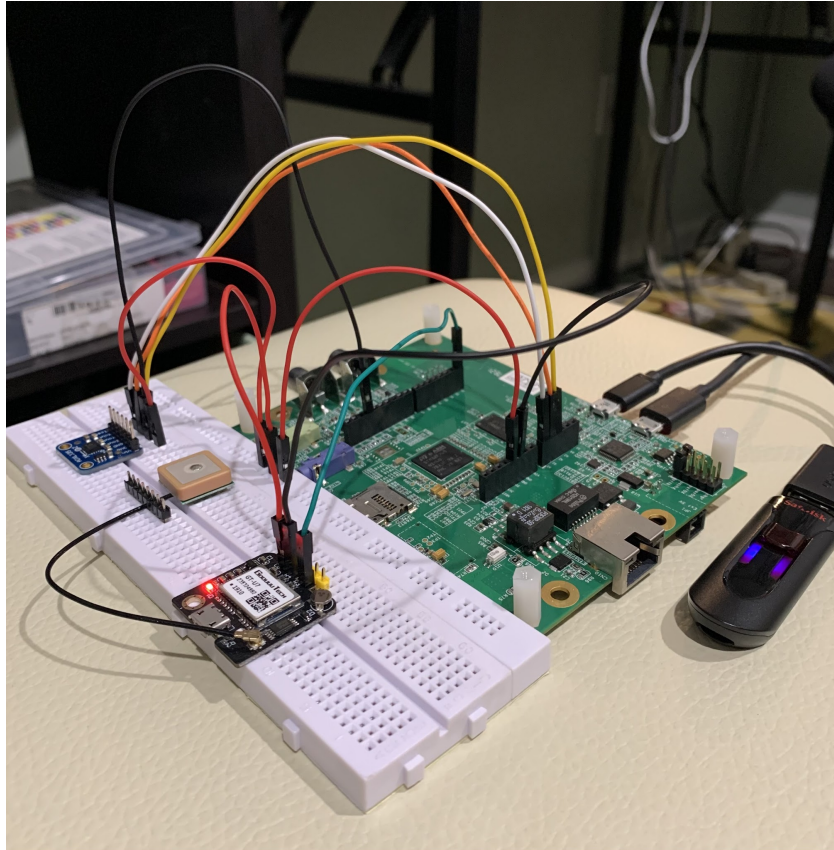


Vehicle Accident Detection System



By Vedant Gannu and Niyanthri Ramaswamy

Introduction and High Level Description

The purpose of the project was to demonstrate a vehicle accident detection system. This functioning was accomplished using an accelerometer and a GPS module. Readings from the accelerometer helped determine when an accident occurred, whereas the GPS module transmitted information that indicated where the accident had occurred. Adafruit ADXL335 accelerometer and GT-U7 GPS module were used in this project.

The code begins with defining its constants and initializing global variables essential for the functioning of the program. Alongside, prototype functions are also defined that aid the main function to execute the program. Inside the main function, with a call to *Sys_Init()*, the microcontroller is booted and system clocks are set. Next the USB host is initialized to enable a USB connection at any point in the program. With the use of the function *configureADC()*, ADC settings are set to enable reading from the Accelerometer as and when needed. The ADC conversions are then started with a call to *HAL_ADC_Start_DMA()*. UART interrupts are then enabled and the UART functionality is configured in the *initUART()* function. The final step that the program takes in the setup process is to calibrate the accelerometer by taking 200 samples and initialing and starting the GPS reading. After valid data is received from the GPS, the terminal screen is cleared and set up for readings, and the program enters into an infinite while loop to detect an accident.

Inside the infinite while loop, while the program waits for an ADC conversion to complete, it constantly updates the GPS information. After the conversion is complete, the program updates both the GPS information and ADC readings to the terminal screen. If the new ADC reading indicates an accident as an acceleration outside the threshold, then crash information is printed, if not then the program returns to the top of the infinite while loop after resetting the ADC conversion flag which is set in the DMA IRQ Handler. If an accident has occurred, the program waits for a USB flash drive to be connected to the microcontroller. When a connection is detected using *USBH_UserProcess* callback function, the program calls *MSC_Application()*. Inside this function, A file is opened/created in the flash drive, inside which the crash information is written. After writing to the USB, the program breaks out of all the infinite loops and reaches the end of the main function and code. This code is explained in the flowchart of Figure A.1 in the Appendix.

To accomplish this task, certain circuitry was needed in order to connect the microcontroller with the accelerometer and the GPS module. Both peripherals were connected to 5V supply and GND in the microcontroller. The Tx pin of the GPS module was connected to D0 pin of the microcontroller. The accelerometer's X axis readings were set to A0 pin, Y axis readings to A1 pin and Z axis readings to A2 pin. The circuit schematic described above can be seen in Figure A.2 in the Appendix.

Low Level Description

Configurations

At the top of the program all the necessary files are included and then global variables are initialized and constants are defined as seen in Table 2.1. The threshold for accident was set to 500 and -500 to accomodate for the sensitivity of our accelerometer in this type of setup and to account for the slower speed in demonstration. Additionally, an enum class was also defined as *MSC_ApplicationTypeDef* with possible values of *APPLICATION_IDLE*, *APPLICATION_START* and *APPLICATION_RUNNING*^[1]. Private function prototypes are also declared before the program enters the main function.

Table 2.1 Global variables and constants

Code	Purpose
#define samples 200	ADC samples taken to calibrate
#define minVal -500	Minimum Threshold for Accident
#define MaxVal 500	Maximum Threshold for Accident
ADC_HandleTypeDef hadc1	ADC Handle
DMA_HandleTypeDef hdma1	DMA Handle
int32_t rawValue[3]	Array to store ADC channel measurements
int conv_complete = 0	Flag to indicate if ADC conversion is complete
int32_t xsample=0	Calibrated x value
int32_t ysample=0	Calibrated y value
int32_t zsample=0	Calibrated z value
int32_t x = 0	Measure x direction acceleration
int32_t y = 0	Measure y direction acceleration
int32_t z = 0	Measure z direction acceleration
char crash_output[100]	File write buffer
FATFS USBDISKFatFs	File system object for USB disk logical drive
FIL MyFile	File object
char USBDISKPath[4]	USB Host logical drive path
USBH_HandleTypeDef husbh	USB Host handle

MSC_ApplicationTypeDef AppliState	Flag to Indicate if USB is connected
GPS_t GPS	Struct from GPS.h that handles GPS information

Inside the main function, the system is booted and initialized with a call to *Sys_Init()*. *\033[2J\033[H* is used to clear the terminal screen and bring the cursor position to home position. *FATFS_LinkDriver(&USBH_Driver, USBDISKPath)* is used to link the USB Host disk I/O driver and the program verifies if it was linked successfully. After which the Host library is initialized with *USBH_Init(&husbh, USBH_UserProcess, 0)*. This function initializes the host stack and the low level. Supported classes are added to *husbh* handle by registering the classes. *USBH_RegisterClass* registers a supported USB Class handler. *USBH_RegisterClass(&husbh, USBH_MSC_CLASS)* connects the MSC class. The program then calls *USBH_Start(&husbh)* to start the host process by enabling the host port VBUS power and starting the low-level operation^[2].

