the car will be the same.

So if we assume that the car will have its motors on for 80% of the time, the average current through the battery pack will be (0.8 * 155mA + 0.2 * 320mA) = 287mA. Therefore, the expected battery life will be (1.3 AH / 287mA) = 4.529 hours. Note that this is for when the car is actively being used and given commands. When it is on, but not being used, it can last for (1.3AH / 155mA) = 8.387 hours.

6. Test Process

This section details all the quality assurance tests that are normally performed during construction of the car. These represent useful ways to troubleshoot an inoperable device.

6.1. Control Board – Power System/Bus Assembly

Assembly of the power system and busses are done in three parts. First, all resistors and capacitors that are not close to U1 (farther than 1’’) are soldered on the control board. Second, U1 is soldered on the board, followed with every nearby component soldered in a counter-clockwise fashion. Finally, the female/male pin connectors are then installed on the board. By soldering these components in this order they can be easily cleaned and corrected in the event of an error or poor soldering joint, without compromising the vital power system components. Additionally, one could test for proper 3.3v output after the constructing the power system, before completing the third step.

6.2. Ultrasonic Cleaner

After assembly of the power system, boards are placed into an ultrasonic cleaner to remove all impurities that could conduct current and possibly damage components.

6.3. Mantis

Tilting the board at an angle of 30 degrees and viewing the PCB through a mantis viewer can reveal improperly soldered joints, as well as missing/damaged components. This should be performed before powering on the system after any soldering is performed.

6.4. Multimeter

After assembly schematics are reviewed, a multimeter may be used to check for “proper shorts” by measuring the resistance across individual pins. Additionally the power system can be checked to ensure a stable 3.3v output is present. If a measurement of 3.3v is not immediately measured, then the power switch should be quickly switched off, as this can indicate potentially irreversible harm being done to the system.

6.5. LCD

All backlights on LCD's are tested to make sure that the adhesive tape is securely attached to the back. Additionally, nearby pins are snipped and reflowed to remove any sharp edges before soldering the LCD, ensuring they do not damage its back. It is critical that the LCD is not installed flush with the PCB. To prevent this, tape is temporarily placed over its pin holes. The LCD pins are placed on this tape, and soldered in place. The tape can be removed after this.

6.6. FETs

FET's can become inoperable due to momentary high currents (possibly the result of bad firmware), compromising the operation of the whole car. Therefore, it is necessary to test every discrete FET individually when the car malfunctions.

To check an NFET, the resistance across the source and drain should be *high* when the gate-source voltage is low (i.e. the car is off). To check a PFET, the resistance across the source and drain should be *low* when the gate-source voltage is low. If either of these tests fail, then that indicates that FET needs to be replaced.

6.7. H-Bridge Voltage

After the H-bridge is constructed, it necessary to check if it is supplying the correct voltage to the motors.

With the power on, a jumper cable can be connected from JV2 pin-12 to JV2 pin-6 to test the right motor (R_FORWARD). To test the left motor (L_FORWARD), a jumper cable is connected from JV2 pin-12 to JV2 pin-2. If installation is correct, a 5.5v - 6.5v voltage will be created across the respective motor test points on the control board. This is the same as the expected output range of the battery pack.

6.8. "Forward Only H-Bridge" Firmware Testing

When performing firmware testing, it may be economical to construct only half of each H-bridge, so that the motors can not be put in reverse. This is done by soldering ONLY Q-21 and Q-41, and NO other FET in figures 6 and 7. This will necessitate shorting jumpers JMP-L and JMP-R to be placed on the control board, but these MUST be removed before constructing the full H-bridges. Otherwise, JMP-L and JMP-R will certainly damage the FET's and the power system.

When these "forward only H-bridges" are constructed, it is no longer possible to damage the H-bridge FETs by firmware, because it is not possible to place the motors in reverse and forward at the same time. This allows for drastic firmware changes to be tested without causing costly FET replacements

6.9. Full H-Bridge Construction and Voltage Testing

After the car's firmware has been tested thoroughly with a "forward only H-bridge", then more components of the full H-bridges in figures 6 and 7 may be added. *<u>Before doing so, make sure JMP-L and JMP-R are removed!</u>*

First, solder NFET's Q12, Q22, Q32, Q42, Q11, and Q31 onto the control board. Next turn the battery power on, and use some simple firmware that only turns on one direction for one motor at a time to test the following:

1) With R_FORWARD off, TP1 and TP3 should be equal to the battery voltage. With R_FORWARD on, TP1 should be half of the battery voltage, and TP3 should be equal to ground.

2) With R_REVERSE off, TP2 and TP4 should equal the battery voltage. With R_REVERSE on, TP2 should be half of the battery voltage and TP4 should be equal to ground.

3) With L_FORWARD off, TP5 and TP7 should equal the battery voltage. With L_FORWARD on, TP5 should be half of the battery voltage, and TP7 should be equal to ground.

4) With L_REVERSE off, TP6 and TP8 should equal the battery voltage. With L_REVERSE on, TP6 should be half of the battery voltage, and TP8 should be equal to ground.

If all the above tests have passed, then PFET's Q13, Q23, Q33, and Q43 may be installed. At this point, the normal firmware may be used. It should be noted, the chance of the normal firmware damaging the car still exists, so that software should be tested thoroughly.

6.10. Switch Debouncing

A common cause of the system switches malfunctioning is not properly "debouncing" signals received from them. The switches are imperfect analog devices, so when the user presses the switch once, it may send several "button press" signals to the MSP's processor. As a result, if the system software does not take this into account, it will appear as if the switches are broken to the user.

To rectify this, there must be a small amount of time that the software ignores switch signals after receiving a single one. This wait time is called the "debounce time". There are many ways to implement this logic, and it depends on whether the switches are being polled or are configured as interrupts. In section 7, a reference implementation of this logic can be found for when the switches are set up as an interrupt.

6.11.IR Calibration

When developing firmware that utilizes the IR detectors and emitters, it can be unclear whether incorrect IR readings are the result of a faulty hardware configuration or faulty code.

The first step in debugging the IR emitters/detectors is printing the detector readings on-screen. Even if these readings are incorrect, printing them onscreen allows one to see what causes the readings to change.

The IR detectors will not produce a useful reading unless the IR LED is on. To see if the IR LED is properly turning on, a digital camera can be used to see if it producing light. Even though most digital cameras have an IR filter, most will still see the IR LED emit a pinkish light. However, this will not work with all cameras, so if the pinkish light is not seen, that does not necessarily mean the IR LED is off.

The IR detectors must also point at the same location the IR LED is shining. Otherwise, the detectors will be sensing IR light from other sources, which will produce noisy readings. So, the detectors' long pins may be bent to face them towards the IR "spotlight". To help reduce the noise further, the back end of the detectors can be covered with a black material, such as electric tape.

To see if the IR emitter/detectors are correctly configured, a white sheet of paper and a black sheet of paper can be used. When placed under the detectors, the black piece of paper should produce a higher reading than the white sheet of paper. If the different sheets of paper do not change the reading, that suggests the detectors' ADC conversions are not being updated. This is commonly the result of incorrect code, namely, the next conversion is not being enabled at the end of the ADC interrupt.

6.12. IOT Disassociation

When used in close proximity to other devices, or due to limited Wi-Fi signal strength, the IOT device occasionally "disassociates", meaning it loses connection to the wireless router. A protocol is included in the software to resolve this problem where if the IOT device disassociates it immediately resets itself, thus reestablishing the connection.

7. Software

The car's runtime consists of a simple "operating system" that enters an intentional infinite loop, where any desired actions can be placed. For example, if one only wants the car to move in reverse at all times, the function "move_reverse ()" can be placed in this loop. A more useful and realistic example would be checking conditions that have changed as a result of interrupts, and then deciding which actions should be taken as a response.

The MSP offers many configuration options for its clocks and timers, and the car has many peripherals attached to it. These items can not configure themselves; they are not "plug-in-and-play". Therefore, it is not sufficient to immediately enter this main loop. Instead, an initialization function (or many) should be placed beforehand so that the timers, port pins, interrupts, ADC, and peripherals can all be configured before their first use.

Interrupts play an important role in the car's operation. Many signals that the car receives are asynchronous and time-critical (an example would be a switch being pressed or a timer completing). Attempting to handle these signals by polling every port pin at every moment would certainly fail. Some of these signals are more difficult to read than simply checking a port pin (in particular, any ADC value), and some port pins should not be set/unset without checking several global conditions. So polling would waste processor time, and considering the MSP CPU's speed, it would most likely make the car unacceptably irresponsive.

So, instead of polling, the port pins are configured to trigger interrupts during certain hardware events. Whenever an interrupt executes, it should quickly handle any necessary "housekeeping duties" (such as selecting the next device to be read in the ADC interrupt), and then alter the global state in a predictable fashion. By using this design methodology, the complexity of the main program loop is massively reduced, and program execution is much more efficient.

This is why the car's software is said to be "background – foreground". The background is the main loop, always taking actions depending on the current global state. The foreground is the set of interrupts that update the global state.

7.1. main.c

This file implements the "operating system". In its most general form, it just calls every initialization function, and then enters a loop where it calls other functions depending on global variables set by interrupts. This function should be kept as simple as possible, so that the car's software can be updated easily. Any sort of even moderately complex logic should be spun off into its own function.

7.2. timers.c

This file provides initialization code for the timers on the MSP, and it also provides "wait" functions. These functions cause the program to pause execution for some specified time, which necessarily requires the use of a timer interrupt. These "wait" functions are used for system timing throughout the code base.

7.3. ports.c

This file provides initial values for ports 1, 2, 3, 4, and J. It uses bit-wise operators extensively, as these ports and their associated registers are represented as 8-bit chars in the code. The initial values must be chosen carefully, as incorrect values can damage the hardware (in particular, the motor H-bridge). These initial values also determine which ports activate interrupts. There is no conditional logic whatsoever in this function, it will always reset the ports to their proper initial states when called.

7.4. adc.c

The ADC10_ISR() function has the responsibility of taking a reading from the ADC. There is only one ADC available on the MSP, but there are 3 analog signals that need to be converted. So, every time this interrupt executes, it must check what reading it is currently taking, and then select which reading it will take next. It places these readings in a global variable.

7.5. interrupts_timer.c

This provides the actual timer interrupt that timers.c is dependent on. This interrupt occurs every 1ms, so it has to be very short otherwise the system will have poor performance. So, it simply updates global counter variables reflecting how many milliseconds have passed, which other pieces of code can reference for timing purposes.

7.6. interrupts_ports.c

This interrupt service routine executes whenever a switch on the MSP board is pressed. It must check which switch has been pressed, and then update a global variable representing that. However, the switches produce noisy signals, so one switch press may seem like several switch presses to the MSP. Therefore, it is necessary to include extra logic that "debounces" the switches, and removes these spurious switch press signals.

7.7. menu.c

There are very few buttons available on the car, so in order to provide the user with a wide range of functionality, it is necessary to use a menu system. This file provides all the functions needed to display a menu, and it calls other functions depending on which menu item is selected. With the use of a generalized menu struct, this functionality is easily customizable.

7.8. interrupts_serial.c

When data, or more accurately "symbols", are received through the car's serial ports, the system must immediately respond to this event otherwise the information will be lost. That is why this file provides an interrupt that places any received data into globally accessible buffers for later processing.

7.9. serial_command_interpret.c

Not all of the data received from the IOT device contains commands for the car; most of these messages are diagnostic messages. Furthermore, in order to support more than one command, or commands with options, some sort of command structure must be specified.

That is the purpose of this file. It provides a serial command interpreter that expects commands in the form "WASNOTWAS.XXXX", where "WASNOTWAS" is the system PIN, and "XXXX" represents the desired action.

