CSC-202 RC Exploration Robot User/Design Manual

Overview: 2
- Parts used: 2

Operation Details 3
- Robot Operating Procedures: 3
- Remote Control Communiation: 3
- Motor Control: 4
- Battery Monitoring: 6
- Ultrasonic Sensors & Arduino: 7
- Environmental Data: 7
- Real-Time Interrupts: 8
- The RC Exploration Clock: 9
- FRAM SPI Storage Chip: 11
- Data Handling in the RC Exploration Vehicle: 19
- Displaying Data on a Computer Screen: 21
- Button- and LCD-based User Interface: 28
- Music and Sound: 31

Future Work Needed: 43
References/Disclosure: 43
Appendix A. (Wiring Diagram) 44

Authors: Rowan Daly, Anthony Lansing

Overview:

The project is a remote-controlled robot that can be used to explore areas that are inaccessible or harmful to humans. The robot is piloted by the user with an RF remote control, and will periodically record environmental data (ambient temperature and light level) to an FRAM memory chip along with the current timestamp. The robot is equipped with three ultrasonic distance sensors, which will prevent the user from accidentally driving into an obstacle by halting the robot if it gets too close. The ultrasonic sensors are monitored by a separate Arduino microcontroller, and any near collisions are also recorded to the FRAM. The robot runs from a standard lithium-polymer battery designed for use with an RC vehicle (nominal voltage of 11.1v), and the battery level is continuously monitored to alert the user when it's running low. Once the robot has returned from its mission, "explore mode" can be turned off and the robot will send all the recorded data to a separate system via a serial connection.

Parts used:

- DRAGON12-Plus2 Rev. D trainer board
- FlySky FS-i6X remote control
- FlySky FS-iA6B RC receiver
- Adafruit MB85RS64V 64kbit FRAM SPI storage chip
- Turnigy 3000mAh 3S 30C Li-Po battery pack
- Parallax #28963 motor mount and wheel kit (included encoders not used)
- Parallax #29144 HB25 DC motor controller
- Parallax #28015 "PING)))" ultrasonic distance sensor
- Arduino Nano microcontroller board

Operation Details

Robot Operating Procedures:

Steps to operate the robot are as follows:

1. Plug the RC battery into the XT-60 adapter
2. Use the menu buttons to select “EXPLORE” mode
3. Use the menu buttons to choose how often data is collected
4. Turn on the remote control
5. Flip the toggle switch to enable power to the motors
6. Pilot the robot as desired
7. When finished operating, press SW5 to end the program
8. Turn off motor power switch
9. Unplug battery

To send recorded data to a PC over serial terminal (such as PuTTY):

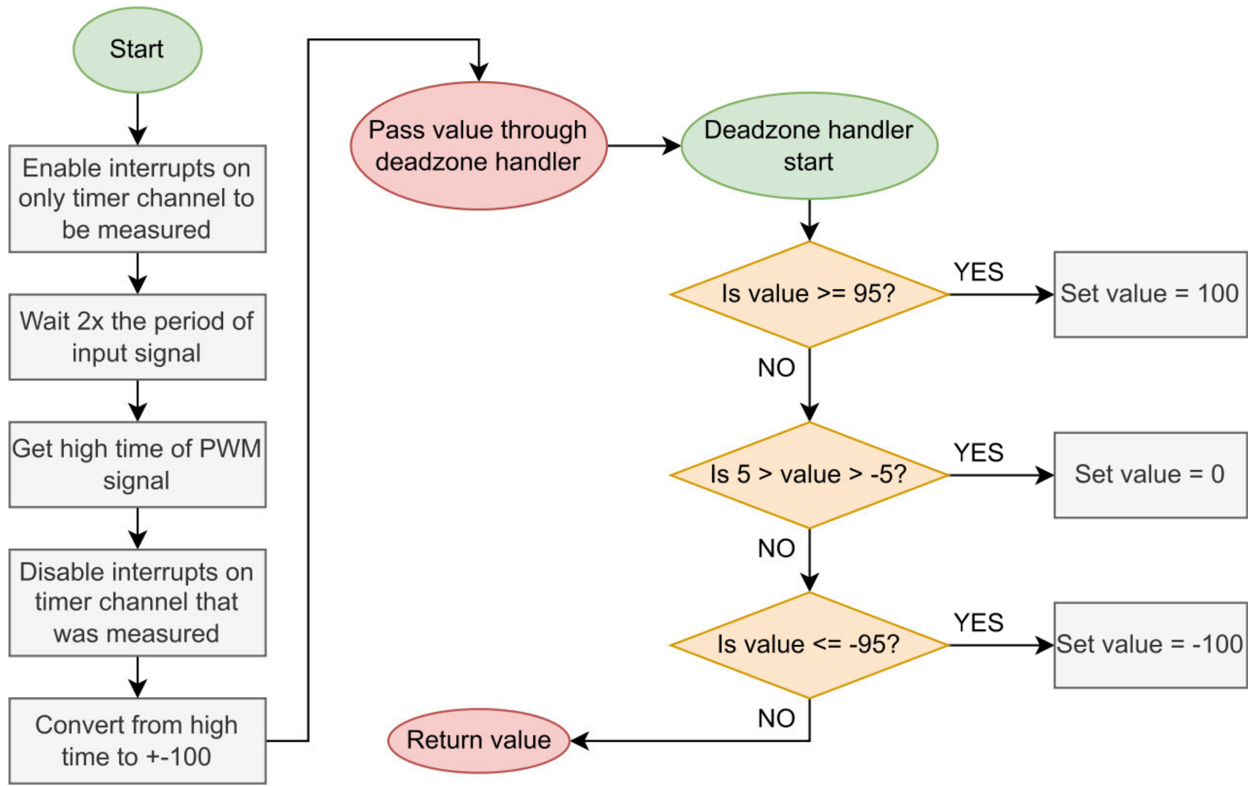
1. Plug the RC battery into the XT-60 adapter
2. Plug the serial adapter into the PC and connect with terminal of choice
3. Use the menu buttons to select “GET DATA”
4. Once data has been transferred, unplug serial adapter and battery.

Remote Control Communiation:

The FS-i6X remote control is a generic remote that is highly configurable to be used with many varieties of RC vehicle. It communicates user input via RF to a corresponding FS-iA6B receiver module. This module is capable of outputting data via PWM (compatible with standard servo motors), PPM (Pulse Position Modulation), and i-BUS (a proprietary serial protocol). The interface is configured via the remote control menu, and for the robot we used two PWM channels, one for forward-back control and one for left-right control. The frequency of the PWM is also configurable, and we chose the maximum of 400Hz to get the fastest response time.

Two functions are used to read the control data signal from the receiver, both stored in rc_read.c. The receiver PWM signal has a high time of 1.5ms if the control stick is in the neutral position, with a range of +- 0.5ms when the control stick is pushed all the way in one direction.

char get_high_time_decimal(char channel_number)



The get_high_time_decimal() function reads the high time of either Port T bit 1 or Port T bit 2 (corresponding to the two output channels) using the timer module. Because the provided get_high_time1() assembly subroutine could only read a single PWM channel, a second subroutine get_high_time2() was created that is almost identical to the first but reads from a second channel. Timer interrupts for each channel are only enabled while the reading is being taken, so that another interrupt does not occur while a reading is in progress.

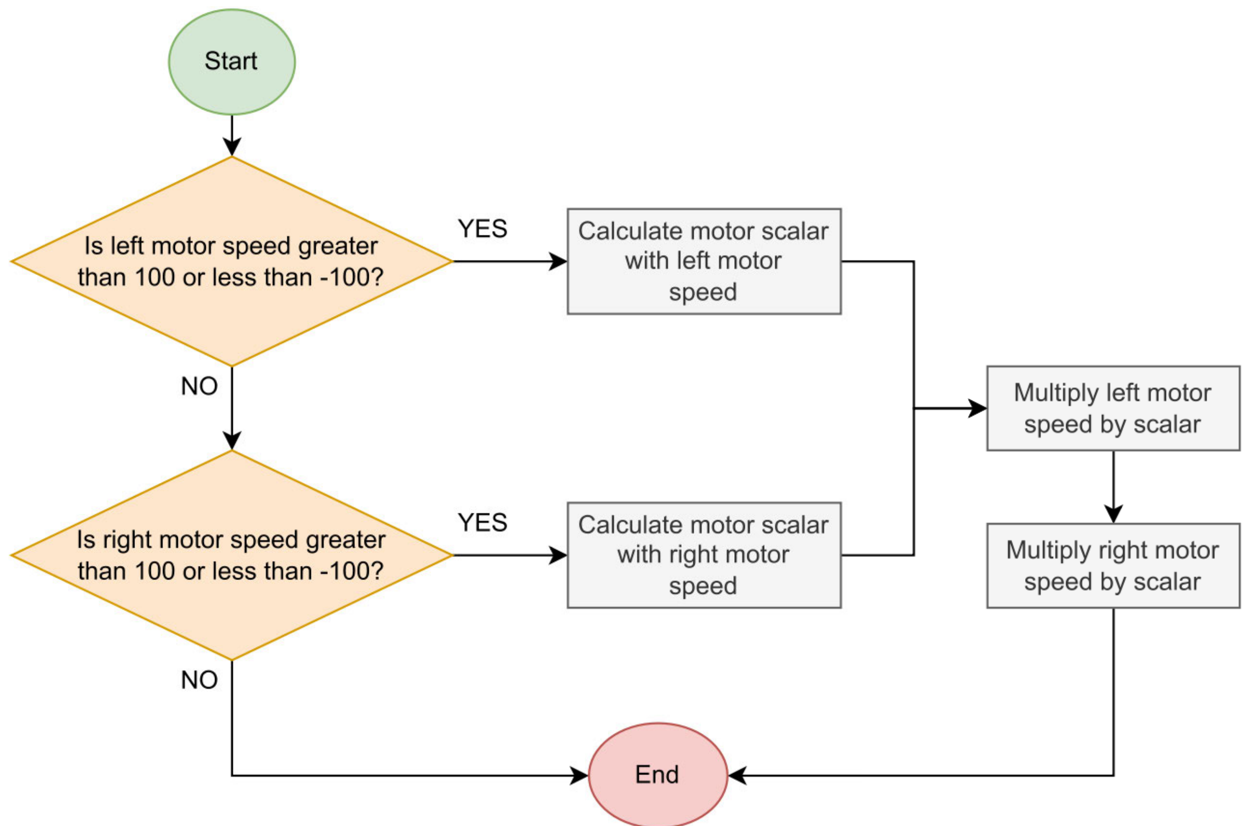
The raw timer return value is fed through a formula to convert the range of the input to +-100, with 0 being neutral. This is then fed through the deadzone_handler() function, which rounds the value to either 100, 0, or –100 if it is within a certain range of any of those values.

Motor Control:

The robot drives using two high-power DC motors via the HB25 motor driver boards. The boards are designed to allow DC motors to be controlled as if they were continuous servos, meaning they take a PWM input pulse between 1ms (full speed reverse) and 2ms (full speed forward) with 1.5ms being no rotation. Each pulse sets the speed that the motor will rotate at until it receives another pulse. The provided DC motor subroutines are used to control each motor via Port P bit 1 and Port P bit 2.

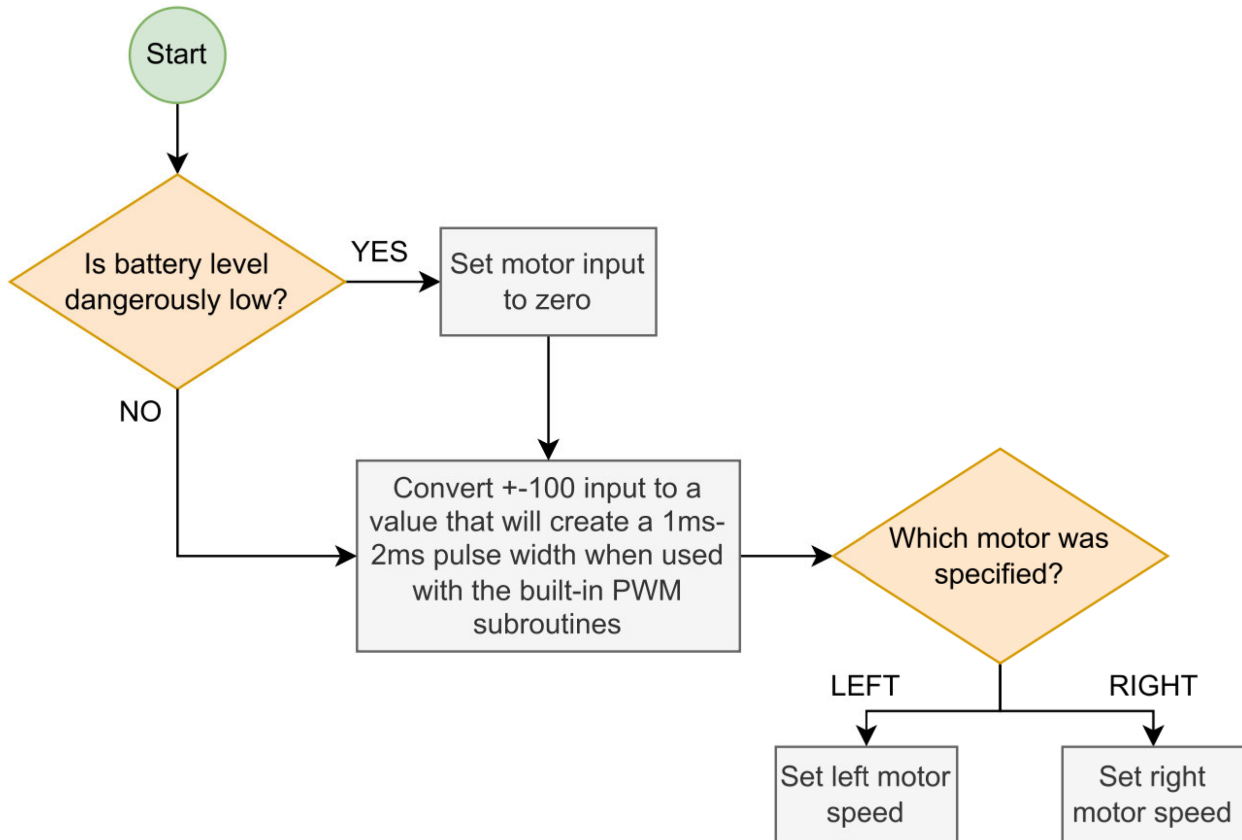
The motor control functions are located inside motor_control.c.

void scale_motor_speed(void)



The function scale_motor_speed() is used to combine the control inputs in such a way to allow control via both input sticks at once, without either value exceeding the +-100 range (see this Stack Exchange post for more information).

void set_motor_speed(char motor_num, char speed_plus_minus_100)



The function set_motor_speed() is used to set the speed of each motor via PWM. The +-100 input value is converted to a range that will provide 1ms-2ms pulse width when used with the provided DC motor PWM subroutines. See the inline comments for details.

The motor driver boards must be powered on **after** the main Dragon12 controller board has completely started up. This is because an invalid signal could be sent over Port P during startup, causing the HB25 drivers to enter an error state where they will refuse to rotate after they receive such an invalid signal.