Next, the ADC is configured by calling the function *configureADC()*. Inside *configureADC()*, firstly the ADC1 clock is enabled by calling *__HAL_RCC_ADC1_CLK_ENABLE()*. ADC1 is used in this setup and is configured to correspond to the variable *hadc1* by calling *hadc1.Instance = ADC1* in the program. The variable *hadc1* is then used to initialize the settings required by the task for the ADC1 channel as shown in Table 2.1.

Table 2.2 ADC1 configuration^[3]

hadc1.Init.{}	Value
ClockPrescaler	ADC_CLOCK_SYNC_PCLK_DIV8
Resolution	ADC_RESOLUTION_12B
ScanConvMode	ADC_SCAN_ENABLE
ContinuousConvMode	ENABLE
DataAlign	ADC_DATAALIGN_RIGHT
NbrOfConversion	3
DMAContinuousRequests	ENABLE
EOCSelection	ADC_EOC_SEQ_CONV

The *ClockPrescaler* variable defines the speed of the clock(ADCLK) for the analog circuitry part of ADC. This impacts the number of samples per seconds as it defines the amount of time used by each conversion cycle. The clock is generated from the peripheral clock divided by a programmable prescaler that allows the ADC to work at a certain frequency. *Resolution* sets the number of conversions made in a second where higher the resolution, the lesser the number of conversions. *ScanConvMode* and *ContinuousConvMode* is used to enable the scan

conversion mode and continuous conversion mode to let the DMA configuration work correctly. *NbrOfConversion* specifies the number of the channels of the regular group that will be converted in scan mode, here it was set to 3, since the program sampled 3 different axis of the accelerometer. *DataAlign* specifies the alignment of the converted result data. *EOCSeclection* is used by the ADC to determine when a conversion is complete. ADC1 is then initialized with these configurations through the call to *HAL_ADC_Init(&hadc1)*.

Also inside the *configureADC()* function, the ADC channels 4, 6 and 12 are then configured using the struct *sConfig* initialized as an *ADC_ChannelConfTypeDef* with the specifications as seen in Table 3.B.2, Table 3.B.3 and Table 3.B.4.

Table 2.3 ADC Channel 6 Configuration^[3]

sConfig.{}	Value
Channel	ADC_Channel_6
Rank	1
SamplingTime	ADC_SAMPLETIME_480CYCLES

Table 2.4 ADC Channel 4 Configuration^[3]

sConfig.{}	Value
Channel	ADC_Channel_4
Rank	2
SamplingTime	ADC_SAMPLETIME_480CYCLES

Table 2.5 ADC Channel 12 Configuration^[3]

sConfig.{}	Value
Channel	ADC_Channel_12
Rank	1
SamplingTime	ADC_SAMPLETIME_480CYCLES

The *Channel* represents the Channel ID. *Rank* corresponds to the rank associated with the channel. *SamplingTime* specifies the sampling time value to be set for the channel and it corresponds to the number of ADC cycles. 480 cycles were determined from the fact that ADCCLK in this task runs at 108MHz, performing 108 cycles every 1us. So, 3us derived from Table 72 in the microcontroller datasheet, multiplied by 108 cycles, lets the program choose 480 cycles as the sampling time. ADC channel is then configured with a call to *HAL_ADC_ConfigChannel(&hadc1, &sConfig)*.

The GPIO port pins PA6, PA4 and PC2 are initialized inside the *HAL_ADC_MspInit(...)* function to function as ADC Channel 6, 4 and 12 's input port from the Analog Discovery. These

port pins correspond to the Arduino connectors A0, A1 and A2 on the board respectively, therefore, the A0, A1 and A2 were defined as *GPIO_InitTypeDef* structs to configure the port pins. GPIOC Clock and GPIOA clock were enabled by calling *__HAL_RCC_GPIOC_CLK_ENABLE()* and *__HAL_RCC_GPIOA_CLK_ENABLE()* in the program. PA6 is set up as seen in Table 2.5, PA4 in Table 2.6 and PC2 in Table 2.7.

Table 2.6 ADC GPIO Pin Configuration as PC2^[4]

A0.{} 	Value
Pin	GPIO_PIN_6
Mode	GPIO_MODE_ANALOG
Pull	GPIO_NOPULL

Table 2.7 ADC GPIO Pin Configuration as PC2^[4]

A1.{} 	Value
Pin	GPIO_PIN_4
Mode	GPIO_MODE_ANALOG
Pull	GPIO_NOPULL

Table 2.8 ADC GPIO Pin Configuration as PC2^[4]

A2.{} 	Value
Pin	GPIO_PIN_2
Mode	GPIO_MODE_ANALOG
Pull	GPIO_NOPULL

Pin corresponds to the pin the program is configuring. *Mode* of the pin is set to Analog to let the port pin function for ADC and the port pin is also configured for no pull, since a pull isn't needed for this functionality. The GPIO port pins are then initialized using *HAL_GPIO_Init(GPIOA, &A0)*, *HAL_GPIO_Init(GPIOA, &A1)* and *HAL_GPIO_Init(GPIOC, &A2)*^[5].

Also inside *HAL_ADC_MspInit(...)*^[5], the DMA is configured. The peripheral clock is enabled using *__HAL_RCC_DMA2_CLK_ENABLE()*. DMA2 Stream0 is used in this setup and is configured to correspond to the variable *hdma1* by calling *hdma1.Instance = DMA2_Stream0* in the program. The variable *hdma1* is then used to initialize the settings required by the task for the DMA2 Stream0 channel as shown in Table 2.8.

Table 2.9 DMA2 Stream0 channel configuration

hdma1.Init.{} 	Value
Channel	DMA_CHANNEL_0
Direction	DMA_PERIPH_TO_MEMORY
PeriphInc	DMA_PINC_DISABLE
MemInc	DMA_MINC_ENABLE
PeriphDataAlignment	DMA_PDATAALIGN_WORD
MemDataAlignment	DMA_MDATAALIGN_WORD
Mode	DMA_CIRCULAR
Priority	DMA_PRIORITY_LOW

Direction variable defines the DMA transfer direction. *PeriphInc* and *MemInc* is disabled since the memory address of the peripheral register and the destination memory location doesn't have to be initialized. *PeriphDataAlignment* specifies the data size of the peripheral and *MemDataAlignment* specifies the memory transfer data size. *Mode* is specified as circular mode so that at the end of the conversion, the program automatically resets the transfer counter and starts transferring again from the first byte of the source buffer. The *Priority* is set to low, which is the priority of concurrent channel requests. All these DMA configurations are then initialized with a call to `HAL_DMA_Init(&hdma1)`^[5].

With the DMA configurations initialized, it is linked to ADC1 by running `__HAL_LINKDMA(hadc,DMA_Handle,hdma1)`^[5]. The DMA Interrupt functionality is then enabled by running `HAL_NVIC_SetPriority(DMA2_Stream0_IRQn, 0, 0)` and `HAL_NVIC_EnableIRQ(DMA2_Stream0_IRQn)`^[5]. The IRQ Handler of DMA2 Stream0 is the function `DMA2_Stream0_IRQHandler()`. Inside which the variable *conv_complete* is set to 1, to indicate that an ADC conversion is complete.

For the GPS to STM32 communication, the program uses USART6 in interrupt mode. `HAL_NVIC_SetPriority(USART6_IRQn, 0, 0)`, `HAL_NVIC_EnableIRQ(USART6_IRQn)`^[5] are used to enable set the priority of the USART6 interrupt and enable USART6 on the NVIC IRQ vector. Within the GPSConfig.h file there is a `_GPS_USART` variable that is of type `UART_HandleTypeDef` data type which contains the Instance and Init fields that allows us to configure the handle for USART6 communication purposes. Next `initUart(&_GPS_USART, (uint32_t) 9600, USART6)`^[5] is called to initialize the GPIO pins and configure USART6 for proper MCU to GPS communication. Table 2.10 shows the USART6 configuration performed.