7.10. custom_string.c

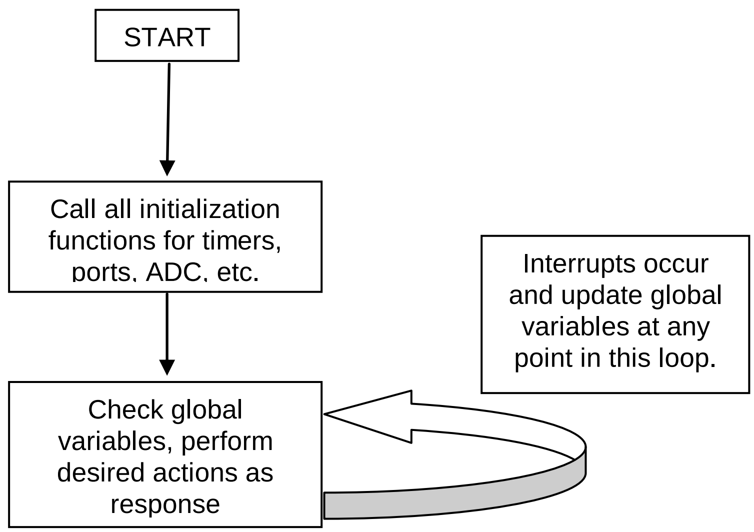
The extensive use of serial communications in this project necessitates the creation of string processing utilities. So, various functions have been defined to carry out string processing. For example, is_same_str() detects if two strings are the same, and set_str() allows one to change the contents of a string.

8. Software Listing

This section contains actual code from the different prototypes of the car. These code print-outs are meant to demonstrate how one could carry out a particular function. They are not meant to be copied verbatim, as they are not intended to operate with one another.

8.1. main.c

Figure 12 – Main Flow Chart



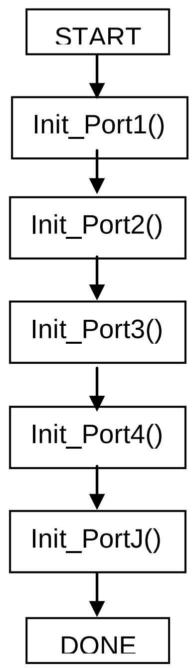

```
//------------------------------------------------------------------------------
//
//   main.c
//
//  Description: This file contains the Main Routine - "While" Operating System
//
//
//  Jim Carlson, edited later by David Gietzen
//  Jan 2016
//  Built with IAR Embedded Workbench Version: V7.3.1.3987 (6.40.1)
//------------------------------------------------------------------------------

//------------------------------------------------------------------------------
#include  "msp430.h"
#include  "functions.h"
#include  "macros.h"

// Global Variables
volatile unsigned char control_state[CNTL_STATE_INDEX];
volatile unsigned int Time_Sequence;
```

```
char led_smclk;
volatile char one_time;
volatile unsigned int five_msec_count;
volatile extern int switch_1_pressed;
volatile extern int switch_2_pressed;
extern char display_line_1[LCD_DISPLAY_LENGTH_1];
extern char display_line_2[LCD_DISPLAY_LENGTH_2];
extern char display_line_3[LCD_DISPLAY_LENGTH_3];
extern char display_line_4[LCD_DISPLAY_LENGTH_4];
extern char *display_1;
extern char *display_2;
extern char *display_3;
extern char *display_4;
extern int current_selected_ADC_channel;
extern int left_detector_reading;
extern int right_detector_reading;
extern int thumb_wheel_reading;
char posL1;
char posL2;
char posL3;
char posL4;
char size_count;
char big;


//===============================================================================
// void main (void)
//
// Description:         This is the "entry point" for program execution. As a
//                      result, it calls all initialization functions, and enters
//                      the main program loop. The operating system is "background -
//                      foreground", where the foreground consists of interrupts
//                       from the port pins.
//
// Arguments:           none
//
// Local Vars:          none
//
// Globals Used:        display_1
//                      display_2
//                      display_3
//                      display_4
//                      posL1
//                      posL2
//                      posL3
//                      posL4
//                      big
//                      movement_flag
//
// David Gietzen, February 2016
//===============================================================================

void main(void){

  //The beginning of main is a good place to put all initial configuration
  //functions that only need to be called once.

  Init_Ports();                              // Initialize Ports
  Init_Clocks();                             // Initialize Clock System
  Init_Conditions();
  Time_Sequence = RESET;
  Init_Timers();                             // Initialize Timers
  one_msec_sleep(SLEEP_TIME);                // 250 msec delay for the clock to settle
  Init_LCD();                                // Initialize LCD
  Init_ADC();
  Init_Motors();


  display_1 = "";
  posL1 = RESET;
```

```
    display_2 = "";
    posL2 = RESET;
    display_3 = "";
    posL3 = RESET;
    display_4 = "";
    posL4 = RESET;

    big = RESET;

    Display_Process();



    //Simply calibrate the IR detectors and then call the project 5 performance.

    one_msec_sleep(DISPLAY_INTERVAL);
    IR_LED_white_calibrate();

    while (ALWAYS)
    {

      //This loop is where code that performs the main actions should be placed.
      //In this case, the car is configured to do a performance, so a function
      //that makes the car move is called in this function.


      IR_LED_manual_calibrate();
      project_5_performance();

      Display_Process();
      one_msec_sleep (DISPLAY_INTERVAL);
    }
  }
```

8.2. ports.c

Figure 13 – Init_Ports Flow Chart




```
//------------------------------------------------------------------------------
//
//  Description: This file contains the initialization for all port pins
//
//
//  Calvin Schmidt
//  Feb 2016
//  Built with IAR Embedded Workbench Version: V7.3.1.3987 (6.40.1)
//------------------------------------------------------------------------------

//------------------------------------------------------------------------------

#include  "msp430.h"
#include  "functions.h"
#include  "macros.h"

void Init_Ports(void){
//------------------------------------------------------------------------------
//
// Description: This file contains the function to call all port initialization functions
//
// Passed: No variables passed
// Locals: No local variables
// Returned: No values returned
//
//
//  Calvin Schmidt
//  Feb 2016
```

```
//  Built with IAR Embedded Workbench Version: V7.3.1.3987 (6.40.1)
//------------------------------------------------------------------------------

//------------------------------------------------------------------------------
  Init_Port1();
  Init_Port2();
  Init_Port3();
  Init_Port4();
  Init_PortJ();
}

void Init_Port1(void){
//------------------------------------------------------------------------------
//Configure Port 1
// SMCLK_OUT (0x01) // SMCLK Out signal
// V_DETECT_R (0x01) //
// V_DETECT_L (0x02) //
// IR_LED (0x04) //
// V_THUMB (0x08) //
// SPI_CS_LCD (0x10) // LCD Chip Select
// RESET_LCD (0x20) // LCD Reset
// SIMO_B (0x40) // SPI mode - slave in/master out of USCI_B0
// SOMI_B (0x80) // SPI mode - slave out/master in of USCI_B0
//
// Passed: No variables passed
// Locals: No local variables
// Returned: No values returned
//
//  Calvin Schmidt
//  Feb 2016
//  Built with IAR Embedded Workbench Version: V7.3.1.3987 (6.40.1)
//------------------------------------------------------------------------------
 P1SEL0 = CLEAR; // P1 set as I/0
 P1SEL1 = CLEAR; // P1 set as I/0
 P1DIR = CLEAR; // Set P1 direction to input
 // Port 1.0
 P1SEL0 |= V_DETECT_R; // V_DETECT_R selected
 P1SEL1 |= V_DETECT_R; // V_DETECT_R selected
 // Port 1.1
 P1SEL0 |= V_DETECT_L; // V_DETECT_L selected
 P1SEL1 |= V_DETECT_L; // V_DETECT_L selected
 // Port 1.2
 P1SEL0 &= ~IR_LED; // IR_LED GPI/O selected
 P1SEL1 &= ~IR_LED; // IR_LED GPI/O selected
 P1OUT |= IR_LED; // P1 IR_LED Port Pin set low
 P1DIR |= IR_LED; // Set P1 IR_LED direction to output
 // Port 1.3
 P1SEL0 |= V_THUMB; // V_THUMB selected
 P1SEL1 |= V_THUMB; // V_THUMB selected
 // Port 1.4
 P1SEL0 &= ~SPI_CS_LCD; // SPI_CS_LCD GPI/O selected
 P1SEL1 &= ~SPI_CS_LCD; // SPI_CS_LCD GPI/O selected
 P1OUT |= SPI_CS_LCD; // P1 SPI_CS_LCD Port Pin set high
 P1DIR |= SPI_CS_LCD; // Set SPI_CS_LCD output direction
 // Port 1.5
 P1SEL0 &= ~RESET_LCD; // RESET_LCD GPI/O selected
 P1SEL1 &= ~RESET_LCD; // RESET_LCD GPI/O selected
 P1OUT &= ~RESET_LCD; // RESET_LCD Port Pin set low
 P1DIR |= RESET_LCD; // Set RESET_LCD output direction
 // Port 1.6
 P1SEL0 &= ~SIMO_B; // SIMO_B selected
 P1SEL1 |= SIMO_B; // SIMO_B selected
 P1DIR |= SIMO_B; // SIMO_B set to Output
 // Port 1.7
 P1SEL0 &= ~SOMI_B; // SOMI_B is used on the LCD
 P1SEL1 |= SOMI_B; // SOMI_B is used on the LCD
 P1DIR &= ~SOMI_B; // SOMI_B set to Input
 P1REN |= SOMI_B; // Enable pullup resistor
//---------------------------------------------------------------
}
```