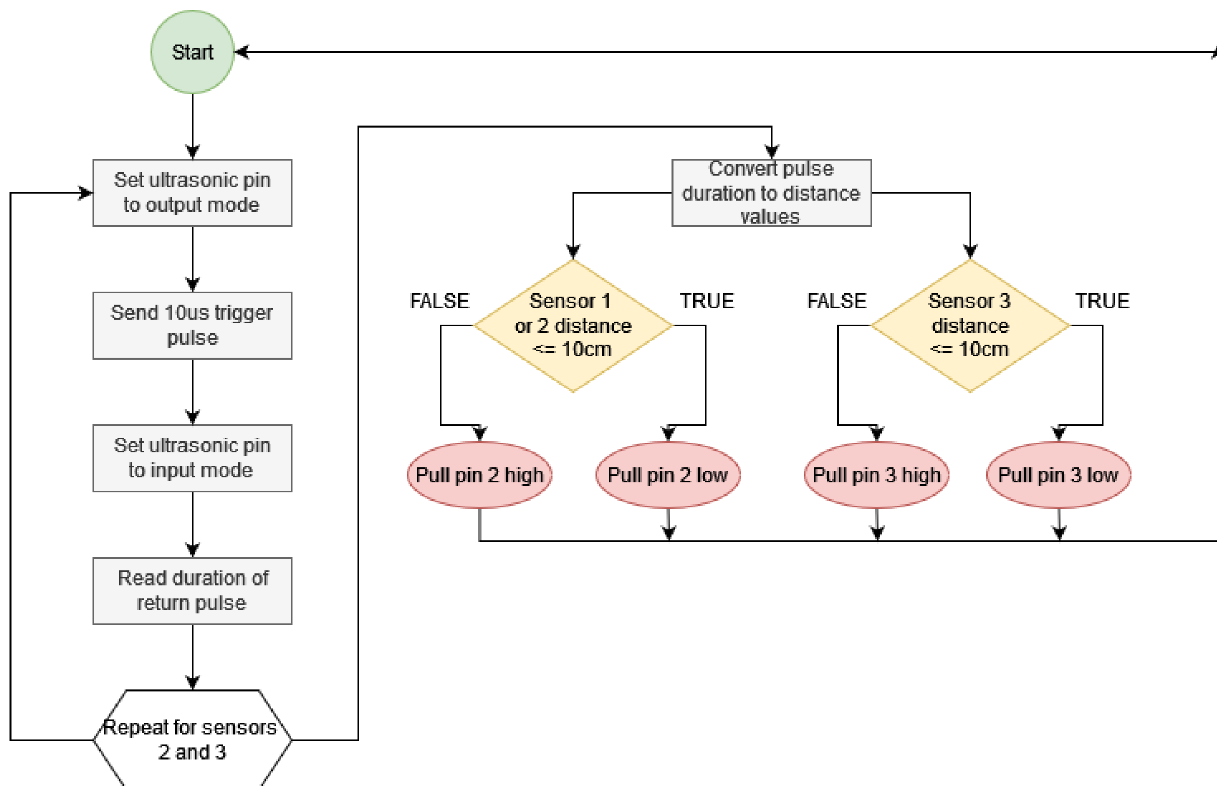
Battery Monitoring:

The battery we used is a standard 3-series lithium-polymer battery, commonly used in RC vehicles. It has a nominal voltage of 11.1v, meaning a maximum range of around 9v to 12.6v. It has an XT-60 connector for power, and a small 4-pin connector for cell balancing during charging. This balance connector is used to monitor the battery voltage via a 10kΩ-6.2kΩ voltage divider, which provides an output voltage range of 3.444v to 4.822v. This is read via the ADC in battery_monitor.c, but only when the robot is not in motion, as the voltage drop caused by the motors would provide a false reading.

The battery level is displayed on the LEDs. If the battery level drops too low (<20%), the robot will sound an alert, and if it drops low enough to risk damage to the battery (<5%), it will prevent motion by setting the global flag g_low_battery_5_percent to true.

Ultrasonic Sensors & Arduino:

Because the timer module is already heavily used and reading three additional input signals would be very complex, we decided to use a separate Arduino Nano microcontroller to monitor the three ultrasonic sensors on the robot. The Arduino code is very simple as there are libraries for interfacing with ultrasonic sensors included. When it detects that an object is too close to the robot, it will pull one of its GPIO pins low (one corresponds to the front two sensors, and one to the back sensors), which are connected to Port H bit 2 and bit 3 respectively on the Dragon12 board. These pins are monitored in the main control loop and the robot will refuse to move in the direction that an object is detected in.



Environmental Data:

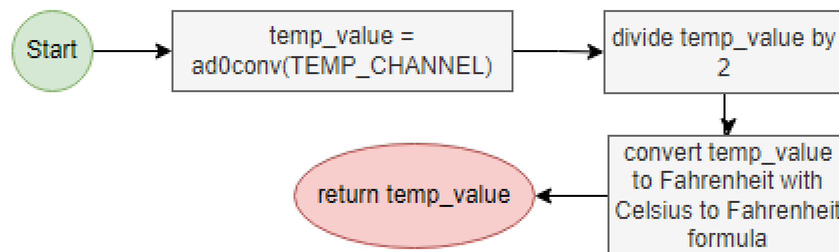
Our design utilizes the Dragon12 Analog-To-Digital converter in order to gather data from the environment. Specifically, we are using the onboard light sensor and temperature sensor to collect the light level and temperature, respectively.

We begin by using the already implemented ad0_enable subroutine to enable the ATD0 for 10-bit resolution with fast flag clearing. Afterwards, any time we need to collect data, we call the functions get_light_level() and get_temperature() which return the values from the light sensor and temperature sensors, respectively.

uint16 get_light_level(void)

The get_light_level() function uses the already implemented ad0_conv subroutine to retrieve a value from channel 4 (the channel associated with the light sensor) and returns.

uint16 get_temperature(void)



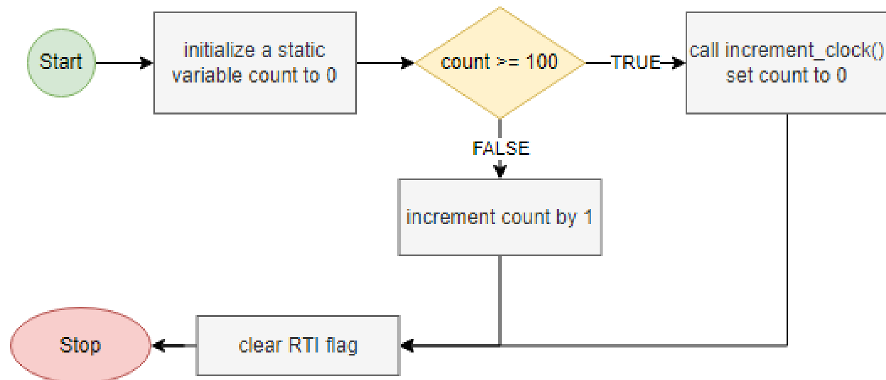
The get_temperature() function also uses the ad0_conv subroutine to retrieve a value from channel 5 (the channel associated with the temperature sensor). However, the value returned from ad0_conv is not useful in its base form and must be converted to Fahrenheit. We first shift the bits of the value from the ATD to the right by 1 bit—dividing by 2—which converts the value to Celsius, and then using the formula for converting Celsius to Fahrenheit to get the final temperature. This temperature is then returned to the calling function.

Real-Time Interrupts:

Our design uses the Dragon12 board's Real-Time Interrupts in order to keep track of the vehicle's time of operation. This is important as we need to know what the environment is like at any given time. It is also helpful to see in the data at what time(s) collisions occurred. The Dragon12 comes with a prebuilt function RTI_init() that enables real-time interrupts to occur every 10.24 milliseconds. Using this information, we can determine that about every 100 interrupts would

equate to one second of operation time. This brings us to the following implementation of the interrupt function:

void interrupt 7 clock_timer(void)

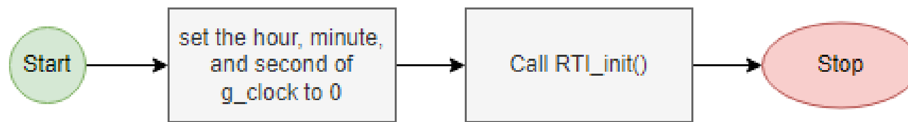


As you can see above, we first initialize a static variable count in order to keep track of the number of interrupts that occur. The variable is made static so that its value is saved for the next time the interrupt occurs. This value will be incremented with every interrupt. At the time that count reaches 100 (indicating that about 1 second has passed) we increment the vehicle's clock with the function increment_clock()—detailed further below—and reset count to 0. At the same time, we are keeping track of the total number of seconds for the operation. The usefulness of this is detailed further below. Finally, we reset the flag with clear_RTI_flag ().

The RC Exploration Clock:

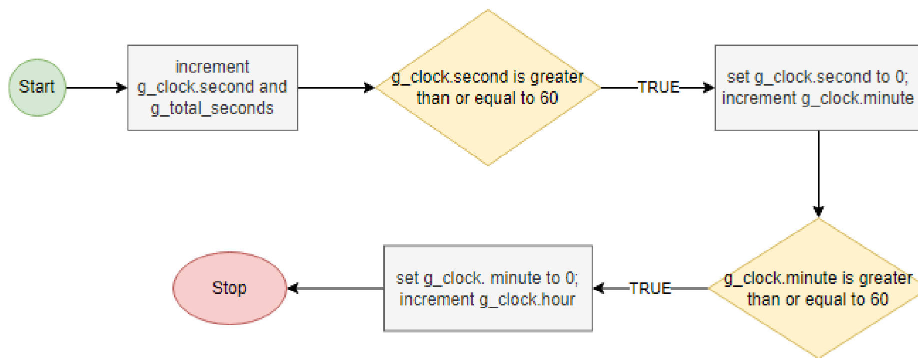
We take an object-oriented approach to keeping track of time on the vehicle. Using a struct, we can keep track of the hours, minutes, and seconds of the vehicle's operation time and then pass the current time to data collection to indicate when environmental data was collected or when a collision occurred. A global variable g_total_seconds is also utilized in order to keep track of the total seconds that have passed. This is useful to check when data should be collected as further discussed below. As noted above, the Dragon12 board's Real-Time Interrupts were utilized in the implementation of the Clock, whose functions are detailed as follows:

void clock_init(void):



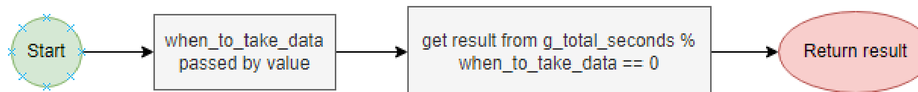
Here we simply set all values to zero and call RTI_init() to begin counting the time.

void increment_clock(void):



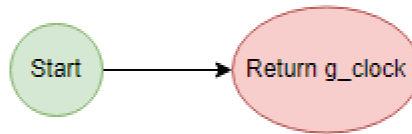
As can be seen above, we increment the seconds of our clock and of the global variables and check whether the seconds have exceeded the maximum allowable (59 seconds). If 59 has been exceeded—meaning the value is either 60 seconds or more—then we reset seconds to 0 and increment the minutes by one. Similarly, if the minutes have exceeded 59, then we reset minutes to 0 and increment hours by 1.

uint8 is_collect_time(uint8 when_to_take_data):



Here we utilize the global variable. By checking the remainder of the global variable g_total_seconds divided by when_to_take_data, we can check when it is time to collect data (based on whatever interval in seconds was provided by the calling function). The function returns true only when the modulo of the two values is 0.

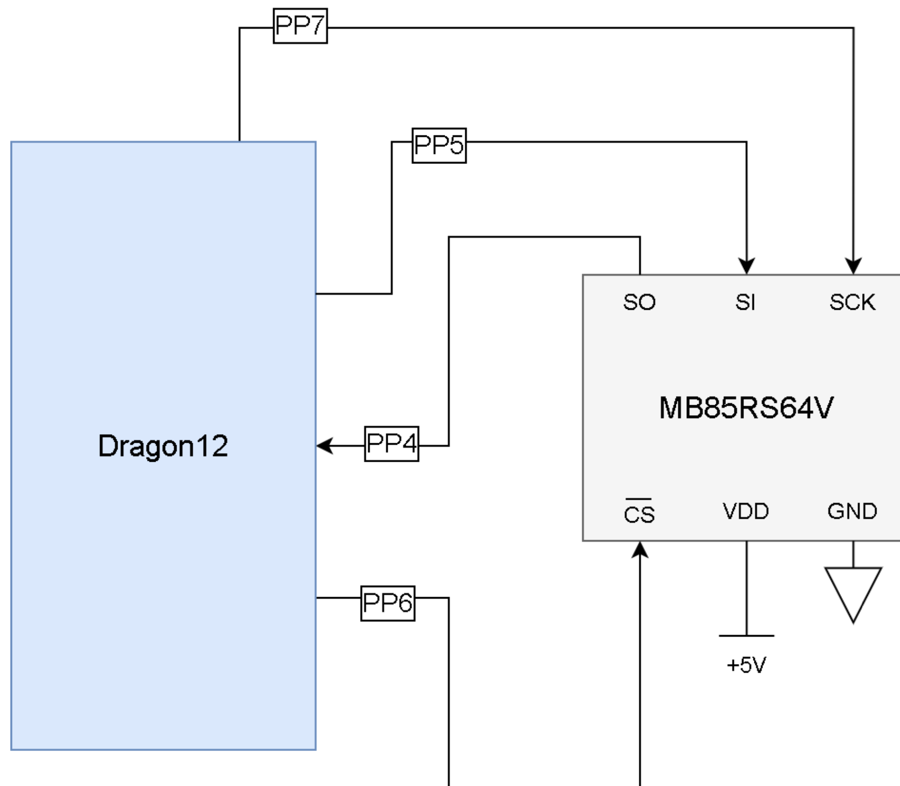
Clock get_time(void):



Here we return the Clock to the calling function. This is needed as when it is time to store the data on the FRAM, we store the data along with the time that it was collected.