Table 2.10 USART6 configuration^[3]

Uhand->{} 	Value
Instance	USART6

Init.BaudRate	9600
Init.WordLength	UART_WORDLENGTH_8B
Init.StopBits	UART_STOPBITS_1
Init.Parity	UART_PARITY_NONE
Init.Mode	UART_MODE_TX_RX
Init.HwFlowCtl	UART_HWCONTROL_NONE

The *Instance* field specifies the USART_TypeDef* to be used, *Init.BaudRate* specifies the baud rate of data transmission from MCU to GPS module (9600), *Init.WordLength* specifies the length of data being sent (8 bit word length), *Init.StopBits* specifies number of stop bits used during communication (1 stop bit), *Init.Parity* specifies level of parity (no parity), *Init.Mode* specifies the mode of operation, and *Init.HwFlowCtl* specifies id hardware flow control is being used. Next `__HAL_UART_ENABLE_IT(Uhand, UART_IT_RXNE)`^[5] is called to enable USART6 receive interrupts for when data is received by the MCU on the RX line from the GPS module.

`HAL_UART_Init(Uhand)` is finally called to initialize the USART6 communication, which then calls `HAL_UART_MspInit(...)` as a callback to set up the associated GPIO pins. PC6 and PC7 for TX6 and RX6 respectively correspond to the Arduino connectors D1 and D0. A single *GPIO_InitTypeDef* struct is used to configure the port pins. GPIOC Clock was enabled by calling `__GPIOC_CLK_ENABLE()`. Table 2.11 and Table 2.12 shows the PC6 and PC7 respectively.

Table 2.11 TX6 GPIO Pin Configuration^[4]

GPIO_InitStruct.{}	Value
Pin	GPIO_PIN_6
Mode	GPIO_MODE_AF_PP
Pull	GPIO_PULLUP
Speed	GPIO_SPEED_HIGH
Alternate	GPIO_AF8_USART6

Table 2.12 RX6 GPIO Pin Configuration^[4]

GPIO_InitStruct.{}	Value
Pin	GPIO_PIN_7
Mode	GPIO_MODE_AF_PP
Pull	GPIO_PULLUP
Speed	GPIO_SPEED_HIGH

Alternate	GPIO_AF8_USART6
-----------	-----------------

If USART6 is the parameter, then PC6 and PC7 will be configured. *Pin* corresponds to the pin the program is configuring. *Mode* of the pin is set to GPIO_MODE_AF_PP for alternate function push and pull mode, *Pull* is set for enabling pullup resistors, *Speed* is set to high speed, and *Alternate* is set for USART6 functioning purposes. The GPIO port pins are then initialized using `HAL_GPIO_Init(GPIOC, &GPIO_InitStruct)[5]` and `__USART6_CLK_ENABLE()` is called to enable the USART6 peripheral clock.

Functionality

The accelerometer is sampled to calibrate the reading operations. For calibration, 200 samples were taken and added to a total. The samples from the x axis read from `rawValue[0]` were added to the variable `xsample`, samples from the x axis read from `rawValue[1]` were added to the variable `ysample` and samples from the x axis read from `rawValue[2]` were added to the variable `zsample`. At the end of sampling, the variables `xsample`, `ysample` and `zsample` were self divided by 200 to get the calibration base value.

Next the program begins the process of locking an initial GPS position before starting the crash alert system. `GPS_Init()` is first called to initialize the `GPS_t` GPS struct^[6] that was globally created in the GPS.h file. The class structure of GPS struct and the GPRMC struct member inside are explained in Tables 2.13 and 2.14 below.

Table 2.13 `GPS_t` GPS Struct Layout^[6]

Struct Field	Purpose
uint8_t rxBuffer[512]	Used for storing NMEA sentences
uint16_t rxIndex	Maintain position in rxBuffer
uint8_t rxTmp	Temporary storage of received GPS byte
uint32_t LastTime	Milliseconds since startup of when recent byte was received
GPRMC_t GPRMC	Struct for storing GPRMC NMEA specific information

Table 2.14 `GPRMC_t` GPRMC Struct Layout

Struct Field	Purpose
char status	Status of incoming data
uint8_t UTC_Hour	UTC hours (24 hour)
uint8_t UTC_Min	UTC Minutes
uint8_t UTC_Sec	UTC Seconds
uint16_t UTC_MicroSec	UTC Microseconds (Not used for this GPS module)
float Latitude	Degrees and minutes of Latitude

double LatitudeDecimal	Degrees format Latitude
char NS_Indicator	North or South
float Longitude	Degrees and minutes of Longitude
double LongitudeDecimal	Degrees format Longitude
char EW_Indicator	East or West
uint8_t UTC_Month	UTC Month
uint8_t UTC_Day	UTC Day
uint8_t UTC_Year	UTC Year

GPS_Init()^[6] sets the *GPRMC.status* to 'V' signifying invalid data, sets *rxIndex* to 0 (beginning of *rxBuffer*), and calls *HAL_UART_Receive_IT(&_GPS_USART,&GPS.rxTmp,1)* to start receiving the first byte from the GPS module and store in *GPS.rxTmp*^[6]. When the byte is received, interrupt is triggered and serviced by *USART6_IRQHandler()* which calls *HAL_UART_IRQHandler(&_GPS_USART)* and the *HAL_UART_RxCpltCallback(...)* which calls *GPS_CallBack()*. *GPS_CallBack()*'s function is to store the "time" when a byte was received in *GPS.LastTime*, place the received byte in *GPS.rxBuffer*^[6] using *GPS.rxIndex*, increment *GPS.rxIndex*, and then call *HAL_UART_Receive_IT(...)* to receive the next byte. The *while (GPS.GPRMC.status == 'V')* is used to ensure that initially the program has recorded a valid location before it officially begins detection. *GPS_Process()*^[6] is the function that is used to extract received data from *GPS.rxBuffer* and store it in *GPS.GPRMC* for later for crash reporting. Within *GPS_Process()*, if the *rxBuffer* has received the available 0183 standard NMEA^[7] GPS data from the GPS module (indicated by 50 milliseconds passing by since the last time that *GPS.LastTime* was updated) and the *rxIndex* is greater than 0, then function moves on to parsing the string. If the "\$GPRMC," substring is present in *GPS.rxBuffer* by the indication that *char *str* does not equal null, then function declares *char* UTC_time, status, Latitude, N_or_S, Longitude, E_or_W, UTC_Date* string variables which will be used to store the substrings that correspond to the different fields of information that will be populated within *GPS.GPRMC*. The *strsep(..., ",")* function is used to split the string on the "," delimiting string of *str* and return a substring for the field of interest. Using this the function is able to query the specific strings of GPRMC NMEA string. If it is the case that *status[0] == 'V'*, then that indicates that the GPS data was invalid. The function prints out ""GPS Standby" to indicate that the GPS is calibrating or invalid data has been received. Notice that the *GPS.GPRMC* fields have not been updated or reset.