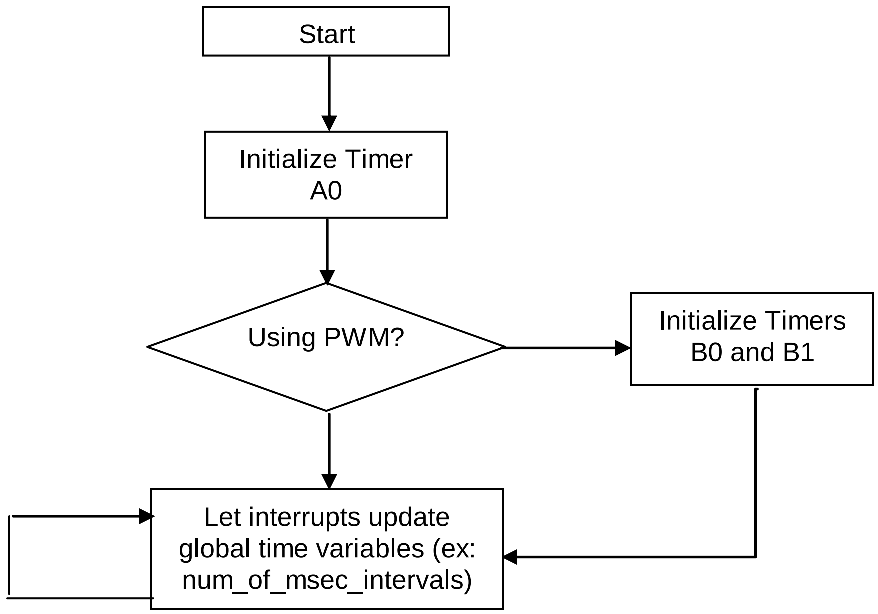
```
void Init_Port2(void){
//------------------------------------------------------------------------------
//Configure Port 2
// USB_TXD (0x01) // PIN 0
// USB_RXD (0x02) // PIN 1
// SPI_SCK (0x04) // PIN 2
// CPU_TXD (0x20) // PIN 5
// CPU_RXD (0x40) // PIN 6
//
// Passed: No variables passed
// Locals: No local variables
// Returned: No values returned
//
//  Calvin Schmidt
//  Feb 2016
//  Built with IAR Embedded Workbench Version: V7.3.1.3987 (6.40.1)
//------------------------------------------------------------------------------
 P2SEL0 = CLEAR; // P1 set as I/0
 P2SEL1 = CLEAR; // P1 set as I/0
 P2DIR = CLEAR; // Set P1 direction to input
 // Port 2.0
 P2SEL1 |= USB_TXD; // USB_TXD selected
 P2SEL0 &= ~USB_TXD; // USB_TXD selected
 // Port 2.1
 P2SEL1 |= USB_RXD; // USB_RXD selected
 P2SEL0 &= ~USB_RXD; // USB_RXD selected
 // Port 2.2
 P2SEL1 |= SPI_SCK; // IR_LED GPI/O selected
 P2SEL0 &= ~SPI_SCK; // IR_LED GPI/O selected
 P2OUT  |= SPI_SCK; // SPI_SCK Port pin set high
 // Port 2.5
 P2SEL1 |= CPU_TXD; // SPI_CS_LCD GPI/O selected
 P2SEL0 &= ~CPU_TXD; // SPI_CS_LCD GPI/O selected
 // Port 2.6
 P2SEL1 |= CPU_RXD; // P1 SPI_CS_LCD Port Pin set high
 P2SEL0 &= ~CPU_RXD; // Set SPI_CS_LCD output direction
//----------------------------------------------------------------
}

void Init_Port3(void){
//------------------------------------------------------------------------------
//Configure Port 3
// EXCEL_X (0x01) // PIN 0
// EXCEL_Y (0x02) // PIN 1
// EXCEL_Z (0x04) // PIN 2
// LCD_BACKLITE (0x08) // PIN 3
// R_FORWARD (0x10) // PIN 4
// R_REVERSE (0x20) // PIN 5
// L_FORWARD (0x40) // PIN 6
// L_REVERSE (0x80) // PIN 7
//
// Passed: No variables passed
// Locals: No local variables
// Returned: No values returned
//
//  Calvin Schmidt
//  Feb 2016
//  Built with IAR Embedded Workbench Version: V7.3.1.3987 (6.40.1)
//------------------------------------------------------------------------------
 P3SEL1 = CLEAR; // P1 set as I/0
 P3SEL0 = CLEAR; // P1 set as I/0
 P3DIR = CLEAR; // Set P1 direction to input
 // Port 3.0
 P3SEL1 &= ~EXCEL_X; // EXCEL_X selected
 P3SEL0 &= ~EXCEL_X; // EXCEL_X selected
 P3DIR  &= ~EXCEL_X; // Set EXCEL_X set direction to Input
 P3OUT  &= ~EXCEL_X; // P3 EXCEL_X Port Pin set low
 P3REN  &= ~EXCEL_X; // pullup resistor
 // Port 3.1
```

```
 P3SEL1 &= ~EXCEL_Y; // EXCEL_Y selected
 P3SEL0 &= ~EXCEL_Y; // EXCEL_Y selected
 P3DIR  &= ~EXCEL_Y; // Set EXCEL_Y set direction to Input
 P3OUT  &= ~EXCEL_Y; // P3 EXCEL_X Port Pin set low
 P3REN  &= ~EXCEL_Y; // pullup resistor
 // Port 3.2
 P3SEL1 &= ~EXCEL_Z; // EXCEL_Z selected
 P3SEL0 &= ~EXCEL_Z; // EXCEL_Z selected
 P3DIR  &= ~EXCEL_Z; // Set EXCEL_Z set direction to Input
 P3OUT  &= ~EXCEL_Z; // P3 EXCEL_X Port Pin set low
 P3REN  &= ~EXCEL_Z; // pullup resistor
 // Port 3.3
 P3SEL1 &= ~LCD_BACKLITE; // LCD_BACKLITE selected
 P3SEL0 &= ~LCD_BACKLITE; // LCD_BACKLITE selected
 P3DIR  |= LCD_BACKLITE; // Set LCD_BACKLITE direction to output
 P3OUT  &= ~LCD_BACKLITE; // P3 LCD_BACKLITE Port Pin set low
 // Port 3.4
 P3SEL1 &= ~R_FORWARD; // R_FORWARD selected
 P3SEL0 |= R_FORWARD; // R_FORWARD selected
 P3DIR  |= R_FORWARD; // Set R_FORWARD direction to Output
 P3OUT  &= ~R_FORWARD; //P3 R_FORWARD Port Pin set low
 // Port 3.5
 P3SEL1 &= ~R_REVERSE; // R_REVERSE selected
 P3SEL0 |= R_REVERSE; // R_REVERSE selected
 P3DIR  |= R_REVERSE; // Set R_REVERSE direction to Output
 P3OUT  &= ~R_REVERSE; //P3 R_REVERSE Port Pin set low
 // Port 3.6
 P3SEL1 &= ~L_FORWARD; // L_FORWARD selected
 P3SEL0 |= L_FORWARD; // L_FORWARD selected
 P3DIR  |= L_FORWARD; // Set L_FORWARD direction to Output
 P3OUT  &= ~L_FORWARD; //P3 L_FORWARD Port Pin set low
 // Port 3.7
 P3SEL1 &= ~L_REVERSE; // L_REVERSE selected
 P3SEL0 |= L_REVERSE; // L_REVERSE selected
 P3DIR  |= L_REVERSE; // Set L_REVERSE direction to Output
 P3OUT  &= ~L_REVERSE; //P3 L_REVERSE Port Pin set low
//----------------------------------------------------------------
}

 void Init_Port4(void){
//------------------------------------------------------------------------------
// Configure Port 4 Pins
// SW1  (0x01) // Switch 1
// SW2  (0x02) // Switch 2
//
// Passed: No variables passed
// Locals: No local variables
// Returned: No values returned
//
//  Calvin Schmidt
//  Feb 2016
//  Built with IAR Embedded Workbench Version: V7.3.1.3987 (6.40.1)
//------------------------------------------------------------------------------
 // Port 4.0
 P4SEL1 &= ~SW1; // SW1 selected
 P4SEL0 &= ~SW1; // SW1 selected
 P4DIR  &= ~SW1; // Set SW1 direction to Input
 P4OUT  |= SW1; //P4 SW1 Port Pin set high
 P4REN  |= SW1; // enabled pullup resistor
 P4IES |= SW1; // SW1 Hi/Lo edge interrupt
 P4IFG &= ~SW1; // IFG SW1 cleared
 P4IE |= SW1; // SW1 interrupt Enabled
 // Port 4.1
 P4SEL1 &= ~SW2; // SW2 selected
 P4SEL0 &= ~SW2; // SW2 selected
 P4DIR  &= ~SW2; // Set SW2 direction to Input
 P4OUT  |= SW2; //P4 SW2 Port Pin set high
 P4REN  |= SW2; // enabled pullup resistor
 P4IES |= SW2; // SW2 Hi/Lo edge interrupt
 P4IFG &= ~SW2; // IFG SW2 cleared
```

```
P4IE |= SW2; // SW2 interrupt enabled
//------------------------------------------------------------------------------
}


void Init_PortJ(void){
//------------------------------------------------------------------------------
// Configure Port J Pins
// IOT_WAKEUP             (0x01) //
// IOT_FACTORY            (0x02) //
// IOT_STA_MINIAP         (0x04) //
// IOT_RESET              (0x08) //
// XINR                   (0x10) // XINR
// XOUTR                  (0x20) // XOUTR
//
// Passed: No variables passed
// Locals: No local variables
// Returned: No values returned
//
//  Calvin Schmidt
//  Feb 2016
//  Built with IAR Embedded Workbench Version: V7.3.1.3987 (6.40.1)
//------------------------------------------------------------------------------
 PJSEL0 = CLEAR; // PJ set as I/0
 PJSEL1 = CLEAR; // PJ set as I/0
 PJDIR = CLEAR; // Set PJ direction to output
 // Port J.0
 PJSEL0 &= ~IOT_WAKEUP;
 PJSEL1 &= ~IOT_WAKEUP;
 PJOUT  &= ~IOT_WAKEUP;
 PJDIR  |= IOT_WAKEUP; // Set PJ Pin 1 direction to output
 // Port J.1
 PJSEL0 &= ~IOT_FACTORY;
 PJSEL1 &= ~IOT_FACTORY;
 PJOUT  &= ~IOT_FACTORY;
 PJDIR  |= IOT_FACTORY; // Set PJ Pin 2 direction to output
 // Port J.2
 PJSEL0 &= ~IOT_STA_MINIAP;
 PJSEL1 &= ~IOT_STA_MINIAP;
 PJOUT  &= ~IOT_STA_MINIAP;
 PJDIR  |= IOT_STA_MINIAP; // Set PJ Pin 3 direction to output
 // Port J.3
 PJSEL0 &= ~IOT_RESET;
 PJSEL1 &= ~IOT_RESET;
 PJDIR  |= IOT_RESET; // Set P3 Pin 4 direction to output
 PJOUT  &= ~IOT_RESET;
// XT1 Setup
// PJSEL0 |= XINR;
// PJSEL0 |= XOUTR;
//------------------------------------------------------------------------------
}
```

8.3. timers.c

Figure 14 - Timers Flow Chart



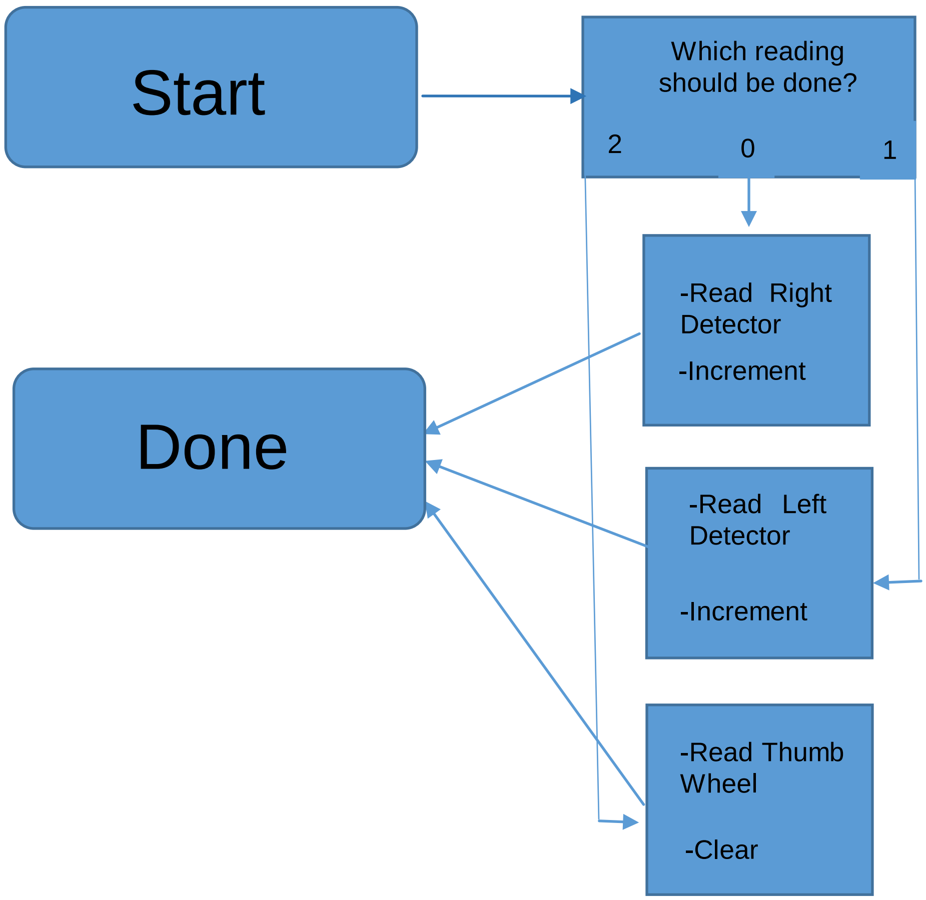

```
//------------------------------------------------------------------------------
//  File Name : timers.c
//
//  Description: Various functions dealing with timers
//
//  Justin Parsons
//  Feb 2016
//  Built with IAR Embedded Workbench Version: V7.3.1.3987 (6.40.1)
//------------------------------------------------------------------------------
#include  "functions.h"
#include  "macros.h"
#include  "msp430.h"
#include  "globals.h"

unsigned int right_forward_rate;
unsigned int right_reverse_rate;
unsigned int left_forward_rate;
unsigned int left_reverse_rate;

void Init_Timers(void){
//------------------------------------------------------------------------------
// Timer Configurations
//  Passed : no variables passed
//  Locals : no variables declared
//  Returned : no values returned
//  Globals : no globals used
//
//  Justin Parsons
//  Feb 2016
//  Built with IAR Embedded Workbench Version: V7.3.1.3987 (6.40.1)
//------------------------------------------------------------------------------
  Init_Timer_A0(); //
//  Init_Timer_A1(); //
//  Init_Timer_B0(); //
```

```
  Init_Timer_B1();
  Init_Timer_B2();   //  Required for provided compiled code to work
//------------------------------------------------------------------------------
}
void Init_Timer_A0(void){
//------------------------------------------------------------------------------
//  Timer A0 initialization sets up A0_0
//  Passed : no variables passed
//  Locals : no variables declared
//  Returned : no values returned
//  Globals : no globals used
//
//  Justin Parsons
//  Feb 2016
//  Built with IAR Embedded Workbench Version: V7.3.1.3987 (6.40.1)
//------------------------------------------------------------------------------
  TA0CTL = TASSEL__SMCLK;       // SMCLK source
  TA0CTL |= TACLR;              // Resets TA0R, clock divider, count direction
  TA0CTL |= MC__UP;             // Up mode
  TA0CTL |= ID__2;              // Divide clock by 2
  TA0CTL &= ~TAIE;              // Disable Overflow Interrupt
  TA0CTL &= ~TAIFG;             // Clear Overflow Interrupt flag
  TA0EX0 = TAIDEX_7;            // Divide clock by an additional 8
  TA0CCR0 = TA0CCR0_INTERVAL;   // CCR0, interrupt every 5 msec
  TA0CCR1 = TA0CCR1_INTERVAL;   // CCR1, interrut every 1 msec
  TA0CCTL0 |= CCIE;             // CCR0 enable interrupt
}

void Init_Timer_B1(void) {
//------------------------------------------------------------------------------
//  Timer B1 initialization sets Up PWM on right wheel
//  Passed : no variables passed
//  Locals : no variables declared
//  Returned : no values returned
//  Globals :   unsigned int right_forward_rate
//              unsigned int right_reverse_rate
//
//  Justin Parsons
//  March 2016
//  Built with IAR Embedded Workbench Version: V7.3.1.3987 (6.40.1)
//------------------------------------------------------------------------------
  TB1CTL = TBSSEL__SMCLK;       // SMCLK
  TB1CTL |= MC_1;               // Up Mode
  TB1CTL |= TBCLR;              // Clear TAR

  right_forward_rate = OFF;      // Set Right Forward Off
  right_reverse_rate = OFF;      // Set Right Reverse Off

  TB1CCR0 = WHEEL_PERIOD;       // PWM Period
  TB1CCTL1 = OUTMOD_7;          // CCR1 reset/set
  TB1CCR1 = right_forward_rate; // P3.4 Right Forward PWM duty cycle
  TB1CCTL2 = OUTMOD_7;          // CCR2 reset/set
  TB1CCR2 = right_reverse_rate; // P3.5 Right Reverse PWM duty cycle
//------------------------------------------------------------------------------
}

void Init_Timer_B2(void) {
//------------------------------------------------------------------------------
//  Timer B2 initialization sets Up PWM on left wheel
//  Passed : no variables passed
//  Locals : no variables declared
//  Returned : no values returned
//  Globals :   unsigned int left_forward_rate
//              unsigned int left_reverse_rate
//
//  Justin Parsons
//  March 2016
//  Built with IAR Embedded Workbench Version: V7.3.1.3987 (6.40.1)
//------------------------------------------------------------------------------
```

```
  TB2CTL = TBSSEL__SMCLK;       // SMCLK
  TB2CTL |= MC_1;               // Up Mode
  TB2CTL |= TBCLR;              // Clear TAR

  left_forward_rate = OFF;      // Set Left Forward Off
  left_reverse_rate = OFF;      // Set Left Reverse Off

  TB2CCR0 = WHEEL_PERIOD;       // PWM Period
  TB2CCTL1 = OUTMOD_7;          // CCR1 reset/set
  TB2CCR1 = left_forward_rate;  // P3.6 Left Forward PWM duty cycle
  TB2CCTL2 = OUTMOD_7;          // CCR2 reset/set
  TB2CCR2 = left_reverse_rate;  // P3.7 Left Reverse PWM duty cycle
//------------------------------------------------------------------------------
}

void Five_msec_Delay(unsigned int delay){
//------------------------------------------------------------------------------
//  Creates a five second delay using a timer intterrupt
//
//  Passed : unsigned int delay          // delay * 5 msec = time delay
//  Locals : no variables declared
//  Returned : no values returned
//  Globals :
//    volatile unsigned int inc_5msec   // interrupt increments this value
//
//  Justin Parsons
//  Feb 2016
//  Built with IAR Embedded Workbench Version: V7.3.1.3987 (6.40.1)
//------------------------------------------------------------------------------
  inc_5msec = CLEAR_REGISTER;   // Reset increment
  while(delay > inc_5msec){}   // stay in loop until delay is over
}
```