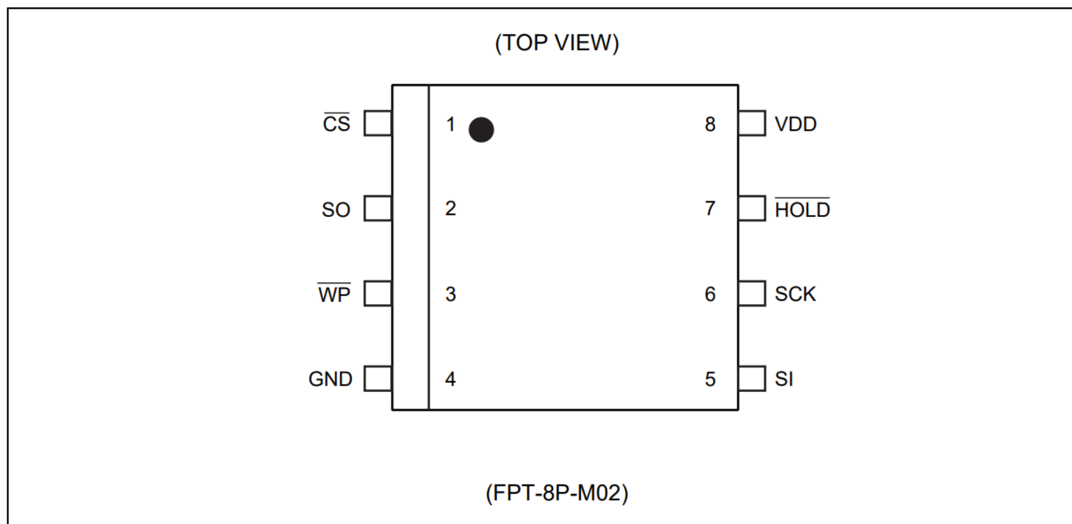
FRAM SPI Storage Chip:

In order to enhance the usefulness of the vehicle, we utilize an Adafruit MB85RS64V 64kbit FRAM SPI storage chip in order to read and write environmental data and collision detection. This allows us to read back any data over serial to be displayed on a computer screen for the user to see. Per the specification provided by the maker (Fujitsu), the MB85RS64V FRAM (Ferroelectric Random Access Memory) chip works much in the same way as an EEPROM and adopts SPI. We connect it to the Dragon12 board over the SPI2 per the below diagram:



The MB85RS64V FRAM has eight pins (although for our purposes we only use six). The six pins utilized in our design are detailed below:

- **Chip select** – this is the slave select pin. This pin must be set to Low before we are able to send any opcodes or data.
- **Serial Clock** – this is the clock input pin for the Dragon12. Data is inputted synchronously to a rising edge while outputted to a falling edge.
- **Serial Data Input** – this is the master out slave in pin. Data from the Dragon12 is transferred here.
- **Serial Data Output** – this is the master in slave out pin. Data from the FRAM is transferred here.
- **Voltage Supply**
- **Ground**



The pin assignment of the Adafruit MB85RS64V 64kbit FRAM SPI storage chip. (Fujitsu)

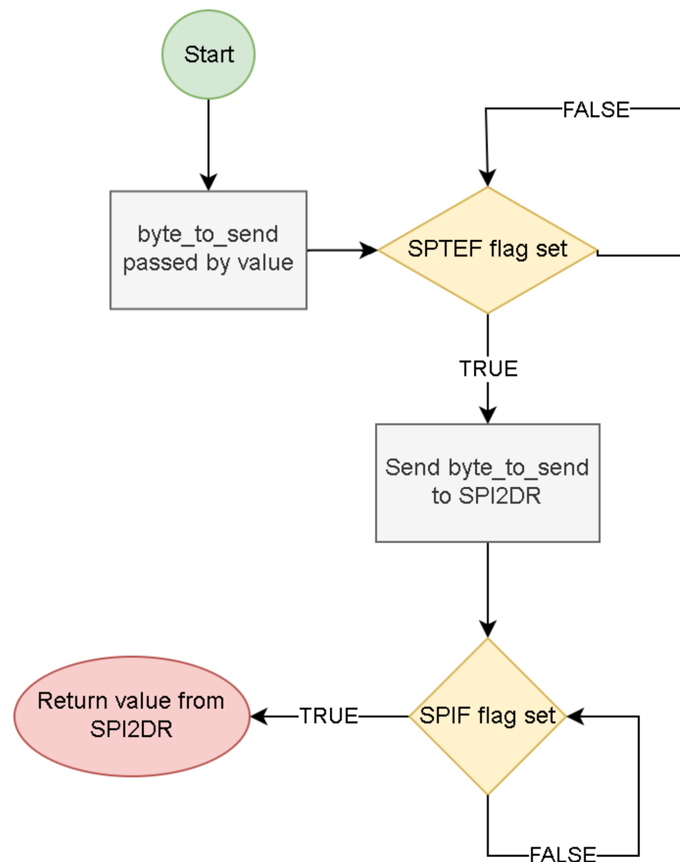
To prepare the FRAM for use on the Dragon12 board, we need to initialize the SPI2. The MB85RS64V FRAM can correspond to two SPI modes: mode 0, which sets CPOL and CPHA to 0; or mode 3, which sets CPOL and CPHA to 1 (Fujitsu). We use mode 0 in our implementation. Hence, the Dragon12 will interface with the FRAM with data shifting in and out on the rising edge of the clock. Further, the FRAM can support up to 20MHz operating frequency. For our implementation, we initialize SPI2 to operate at 12MHz.

With the above in mind, we start by initializing SPI2. This is done in the function eeprom_init(). We do so following the below pseudocode:

// Set the clock speed to 12MHZ
// Set Dragon12 as master with CPOL and CPHA set to 0
// Clear SPI2CR2 to disable MODF
// Set SS2 as an output port
// Set SS2 High until we are ready to send data

Once the Dragon12 has been set up as master per the above, we can start sending operation codes (opcodes), memory addresses, and data. We implement the following function in order to do so:

uint8 send_byte(uint8 byte_to_send):



We start by checking that the SPTEF flag of SPI2SR has been cleared, indicating that there is not already data in the register. We then put byte_to_send into SPI2DR for the transfer and wait for the transfer to complete. Like with any SPI, data was returned from the FRAM. This could be useful data or junk data depending on the command being executed. Either way, we return that value to the calling function.

The MB85RS64V FRAM comes with a total of seven opcodes that can be used to interface with the device; however, for our purposes, we use the following four:

- **WREN** (Set Write Enable Latch) which corresponds with opcode 0x06. Sending this code to the FRAM tells it to enable writing.
- **WRDI** (Reset Write Enable Latch) which corresponds with opcode 0x04. Sending this code to the FRAM tells it to disable writing.
- **READ** (Read Memory Code) which corresponds with opcode 0x03. Sending this code to the FRAM tells it to send data back from a memory location.
- **WRITE** (Write Memory Code) which corresponds with opcode 0x02. Sending this code to the FRAM tells it to write data to a memory location.

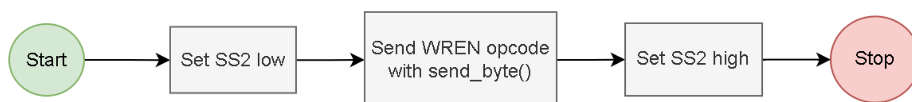
It should be noted that before any bytes can be sent, we must set the SS2 low, and when we are finished we must set it to high. We use the pre-built SS2_LO and SS2_HI subroutines to accomplish this.

Below we go into further detail on how the above opcodes are used to interface with the FRAM:

Writing data to the MB85RS64V FRAM:

In order to write data to the FRAM, we must first enable writing by sending WREN to the FRAM. We first set SS2 to low and utilizing the above detailed send_byte function, we send the WREN opcode to the FRAM. We then set SS2 high to indicate that we are done sending the command. We have implemented the write_enable() function to do accomplish this.

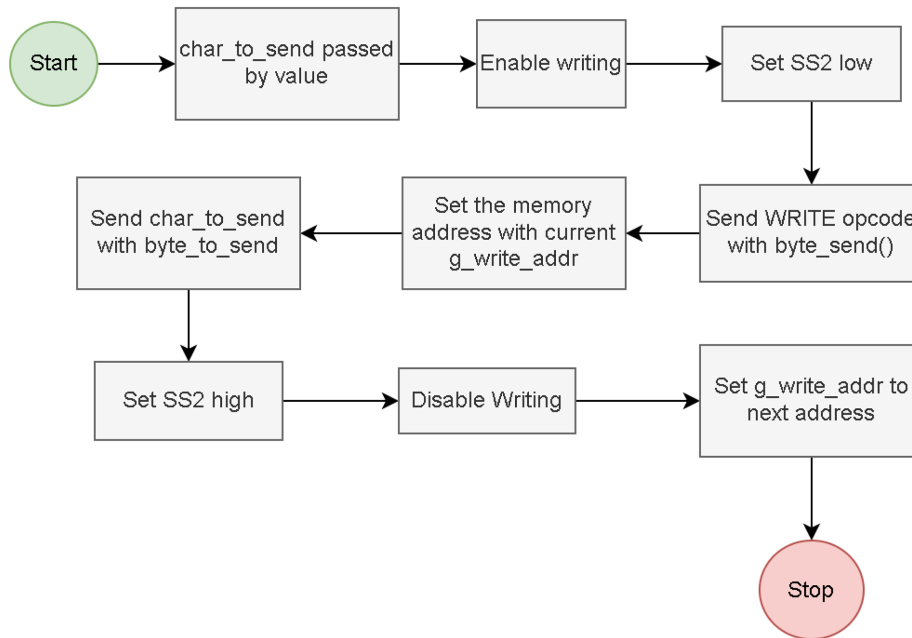
void write_enable(void):



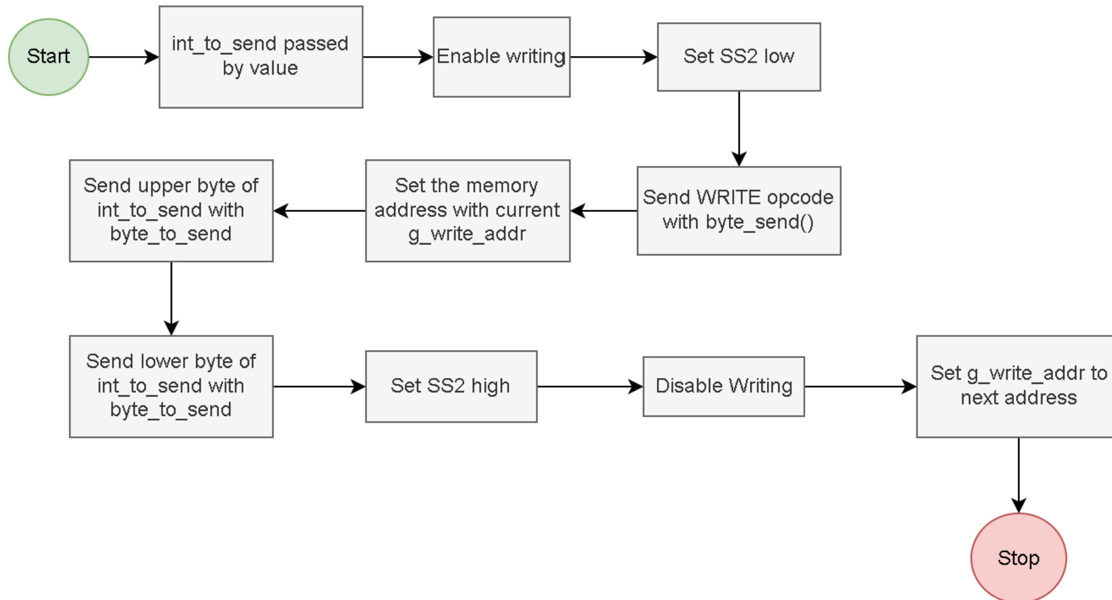
Now that writing has been enabled, we can begin writing to the FRAM. According to the specification provided by the Fujitsu, in order to write to the chip, we need three things: the WRITE opcode, the 16-bit memory address we wish to write to, and the 8 bits of writing data. Once SS2 is set low, we must send those 3 sets of data in that order. However, the FRAM has the added functionality of automatic address incrementation. What this means is that we can continue to send data one byte at a time and the FRAM will continue to write that data, automatically incrementing to the next memory address. We set SS2 high when we are finished writing.

With the above in mind, we have implemented two functions that allow us to write an unsigned char and an unsigned int to the FRAM.

void write_char(uint8 char_to_send):

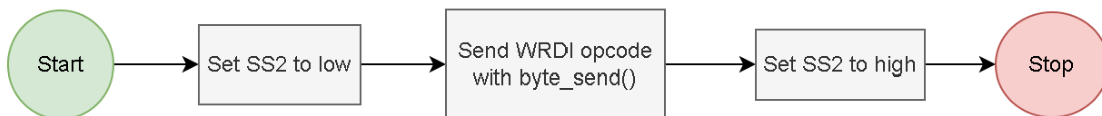


void write_int(uint16 int_to_send):



The implementations of the above functions are largely the same with the difference being that write_int must send the data one byte at a time with the upper byte being sent first. Otherwise, the method of writing is the same. We first enable writing by calling write_enable(). Next, we set SS2 low and send the WRITE opcode followed by the memory address we wish to write to. Then we send the data. Finally, we set SS2 high to indicate that we are finished writing and then disable writing by sending the WRDI opcode to the MB85RS64V FRAM. For our design, we have implemented a separate function write_disable() to do this for us.

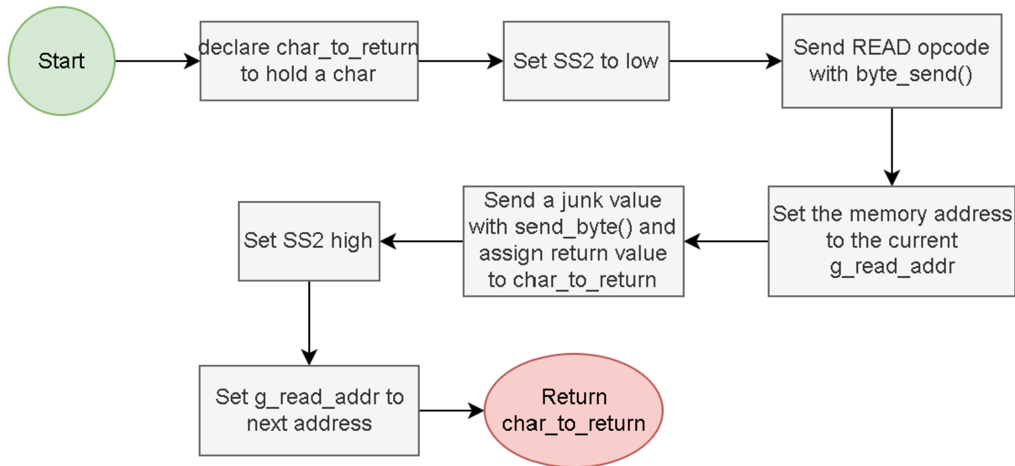
void write_disable(void):



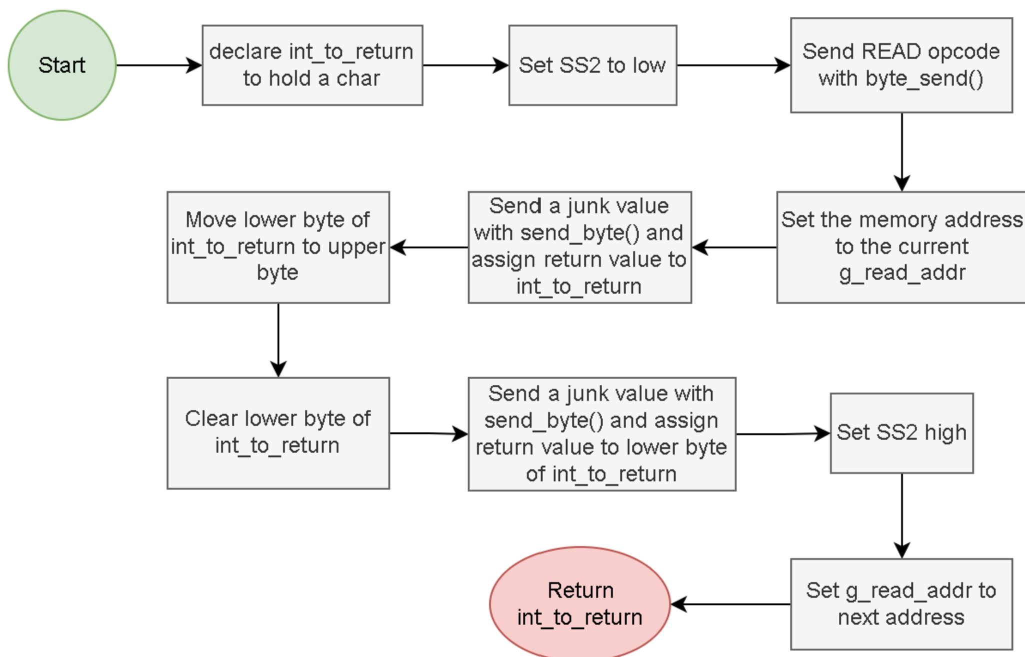
Reading data from the MB85RS64V FRAM:

Reading data from the MB85RS64V FRAM works much in the same way as writing data, structurally speaking. We must set SS2 low in order to send the READ opcode, followed by the memory address we wish to read from. However, reading differs in that we care what is being returned from the send_byte function. After sending the memory address, we can use the send_byte function to send a junk value to the FRAM (which will be ignored since the chip is not in write mode) and this will allow the FRAM to shift in the value from the memory location we provided. We can then return that value to the calling function. Like with writing, we have implemented two functions to read an unsigned char and an unsigned int from the FRAM:

uint8 read_char(void):



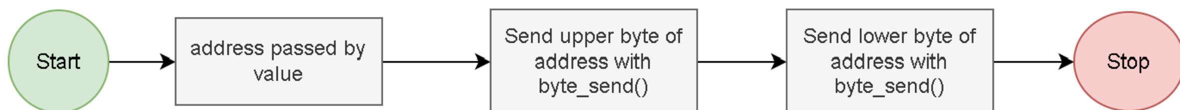
uint16 read_int(void):



The above functions behave very similarly; however, when reading an integer, we must do so in bytes. Therefore, we first read the upper byte from the FRAM and assign that to an unsigned integer. Then we use bitwise manipulation to shift the byte we just read into the upper byte of our variable before reading the lower byte in from the FRAM. We can then return the integer to the calling function.