If the data was valid, then *GPS.GPRMC* is reset, *GPS.GPRMC.status*, *GPS.GPRMC.NS_Indicator*, and *GPS.GPRMC.EW_Indicator* are set. A *char time[3]* is declared to store bytes of *UTC_time* for the individual sections of the time. First the function extracts the hours string from *UTC_time*, and uses *(uint8_t)atoi(time)* to convert the string to the proper data type for *GPS.GPRMC.UTC_Hour* struct field. This process is repeated for Minutes and Seconds. Identical process is performed for extracting and storing date fields inside of *GPS.GPRMC*, except *char date[3]* is used. Next the latitude and longitude strings are converted

from strings to floats using *strtof(...)*, stored in *GPS.GPRMC.Latitude* and *GPS.GPRMC.Longitude*, and *convertDegMinToDecDeg()*^[6] is used to convert the DDSS.SSSSS formatted latitude and longitude numbers decimal degrees. *convertDegMinToDecDeg()* essentially takes a measurement and direction indication, and returns the result of **DD + SSSSS/60**^[8]. These decimal degrees are then stored in the *Latitude* and *Longitude* fields of *GPS.GPRMC*. *GPS.rxIndex* and the contents of *GPS.rxBuffer* as reset for new valid data arrival.

Finally the function calls *HAL_UART_Receive_IT(...)* to prompt the GPS module to start sending new data and exits to the main routine. Once *GPS.GPRMC.status == 'A'*, the program prints "*GPS Initialized* " and a HAL delay is called for setting ensuring that the DMA and ADC are initialized before the main functionality is started.

Next, the terminal screen is set up using VT100-ANSI escape sequences. The following Table 2.15 summarizes how escape sequences were used to set up the terminal. Once these attributes were initialized, *fflush(stdout)* was used to set the attributes to the terminal by clearing the stdout buffer.

Table 2.15 Escape sequence used to set up terminal screen for the program

Escape Sequence	Purpose
\033[2J\033[H	Erase screen & move cursor to home position
\033[5;0H	To print "GPS" on row 0
\033[11;0H	To print "Accelerometer Readings" on row 11
\033[s	To save the cursor's current location for Accelerometer Reading scrolling
\033[12;24r	To enable scrolling of Accelerometer Readings on lines 12 to 24

The program then enters into an infinite while loop, to detect when an accident happens. The program waits for conversion to complete, by looping through a while loop till the *conv_complete* flag is set, indicating that the ADC conversion is complete. While waiting for an updated ADC conversion, *GPS_Process()* is called to initiate an update of the GPS values in the meanwhile. Once the flag indicates that the conversion is complete, the readings are calculated the following way for each axis:

$$\begin{aligned}
 x &= xsample - rawValue[0] \\
 y &= ysample - rawValue[1] \\
 z &= zsample - rawValue[2]
 \end{aligned}$$

x, *y* and *z* values are then printed to the terminal using the printf statement, ("*\033[uX:%ld Y:%ld Z:%ld\r\n*",*x,y,z*). *\033[u* saves the accelerometer's last position in the scrolling section. *\033[s* is then printed to save the cursor's current location for the next output. The gps information is printed using the string formatting as seen in Table 2.16.

Table 2.16 Print sequence for GPS information display

Escape Sequence	Purpose
\"%d:%d:%d (UTC)\"	GPS.GPRMC.UTC_Hour, GPS.GPRMC.UTC_Min, GPS.GPRMC.UTC_Sec
\"%d-%d-%d (UTC)\"	GPS.GPRMC.UTC_Month, GPS.GPRMC.UTC_Day, GPS.GPRMC.UTC_Year
\"%f %c\"	GPS.GPRMC.LatitudeDecimal, GPS.GPRMC.NS_Indicator
\"%f %c\"	GPS.GPRMC.LongitudeDecimal, GPS.GPRMC.EW_Indicator

With the terminal updated with new values, next the program checks if an accident has occurred by checking if one of *x*, *y* or *z* values are either greater than the *maxVal* threshold or less than the *MinVal* threshold. If they aren't, *conv_complete* is reset to start a new conversion. If they are, it indicates that an accident has occurred because of the huge acceleration read by the accelerometer. The terminal is cleared and the cursor is returned to home position with printing \033[2J\033[H. The user is then indicated on the terminal that an accident has occurred. The GPS coordinates where the crash happened is printed in the same manner as seen in Table 2.10 and *sprintf* stores the string in the same format as Table 2.10, in a variable *crash_output*.

The program then enters into an infinite while loop to run on whatever device is connected to the microcontroller. *USBH_Process(&usbh)* is called a host process function that implements the core state machine in standalone mode operation. Its callback is defined as *USBH_UserProcess* when the USBH driver was initialized earlier in the main function. The *USBH_UserProcess* callback^[2] handles the USB host core events and helps determine when the usb device is connected or disconnected. Inside the callback function, another switch case loop is implemented, where *id* is used to determine which part of the function is implemented. If the *id* is set to *HOST_USER_CLASS_ACTIVE*, *AppliState* is set to *APPLICATION_START*, indicating the main function that a flash drive has been connected to the microcontroller. If the *id* is set to *HOST_USER_DISCONNECTION*, it means that no usb device has been connected to the microcontroller. In this case, *AppliState* is set to *APPLICATION_IDLE*, *f_mount()* is used to unregister the work area of the volume. Both variables, *mouse_connection* and *msc_connection* are set to 0, to indicate the main function that a flash drive is connected to the microcontroller.^[1]

After the *UserProcess*, next a switch case loop determines its condition using the variable *AppliState*. The case takes action only when *AppliState* has a value of *APPLICATION_START*. Inside *APPLICATION_START*, it means that the flash drive is connected, therefore, the function *MSC_Application(crash_output)* is called. Inside *MSC_Application(crash_output)*^[1], a *FRESULT* type variable called *res* is initialized where *FRESULT* type variables are FatFs function common result code. A *uint32_t* type variable called *byteswritten* is initialized to keep count of bytes written. Additionally a *uint8_t* array called *wtext* is initialized with the string "ACCIDENT DETECTED \r\n". Another *uint8_t* array called *output*

that takes the value of *crash_output* casted into a *uint8_t* array type. The program then calls *f_mount(&USBDISKFatFs, (TCHAR const*)USBDISKPath, 0)*^[9] and waits till the function's return value is *FR_OK*. In the *f_mount()* function, *USBDISKFatFs* represents the Pointer to the file system object to be registered and cleared, *USBDiskPath* represents the pointer to the null-terminated string that specifies the logical drive and *0* represents the mounting option where the system has been asked to not mount now but to be mounted on the first access to the volume. If the return value is *FR_OK*, it means that the mounting was successful.

The program then indicates the user on the terminal that it is starting to create a report file. The code calls *f_open(&MyFile, "STM32.TXT", FA_CREATE_ALWAYS | FA_WRITE)*^[9] till the function's return value is *FR_OK* to ensure the operation was successful. In *f_open()* function, *&MyFile* represents the pointer to the blank file structure, "*STM32.TXT*" represents the file name and *FA_CREATE_ALWAYS | FA_WRITE* represents the mode flags. Once the file is opened successfully, the program indicates to the user through the terminal that it would now start writing the report to the file. The following two *f_write* operations are called to print the text in *w_text* and *output* array:

```
f_write(&MyFile, wtext, sizeof(wtext), (void *)&byteswritten) [9]
f_write(&MyFile, output, strlen(output) * sizeof(uint8_t), (void *)&byteswritten) [9]
```

In this function call, the first input variable represents the pointer to the file object structure, the second variable is the pointer to the data to be written, the third specifies the number of bytes to write and the fourth variable is a pointer to the variable to return the number of bytes written. The return values of both functions are checked if it was *FR_OK* and a non zero number of bytes were written. The program then indicates to the user that the report has been written and with a call to *f_close(&MyFile)*, the program closes the open text file. Additionally, the program unlinks the USB disk I/O driver by calling *FATFS_UnLinkDriver(USBDISKPath)*. Back in the main function, the program breaks out of the infinite loops to reach the end of the main function.