8.4. adc.c

Figure 15 – ADCISR_10 Flow Chart



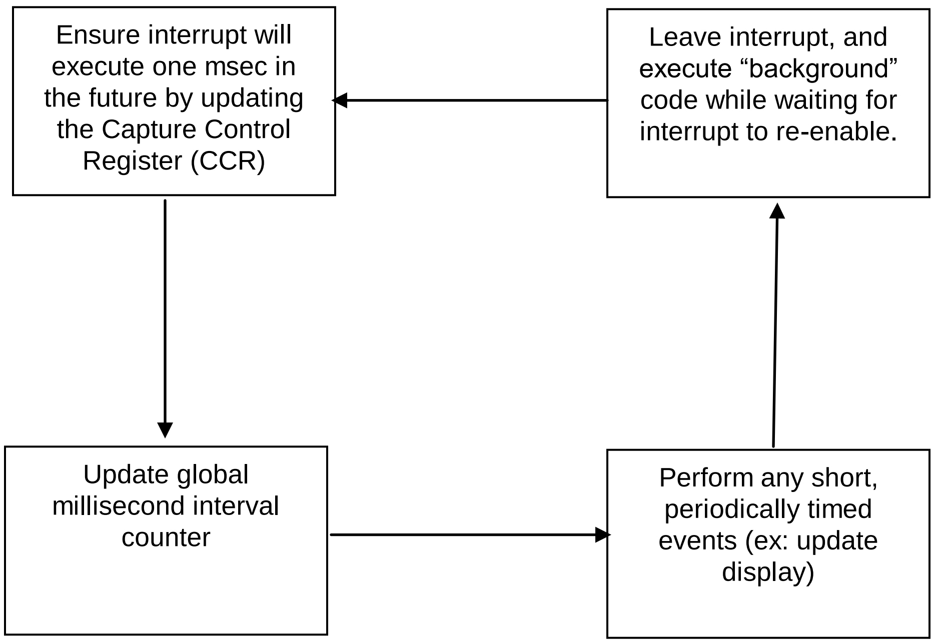

```
//------------------------------------------------------------------------------
//
//  Description: This file contains ADC.c
//
//  Jared Abell
//  Mar 2016
//  Built with IAR Embedded Workbench Version: V7.3.1.3987 (6.40.1)
//------------------------------------------------------------------------------

unsigned int ADC_Thumb;
unsigned int channel;
unsigned int leftdetector;
unsigned int rightdetector;

#include  "msp430.h"
#include  "functions.h"
#include  "macros.h"
#include "msp430fr5739.h"
```

```
//==========================================================================
// Function name: ADC10_VECTOR
//
// Description: ISR for the ADC
//
// Author: Jared Abell
// Date: Feb 16
// Compiler: Built with IAR Embedded Workbench Version: V7.3.1.3987 (6.40.1)
//==========================================================================

//------------------------------------------------------------------------------
// ADC10 interrupt service routine
// ADC_Right_Detector; // A00 ADC10INCH_0 - P1.0
// ADC_Left_Detector; // A01 ADC10INCH_1 - P1.1
// ADC_Thumb; // A03 ADC10INCH_3 - P1.3
// ADC_Temp; // A10 ADC10INCH_10 - Temperature REF module
// ADC_Bat; // A11 ADC10INCH_11 - Internal
#pragma vector=ADC10_VECTOR
__interrupt void ADC10_ISR(void){
 switch(__even_in_range(ADC10IV,ADCswitch)) {
 case nointerrupt: break; // No interrupt
 case cro: break; // conversion result overflow
 case cto: break; // conversion time overflow
 case adc10hi: break; // ADC10HI
 case adc10lo: break; // ADC10LO
 case adc10in: break; // ADC10IN
 case maincaseADC:
   // Need this to change the ADC10INCH_x value.
       ADC10CTL0 &= ~ADC10ENC; // Turn off the ENC bit of the ADC10CTL0
      switch (channel){
            case caseA1:
            ADC10MCTL0 = ADC10INCH_1; // Next channel A1
            rightdetector = ADC10MEM0; // Current Channel result for A0
            channel=channel+channel_increment;
            break;
            case caseA3:
            ADC10MCTL0 = ADC10INCH_3; // Next channel A3
            leftdetector = ADC10MEM0; // Current Channel result for A1
            channel=channel+channel_increment;
            break;
            case caseA0:
            ADC10MCTL0 = ADC10INCH_0; // Next channel A0
            ADC_Thumb = ADC10MEM0; // Current Channel result for A3
            channel=clear;
            break;
      default:
       break;
}
ADC10CTL0 |= ADC10ENC; // Turn on the ENC bit of the ADC10CTL0
ADC10CTL0 |= ADC10SC; // Start next sample.
break;
 }
}
//------------------------------------------------------------------------------


//==========================================================================
// Function name: ADC_Process
//
// Description: start the sampling and conversion
//
// Author: Jared Abell
// Date: Feb 16
// Compiler: Built with IAR Embedded Workbench Version: V7.3.1.3987 (6.40.1)
//==========================================================================
//------------------------------------------------------------------------------
void ADC_Process(void){
 while (ADC10CTL1 & BUSY); // Wait if ADC10 core is active
 ADC10CTL0 |= ADC10ENC + ADC10SC; // Sampling and conversion start
}
```

```
//------------------------------------------------------------------------------
```

8.5. interrupts_timer.c

Figure 16 - Timer0_A0_ISR

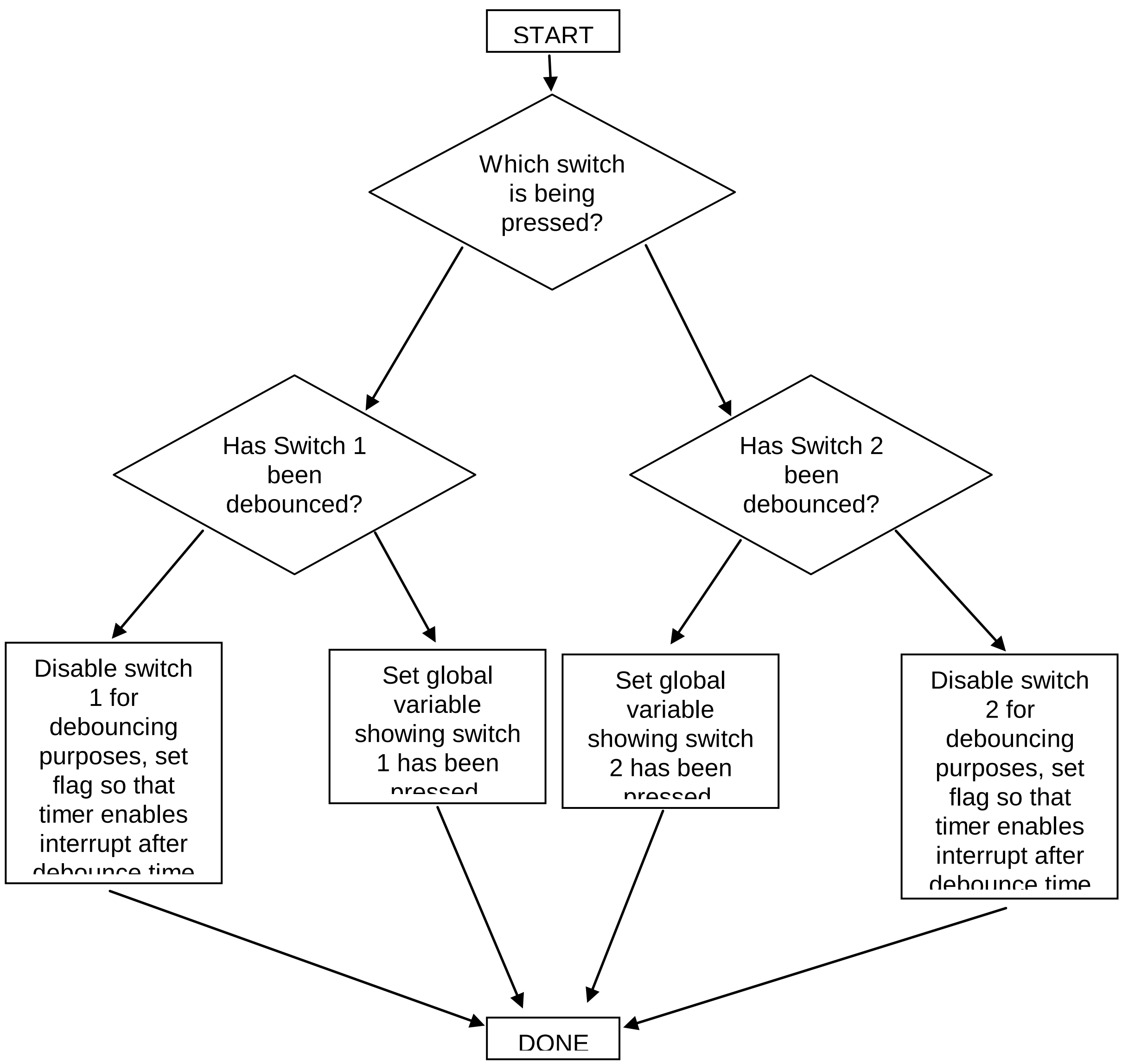



```
//------------------------------------------------------------------------------
//
//  Description: This file contains the interrupts for timers
//
//
//  Calvin Schmidt
//  Feb 2016
//  Built with IAR Embedded Workbench Version: V7.3.1.3987 (6.40.1)
//------------------------------------------------------------------------------

#include  "msp430.h"
#include  "functions.h"
#include  "macros.h"

//Display Globals
extern char display_line_1[LENGTHOFDISPLAY];
extern char display_line_2[LENGTHOFDISPLAY];
extern char display_line_3[LENGTHOFDISPLAY];
extern char display_line_4[LENGTHOFDISPLAY];
extern char *display_1;
extern char *display_2;
extern char *display_3;
extern char *display_4;
extern char posL1;
extern char posL2;
extern char posL3;
extern char posL4;
// Timer counter globals
extern unsigned int Timer_A0_Counter;
extern unsigned int Timer_Counter;
extern int Current_Time;
extern int msec_count;
extern int enable_timer_counter;
```

```
extern char msec_display[6];
// ADC globals
extern int ADC_Thumb;
extern int ADC_Left_Detector;
extern int ADC_Right_Detector;
extern char adc_char[5];



//------------------------------------------------------------------------------
// TimerA0 0 Interrupt handler
#pragma vector = TIMER0_A0_VECTOR
__interrupt void Timer0_A0_ISR(void){
//------------------------------------------------------------------------------
// Description: Interrupt that will occur every 50ms, as set by the clock speed and
TA0CCR0 interval macro.
//              Performs function calls based on current Timer_Counter phase to perform
movement for Project 4
//
//
//  Calvin Schmidt
//  Feb 2016
//  Built with IAR Embedded Workbench Version: V7.3.1.3987 (6.40.1)
//------------------------------------------------------------------------------
 TA0CCR0 += TA0CCR0_INTERVAL; // Add Offset to TACCR0
 Timer_Counter++; // Number of times timer interrupt has been called

 if (enable_timer_counter) {
   dec_to_char(Timer_Counter * INTERVAL);
   display_1 = &msec_display[0];
   Display_Process();
 }
}
//-----------------------------------------------------------------------------


//-----------------------------------------------------------------------------
// TimerA0 1-2, Overflow Interrupt Vector (TAIV) handler
#pragma vector=TIMER0_A1_VECTOR
__interrupt void TIMER0_A1_ISR(void){
  switch(__even_in_range(TA0IV,14)){
  case 0: break; // No interrupt
  case 2: // CCR1 not used
    msec_count++;
    TA0CCR1 += TA0CCR1_INTERVAL; // Add Offset to TACCR1
  break;
  case 4: // CCR2 not used
    TA0CCR2 += TA0CCR2_INTERVAL; // Add Offset to TACCR2
  break;
  case 14: // overflow
  break;
  default: break;
}
}
//-----------------------------------------------------------------------------
```