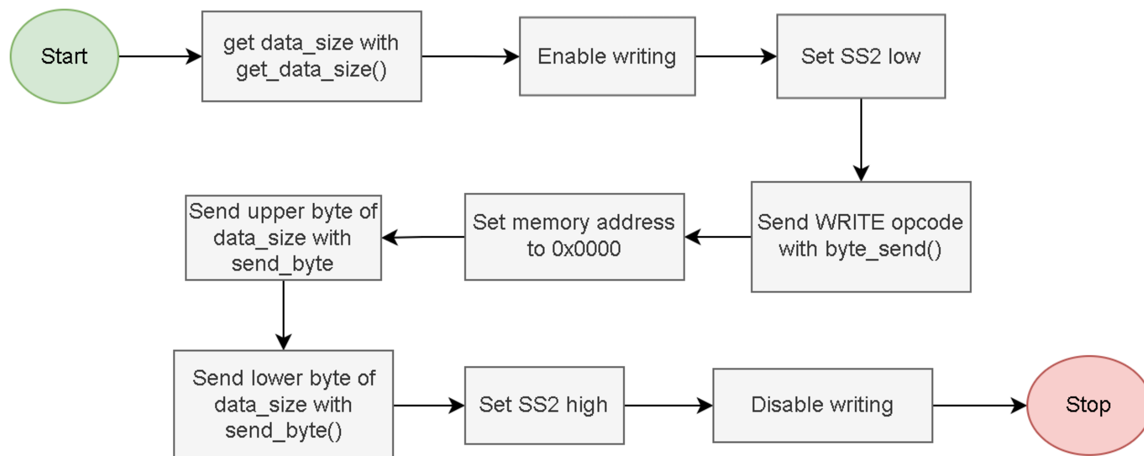
Finally, in order to keep track of the memory address, we utilize two global variables: g_read_addr for the memory addresses we read from and g_write_addr for the memory addresses we write to. As would have been seen in the flowcharts for the reading and writing functions, we must increment the memory addresses each time we read/write to ensure we are always at the appropriate memory address. Before writing/reading any data, we set the memory address with the following function:

void set_mem_addr(uint16 address):



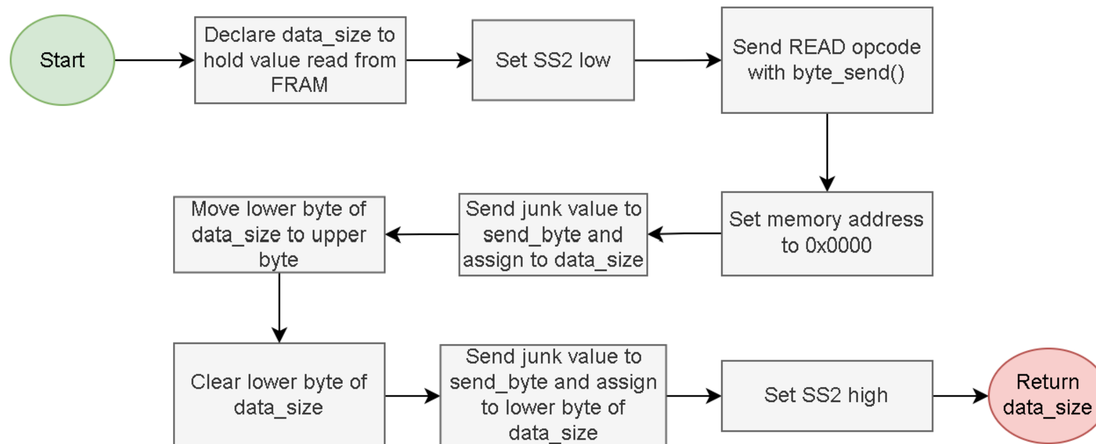
Both addresses are set to begin at address 0x0002. This is because we have reserved the first two bytes of memory in the FRAM for the number of data entries stored on the chip. With this in mind, our design utilizes two more functions:

void complete_write(void):



This function is very similar to the write_int function with the only difference being that we are writing the size of the data to memory location 0x0000.

uint16 read_data_size(void):



This function is very similar to the read_int function with the only difference being that we are reading the size of the date from memory location 0x0000.

Data Handling in the RC Exploration Vehicle:

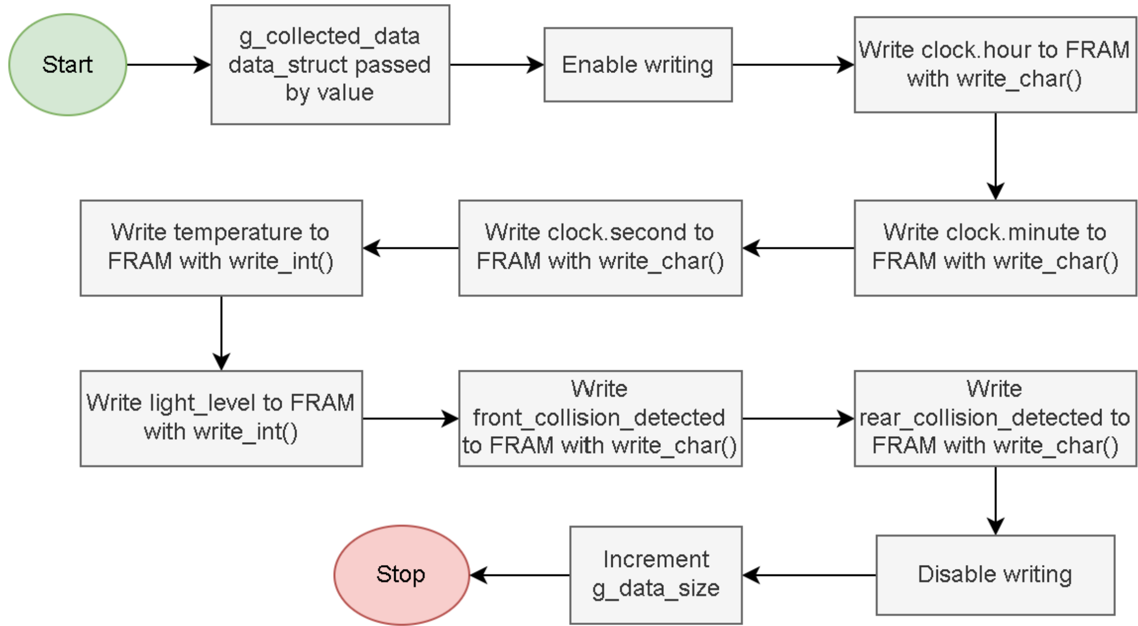
To make it easier to read and write data to the FRAM, we have gone with an object-oriented approach to data handling. We use a struct to keep all data collected. This includes not only the environmental data, but also our collision detection variables. This way we are always assured that the data we read from the FRAM are the expected values.

The struct g_collected_data we have implemented includes the following values:

- **clock**—regardless of whether it is data collection or a collision detection, the time of occurrence is needed, so g_collected_data contains a Clock struct.
- **temperature**—the temperature from the environment. This value will be 0 if a collision is being written to the FRAM instead.
- **light_level**—the light level from the environment. This value will be 0 if a collision is being written to the FRAM instead.
- **front_collision_detected**—TRUE when a front collision is detected. This will be FALSE when there is no collision or when a rear collision is detected instead.
- **rear_collision_detected**—TRUE when a rear collision is detected. This will be FALSE when there is no collision or when a front collision is detected.

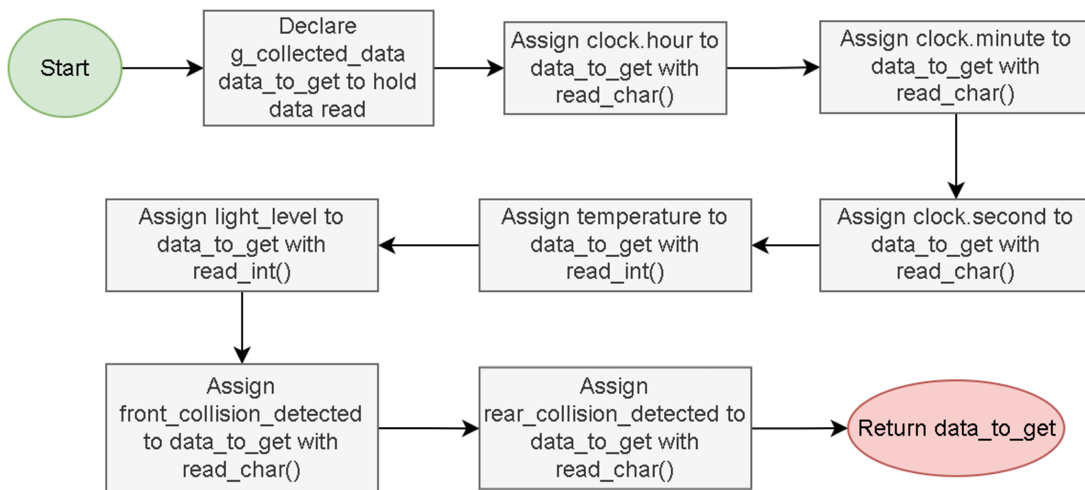
With our struct now defined, we have implemented the following functions to handle the reading and writing of data:

void write_data(g_collected_data):



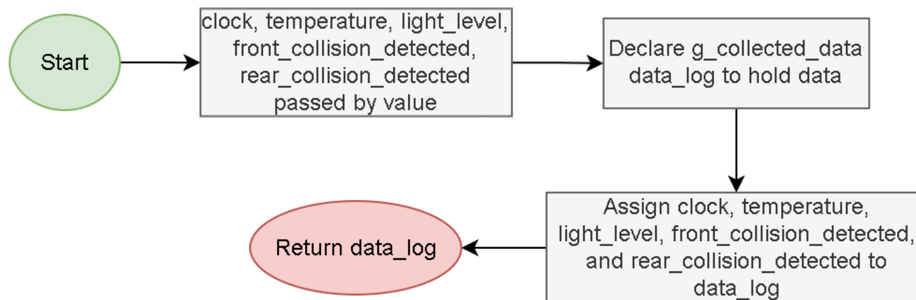
This function uses the previously discussed write_enable(), write_char(), and write_int() functions in order to write a g_collected_data struct to the FRAM. We also increment the size of the data collected thus far so that we know the size of the data set when reading from the FRAM.

g_collected_data get_data(void):



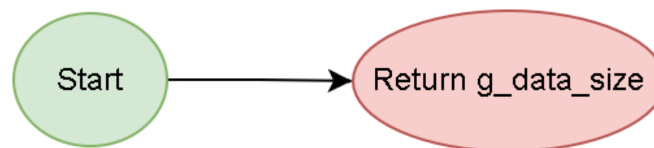
This functions creates reads a data struct from the FRAM and returns it to the calling function. We use the previously discussed functions read_char() and read_int() to accomplish this.

g_collected_data make_data_log(Clock clock, uint16 temperature, uint16 temperature, uint16 light_level, uint8 front_collision_detected, uint8 rear_collision_detected):



This function creates a g_collected_data struct using the provided arguments and returns it to the calling function.

uint16 get_data_size(void):



Here we return g_data_size to the calling function. For our purposes, it would be the FRAM that we would need this information.

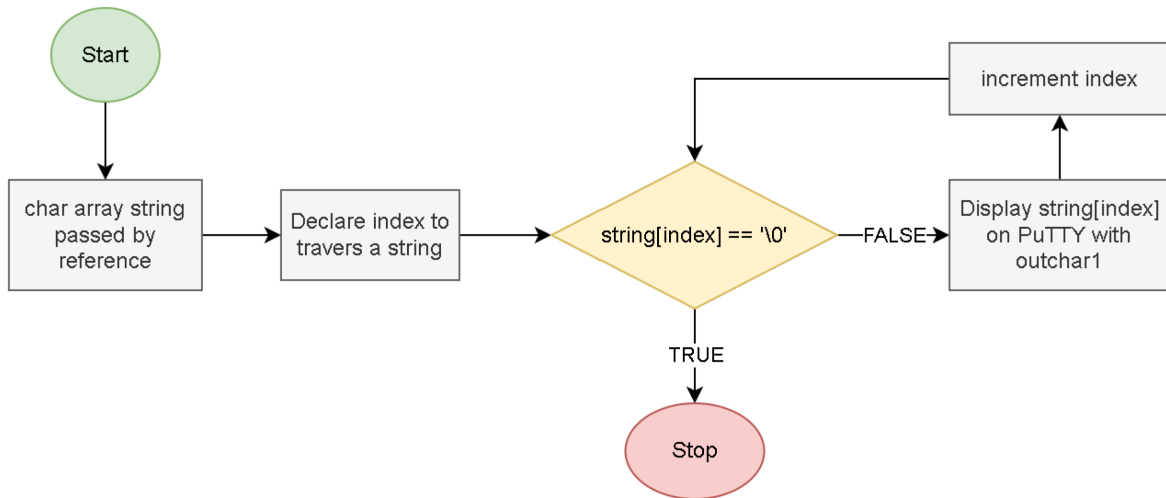
Displaying Data on a Computer Screen:

Data that the vehicle has gathered can be displayed on PuTTY by connecting a USB serial cable from the Dragon12 to the computer. After selecting “Get Data” from the menu, data is read in from the FRAM and printed on the screen.

We start by calling the premade assembly subroutine SCI1_init in order to set up SCI1 with a baud rate of 9600 at 8-bits with no parity. Once SCI1 is ready, we are ready to display data on PuTTY.