Results and Analysis

```
GPS Standby
$GPRMC,021707.49,V,,,,,121220,,,N*71
$GPVTG,,,,,,N*30
$GPGGA,021707.49,,,,,0,03,41.83,,,,,*65
$GPGSA,A,1,30,07,14,,,,,41.84,41.83,1.00*07
$GPGSV,4,1,14,01,49,146,32,07,62,197,35,08,46,050,,10,01,039,31*7A
$GPGSV,4,2,14,11,70,088,38,13,14,310,24,14,42,302,33,17,14,240,*72
$GPGSV,4,3,14,21,56,097,,23,,,32,27,10,056,,28,34,298,*4C
$GPGSV,4,4,14,30,63,274,38,51,31,226,29*7B
$GPGLL,,,,,021707.49,V,N*44
$GPRMC,021708.00,V,,,,,121220,,,N*73
$GPVTG,,,,,,N*30
$GPGGA,021708.00,,,,,0,04,41
GPS Standby
$GPRMC,021709.00,A,4104.70868,N,07332.05573,W,0.051,,121220,,A*68
$GPVTG,,T,,M,0.051,N,0.094,K,A*2A
$GPGGA,021709.00,4104.70868,N,07332.05573,W,1,06,2.36,5.6,M,-34.2,M,,*68
$GPGSA,A,3,30,07,21,14,28,08,,,,,6.72,2.36,6.29*0B
$GPGSV,4,1,16,01,49,146,,07,62,197,35,08,45,050,31,10,01,039,19*72
$GPGSV,4,2,16,11,70,088,38,13,14,310,20,14,42,302,35,17,14,240,*72
$GPGSV,4,3,16,20,,,25,21,56,097,32,22,,,22,23,,,22*42
$GPGSV,4,4,16,27,10,056,26,28,34,298,31,30,63,274,38,51,31,226,27*78
$GPGLL,4104.70868,
GPS Initialized
```

Figure 3.1 GPS Initialization

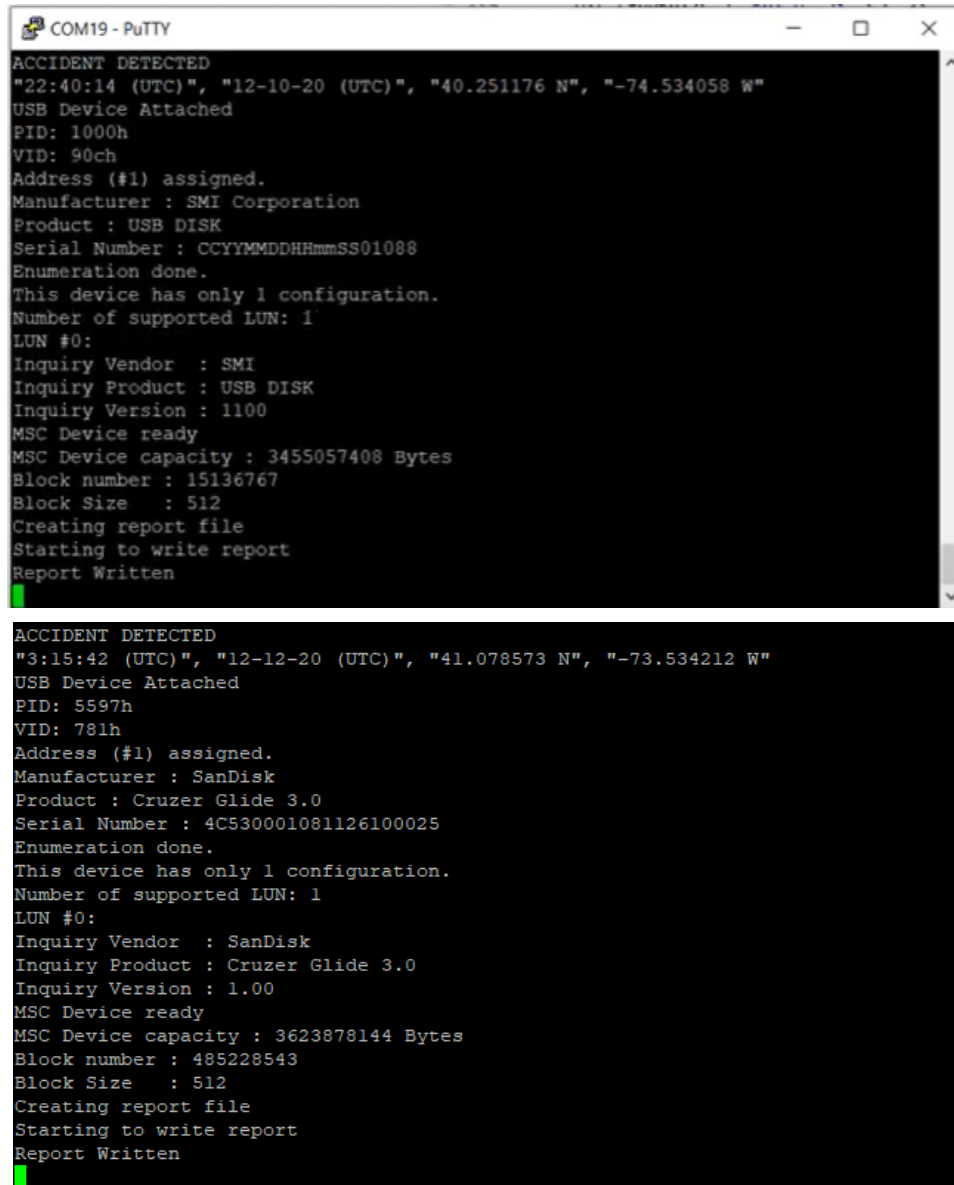
```
COM19 - PuTTY
"22:40:14 (UTC)", "12-10-20 (UTC)", "40.251176 N", "-74.534058 W"
$GPRMC,224014.00,A,4015.07062,N,07432.04359,W,0.106,,101220,,,A*6E
$GPVTG,,T,,M,0.106,N,0.197,K,A*2B
$GPGGA,224014.00,4015.07062,N,07432.04359,W,1,06,1.50,51.3,M,-34.5,M,,*5F
$GPGSA,A,3,16,04,27,09,26,31,,,,,2.67,1.50,2.21*0D
$GPGSV,3,1,12,03,06,223,,04,67,243,16,07,12,296,16,08,25,186,15*73
$GPGSV,3,2,12,09,41,303,36,16,71,028,11,18,05,064,16,26,39,054,30*7C
$GPGSV,3,3,12,27,52,152,28,31,15,104,11,46,18,245,,51,33,225,*7E
$GPGSV,3,4,12,27,52,152,28,31,15,104,11,46,18,245,,51,33,225,*7E
$GPGLL,4015.07062,N,07432.04359,W,224014.00,A,A*70

Accelerometer Readings
X:88 Y:54 Z:-50
X:91 Y:44 Z:-54
X:103 Y:38 Z:-15
X:147 Y:42 Z:-35
X:101 Y:33 Z:-20
X:85 Y:46 Z:-28
X:105 Y:40 Z:-38
X:93 Y:47 Z:-53
X:97 Y:54 Z:-11
X:94 Y:61 Z:-11
X:104 Y:39 Z:-30
X:123 Y:35 Z:-36

GPS Initialized
"3:15:42 (UTC)", "12-12-20 (UTC)", "41.078573 N", "-73.534212 W"
$GPRMC,031542.00,A,4104.71451,N,07332.05286,W,0.136,,121220,,,A*6E
$GPVTG,,T,,M,0.136,N,0.252,K,A*22
$GPGGA,031542.00,4104.71451,N,07332.05286,W,1,08,1.68,55.0,M,-34.2,M,,*5B
$GPGSA,A,3,01,14,28,30,21,07,17,19,,,,,2.79,1.68,2.23*05
$GPGSV,3,1,12,01,65,095,35,03,08,136,17,07,33,185,17,08,23,063,*78
$GPGSV,3,2,12,11,48,055,30,13,11,286,22,14,65,321,31,17,34,260,30*78
$GPGSV,3,3,12,19,13,251,29,21,47,057,36,28,56,315,34,30,53,220,38*7D
$GPGSV,3,4,12,19,13,251,29,21,47,057,36,28,56,315,34,30,53,220,38*7D
$GPGLL,4104.71451,N,07332.05286,W,031542.00,A,A*71

Accelerometer Readings
X:-80 Y:-74 Z:-83
X:231 Y:172 Z:194
X:-76 Y:-92 Z:-71
X:172 Y:187 Z:148
X:-66 Y:-115 Z:-93
X:180 Y:190 Z:143
X:-102 Y:-89 Z:-61
X:212 Y:157 Z:194
X:-87 Y:-86 Z:-56
X:313 Y:337 Z:218
X:-124 Y:-189 Z:-90
X:512 Y:467 Z:430
```