8.6. interrupts_ports.c

Figure 17 – Switches ISR Flow Chart




```
//------------------------------------------------------------------------------
//
//  interrupts_ports.c
//
//  Description: This file has the responsibility of handling all port-based interrupts.
//               This means it includes code for handling switches and peripherals,
//               as well as functions necessary for logic within their interrupts.
//
//  David Gietzen
//  Feb 2016
//  Built with IAR Embedded Workbench Version: V7.3.1.3987 (6.40.1)
//------------------------------------------------------------------------------
```

```
#include "msp430.h"
#include "functions.h"
#include "macros.h"

volatile char switch_1_debounce_flag = RESET;
volatile unsigned int switch_1_debounce_time;

volatile char switch_2_debounce_flag = RESET;
volatile unsigned int switch_2_debounce_time;

volatile int switch_1_pressed = RESET; //This holds the number of times switch 1 has been
pressed.
                                        //Decrement after reading this.
volatile int switch_2_pressed = RESET; //This holds the number of times switch 2 has been
pressed.
                                        //Decrement after reading this.

extern char *display_1;
extern char *display_2;
extern char *display_3;
extern char *display_4;

//===============================================================================
// void Switches_ISR (void)
//
// Description:          This interrupt service routine processes what should
//                       happen when the switches are pressed. As a result of
//                       switches having a noisy signal, they need to be "debounced"
//                       so that one button press does not count as multiple ones,
//                       and so the timer interrupt must be entered several times
//                       to complete. Why not just call a wait() function inside
//                       this interrupt? Because that would make one button press
//                       block another.
//
//                       In summary, whatever you want a switch to do, place that
//                       at //PLACE SWITCH X ACTION HERE - BEGIN
//
// Arguments:            none
//
// Local Vars:           none
//
// Globals Used:         P4IFG
//                       switch_1_debounce_flag
//                       switch_1_debounce_time
//                       switch_2_debounce_flag
//                       switch_2_debounce_time
//                       display_4
//
// David Gietzen, February 2016
//===============================================================================

#pragma vector = PORT4_VECTOR
__interrupt void Switches_ISR(void)
{
  //The logic of this function is somewhat messy, but necessary. The MSP430 switches
  //produce multiple "pressed" signals whenever the switch is pressed just once.
  //So how do we fix that? We could just temporarily disable the switch and call a
  //wait() function here, but since the MSP430 has blocking interrupts, the timer
  //won't finish because the timer relies on a different interrupt to update! So,
  //how can we get around this? We can disable the switch interrupt, leave, and
  //then in the timer interrupt have it call this interrupt when the "debounce" time
  //finishes.

  //In summary, the logic here needs TWO SEPARATE interrupts to work, this one called
  //Switches_ISR (), and Timer0_A0_ISR(). So changes here must be checked against
  //that interrupt as well.


  if (P4IFG & SW1) //If switch 1 pressed
  {
```

```
    disable_switch_1_interrupt();       //Disable temporarily so spurious signals don't
trigger interrupt

    if (switch_1_debounce_flag == RESET) //Enter if debounce not started
    {
     switch_1_debounce_time = RESET;    //Reset the debounce timer

     switch_1_debounce_flag = ALWAYS;   //Tell the timer to start updating the debounce
time

     LCD_backlight_toggle();
    }

    else
    {
      LCD_backlight_toggle();

      //PLACE SWITCH 1 ACTION HERE - BEGIN

      display_4 = " Switch 1";

      switch_1_pressed += ALWAYS;

      //PLACE SWITCH 1 ACTION HERE - END

      switch_1_debounce_flag = RESET;   //Set so this interrupt re-enters correctly when
switch pressed

      clear_then_enable_switch_1_interrupt();
    }
  }


  if (P4IFG & SW2) //If switch 2 pressed
  {
    disable_switch_2_interrupt();

    if (switch_2_debounce_flag == RESET)
    {
     switch_2_debounce_time = RESET;

     switch_2_debounce_flag = ALWAYS;

     LCD_backlight_toggle();
    }

    else
    {
      LCD_backlight_toggle();

      //PLACE SWITCH 2 ACTION HERE - BEGIN

      display_4 = " Switch 2";

      switch_2_pressed += ALWAYS;

      //PLACE SWITCH 2 ACTION HERE - END

      switch_2_debounce_flag = RESET;

      clear_then_enable_switch_2_interrupt();
    }
  }

}



//What follows here are just “helper” functions that set port pins. They do not
//contain any complex logic.
```

```
//==============================================================================
// void disable_switch_1_interrupt (void)
//
// Description:         Effectively disables switch 1 doing anything, by disabling
//                      it's code in the port 4 interrupt.
//
// Arguments:           none
//
// Local Vars:          none
//
// Globals Used:        P4IE
//
// David Gietzen, February 2016
//==============================================================================

void disable_switch_1_interrupt ()
{
  P4IE  &= ~SW1;  //Disables interrupts execution
}



//==============================================================================
// void enable_switch_1_interrupt (void)
//
// Description:         Enables switch 1, allows switch 1's code to execute within
//                      port 4 interrupt
//
// Arguments:           none
//
// Local Vars:          none
//
// Globals Used:        P4IE
//
// David Gietzen, February 2016
//==============================================================================

void enable_switch_1_interrupt ()
{
  P4IE  |= SW1;         //Enable interrupts for switch 2
}



//==============================================================================
// void clear_switch_1_interrupt_flag (void)
//
// Description:         Clears any call to switch 1's code in port 4 interrupt.
//
// Arguments:           none
//
// Local Vars:          none
//
// Globals Used:        P4IFG
//
// David Gietzen, February 2016
//==============================================================================

void clear_switch_1_interrupt_flag ()
{
  P4IFG &= ~SW1;        //Clear any spurious interrupt calls
}



//==============================================================================
// void set_switch_1_interrupt_flag (void)
```

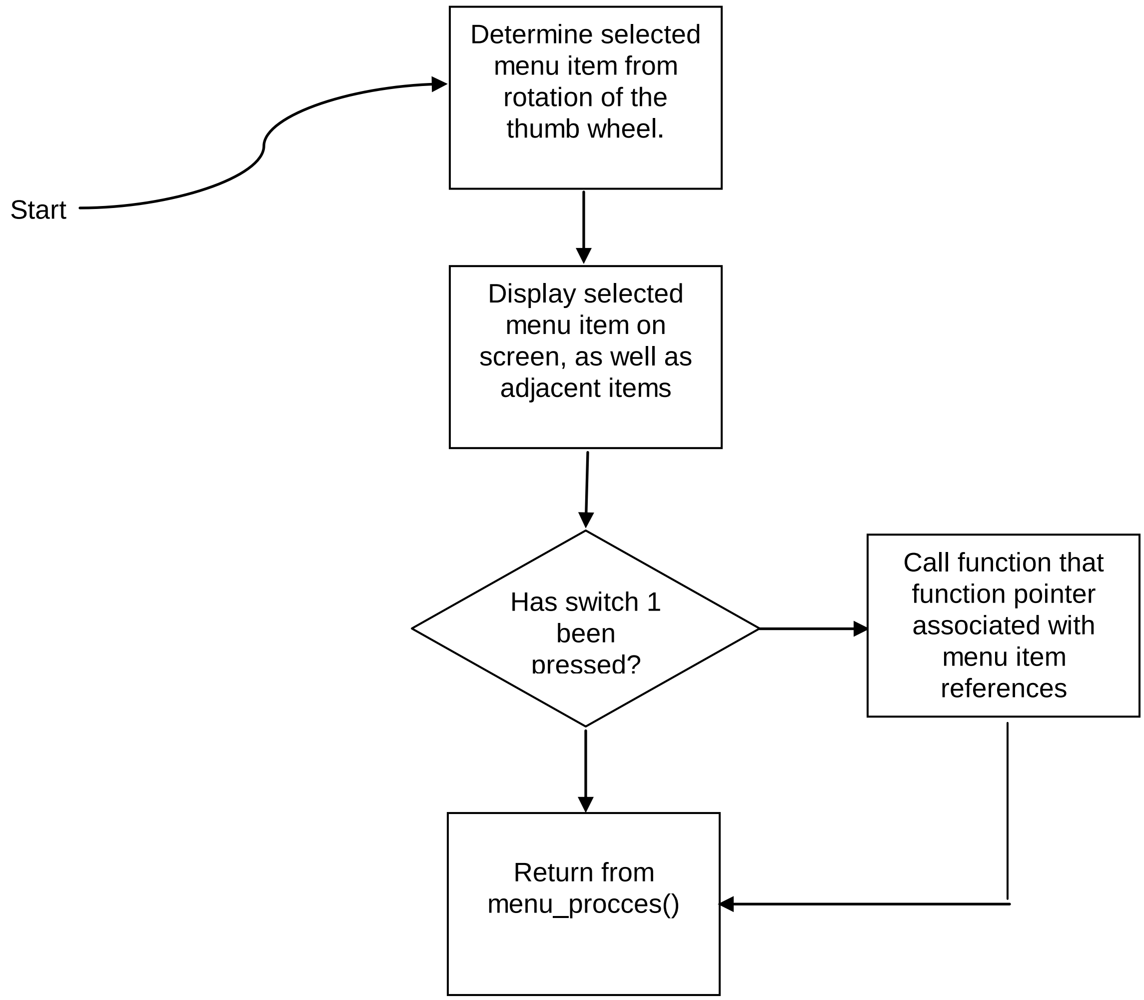
```
//
// Description:         Makes it so that switch 1's code in port 4 interrupt is
//                      immediately executed
//
// Arguments:           none
//
// Local Vars:          none
//
// Globals Used:        P4IFG
//
// David Gietzen, February 2016
//===============================================================================

void set_switch_1_interrupt_flag ()
{
  P4IFG |= SW1;
}



//===============================================================================
// void clear_then_enable_switch_1_interrupt (void)
//
// Description:         Clear the switch 1 interrupt flag, and then enable
//                      switch 1's code in port 4 interrupt.
//
// Arguments:           none
//
// Local Vars:          none
//
// Globals Used:        P4IFG
//                      P4IE
//
// David Gietzen, February 2016
//===============================================================================

void clear_then_enable_switch_1_interrupt ()
{
  clear_switch_1_interrupt_flag();
  enable_switch_1_interrupt();
}



//===============================================================================
// void disable_switch_2_interrupt (void)
//
// Description:         Effectively disables switch 2 doing anything, by disabling
//                      it's code in the port 4 interrupt.
//
// Arguments:           none
//
// Local Vars:          none
//
// Globals Used:        P4IE
//
// David Gietzen, February 2016
//===============================================================================

void disable_switch_2_interrupt ()
{
  P4IE  &= ~SW2;        //Disables interrupts execution
}



//===============================================================================
// void enable_switch_2_interrupt (void)
//
// Description:         Enables switch 2, allows switch 2's code to execute within
```

```
//                        port 4 interrupt
//
// Arguments:             none
//
// Local Vars:            none
//
// Globals Used:          P4IE
//
// David Gietzen, February 2016
//===============================================================================

void enable_switch_2_interrupt ()
{
  P4IE  |= SW2;           //Enable interrupts for switch 2
}



//===============================================================================
// void clear_switch_2_interrupt_flag (void)
//
// Description:           Clears any call to switch 1's code in port 4 interrupt.
//
// Arguments:             none
//
// Local Vars:            none
//
// Globals Used:          P4IFG
//
// David Gietzen, February 2016
//===============================================================================

void clear_switch_2_interrupt_flag ()
{
  P4IFG &= ~SW2;          //Clear any spurious interrupt calls
}



//===============================================================================
// void set_switch_2_interrupt_flag (void)
//
// Description:           Makes it so that switch 2's code in port 4 interrupt is
//                        immediately executed
//
// Arguments:             none
//
// Local Vars:            none
//
// Globals Used:          P4IFG
//
// David Gietzen, February 2016
//===============================================================================

void set_switch_2_interrupt_flag ()
{
  P4IFG |= SW2;
}



//===============================================================================
// void clear_then_enable_switch_2_interrupt (void)
//
// Description:           Clear the switch 2 interrupt flag, and then enable
//                        switch 2's code in port 4 interrupt.
//
// Arguments:             none
//
// Local Vars:            none
```

```
//
// Globals Used:         P4IFG
//                       P4IE
//
// David Gietzen, February 2016
//================================================================================

void clear_then_enable_switch_2_interrupt ()
{
  clear_switch_2_interrupt_flag();
  enable_switch_2_interrupt();
}
```