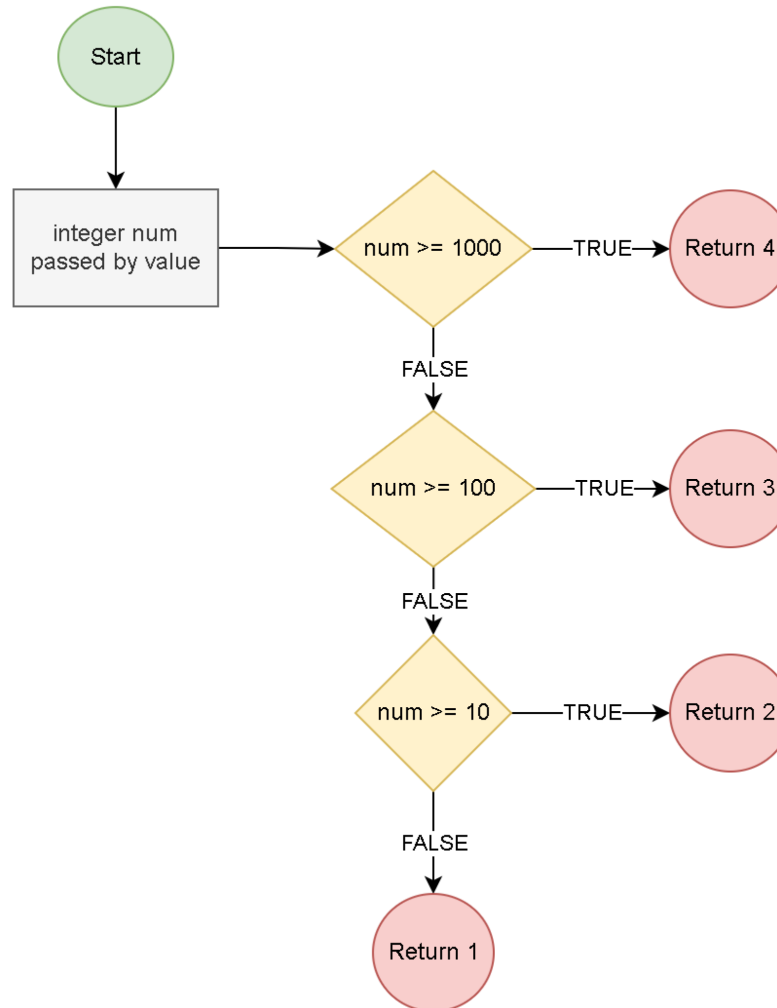
We have implemented three functions to accomplish the task of printing strings and numbers to the screen:

void alt_print(char * string):



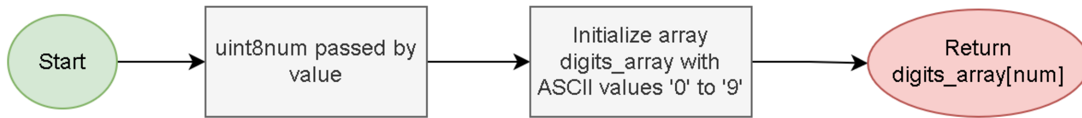
In this function we iterate over a C-string until the null terminating character is encountered, display each character in the C-string using the Dragon12 premade assembly subroutine outchar1.

uint8 get_num_length(uint16 num):



In order to display an integer to the screen, we need to know how many digits it contains so that when we convert it to a string, we can place the null character at the end of it. We do not expect an integer to ever be more than four digits long.

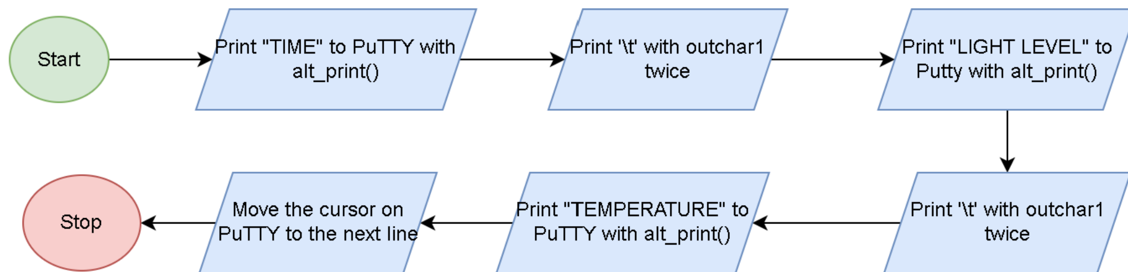
uint8 get_num_char(uint8 num):



This function is for converting a number to an ASCII value that we can then display on PuTTY. For this function, we create an array digits_array that holds the ASCII values for the numbers 0-9 in that order. Then it is only a simple matter of returning the element at index num which was provided by the calling function.

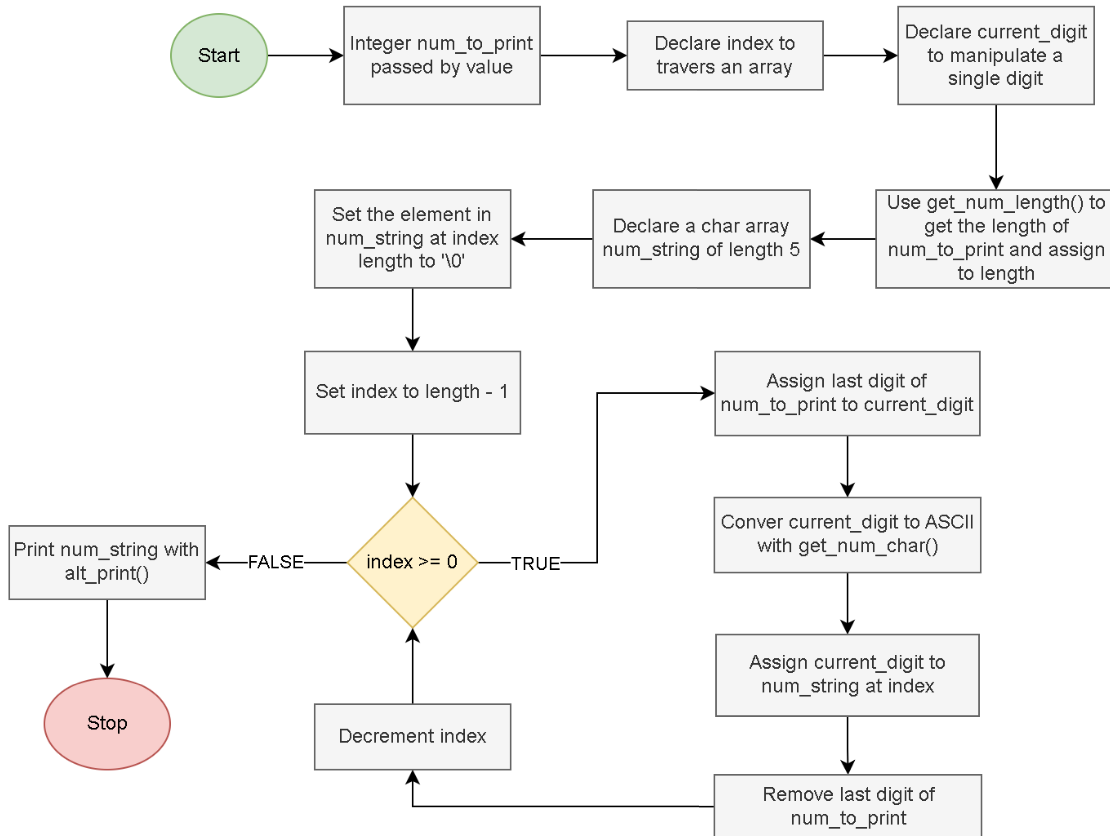
With the above three functions implemented, we can display all data using the following functions:

void write_labels(void):



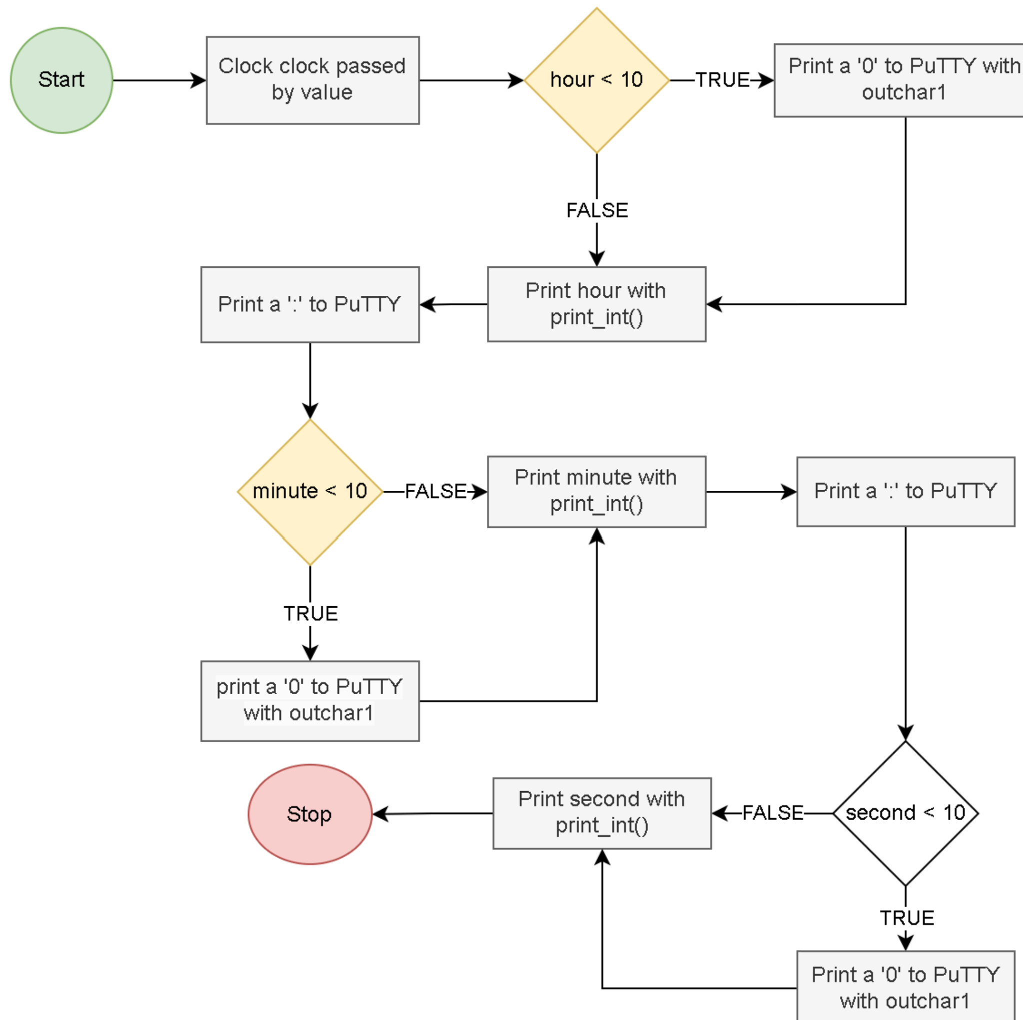
Using our alt_print() function and the Dragon12 outchar1 assembly subroutine, we display the headers of our data chart to PuTTY.

void print_int(uint16 int_to_print):



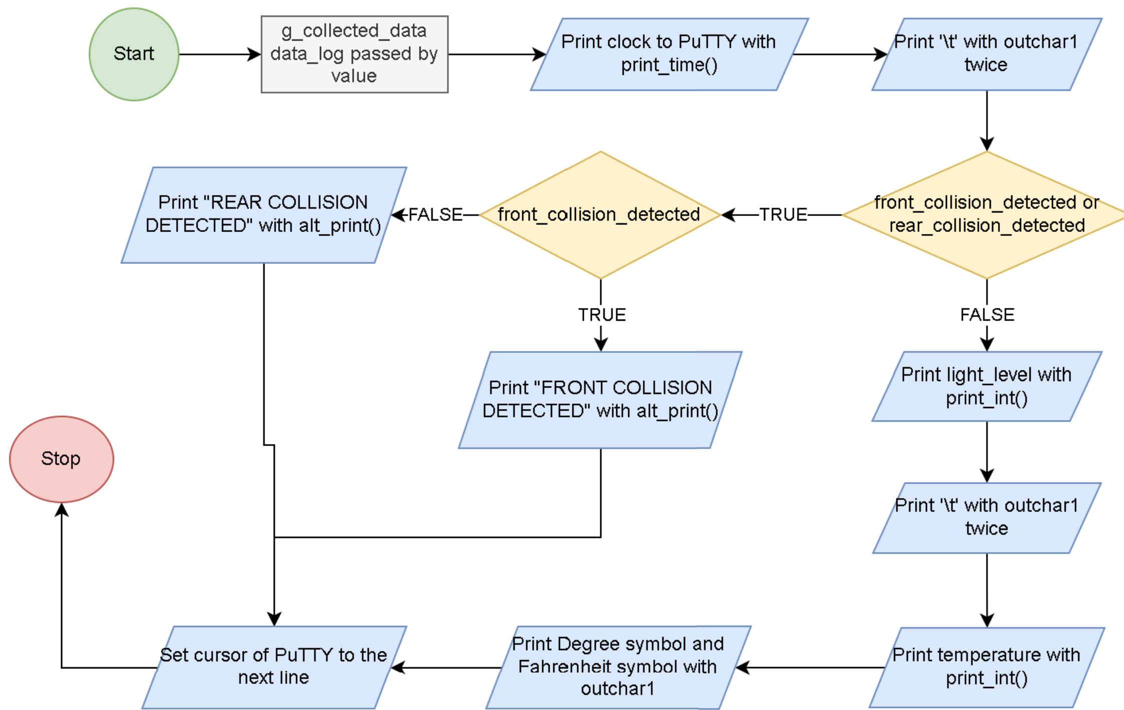
This function converts an integer to a string and prints that string to PuTTY. We start by declaring an empty char array of length 5 as we do not expect to have a number greater than 4 digits long with the final element being the null terminating character. Next, we get the number of digits in the integer and we set that element in the array to the null terminating character. Then we go through each digit of the integer, starting with the last digit, and use the get_num_char() function to convert each digit to its ASCII equivalent and assign it to the array. Finally, we print the string to PuTTY with alt_print().

void print_time(Clock clock):



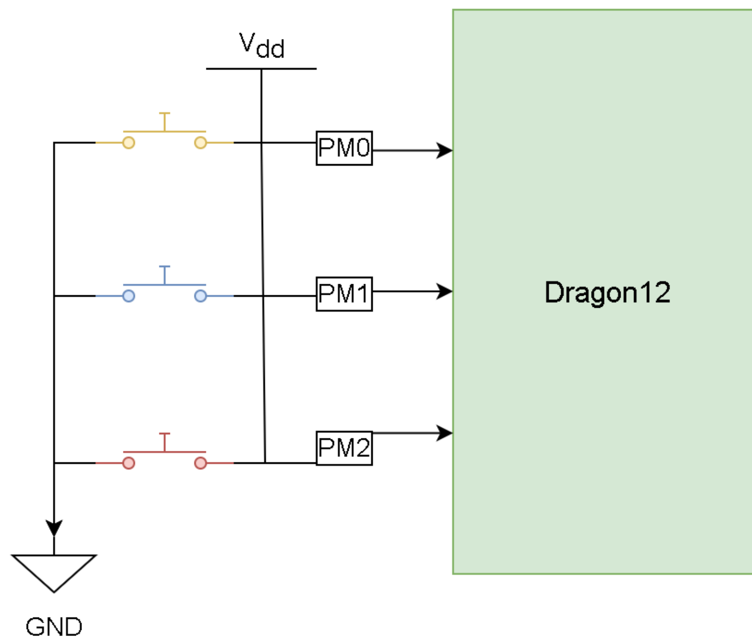
This function displays the time to PuTTY. For each value of clock—hours, minutes, seconds—we first check if it is less than ten. If it is, we first print '0' to the screen using outchar1 before print the clock value using print_int(). If the clock value is 10 or greater, we can just use print_int().