Figure 3.2 Accelerometer and GPS information after program is initialized. (Top = Partner 1, Bottom = Partner 2)



The image displays two screenshots of a PuTTY terminal window titled 'COM19 - PuTTY'. The top screenshot shows the output for 'Partner 1', and the bottom screenshot shows the output for 'Partner 2'. Both logs start with 'ACCIDENT DETECTED' followed by a timestamp and coordinates. They then report a USB device attachment with details like PID, VID, and manufacturer. The logs also show the device's configuration, LUN information, and MSC device details such as capacity and block size. Finally, they report the creation and writing of a report file.

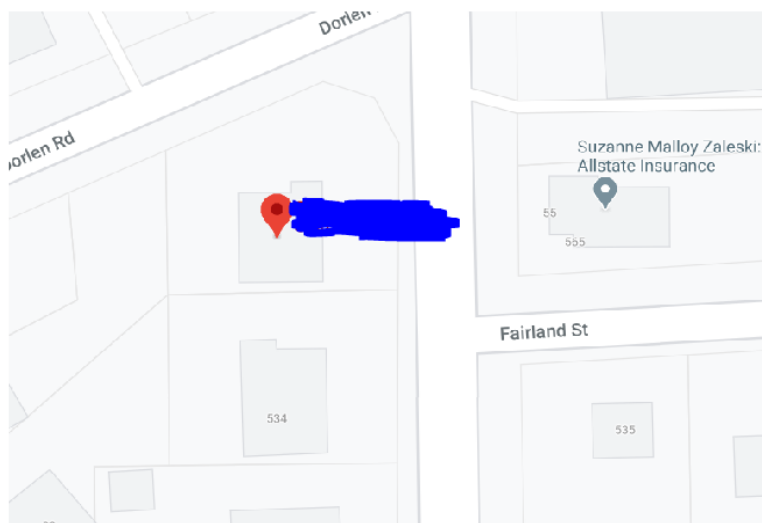
```
COM19 - PuTTY
ACCIDENT DETECTED
"22:40:14 (UTC)", "12-10-20 (UTC)", "40.251176 N", "-74.534058 W"
USB Device Attached
PID: 1000h
VID: 90ch
Address (#1) assigned.
Manufacturer : SMI Corporation
Product : USB DISK
Serial Number : CCYYMMDDHHmmSS01088
Enumeration done.
This device has only 1 configuration.
Number of supported LUN: 1
LUN #0:
Inquiry Vendor : SMI
Inquiry Product : USB DISK
Inquiry Version : 1100
MSC Device ready
MSC Device capacity : 3455057408 Bytes
Block number : 15136767
Block Size : 512
Creating report file
Starting to write report
Report Written

ACCIDENT DETECTED
"3:15:42 (UTC)", "12-12-20 (UTC)", "41.078573 N", "-73.534212 W"
USB Device Attached
PID: 5597h
VID: 781h
Address (#1) assigned.
Manufacturer : SanDisk
Product : Cruzer Glide 3.0
Serial Number : 4C530001081126100025
Enumeration done.
This device has only 1 configuration.
Number of supported LUN: 1
LUN #0:
Inquiry Vendor : SanDisk
Inquiry Product : Cruzer Glide 3.0
Inquiry Version : 1.00
MSC Device ready
MSC Device capacity : 3623878144 Bytes
Block number : 485228543
Block Size : 512
Creating report file
Starting to write report
Report Written
```

Figure 3.3 Output after an Accident has occurred (Top = Partner 1, Bottom = Partner 2)



Map for 40.255176,-74.534058



Map for 41.078573, -73.53421

Figure 3.4 GPS location of where the accident occurred (Top = Partner 1, Bottom = Partner 2)