8.7. menu.c

Figure 18 - menu_process Flowchart




```
//Menu.c
//Provides menu subsystem

#include  "msp430.h"
#include  "functions.h"
#include  "macros.h"
#include  "string.h"

volatile extern int switch_1_pressed;
volatile extern int switch_2_pressed;
extern char display_line_1[LCD_DISPLAY_LENGTH_1];
extern char display_line_2[LCD_DISPLAY_LENGTH_2];
extern char display_line_3[LCD_DISPLAY_LENGTH_3];
extern char display_line_4[LCD_DISPLAY_LENGTH_4];
extern char *display_1;
extern char *display_2;
extern char *display_3;
extern char *display_4;

//===============================================================================
// struct Menu
//
// Description:         This struct creates a Menu object that is ready to be
//                      processed by menu_process(). It uses function pointers
//                      to provide actions associated with a menu items
//
// David Gietzen, April 2016
//===============================================================================
```

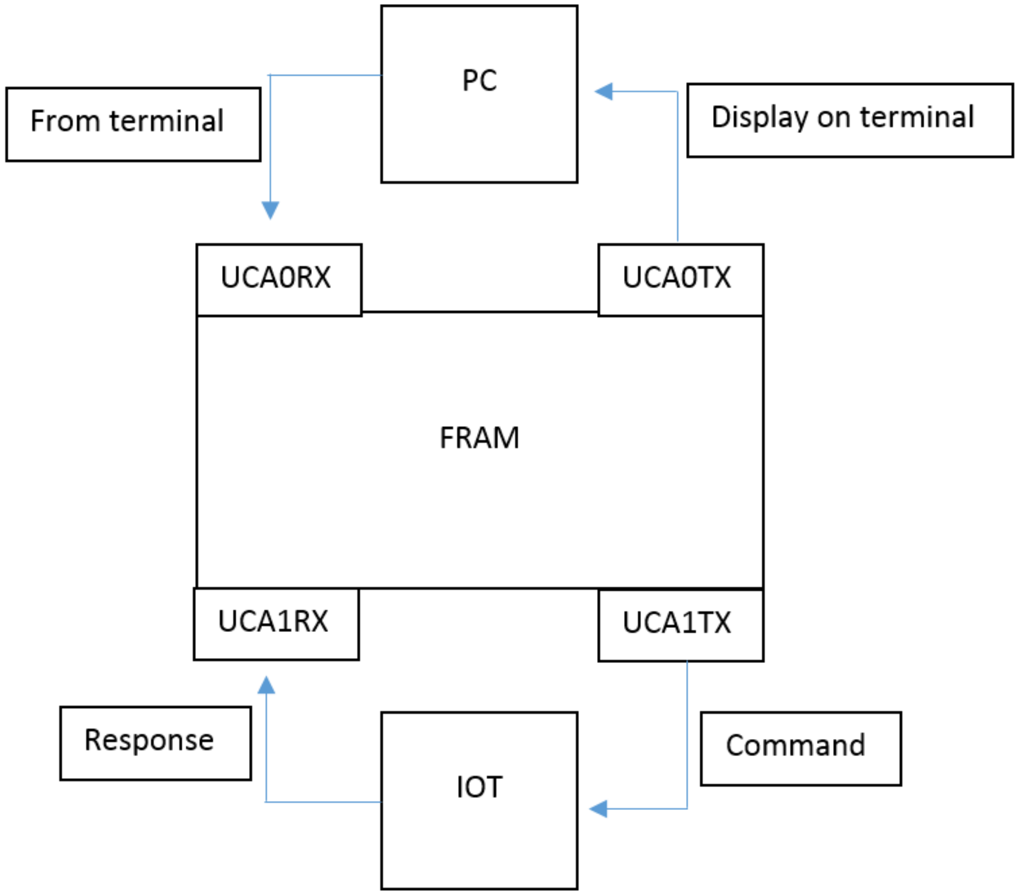
```
struct Menu
{
  char num_items_in_menu;
  char * line_str[MAX_ITEMS_IN_MENU];                   //Line to display on screen for the menu
item
  void (*line_function_ptr[MAX_ITEMS_IN_MENU])(void);       //Function pointer to function to call
when line is selected
};


//
//GLOBAL VARIABLES
//

char * resistor_strings[NUM_RESISTOR_STRINGS] = {" Black",       " 0",
                                                 " Brown",       " 1",
                                                 " Red",         " 2",
                                                 " Orange",      " 3",
                                                 " Yellow",      " 4",
                                                 " Green",       " 5",
                                                 " Blue",        " 6",
                                                 " Violet",      " 7",
                                                 " Gray",        " 8",
                                                 " White",       " 9"};


char * top_banner =     "RED AND WHITE  RED AND WHITE  ";
char * bottom_banner =  "WHITE AND RED  WHITE AND RED  ";

char * NCSU_song =              "          "
                     "We're the Red and White from State. "
                        "And we know we are the best. "
                          "A hand behind our back, "
                        "We can take on all the rest. "
                        "Come over the hill, Carolina. "
                       "Devils and Deacs stand in line. "
                      "The Red and White from N.C.State. "
                                "Go State! ";


struct Menu main_menu = {

  .num_items_in_menu = MAIN_MENU_LENGTH,

  .line_str = {"Resistors", "Shapes", "Song", "","","  David"," Gietzen","","How'd you","get
here?"},

  .line_function_ptr = {&resistor_demonstrate,
                        &shape_demonstrate,
                        &song_demonstrate,
                        NULL,
                        NULL,
                        NULL,
                        NULL,
                        NULL,
                        NULL,
                        NULL}
};


struct Menu shapes_menu = {

        .num_items_in_menu = SHAPES_MENU_LENGTH,

        .line_str = { "Circle",
                      "Square",
                      "Triangle",
                      "Octagon",
                      "Pentagon",
                      "Hexagon",
                      "Cube",
                      "Oval",
                      "Sphere",
                      "Cylinder"},

        .line_function_ptr = {NULL}
};
```

```
struct Menu * current_selected_menu = &main_menu;                    //Pointer that points to the
currently selected menu.



//===============================================================================
// void update_menu()
//
// Description:         This is a general function that displays and processes
//                      whatever menu is set as (current_selected_menu). It
//                      exists to simplify calling menu_process() out of this
//                      file
//
// Arguments:           none
//
// Local Vars:          none
//
// Globals Used:       current_selected_menu
//
// David Gietzen, April 2016
//===============================================================================

void update_menu()
{
  menu_process (current_selected_menu);
}


//===============================================================================
// void menu_process (void * menu_ptr_to_use)
//
// Description:         Processes user input for the given menu, and also displays
//                      it on screen.
//
// Arguments:           menu_ptr_to_use         - What menu to process
//
// Local Vars:          none
//
// Globals Used:        none
//
// David Gietzen, April 2016
//===============================================================================

void menu_process (void * menu_ptr_to_use)
{
  struct Menu * menu_ptr = (struct Menu *) menu_ptr_to_use;

  //Clear the screen

  screen_clear();

  //Check what item the thumb wheel is selecting

  int menu_index = get_thumb_index(menu_ptr->num_items_in_menu);

  if (menu_index >= menu_ptr->num_items_in_menu) //If the menu has less than the max number of
entries, then don't go past that.
  {
    menu_index = (menu_ptr->num_items_in_menu - ALWAYS);
  }



  //Display menu selection on screen.

  if (menu_index > RESET)
  {
    display_1 = menu_ptr->line_str[menu_index - ALWAYS];
  }

  display_2 = menu_ptr->line_str[menu_index];

  if (menu_index < (menu_ptr->num_items_in_menu - ALWAYS))
  {
    display_3 = menu_ptr->line_str[menu_index + ALWAYS];
```

```
    }

    //Process any button press

    if (was_switch_1_pressed())
    {
        flush_switches();

        if (menu_ptr->line_function_ptr[menu_index] != NULL)  //Do not execute NULL fucntions!
        {
            menu_ptr->line_function_ptr[menu_index]();      //Execute selected function
        }
    }


    //Update the screen if possible

    try_display_update();

    return;
}
```

8.8. interrupts_serial.c

Figure 19 – USCI_A0_ISR Flowchart




```
//------------------------------------------------------------------------------
//
//  Description: This file contains interrupt_serial.c The file handles the
//  UCA0/UCA1 serial ports. Included in this is the code for the recieve and
//  and transmit interrupts for both ports. The function of the code is to
//  handle communication between the PC, Fram board, and IOT Device.
//
//
//
//  Jared Abell
//  Apr 16
//  Built with IAR Embedded Workbench Version: V7.3.1.3987 (6.40.1)
//------------------------------------------------------------------------------


#include  "msp430.h"
#include  "functions.h"
#include  "macros.h"
#include "msp430fr5739.h"



extern char *display_1;
extern char *display_2;
extern char *display_3;
extern char *display_4;
extern char USB_Char_Rx[SMALL_RING_SIZE];
extern char USB_Char_Tx[SMALL_RING_SIZE];
extern char IOT_Char_Rx[SMALL_RING_SIZE];
extern char IOT_Char_Tx[SMALL_RING_SIZE];
extern unsigned int usb_tx_ring_wr;
extern unsigned int usb_rx_ring_wr;
extern unsigned int iot_tx_ring_wr;
```