void write_to_putty(g_collected_data data_log):



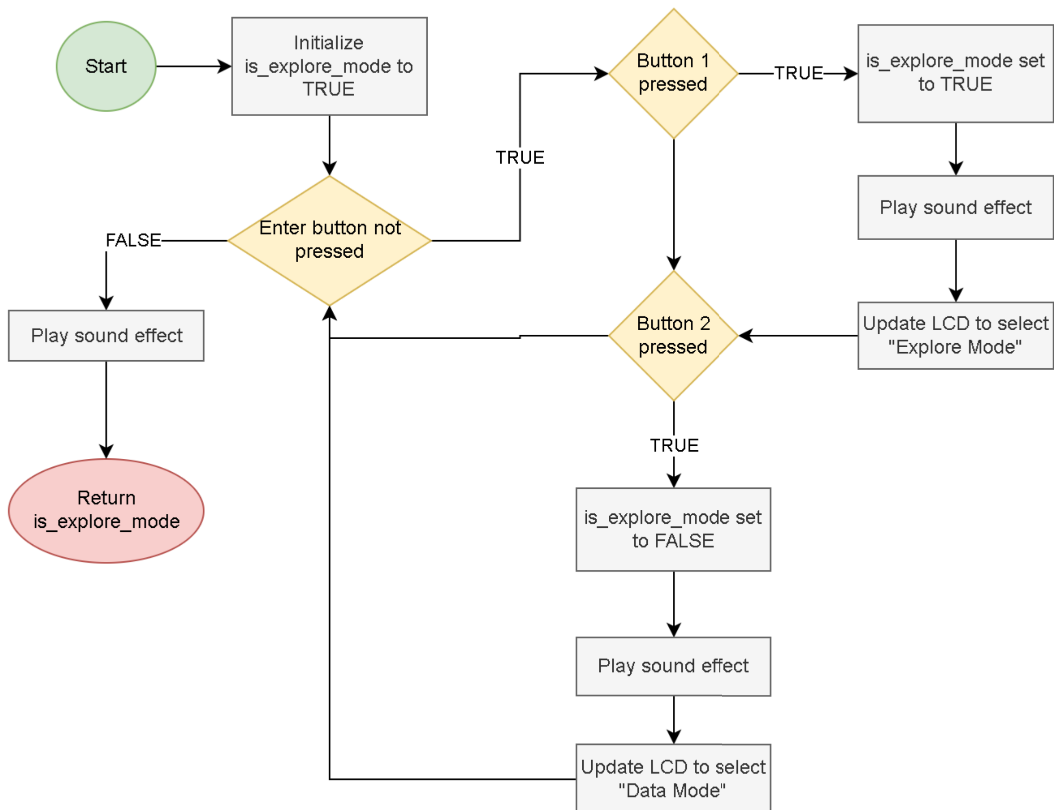
With the provided g_collected_data argument provided by the calling function, we display all values stored in data_log. We print data_log.clock using print_time(). However, depending on whether there was a collision detected impacts how the data is displayed. If either a front or rear collision was detected, we do not display the environmental data. We instead display whether or not a front or rear collision was detected using alt_print. If there was no collision detected, we display the light level and temperature using print_int(). The outchar1 calls we make are for formatting purposes.

Button- and LCD-based User Interface:



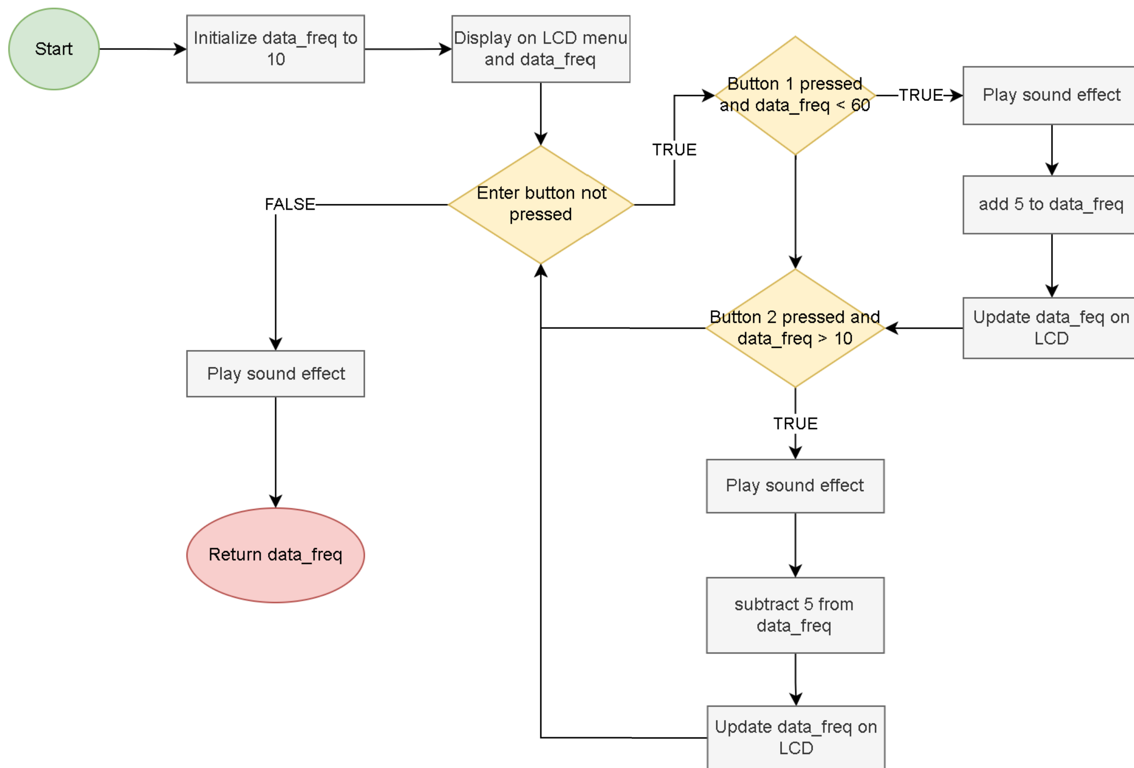
We use three buttons on the Dragon12 connected through Port M, bits 0-2, for traversing through a menu. We use the following three functions for getting input from a user:

uint8 get_mode(void):



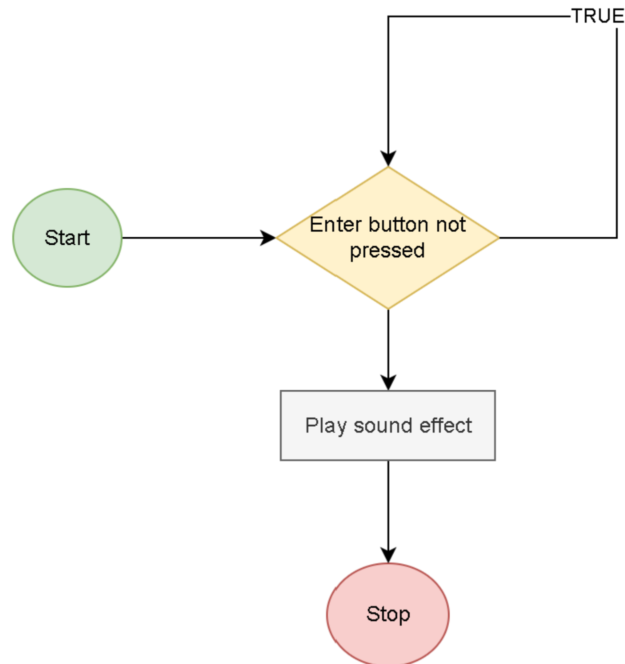
In this function, we use a while loop to wait for a decision from the user. As long as the enter button was not pressed (Port M bit 2) we check whether button 1 was pressed (Port M bit 0) or button 2 was pressed (Port M bit 1). If button 1 was pressed, we set is_explore_mode to TRUE and move the '*' character to the first line to show that "Explore Mode" was selected. If button 2 was pressed, we set is_explore_mode to FALSE and move the '*' characer to the second line to show that "Get Data" was selected. We use a sound effect each time. Once the enter button is pressed, we can then return is_explore_mode.

uint8 get_frequency(void):



Similar to get_mode(), we use a while loop to wait for a decision from the user; however, we increment or decrement the frequency at which data is collected with the pressing of buttons 1 and 2, respectively. The frequency of data is collected cannot be incremented over 60 seconds or decremented less than 10 seconds. Since the enter button is pressed, we return the frequency at which data is collected.

void wait_for_enter_press(void):



In this function, we simply have a while loop that waits for the enter button to be pressed.

Music and Sound:

There are three premade assembly subroutines provided for the Dragon12 that could allow us to use the timer to pulse sound through the speaker for PT5; however, since we also use the timer to control the motors, using the assembly subroutines as written would interfere with the operation of the vehicle. Therefore, we have implemented our own assembly subroutines noise_init, start_sound, and stop_sound to work with the timer and the speaker on PT5 without interfering with the motor control.

void noise_init(void):

The design of this subroutine is very similar to the premade sound_init subroutine. We set channels 5 and 7 as output compares in the TIOS register and set the frequency to 1.5MHz. We then set the pulse train out PT5 on a TC7 match. Lastly, which differs from sound_init, we disconnect the timer from the output pins so that the Dragon12 does not try to pulse through the speaker while the timer is running but sound has not been turned on. The following is the pseudocode we follow:

```
// Set TIOS for output compares on Channels 5 and 7
// Set TSCR2 for 1.5MHz
// Enable the timer
// Set TC5 and TC7 to the value in TCNT
// Clear OC7M and OC7D so no pulse out PT5
// Disconnect the timer from the output pins
```

void start_sound (void):

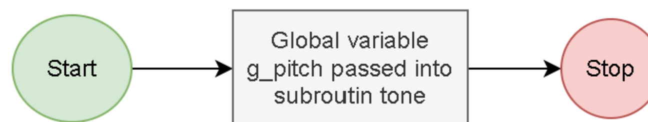
In this subroutine, we enable TC5 interrupts in TIE and set OC7M and OC7D to pulse out PT5 and go high on a TC7 match.

void stop_sound (void):

In this subroutine, we disable TC5 interrupts in TIE, clear OC7M and OC7D so there is no pulse out PT5, and disconnect the timer from the output pins.

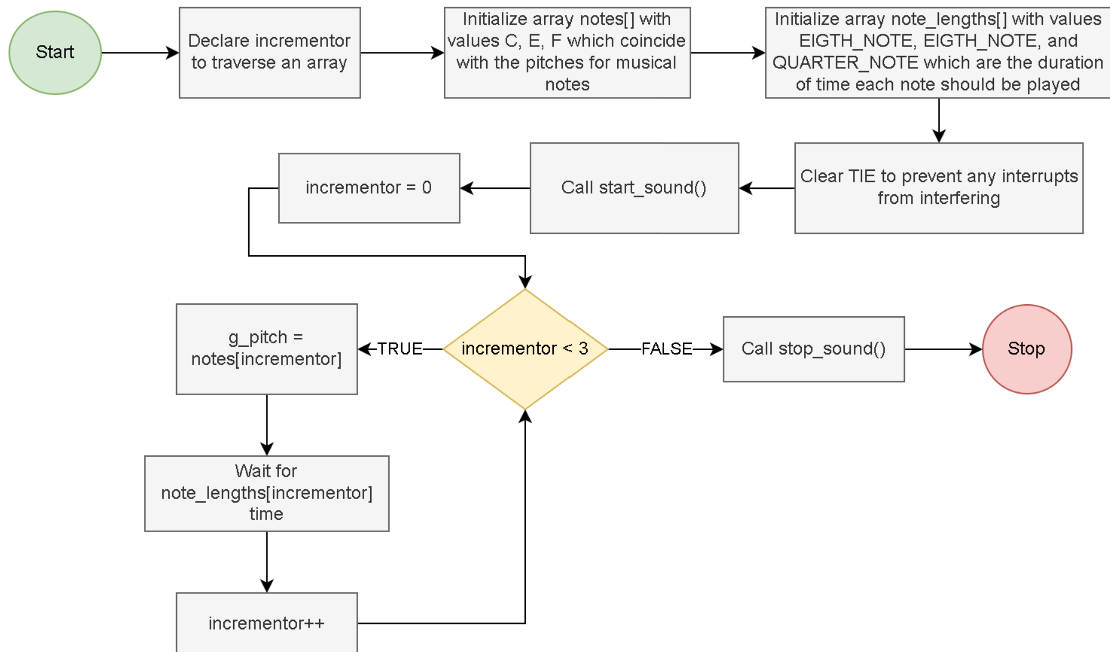
With the above subroutines implemented, we can start playing musical notes on the speaker. After initializing the pulse train for the speaker using noise_init, we use the premade assembly subroutine tone and an interrupt function noise_maker to play individual notes whenever we call start_sound. In addition, we have a global variable g_pitch which holds the pitch of a musical note. This variable is passed into the tone assembly subroutine.

void interrupt 13 noise_maker(void):

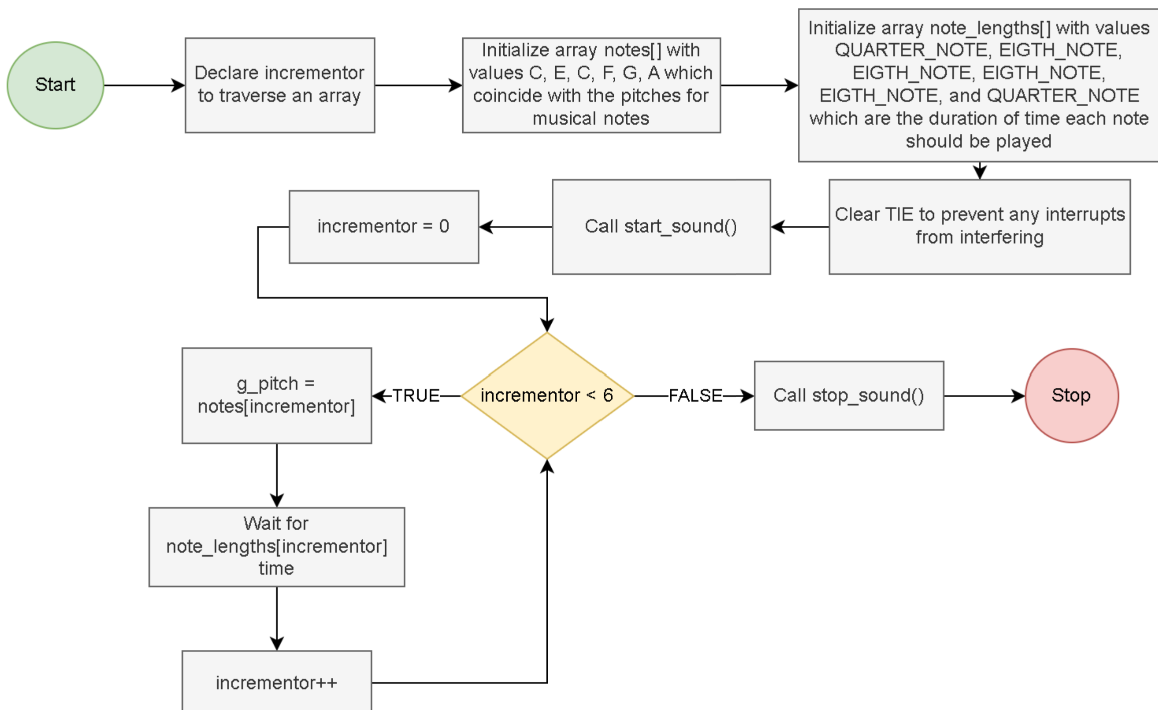


The following functions are implemented to play various songs and sound effects:

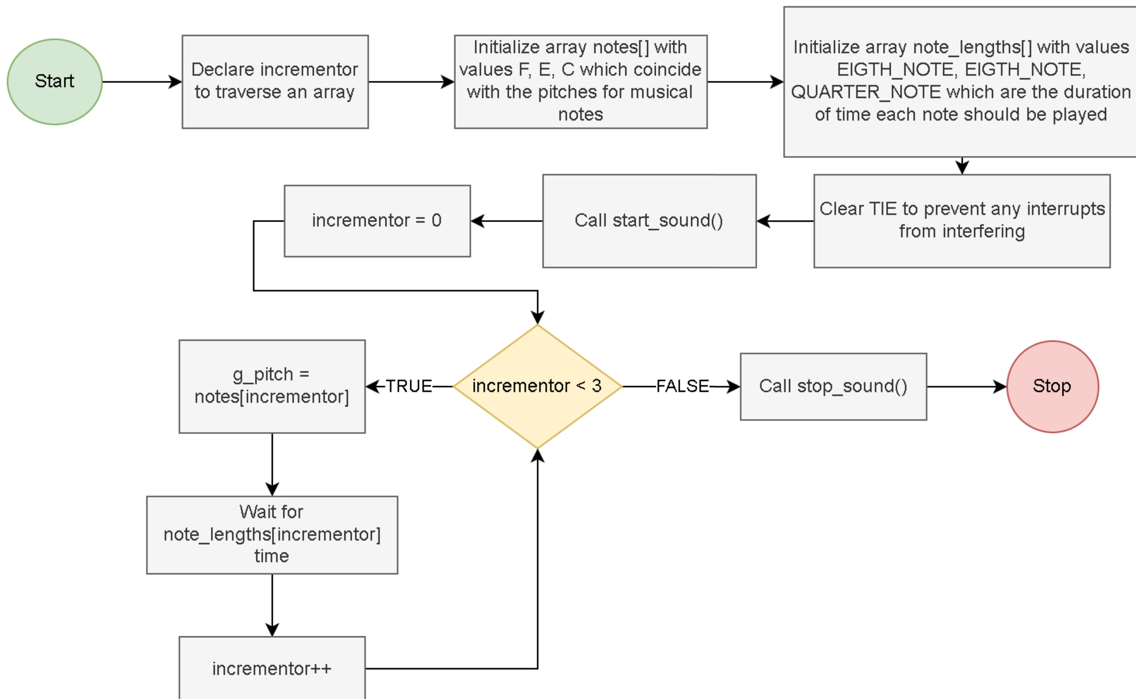
void start_jingle(void):



void explore_jingle(void):

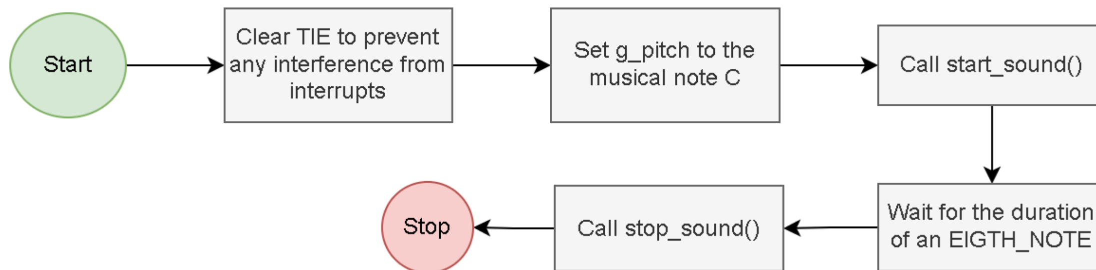


void end_jingle(void):



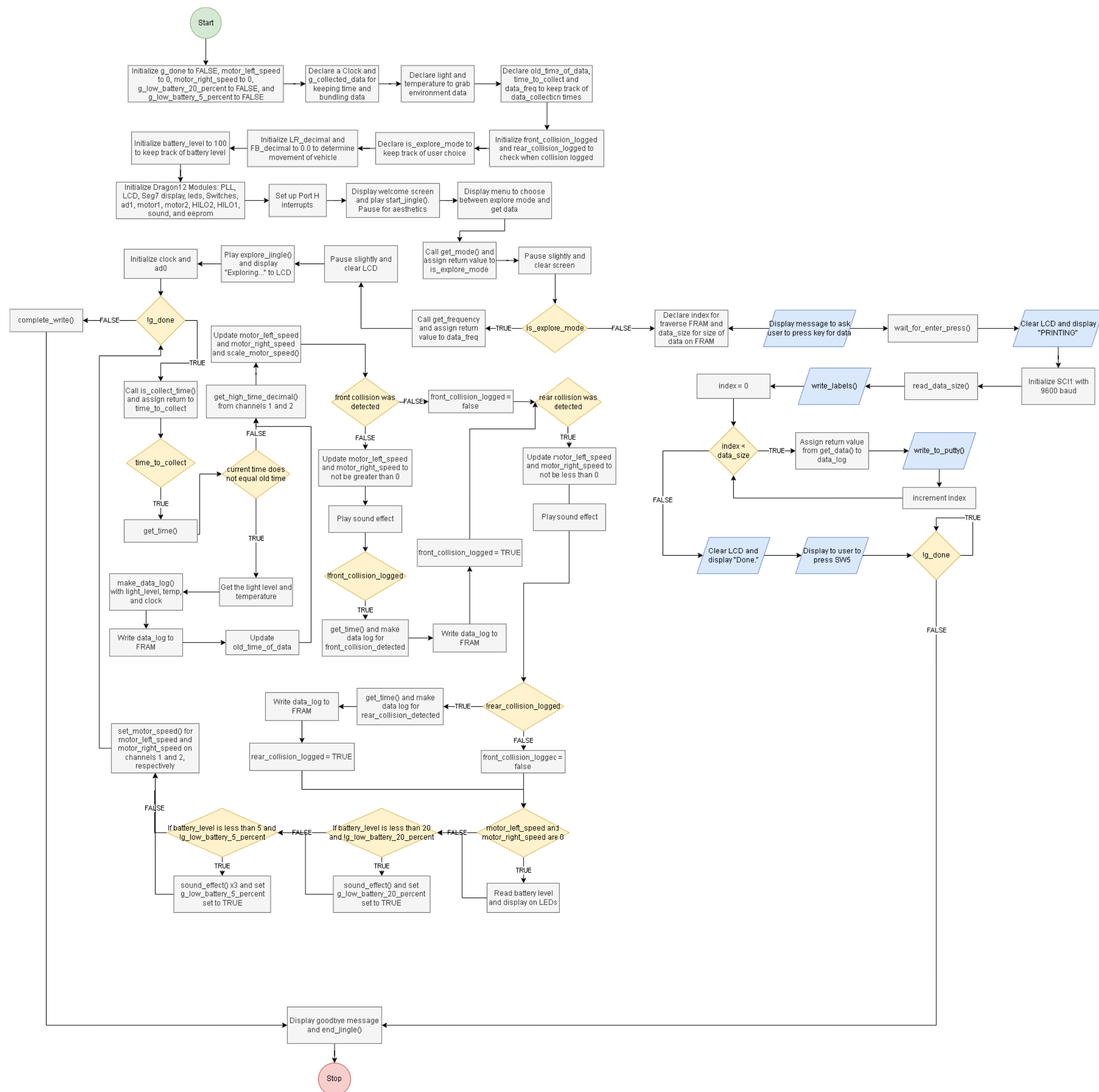
The above functions have the same structure. We initialize two arrays: one that contains the pitches for the musical notes to be plays, and one for the durations each note should be played. We then clear TIE to ensure that there weren't already interrupts being enabled. Next, we call start_sound and iterate over the arrays setting g_pitch to the current pitch and waiting for the current duration using the premade assembly routine ms_delay. Finally, we call stop_sound.

void sound_effect(void):



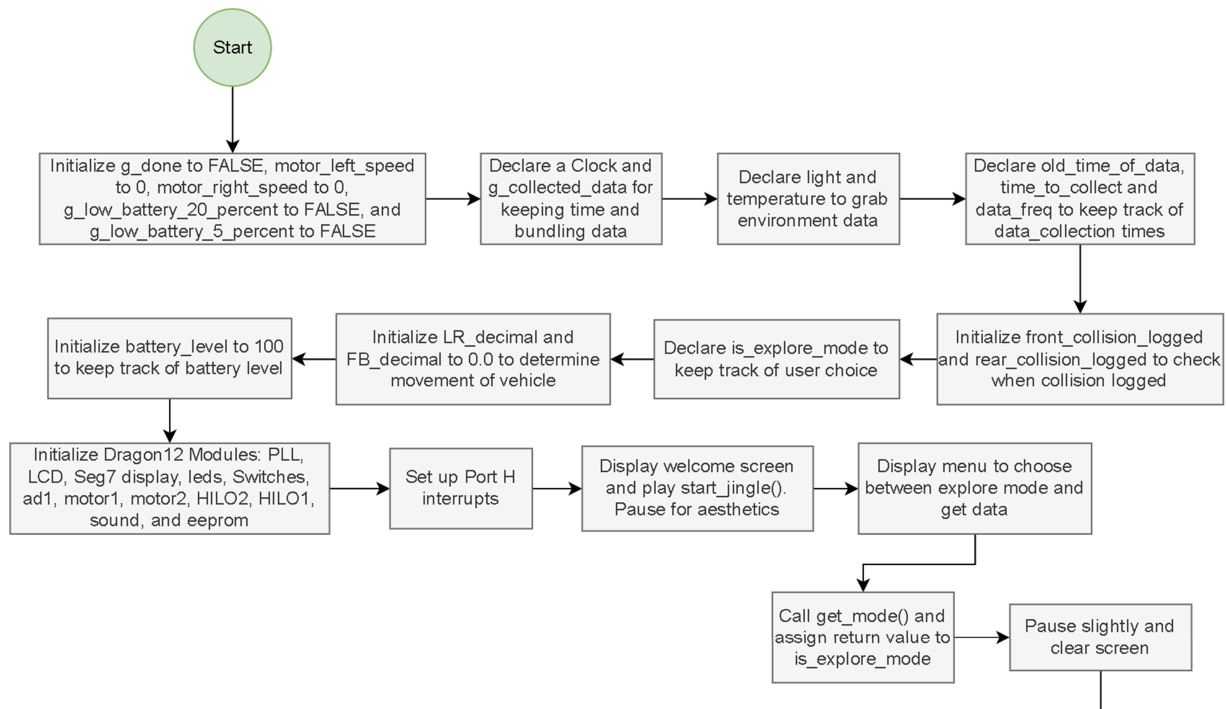
In this function, we clear TIE to ensure that there were no other interrupts interfering. Then we set g_pitch to the musical note 'C', start the sound, wait a moment, and finally ending the sound.

Putting It All Together:



Above is the entire main() function. It is quite large so we will be breaking it down into parts.

Part 1: Initializing the program

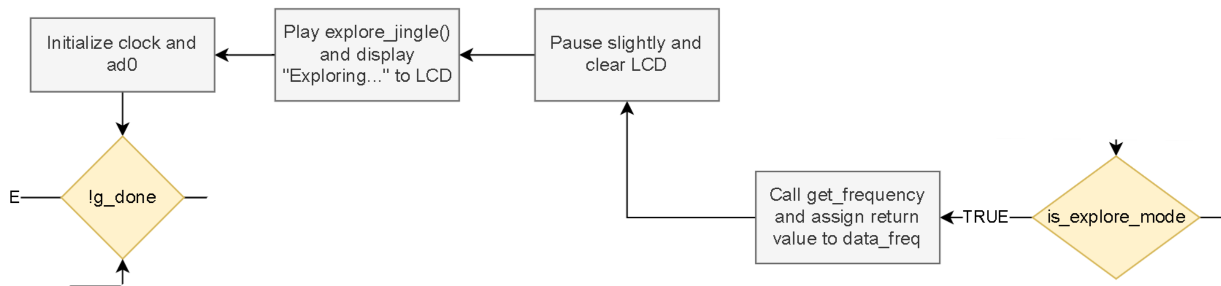


We begin by initializing our global variables. Some of note: g_done is our flag that allows us to step out and end the program when SW5 is pressed on the Dragon12. motor_left_speed and motor_right_speed are set to 0 and will help determine both the direction and speed the wheels on the vehicle spin. front_collision_logged and rear_collision_logged are flags that prevent the vehicle from trying to log the same collision multiple times. LR_decimal and FB_decimal are for reading data from the remote control that we can then apply to the motor speed and direction.

Once our variables are declared/initialized, we initialize all the systems needed to start the vehicle. Of note is the 7-segment display. Although we disable it in our design, due to the construction of the Dragon12, lights on the 7-segment display come on during driving of the vehicle. This is due to the fact that both the 7-segment display and PWM use Port P.

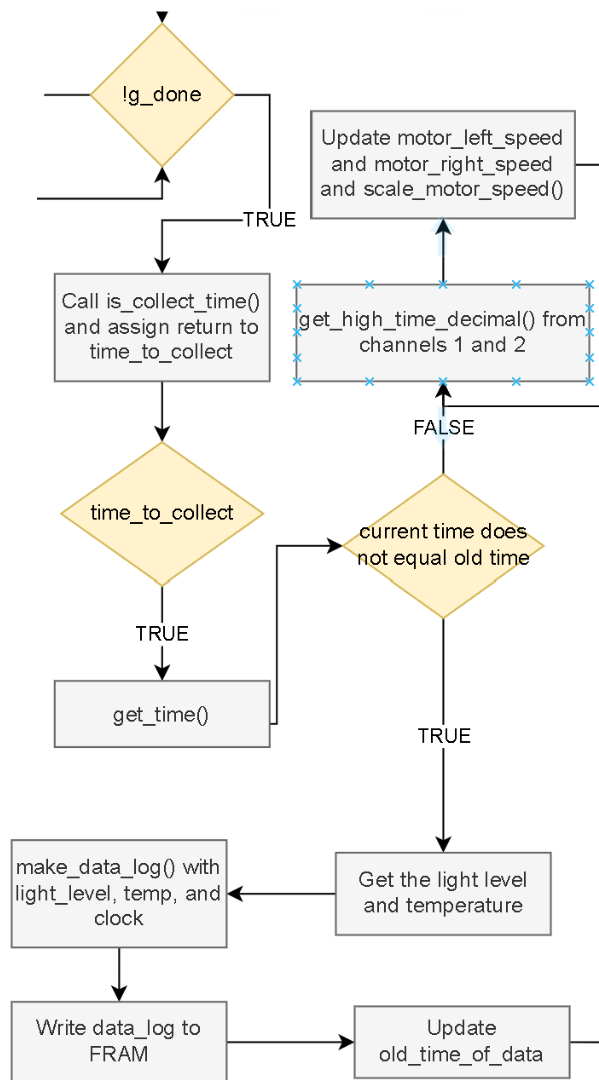
Once our Port H interrupts are prepared and the vehicle started, the vehicle waits for a selection from the user with get_mode().

Explore Mode



If explore mode is selected, we get the desired frequency of data collected from the user with get_frequency(). Then we play music to indicate that exploring has started and we initialize the clock and ad0 for collecting environmental data. At this point we enter the main loop that only finishes once SW5 is pressed.

