Aasta Gandhi (apg67)
Khyati Sipani (ks965)

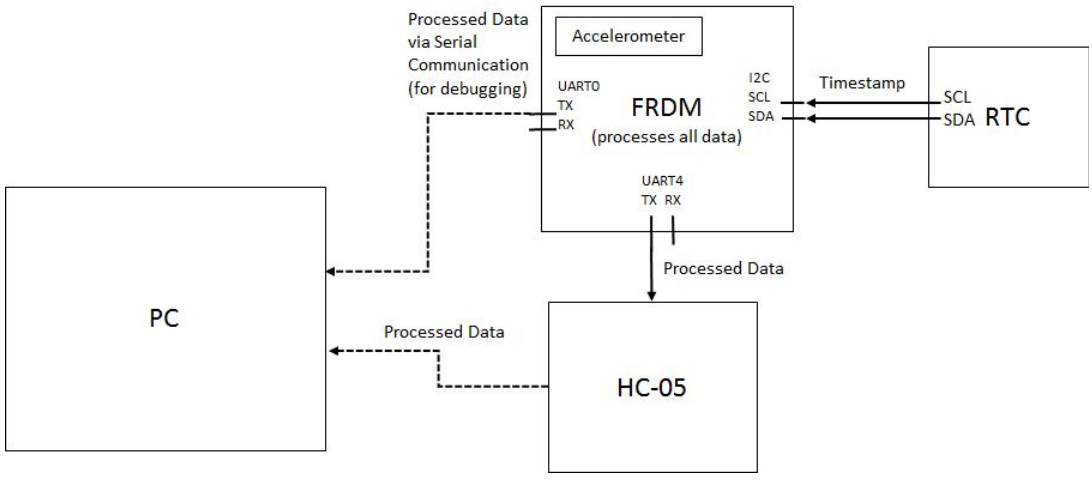
3140 Final Project: Speed Tracker

Introduction

The goal of this Speed Tracking device is to continuously monitor if a vehicle speeds more accurately. Using the accelerometer to check if a moving system is “speeding” (i.e moving faster than set ranges), the speed tracker will time stamp every instance of speeding using a real time clock, and send that data to the main system via bluetooth. It will also caution the user by blinking the LED on the board with a different color to indicate different modes of speeding. The motivation for this project is to make current speeding systems more efficient and effective at monitoring if vehicles are actually speeding intentionally as well as to caution the driver that they are speeding.