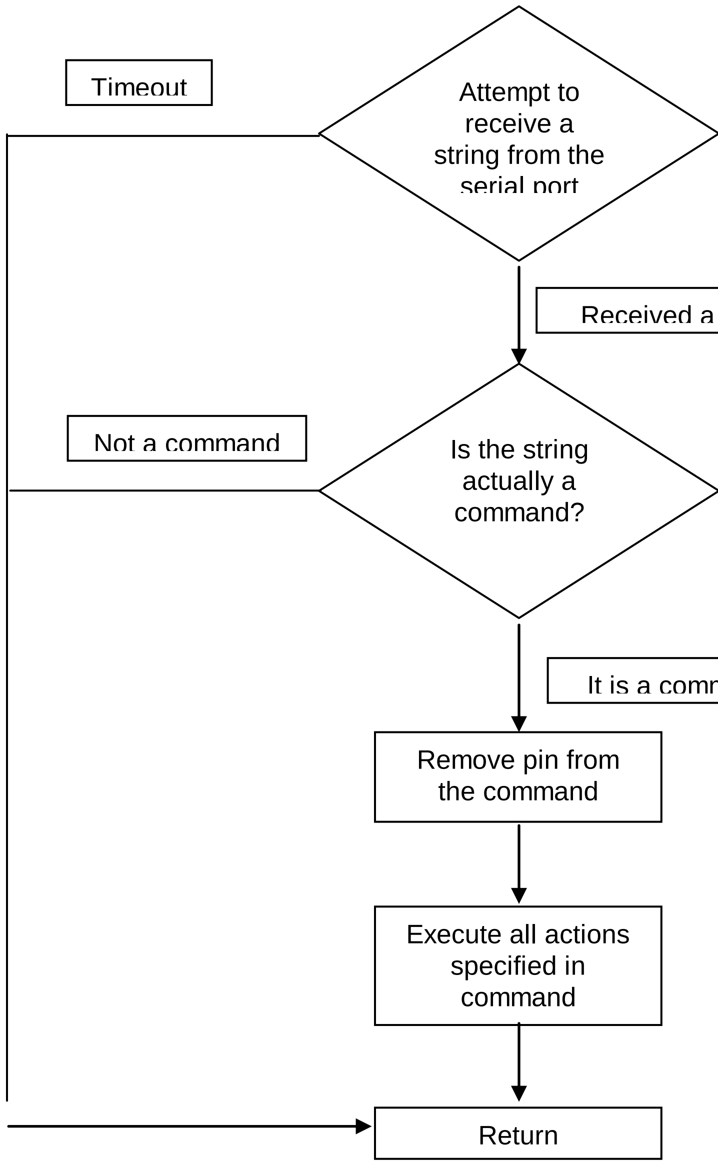
```
extern unsigned int iot_rx_ring_wr;
extern volatile unsigned int bootfinished;
unsigned int commandflag;
extern int stoptransmit;
int iotflag = clear;
volatile unsigned int messagedone;
extern unsigned int flagforline;



//=============================================================================
// Function name: USCI_A0_ISR
//
// Description: Sets up the UCA0 Transmit and Recieve buffers. USB_Char_Rx reads
// characters transmitted by the PC and IOT device. Code handling the array to
// make sure it is accurately received and then cleared out for the next
// instruction are handled in my main.
//
// Author: Jared Abell
// Date: Apr 16
// Compiler: Built with IAR Embedded Workbench Version: V7.3.1.3987 (6.40.1)
//=============================================================================

//-----------------------------------------------------------------------------
#pragma vector=USCI_A0_VECTOR
__interrupt void USCI_A0_ISR(void){
 unsigned int temp;
 switch(__even_in_range(UCA0IV,switchA0)){
 case clear: // Vector 0 - no interrupt
 break;
 case RXA0: // Vector 2 - RXIFG


 //If this is the first character from the PC, set the bootfinished flag
 if (!iotflag)
 {
 iotflag = ALWAYS;
 bootfinished = ALWAYS; //move to on for first character
 }

 temp = usb_rx_ring_wr;

 //Read the character in the USB Receive Array
 USB_Char_Rx[temp] = UCA0RXBUF; // RX -> USB_Char_Rx character


 //Signal a command is being sent if first character is a '.'
 if (USB_Char_Rx[temp] == '.')
 {
 commandflag = ALWAYS;
 }

 if (++usb_rx_ring_wr >= (SMALL_RING_SIZE))
 {
 usb_rx_ring_wr = BEGINNING; // Circular buffer back to beginning
 }

 break;
 case TXA0: // Vector 4 - TXIFG

 //Reads whatever is transmitted by the UCA0TXBUF to ensure
 //correct transmission
 temp = usb_tx_ring_wr;

 USB_Char_Tx[temp] = UCA0TXBUF;

 if (++usb_tx_ring_wr >= (SMALL_RING_SIZE))
 {
 usb_tx_ring_wr = BEGINNING; // Circular buffer back to beginning
 }

 break;
 default: break;
 }
}
//-----------------------------------------------------------------------------
```

```
//=============================================================================
// Function name: USCI_A1_ISR
//
// Description: Serial interrupt for IOT. This handles the communication
// involving the commands sent to the IOT device and the responses sent by the
// IOT Device. Anything sent to the IOT device uses UCA1TXBUF. The responses
// received from the IOT device are received throug UCA0RXBUF and immediately
// tranferred back to the fram board/PC
//
// Author: Jared Abell
// Date: Apr 16
// Compiler: Built with IAR Embedded Workbench Version: V7.3.1.3987 (6.40.1)
//=============================================================================
//-------------------------------------------------------------------------------

#pragma vector=USCI_A1_VECTOR
__interrupt void USCI_A1_ISR(void){
 unsigned int temp;
 switch(__even_in_range(UCA1IV,switchA1)){
 case clear: // Vector 0 - no interrupt
 break;
 case RXA1: // Vector 2 - RXIFG

 temp = iot_rx_ring_wr;

 //Read the character sent by IOT device
 IOT_Char_Rx[temp] = UCA1RXBUF; // RX -> USB_Char_Rx character

 //Set the command flag if first character is a '.'
 if (IOT_Char_Rx[temp] == '.')
 {
 commandflag = ALWAYS;

    //If second character is a '.', clear flagforline so car returns to IOT
    //control
    if(IOT_Char_Rx[temp+ALWAYS == '.')
        {
        flagforline = clear;
        }
 }

 //So long as boot is finished, send IOT responses back to FRAM board,PC
 if (bootfinished == ALWAYS)
 {
 UCA0TXBUF = UCA1RXBUF;
 }


 if (++iot_rx_ring_wr >= (SMALL_RING_SIZE))
 {
 iot_rx_ring_wr = BEGINNING; // Circular buffer back to beginning
 }


 //Set messagedone flag when a new line is read. Set array index
 //back to the BEGINNING
 if (IOT_Char_Rx[temp] == '\n')
 {
 iot_rx_ring_wr = BEGINNING;
 messagedone = ALWAYS;
 }




 break;

 case TXA1: // Vector 4 - TXIFG


 //Ensure what is being transmitted to the IOT device is a correct command
 temp = iot_tx_ring_wr;

 IOT_Char_Tx[temp] = UCA1TXBUF;
```

```
 if (++iot_tx_ring_wr >= (SMALL_RING_SIZE))
 {
 iot_tx_ring_wr = BEGINNING; // Circular buffer back to beginning
 }
 break;
 default: break;
 }
}
//------------------------------------------------------------------------------
```

8.9. serial_command_interpret.c

Figure 20 - serial_command_interpret Flowchart




```
//-----------------------------------------------------------------------------
//
//  serial_command_interpret.c
//
//  Description: This file provides the seiral command interpreter. This
//               interpreter has the responsibility of receiving commands from
//               the IOT device, and then deciding what actions to take upon
//               receiving that command. As this function must communicate with
//               IOT device, it must have extra logic within it to prevent
//               disassociations and infinite loops due to stomped
//               communications.
//
//  David Gietzen
//  Apr 2016
//  Built with IAR Embedded Workbench Version: V7.3.1.3987 (6.40.1)
//-----------------------------------------------------------------------------

#include  "msp430.h"
#include  "functions.h"
#include  "macros.h"
```

```
#define COMMAND_LENGTH                  (60)
#define ACTUAL_COMMAND_LENGTH           (20)
#define ONE_SEC_MS                      (1000)
#define MAX_TIMEOUT_COUNT               (2000)
#define DISASSOCIATION_STR_LENGTH       (8)
#define IOT_STATUS_STR_LENGTH           (5)

extern char *display_1;
extern char *display_2;
extern char *display_3;
extern char *display_4;

extern const char * timeout_str;
extern const char * timeout_str_usb;
extern const char * timeout_str_cpu;

extern volatile unsigned int number_msec_intervals;

char command_str[COMMAND_LENGTH] = {'\0'};                          //This holds the received command
char actual_command_str[ACTUAL_COMMAND_LENGTH] = {'\0'};            //This is the received command with
                                                                    //the password stripped off

const char * DISASSOCIATION_STR = "+WIND:41";
const char * IOT_STATUS_STR     = "+WIND";


const char * PIN_STR            = "WASNOTWAS";
const char * GOOD_PIN_STR       = "Password Confirmed";
const char * BAD_PIN_STR        = "Bad Password Given";

const char  COM_EMPTY          = '.';
const char  COM_9600_BAUD      = 'b';
const char  COM_115200_BAUD    = 'B';
const char  COM_RESET_IOT      = 'r';
const char  COM_CONNECT_IOT    = 'C';
const char  COM_BLACK_LINE_IOT = 'L';
const char  COM_FORWARD_IOT    = 'F';
const char  COM_REVERSE_IOT    = 'R';
const char  COM_RIGHT_TURN_IOT = 'T';
const char  COM_LEFT_TURN_IOT  = 't';


//================================================================================
// void serial_command_interpret()
//
// Description:         This function goes into a loop that attempts to receive
//                      commands from the IOT module and USB port.
//
// Arguments:           none
//
// Local Vars:          none
//
// Globals Used:        display_1
//                      display_1
//                      display_1
//                      display_1
//                      command_str
//                      actual_command_str
//
// David Gietzen, February 2016
//================================================================================

void serial_command_interpret()
{
  //Empty the ring's of old data

  flush_ring_usb();
  flush_ring_cpu();

  //Inform computer that car is prepared to receive command
  send_str_usb ("Ready for a command!\n", LARGE_LENGTH);


  //Read a command from either the IOT device or the PC

  int timeout_happened = ALWAYS;         //Represents if a timeout occured when receiving a command.
The initial value should be 1 for the following loop.
  int iot_or_usb = RESET;               //Selects which source to read a command from
```

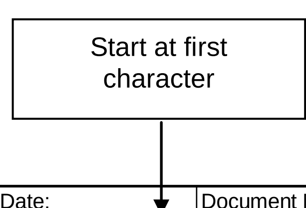
```
//Try receiving a command from both the USB serial port and IOT serial port

int timeout_count = RESET;

while ( timeout_happened ||
        is_same_str (command_str, "\r", ALWAYS) ||
        is_same_str (command_str, "\n", ALWAYS) ||
        is_same_str (command_str, "\r\n", ALWAYS))
{
    if (RESET == iot_or_usb)
    {
      timeout_happened = read_str_usb (command_str, COMMAND_LENGTH, '\r');

      iot_or_usb = ALWAYS;
    }
    else
    {
      timeout_happened = read_str_cpu (command_str, COMMAND_LENGTH, '\n');

      iot_or_usb = RESET;
    }

    try_display_update();

    timeout_count += ALWAYS;


    //Prevents potential infinite loop caused by an IOT disassociation

    if (number_msec_intervals > DIS_WAIT_MS)
    {
              indent_and_display (RESET, "Re-assoc", DISPLAY_2_ID);

              //Reset IOT device upon disassociation

              send_str_cpu("AT+CFUN=1\r", LARGE_LENGTH);

              PJOUT &= ~IOT_RESET;                         //Resets the IOT device
              one_msec_sleep(WAIT_RESET_MS);
              PJOUT |= IOT_RESET;

              number_msec_intervals = RESET;

              return;
    }


}


//This is for performing diagnostics
send_str_usb(command_str, COMMAND_LENGTH);

//Clear the scren, because now we will start displaying stuff on screen

screen_clear();
lcd_BIG_mid();
try_display_update();


//Check for disassociation from IOT device, reconnect on that event
if (is_same_str(DISASSOCIATION_STR, command_str, DISASSOCIATION_STR_LENGTH))
{

    indent_and_display (RESET, "Re-assoc", DISPLAY_2_ID);
    one_msec_sleep(ONE_SEC_MS>>ALWAYS);

    //Reset IOT device upon disassociation

    send_str_cpu("AT+CFUN=1\r", LARGE_LENGTH);

    PJOUT &= ~IOT_RESET;                         //Resets the IOT device
    one_msec_sleep(WAIT_RESET_MS);
    PJOUT |= IOT_RESET;

    return;
```