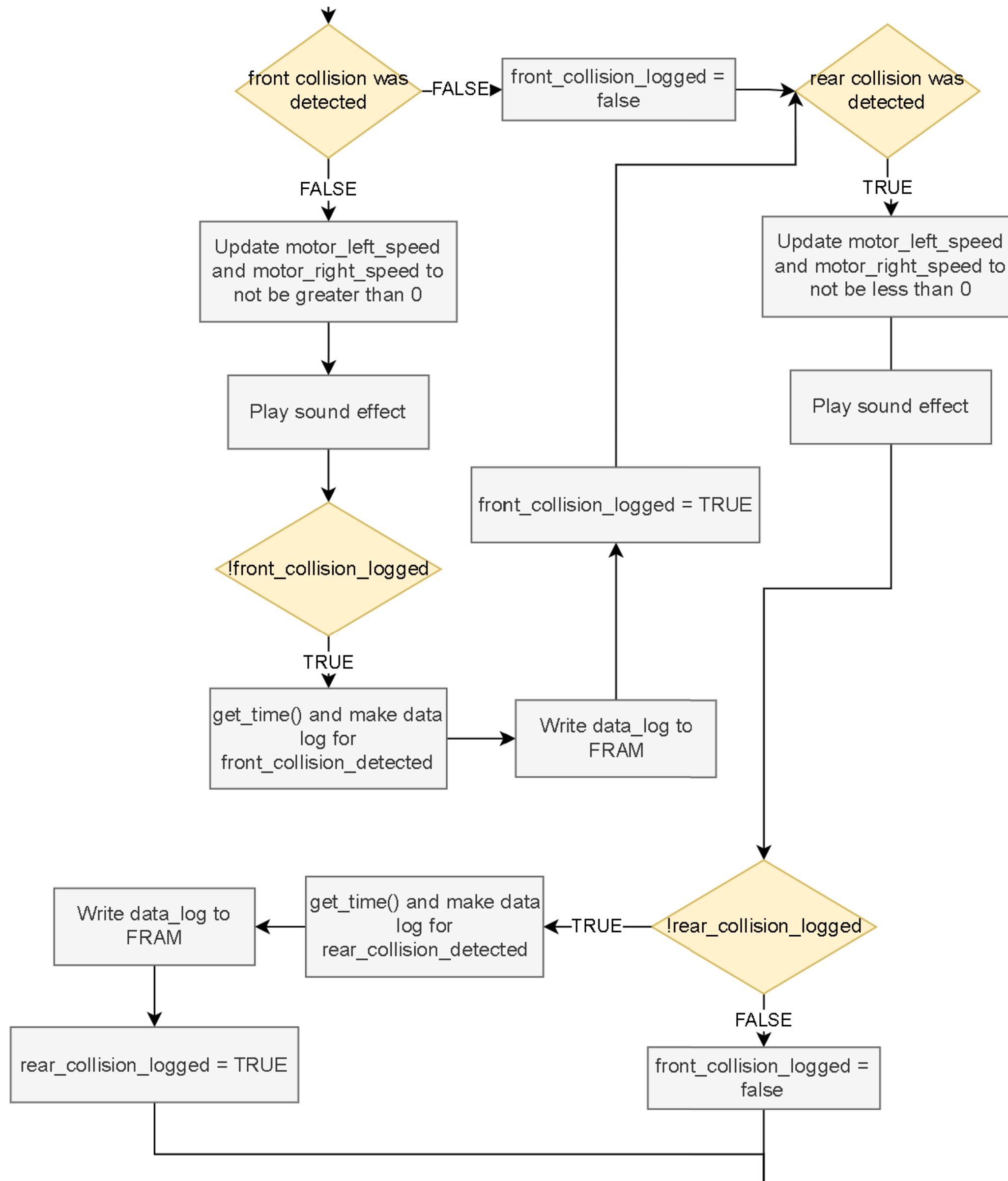
Collecting Data



This first portion of our explore mode loop is to check whether it is time to collect data. We need two variables for this which were initialize at the start of this program: old_time_of_data and time_to_collect. In order to prevent the program from collecting data with every iteration of the loop, we need a way to check whether data had already been collected at that time interval. old_time_of_data is initialized with an unlikely value at the start of the program—we used 99—and when the above code portion is encountered, we get the current time and compare the seconds of the current time to the old_time_of_data. As long as they do not match, data can be collected and written to the FRAM. At the end of this if-block, we update the old_time_of_data.

After data is collected (or not), we get the updated values from the remote control and convert them to values that can be applied to the motors. However, these values may change depending on outside conditions as will soon be seen.

Detecting Collisions

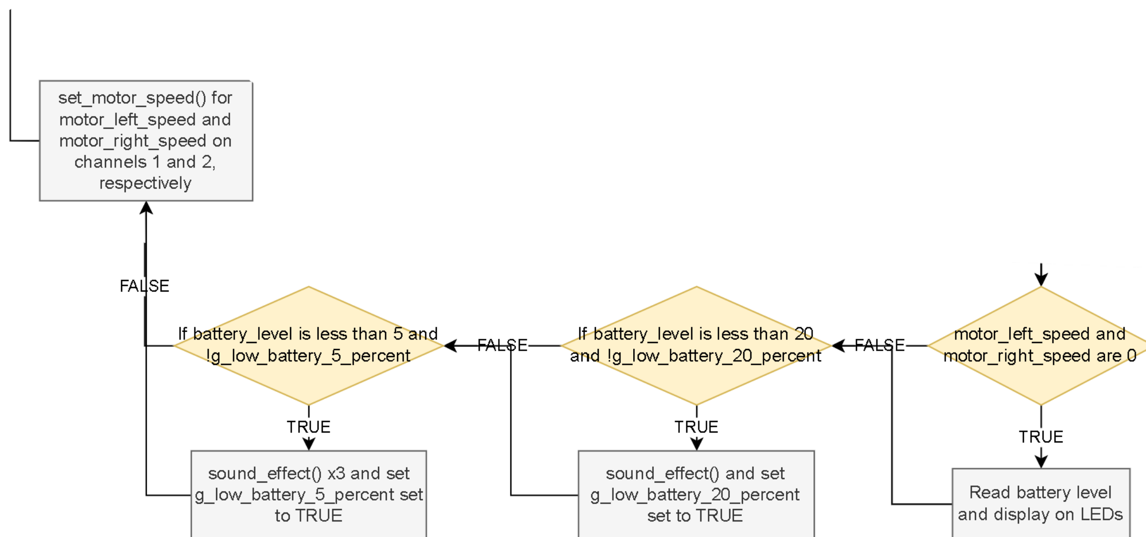


After collecting data and getting updated values from the remote control, our next step is to check for collisions. Our logic in selecting this specific order of events is that these sections of code may modify the motor_right_speed and motor_left_speed values depending on whether a collision was detected. If a front collision was detected, we modify motor_right_speed and motor_left_speed so that neither is greater than 0. If either one is, then we set that variable to 0

to prevent any forward movement in that motor. The same logic applies with rear collision, only we prevent motor_right_speed and motor_left_speed from being less than 0.

It is in this code that we log the collisions as well. We use flags to prevent the same collision from being logged more than once. When a front or rear collision is detected the first time, it is logged and written to the FRAM; however, the flag is set to TRUE so that we do not enter that if-block again. The flag is set to FALSE again once a collision is no longer detected.

Battery Monitoring

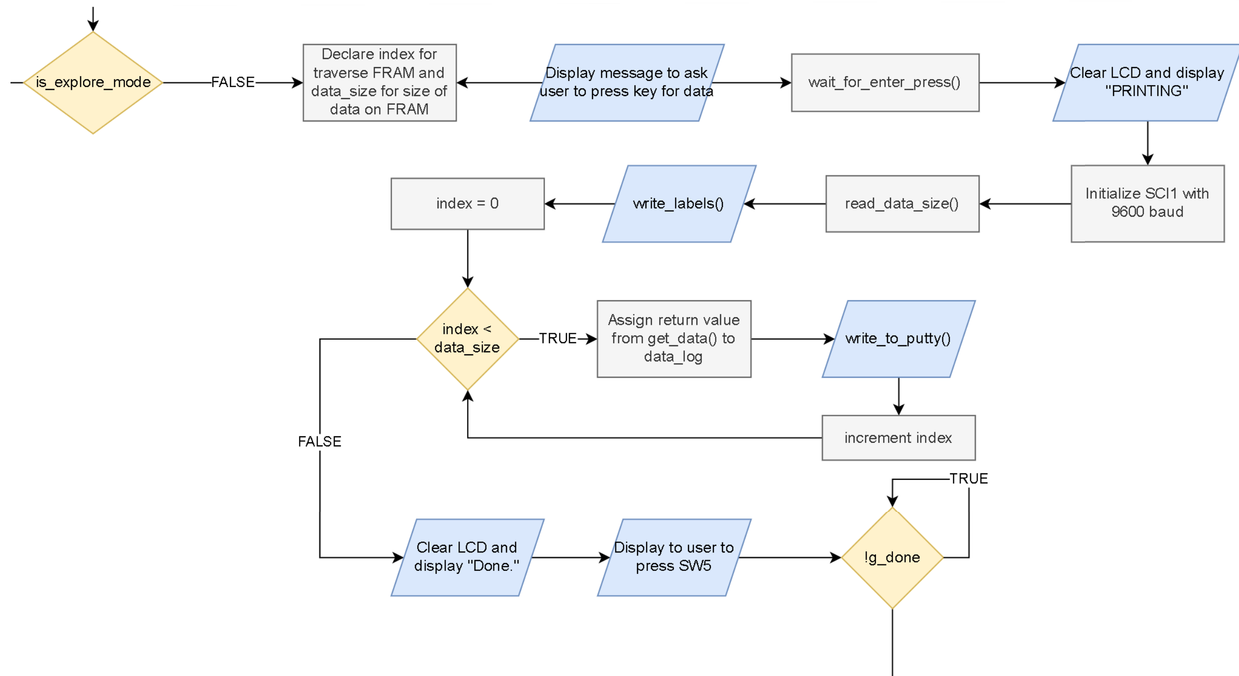


The final portion of the explore mode loop is detecting battery level. We update the battery level on the LEDs only when the vehicle has stopped moving, however briefly. This saves on energy use as the vehicle is not expected to be constantly moving in one direction or another, so only updating the display when the vehicle has stopped is satisfactory.

We also check for battery levels and alert the user. With flags, we prevent the vehicle from constantly beeping when low on battery. One beep is enough at 20% to give the user warning. At 5% the vehicle is no longer allowed to keep moving.

Lastly, we set the speeds of our motors and the vehicle moves. This completes the explore more loop.

Part 2: Data Mode

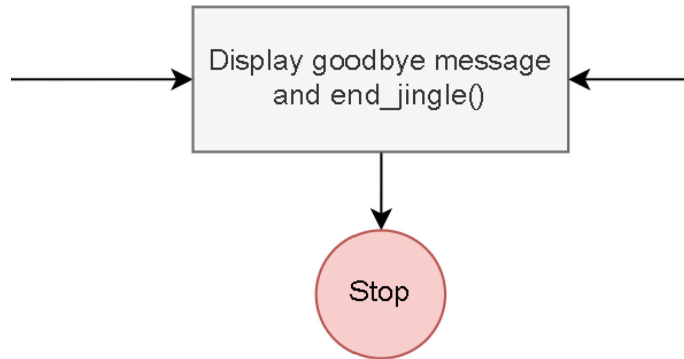


This portion of the code is much simpler than Explore Mode as much of the work of has already been done in eeprom.c and output.c. When the user selects to Get Data, we enter the Data Mode portion of the code. It is here that we read the data from the FRAM.

Ideally, the user will have already connected his/her computer to the serial port, but we display a message to press enter when ready to give him/her time to get set up.

We initialize SC1 with a 9600 baud rate and read the size of the data from the FRAM. With data_size, we can then iterate through the FRAM much like we would an array, and print out each data_log. Once finished, the program waits for the user to press SW5.

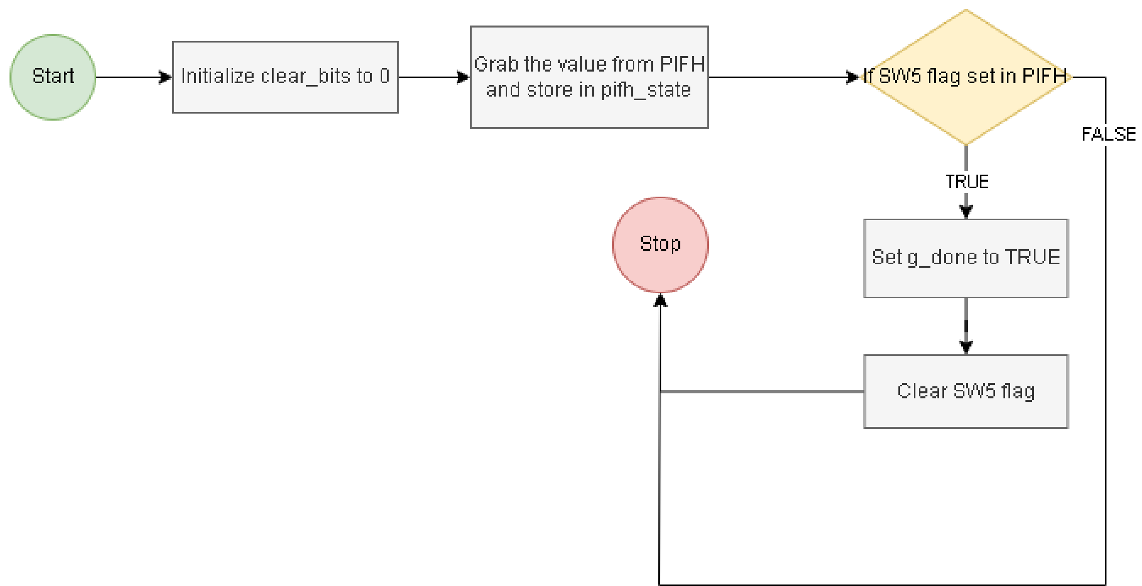
Once SW5 is pressed, either in explore mode or data mode, a final message is displayed and the program ends.



Briefly on the Interrupts:

Our main.c uses 3 interrupts:

void interrupt 25 detect_switches(void):



For detecting when SW5 is detected. We can then set g_done to TRUE and end the program.

void interrupt 9 handler1(void)

void interrupt 10 handler2(void):

Handler1() and Handler2() are only used to measure the high/low time of the RC receiver pulse width. They are triggered on the rising and falling edges of the PWM signals on channel 1 and channel 2. See the Timer chapter in *Programming the DRAGON12-Plus-USB* for more details.

Future Work Needed:

Features we did not have time to implement:

- Replace the motor driver power switch with a relay, so that the Dragon12 board can enable power to the drivers automatically after it has started up.
- Improve wire management
- Instead of using SW5 to end the program, set a button on the remote control to do so.
- Add a humidity sensor to also collect humidity readings.

References/Disclosure:

Haskell, Richard E., and Darrin M. Hanna. *Programming the DRAGON12-Plus-USB in C and Assembly Language Using CodeWarrior*. LBE Books, 2011.

McMahon, Russell. "Algorithm for Mixing 2 Axis Analog Input to Control a Differential Motor Drive." *StackExchange*, 19 Sept. 2011, https://electronics.stackexchange.com/a/19671.

MB85RS64V. FUJITSU SEMICONDUCTOR LIMITED, Sept. 2013.

Appendix A. (Wiring Diagram)