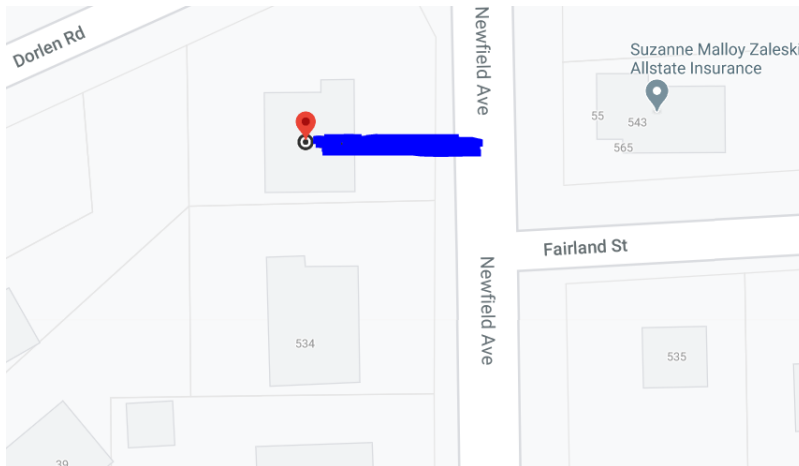
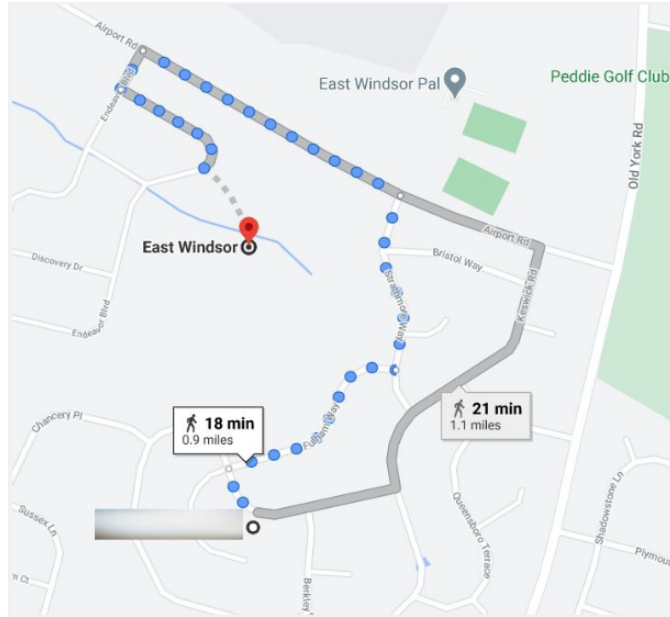
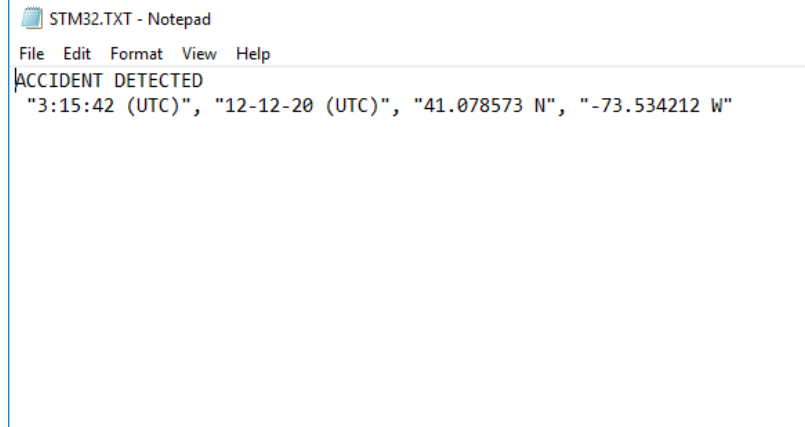


Figure 3.5 Difference between Expected location and Measured Location (Top = Partner 1, Bottom = Partner 2)





```
STM32.TXT - Notepad
File Edit Format View Help
ACCIDENT DETECTED
"3:15:42 (UTC)", "12-12-20 (UTC)", "41.078573 N", "-73.534212 W"
```

Figure 3.6 Crash output written in the USB flash drive (Top = Partner 1, Bottom = Partner 2)

The program has been implemented such that we are able to display accelerometer and GPS locational data in scrolling form on the Putty terminal to see the changing data from the peripherals. Figure 3.1 is meant to show output to Putty terminal when the MCU is first booted up and the GPS and accelerometer are being set up. Since the `_GPS_DEBUG` flag was enabled in the `GPSCfg.h` file, other GPS related 0183 standard NMEA strings besides GPRMC have been outputted to the terminal. As mentioned in the Low Level description, when the GPS is receiving invalid data from the GPS due to location changes or initial boot up, the GPS will send no usable data. Figure 3.1 shows the “GPS Standby” string and empty NMEA strings with ‘V’ character within the GPRMC string to represent the receipt of invalid data. At the bottom we can see that a GPS lock has been established with the indicated string: “GPS Initialized”. Since this string only shows up after the accelerometer has been calibrated with the 200 samples, we know that the accelerometer and GPS have now finished calibration and monitoring can start. In Figure 3.2 we see the screen is split into 2 with GPS updates on the top half and accelerometer readings on the bottom half. GPS date and time of capture, along with acquired latitude and longitude in degrees have been displayed correctly on the top half along with the NMEA strings from which this data was extracted (`_GPS_DEBUG = 1`). Figure 3.3 shows Putty terminal when an accident has occurred. We can see that UTC time, date, and latitude and longitude coordinates are displayed at the top and that the USB disk drive has been connected properly as seen by the USB details on the terminal. Figure 3.6 shows that the locational data of the accident was properly written to the USB as seen by the `STM32.txt` file’s contents.

Overall the results show that the project was able to capture the occurrence of accidents by monitoring accelerometer readings and GPS data to output crash occurrence location to a USB storage device. One of the difficulties the team faced was that the accelerometer had a tendency to give non-smooth values even when the MCU and peripherals were set on a flat surface. There’s the possibility that because of accelerometer sensitivity and other signals, the data was getting affected by noise especially since it was resting on a breadboard with an antenna device. In terms of the GPS unit, there were instances where the GPS took long periods of time to boot up and start sending valid data from a long power off. We believe that this was caused by the fact that testing was performed indoors where signal disruption is more

likely to occur. Figure 3.5 shows an example of this where the GPS captured a location that was 0.9 miles from the stationary accident testing sight for partner 1 and spot on for partner 2. The GPS and accelerometer were implemented to operate asynchronously since the systems don't rely on one another. The accelerometer requires immediate readings/updates using DMA, while the GPS just needs to track the most recent location.

Conclusion

Through this project, we learnt how to implement an accelerometer and a GPS module using knowledge about ADC, DMA, and UART communication that was learnt in the past. The project demonstrates successful ADC conversions and accurate GPS location to help serve the main function of its task. This project has many real life applications. While currently the crash output is only being communicated to a USB, an extension for this project can be made using a GSM module where this information along with GPS location can be immediately sent to emergency response personnel to help save lives. Another variation of the same technology can be used to help old people, where their close ones can be notified if they become immobile for any reason. With a noble purpose, this project proves to be a successful step towards something that can be improved in the future for better and more versatile functionality.

References

- [1] Vpecanins, "vpecanins/stm32-usb-fatfs," *GitHub*. [Online]. Available: https://github.com/vpecanins/stm32-usb-fatfs/blob/master/Projects/STM32F401-Discovery/Applications/FatFs/FatFs_USBDisk/Src/main.c. [Accessed: 12-Dec-2020].
- [2] ST, "STM32Cube USB host library," 2015. [Online]. Available: https://www.ecse.rpi.edu/courses/F19/ECSE-4790/Documents/UM1720-stm32_USB_Host_Library.pdf. [Accessed: 04-Dec-2020].
- [3] Noviello, C., n.d. *Mastering STM32*.
- [4] "RM0410 Reference manual," *Microprocessor Systems - Fall 2020*. [Online]. Available: https://www.ecse.rpi.edu/courses/F20/ECSE-4790/Documents/RM0410-stm32f7_Reference_Manual.pdf.
- [5] "UM1905 User Manual: Description of STM32F7 HAL and Low-layer drivers," *Microprocessor Systems - Fall 2020*. [Online]. Available: https://www.ecse.rpi.edu/courses/F20/ECSE-4790/Documents/UM1905-stm32f7_HAL_and_LL_Drivers.pdf.
- [6] nimaltd, "nimaltd/GPS," *GitHub*, 19-Mar-2019. [Online]. Available: <https://github.com/nimaltd/GPS>. [Accessed: 08-Dec-2020].
- [7] "NMEA Format_v0.1." [Online]. Available: http://navspark.mybigcommerce.com/content/NMEA_Format_v0.1.pdf. [Accessed: 10-Dec-2020].
- [8] "Convert NMEA sentence Lat and Lon to Decimal Degrees," *Convert NMEA sentence Lat and Lon to Decimal Degrees - Raspberry Pi Forums*, 22-Feb-2017. [Online]. Available: <https://www.raspberrypi.org/forums/viewtopic.php?t=175163>. [Accessed: 10-Dec-2020].
- [9] "f_readdir," Elm Chan. [Online]. Available: http://www.elm-chan.org/docs/fat_e.html