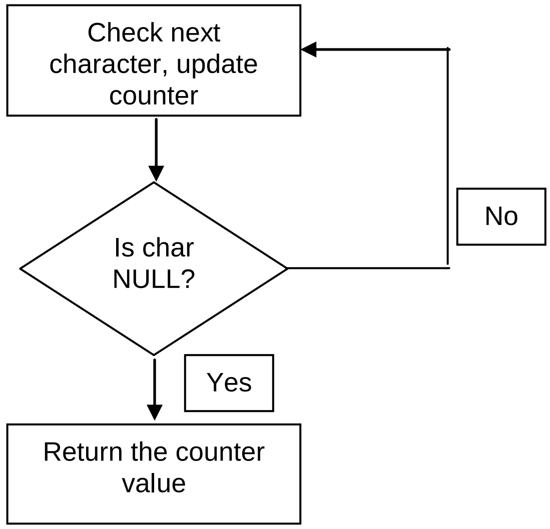
```
    }

    //Check for IOT status string; if it is one, then ignore this command

    if (is_same_str(IOT_STATUS_STR, command_str, IOT_STATUS_STR_LENGTH))
    {
        indent_and_display (RESET, " WIFI STAT", DISPLAY_2_ID);

        try_display_update();

        return;
    }



    //Check if the command has the password. If it does not, then leave the
    //function.

    if (is_same_str (command_str, PIN_STR, str_length (PIN_STR)))
    {
           send_str_usb (GOOD_PIN_STR, LARGE_LENGTH);
    }

    else
    {
           send_str_usb (BAD_PIN_STR, LARGE_LENGTH);

           indent_and_display (RESET, " BAD PASS", DISPLAY_2_ID);

           try_display_update();

           return;        //Exit the function, no command to interpret
    }


    //Pull the command(s) out of the received string by stripping off the password

    set_str(actual_command_str,    ACTUAL_COMMAND_LENGTH,    command_str    +    str_length(PIN_STR),
  COMMAND_LENGTH - str_length(PIN_STR));

    char * actual_command_str_ptr = actual_command_str;

    actual_command_str_ptr++; //Ignore initial dot

    char display_command[COMMAND_LENGTH] = {'\0'};

    while  ((NULL_CHAR  !=  *actual_command_str_ptr)  &&  ('\n'  !=  *actual_command_str_ptr)  &&  ('\r'  !=
  *actual_command_str_ptr))
    {

      //Display command on screen
      screen_clear();
      display_command[RESET] = *actual_command_str_ptr;
      indent_and_display (ALWAYS + ALWAYS, "Command", DISPLAY_1_ID);
      indent_and_display ((DISPLAYED_CHARS - ALWAYS) >> ALWAYS, display_command, DISPLAY_2_ID);

      one_msec_sleep(WAIT_SMALL_MS >> ALWAYS);
      try_display_update();


      //Interpret the command
      if (COM_EMPTY == *actual_command_str_ptr)
      {
        send_str_usb ("No Command", LARGE_LENGTH);
      }

      else if (COM_115200_BAUD == *actual_command_str_ptr)
      {
        send_str_usb ("115200 Baud Set", LARGE_LENGTH);
        Init_Serial_UCA1_115200_Baud();
      }

      else if (COM_9600_BAUD == *actual_command_str_ptr)
      {
        send_str_usb ("9600 Baud Set", LARGE_LENGTH);
        Init_Serial_UCA1_9600_Baud();
      }
```

```
      else if (COM_RESET_IOT == *actual_command_str_ptr)
      {
        reset_IOT();
      }

      else if (COM_CONNECT_IOT == *actual_command_str_ptr)
      {
        connect_IOT();
      }

      else if (COM_FORWARD_IOT == *actual_command_str_ptr)
      {
        motors_set_direction(FORWARD, FAST_FORWARD_L, FORWARD, FAST_FORWARD_R);
        one_msec_sleep((MIN_MOTOR_TIME_FORWARD_MS<<ALWAYS<<ALWAYS)          +
MIN_MOTOR_TIME_FORWARD_MS<<ALWAYS);
        motors_off();
      }

      else if (COM_REVERSE_IOT == *actual_command_str_ptr)
      {
        motors_set_direction(REVERSE, FAST_REVERSE_L, REVERSE, FAST_REVERSE_R);
        one_msec_sleep(MIN_MOTOR_TIME_REVERSE_MS);
        motors_off();
      }

      else if (COM_RIGHT_TURN_IOT == *actual_command_str_ptr)
      {
        motors_set_direction(FORWARD, FAST_FORWARD_L, OFF, RESET);
        one_msec_sleep(TURN_TIME_RIGHT_MS);
        motors_off();
      }

      else if (COM_LEFT_TURN_IOT == *actual_command_str_ptr)
      {
        motors_set_direction(OFF, RESET, FORWARD, FAST_FORWARD_R);
        one_msec_sleep(TURN_TIME_LEFT_MS);
        motors_off();
      }

      else if (COM_BLACK_LINE_IOT == *actual_command_str_ptr)
      {
        follow_black_line();
        motors_off();
      }

      else
      {
        //Inform user that command was misunderstood
        send_str_usb ("Command not understood.\n", LARGE_LENGTH);
      }

      //Look at the next command
      actual_command_str_ptr++;
    }

    screen_clear();

    //Clear command string
    set_str(command_str, COMMAND_LENGTH, "", ALWAYS);

    //Inform the computer that the command has completed.
    send_str_usb ("\nDone.\n", LARGE_LENGTH);
  }
```

8.10. custom_string.c

Figure 21 - get_str_length Flowchart






```
//------------------------------------------------------------------------------
//
//  custom_string.c
//
//  Description: This file provides various functions needed to quickly
//          process strings.
//
//  David Gietzen
//  Apr 2016
//  Built with IAR Embedded Workbench Version: V7.3.1.3987 (6.40.1)
//------------------------------------------------------------------------------

#include  "msp430.h"
#include  "functions.h"
#include  "macros.h"

//==============================================================================
// int get_str_length (char * str, int max)
//
// Description:         This returns the length in char's of (str), NOT including
//                      the null terminator character. If a null terminator is
//                      not found within (max) char's, then that length is returned.
//
// Arguments:           str                     - The string to get the length
//                                                of
//
//                      max                     - The max number of characters
//                                                to check.
//
// Local Vars:          none
//
// Globals Used:        none
//
// David Gietzen, April 2016
//==============================================================================

int get_str_length (const char * str, unsigned int max)
{
  int length = RESET;

  for (int i = RESET; i < max; i++)
  {
    if (NULL_CHAR == str[i])
    {
      //length = i + ALWAYS;

      length = i;

      break;
    }
```

```
    length = i;
  }

  return length;
}



//==============================================================================
// int get_str_length (char * str)
//
// Description:         This returns the length in char's of (str), checking
//                      ONLY UP TO 65535 characters. Does NOT include the NULL
//                      character in the return value.
//
// Arguments:           str                     - The string to get the length
//                                                of
//
// Local Vars:          none
//
// Globals Used:        none
//
// David Gietzen, April 2016
//==============================================================================

int str_length (const char *str)
{
  return get_str_length (str, UINT_MAX_VALUE);
}


//==============================================================================
// int get_str_length_terminator (char * str, int max)
//
// Description:         This returns the length in char's of (str), NOT including
//                      the null terminator character. If a null terminator is
//                      not found within (max) char's, then that length is returned.
//                      This is like the above function, but it also takes an argument
//                      for the terminator.
//
// Arguments:           str                     - The string to get the length
//                                                of
//
//                      max                     - The max number of characters
//                                                to check
//
// Local Vars:          none
//
// Globals Used:        none
//
// David Gietzen, April 2016
//==============================================================================

int get_str_length_terminator (const char * str, char terminator, unsigned int max)
{
  int length = RESET;

  for (int i = RESET; i < max; i++)
  {
    if (terminator == str[i])
    {
      //length = i + ALWAYS;

      length = i;

      break;
    }

    length = i;
  }

  return length;
}


//==============================================================================
// int get_str_length_terminator (char * str, char terminator)
```

```
//
// Description:         This returns the length in char's of (str), checking
//                      ONLY UP TO 65535 characters. This stops reading at the
//                      given terminator character. This DOES not include the
//                      NULL terminator in the return value.
//
// Arguments:           str                     - The string to get the length
//                                                of
//
//                      terminator              - The character to stop reading
//                                                at
//
// Local Vars:          none
//
// Globals Used:        none
//
// David Gietzen, April 2016
//==============================================================================

int str_length_terminator (const char *str, char terminator)
{
  return get_str_length_terminator (str, terminator, UINT_MAX_VALUE);
}


//==============================================================================
//void set_str(char * what_to_set_str, unsigned int set_length, const char * copy_from_str,

unsigned int from_length)
//
// Description:         Copies as much as possible from (copy_from_str) and puts
//                      it in (what_to_set_str).
//
// Arguments:           what_to_set_str         -The string to write to
//
//                      set_length              -The length of what_to_set_str
//
//                      copy_from_str           -The string to read from
//
//                      from_length             -The length of copy_from_str
//
// Local Vars:          none
//
// Globals Used:        none

// David Gietzen, February 2016
//==============================================================================

void set_str(char * what_to_set_str, unsigned int set_length, const char * copy_from_str,

unsigned int from_length)
{
  //Clear destination string

  for (int i = RESET; i < set_length; i = i + ALWAYS)
  {
    what_to_set_str[i] = NULL_CHAR;
  }

  //Copy as much as possible

  for (int i = RESET; (i < (set_length - ALWAYS)) && (i < from_length); i = i + ALWAYS)
  {
    what_to_set_str[i] = copy_from_str[i];

    if (copy_from_str[i] == NULL_CHAR)  //Leave if end of source string detected
    {
      return;
    }
  }

  return;
}

//==============================================================================
//char is_same_str (const char * first_str, const char * second_str, unsigned int

length_to_check)
```

```
//
// Description:          Returns a boolean value that is true when the two strings
//                       are the same (up to the provided length). This will
//                       stop immediately if a null is detected in one of the strings.
//
// Arguments:            first_str               -First string to read
//
//                       second_str              -Second string to read
//
//                       length_to_check         -The amount of characters to read
//                                                in BOTH strings
//
// Local Vars:           none
//
// Globals Used:         none

// David Gietzen, February 2016
//================================================================================


char is_same_str (const char * first_str, const char * second_str, unsigned int

length_to_check)
{
  char result = ALWAYS;  //Result is (True) initially, following loop tests if false

  for (int i = RESET; i < length_to_check; i = i + ALWAYS)
  {
    if (first_str[i] != second_str[i])
    {
      result = RESET;   //Strings were NOT same
    }

    if ((NULL_CHAR == first_str[i]) || (NULL_CHAR == second_str[i]))
    {
      break;    //One of the strings ended, quit incrementing through str
    }
  }

  return result;
}
```

9. Conclusion

This is the first class where we have directly witnessed the connection between software and hardware. For some of our team members, it solidified their choice in computer engineering. The projects clearly demonstrate one of the main differences between high-level systems and low-level systems; your software has complete control over the hardware, and so bugs in the software/hardware have immediate implications for the other side.

When you are not working with an OS that can automatically schedule tasks for you, handling the control flow becomes an involved process. For example, if your code spends too long in a particular interrupt, your code may malfunction in unexpected ways. Timer interrupts won't occur at the right time, causing the several errors in the sequence of code executed. In order to make a stable final design, determining the flow of execution requires much care and thought.

As far as the class itself is concerned, our team enjoyed the class and its direct access to working with hardware. Being able to receive quick help from Mr. Carlson on issues regarding our projects was a major benefit. It is definitely apparent that he cares about this class and the students learning. Teaching a class like ECE306, working a full-time job, and responding to problems quickly demonstrates his commitment to our success, and our team greatly appreciates it. This is one of the best classes at NC State.