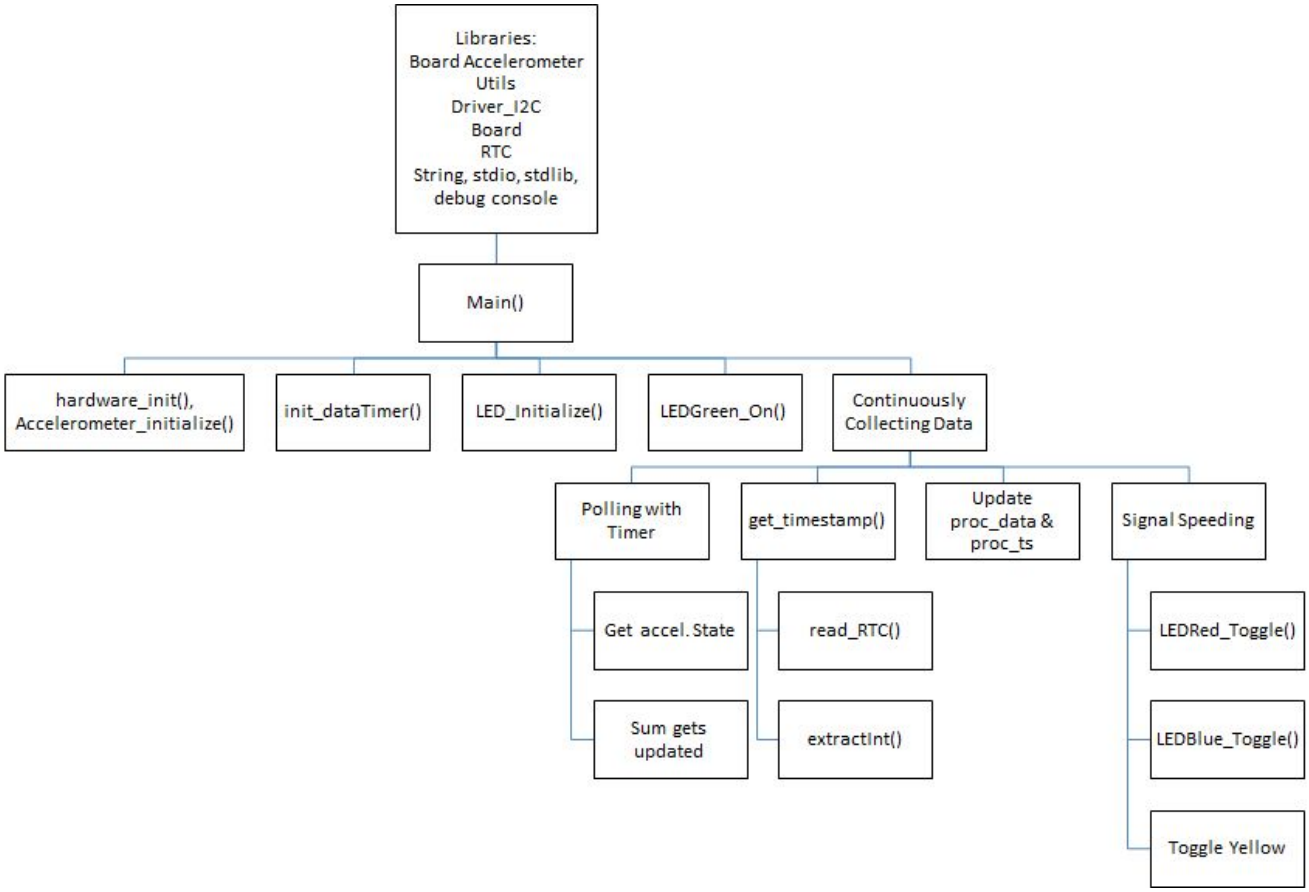
System diagram

Fig 1. Higher Level Schematic of Hardware and connections. In this diagram, the RTC is connected to the board via the SCL and SDA pins (following the I2C protocol). When the board is not powered, the RTC runs on an external battery cell. The HC-05 Bluetooth module is connected to the FRDM via universal asynchronous transmitter and receiver (UART).



Aasta Gandhi (apg67)
Khyati Sipani (ks965)

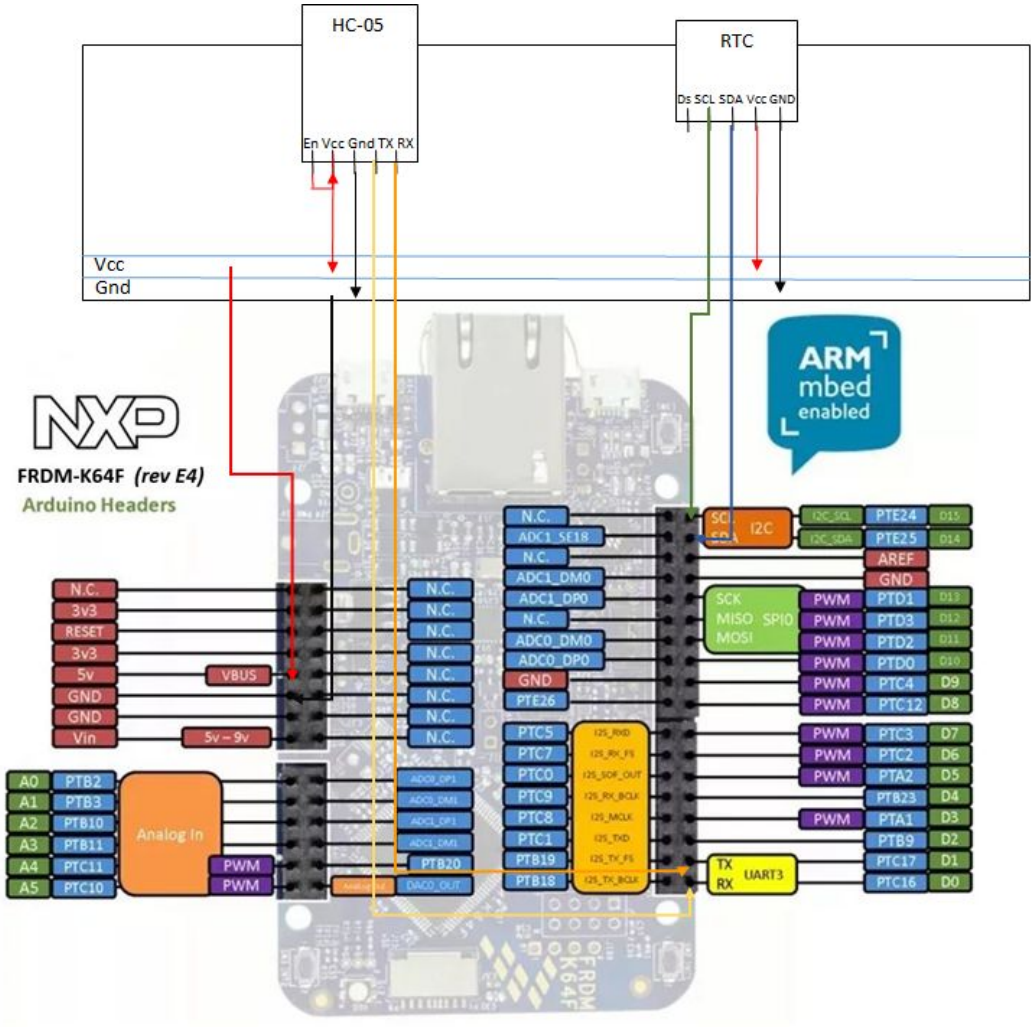
Fig 2. Higher Level Schematic of Software. This flowchart shows all the methods and libraries that were used to process our data and run our algorithm. We wrote and modified Utils.c/Utils.h, RTC.h/RTC.c, changed the configuration for I2C in RTE_device.h() for the RTC, and retrieved values from the Accelerometer libraries which we downloaded from the tutorial.