Appendix

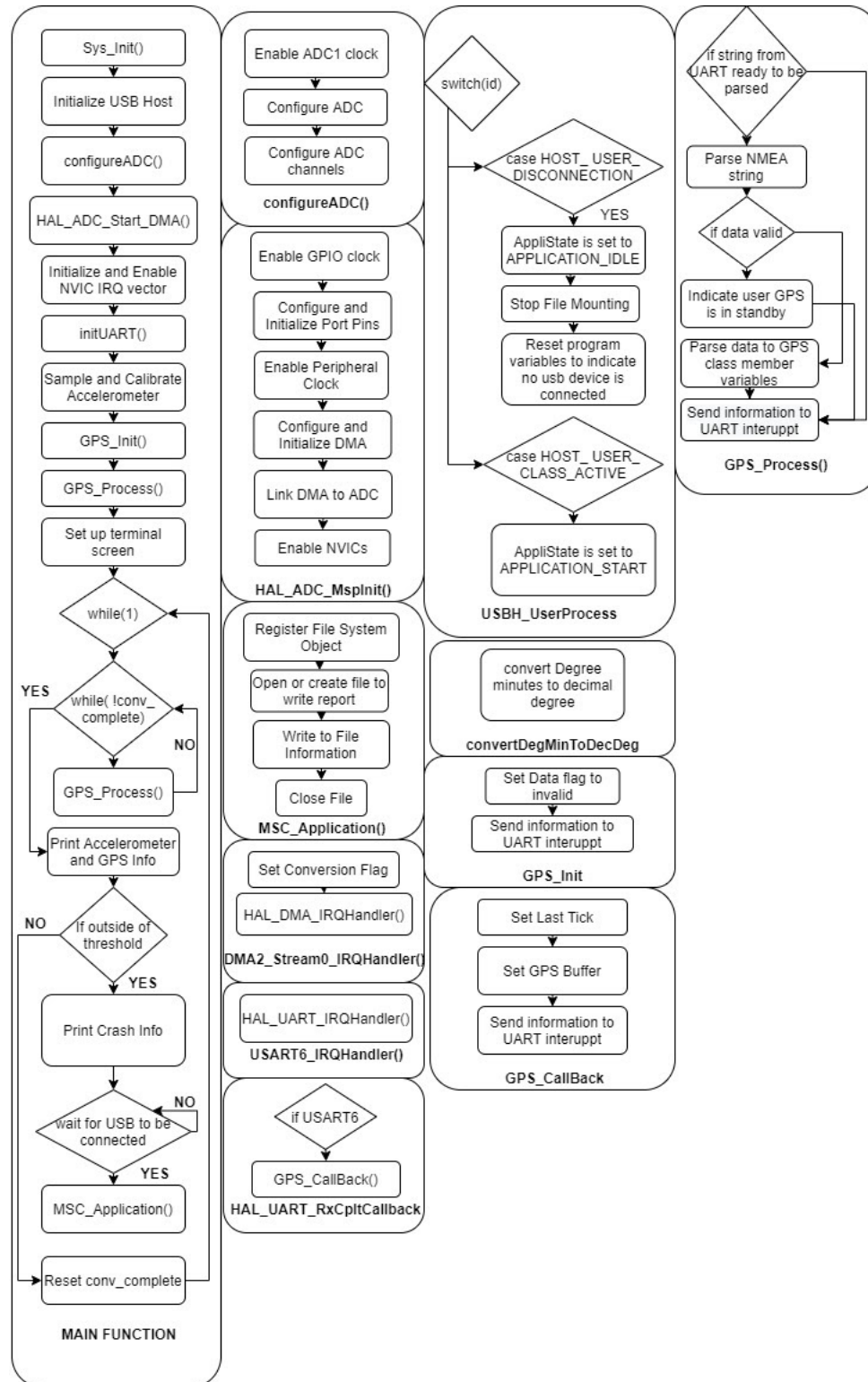


Figure A.1 Code Flowchart

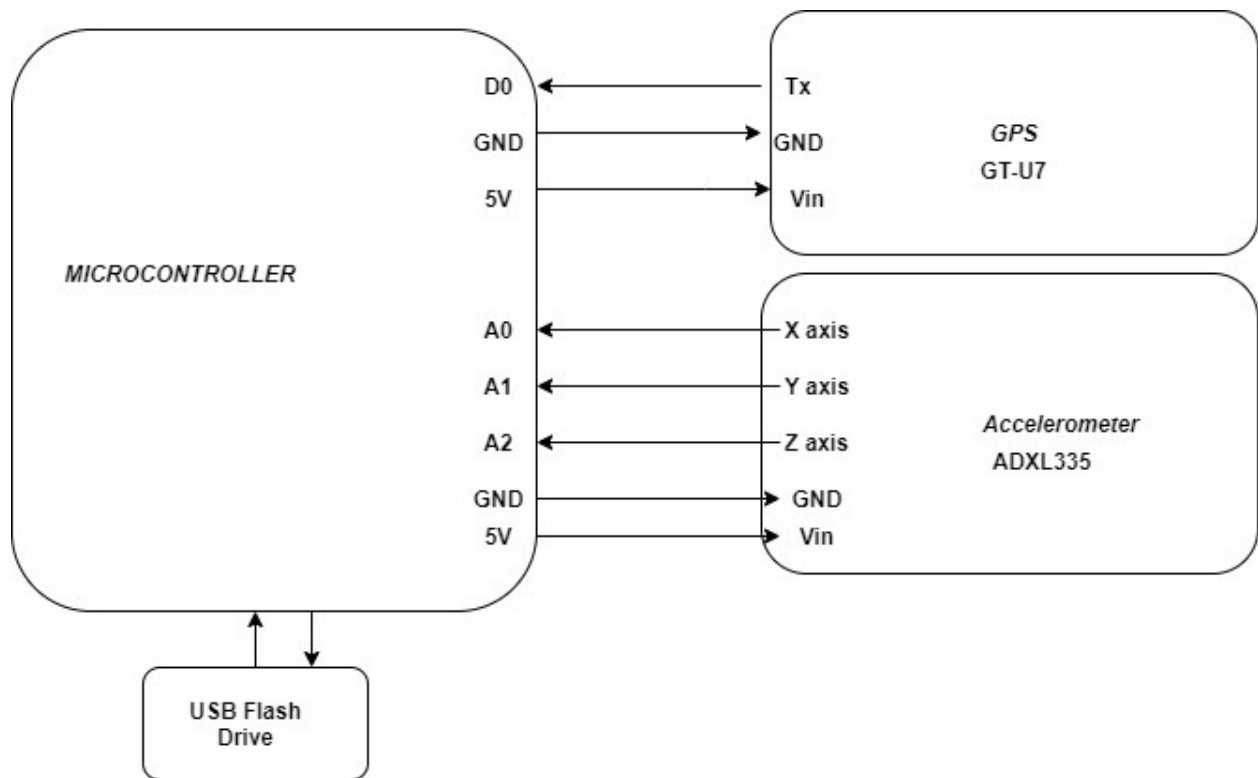


Figure A.2 Circuit Schematic