Aasta Gandhi (apg67)
Khyati Sipani (ks965)

Hardware description

Fig 3. Pin Connections made between the RTC, HC-05, the FRDM and breadboard.



Bill of materials:

1. **RTC:** Donop DS3231 AT24C32 IIC module precision RTC, Price: $4.60
2. **Bluetooth module:** DSD TECH HC-05 Bluetooth Serial Pass-through Module x1, Price: $7.99

Software description

Processing Data

To store data, we initialize two arrays (float proc_data[180] and char* proc_ts[180]) that act as circular buffer. Proc_data[180] will contain the processed velocities from each second for up to 3 minutes, and then start rewriting the values in the array, assuming that the data has already been transferred to the PC. Proc_ts contains the corresponding time stamps for the computed values of

velocity. We chose approximately 3 minutes because we are writing and storing directly on the on-chip RAM which has limited space (256 kb including stack, program etc).

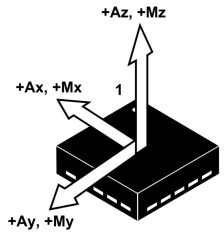
We used polling as our method to process data because our timer load value was 120000 in order to collect data every 2 ms, and we initialized our timer by enabling the timer and setting the load value.

To process data, we had to derive and average the velocity every one second. To do this we collected the x state of the accelerometer in the x direction every 2 ms by polling (we used a timer with a load value equivalent to 2 ms and checked if TFLG == 1). For every one second, we collect 500 accelerations (we wait for the timer to expire 500 times) and then average those data points. In order to calculate the current velocity, we then add the current acceleration to the previous velocity (prevVelocity) using the formula: $v_{curr} = v_{prev} + a_{avg}t$, where t is 1 second, and store the current velocity to proc_data and prevVelocity for the next iteration. We use prevVelocity instead of proc_data[j-1] to avoid the case in which j=0 and cannot access proc_data[j-1] which would have to be proc_data[179]. We then call get_timestamp() to collect the timestamp from the RTC and store it's address in the proc_ts array.

To signal the user if they are speeding, we had a num_speed variable that checks if the user's speeding is intentional or not (i.e to account for sudden jerks, movements, speeding by accident). If num_speed is greater than some defined number, x, that means speeding has occurred x or more times and the user will be signaled by a change in color in the LED. If speeding is occurring at greater than or equal to two times a defined speed limit and a margin of error, then the red LED will toggle. If speeding occurs 1.5 times to two times the speed limit + margin, the blue light will flash. If speeding has occurred up to 1.5 times the total limit, a yellow light (Red and Green LED) will flash. If the user is not speeding and the system is on, the green light will remain on. At the end of this process, we increment our counter for proc_data and proc_ts to iterate over the next second or set our counter to zero if the array is full.

Accelerometer

The accelerometer collects data in 6 axes, but we only processed data from the x-axis to observe horizontal movement; this was called pstate.x. The sensor collects data in milli-gravity which we converted to decimeters/second^2 (dm/s^2) by multiplying the state of the accelerometer to 0.0980665. We chose decimeters as our unit for displacement because meters makes the values too small to demo with and interpret and centimeters/millimeters outputs values that are larger and more sensitive to change in movements.



Top view

When processing data, we had to take into account the orientation of the board and which direction, relative to the moving object, is the positive and negative x axis (we followed the orientation on the right). This orientation also affects what accelerations are positive and negative. For example, if we are increasing speed in the positive x direction, pstate.x will be positive, but if we are decreasing speed in the positive x direction, pstate.x will negative. In the negative x direction, pstate.x will be negative when accelerating and positive when decelerating. We accounted for this by orienting the accelerometer and testing in the right direction rather than trying to algebraically manipulate collected data.

Aasta Gandhi (apg67)
Khyati Sipani (ks965)

Real Time Clock

The real time clock is used to collect time stamps every second, which we associate with the speeds that we collect. We used I^2C to communicate with the real time clock. I first set the clock by defining a set_clock() method which is in RTC.c Using the datasheet of the DS1307 real time clock, we figured out which registers we had to write to and what value we had to write to them.
Figure 4 shows this table.

ADDRESS	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0	FUNCTION	RANGE
00h	CH	10 Seconds			Seconds				Seconds	00–59
01h	0	10 Minutes			Minutes				Minutes	00–59
02h	0	12	10 Hour	10 Hour	Hours				Hours	1–12 +AM/PM 00–23
		24	PM/ AM							
03h	0	0	0	0	0	DAY			Day	01–07
04h	0	0	10 Date		Date				Date	01–31
05h	0	0	0	10 Month	Month				Month	01–12
06h	10 Year				Year				Year	00–99
07h	OUT	0	0	SQWE	0	0	RS1	RS0	Control	—
08h–3Fh									RAM 56 x 8	00h–FFh

Fig. 4

We did not need to initialize the I2C as that was already done by Accelerometer_Intialize(), which also used I2C0. We did have to set the bus speed to ARM_I2C_BUS_SPEED_STANDARD though (100 kHz).

Next, In Driver_I2C.h, the library associated with the I2C Driver, there were two functions called Master_Transmit() and Master_Receive. Master_Transmit() has parameters addr, *data, num, and xfer_pending. Since I was using the method MasterTransmit() to set the RTC clock, I called the function with the slave address (RTC Address) which was obtained from the data sheet, a pointer to an array called “data” where data[0] contained the address of the seconds register, and data[1]...data[8] contained the values I wanted to write to the seconds register, minutes register, hours register, and so on respectively. The argument num I passed in was 9 since I was passing in 9 bytes. Xfer_pending means that the transfer condition is still pending, and passing this as true would mean that the STOP condition would not be generated. Hence, I set this parameter to false. As a check, I put a while(ptrI2C->GetStatus.busy()); followed by if(ptr->getDataCount()!=9 {return -1}; Else, it returns a zero. The purpose of that was to check if the driver is busy and wait if it is, and to check if all the bytes were written by using getDataCount(). This was also done in the implementation of the accelerometer, hence I followed the same pattern.

When I first set the clock, I made a mistake with setting the data. Hence, I modified set_clock() to just set the date. Hence, the implementation of set_clock() in our code currently only sets the date of the clock. Once the date of the clock is set, there is no need to call set_clock() again as the RTC remembers that date.

Next, I implemented a method called read_RTC(). read_RTC() reads the seconds, months, hours, day, date, month, and year registers in order to generate a timestamp. In order to read off the RTC, I

first called MasterTransmit() with the RTC slave address, and a pointer to an array of length one contained the address of the seconds register, which was the register I wanted to read from. I put the same check as in set_clock for the same purposes: while(ptrI2C->GetStatus.busy()); followed by if(ptr->getDataCount()!=9 {return -1}; Then I called MasterRecieve with the RTC slave address, a pointer of an empty array, and with the number of bytes parameter as 7 as I wanted the 7 timekeeper register values. Following that, I checked the status of the I2C and used getDataCount() again. This saved the values of the timekeeper registers in the empty array.

Finally, in order to generate the timestamp - in my main method I had a get_timestamp() method. Whenever I needed a time stamp, this method called read_RTC() and saved all the values in an array called timestamp. Next, I processed all the values in this array by extracting the “10 second” bits and “second bits” for example (see Figure 1) and doing a conversion as such: seconds = 10*(tenseconds) + seconds. The bits were extracted using the helper method extractInt(). I did such conversions for all the values to obtain integer values for the quantities of time. Then, I concatenated all these integer values using sprintf and stored them in a string that was initialized as a global variable (str[25]). Once get_timestamp() was over, str was assigned to proc_ts[j], which is an array of corresponding time stamps for the collected values of velocity.

Bluetooth

The purpose of the bluetooth is to send the data wirelessly to the PC while a vehicle/user is moving. To set up the bluetooth, we used the universal asynchronized receiver/transmitter (UART) channels. To test serial communication (i.e directly from the FRDM to PC), we initialized UART channel 0 and tested by printing simple arrays of chars by waiting for the transmitting buffer to be empty and ready, and then transmitting the data. Here, we iterated over the array and were successful. Similarly, to test the bluetooth, we first paired the bluetooth with the PC, initialized UART4 and then attempted to transmit the data which was unsuccessful because TeraTerm (the interface we used to receive data from the bluetooth) could not open the COM9 port. To debug this, we tried to configure the bluetooth and tested wireless and serial communication with the arduino. When tested with the arduino, serial communication was successful, but wireless was not. A potential issue is that the bluetooth is not automatically configured with the COM9 port which might have prevented Tera Term from opening the port (because nothing was connected), and why we could not alter bluetooth configurations using AT Commands in arduino.

Testing

We used PuTTY mainly for testing purposes. For example, to ensure that the RTC was collecting the timestamp properly, we used debug_printf() statements to ensure that they were correct. We also printed out values of velocity on PuTTy, and checked if they were reasonable by walking with the board and testing various speeds. We verified speeding with the blinking LEDs. To test if the outputted values were reasonable, we also calculated values ourself and compared them with out program. This fixed a lot of casting issues - initially, we defined our previous velocity, proc_data array and sum in int because we wanted to estimate speed and make the accelerometer sensitivity almost negligible and to easily display on putty, but we later switched to float to detect smaller variations in movement and to

calculate values more consistently with the accelerometer state (which calculates values to more than 5 digits in a defined struct, int8_t). We also used debug mode and various counters that we added to our watch make sure our circular buffer and polling was incrementing correctly and changing values consistently.

To test and resolve issues with the bluetooth, we used test arrays of type char and int to transfer data through both UART channels. We also tested with the arduino. However, as described above, we were unsuccessful in debugging this.

Results and challenges

We faced many challenges while working on the project. While trying to make the RTC work, we used a logic analyzer (after testing with an oscilloscope) provided by our project mentor to see why our RTC wasn't setting properly. Since we could see what values were being sent to the bus, we figured that we were writing to the wrong slave address, and that fixed the issues we were having with the RTC.

The most complicated part of our design was trying to reduce error in velocity computation. The accelerometer is highly sensitive, and did not work exactly the way we want to. As described in the Accelerometer section, the sensor is sensitive to the order of 10^-6 m/s^2 which means that any small movement triggers changes in values. Consequently, if we first speed and then move at a constant value, x, then put the board down on a flat surface, the board will assume you are still moving at that constant speed x, when you're not really moving at all. Here, the board correctly reads that the acceleration is zero, but cannot determine whether the velocity is zero. There were two ways we attempted to correct this:
1) Have a counter that causes the current value to automatically become zero when the user has been moving at a constant speed of zero for a defined number of seconds (continuously compares current velocity to previous velocity). This assumes that no one can travel at constant speed perfectly for an extended amount of time, and if their velocity is being computed as the same it probably means there is some error with the values we are getting or that the board is still. However, we found that the velocity still increased because the acceleration was not perfectly zero (it was rounded to zero or one when casted as an int to print in putty) which also tended to increase the velocity constantly. This occurred in both the negative and positive direction. Additionally, the previous and current velocity will never exactly be the same due to very slight differences in value (i.e to the order of 10^-4). We could have rounded our float values, but because the data is unstable already and casting to float also causes slight changes in value because of the way it is stored in memory, we decided not to tamper with these values further.
2) We attempted to compare the current acceleration to the previous acceleration in a similar manner as above, and found similar issues where the acceleration was not exactly zero so the velocity would also not be exactly zero at any point.

Another challenge was implementing the bluetooth. We were mainly having technical issues with the bluetooth because we were unable to configure and set it up properly as mentioned above.

Aasta Gandhi (apg67)
Khyati Sipani (ks965)

Work distribution

Khyati implemented and debugged the RTC. Aasta implemented and debugged the Bluetooth. We both designed and coded the main algorithm together, debugged it and put the entire system together to test. We checked through each other's code and helped resolve problems along the way. We communicated mostly by working on the project together, in office hours or messaging each other. For the project proposal, we wrote our respective sections and split up the writing of the algorithm, debugging and testing.

References, software reuse

1) K64 Sub-Family Reference Manual
2) http://stackoverflow.com/questions/8011700/how-do-i-extract-specific-n-bits-of-a-32-bit-unsigned-integer-in-c (Retreived May 9, 2017) -- For helper method extractInt()
3) Bluetooth tutorial: https://developer.mbed.org/media/uploads/defrost/frdm-k64_adc_configuration.pdf
4) HC-05 Tutorial: http://cdn.makezine.com/uploads/2014/03/hc_hc-05-user-instructions-bluetooth.pdf