Crash Buddy:
Crash Detection For Adventure Athletes

Developed by:
Joshua An
Bradley Bares
Matthew Chan
Stanley Charles
Dominik Ritzenhoff
Brett Schneider

Advisor:
Charles DiMarzio

TABLE OF CONTENTS

INTRODUCTION

Workers, adventurers, thrill-seekers, and athletes alike use helmets to protect their heads from impact during their respective activities; however, helmets do not protect one against a loss of consciousness or brain damage. These two conditions can also lead to short and long-term complications without proper attention. When a loss of consciousness occurs, individuals may not be able to treat other injuries in a timely manner. Additionally, individuals may become disoriented or lost and may require assistance to return from their activity (e.g. skiers, bikers, etc.). In cases of brain injury, the most common condition is mild traumatic brain injury (MTBI) – also known as a concussion [1]. Concussions themselves are complicated to diagnose, and they can lead to a variety of long-term complications such as changes in personality, memory problems, and sensitivity to light or noise [2]. There is a clear need for a device that can alert a helmet user's emergency contacts of a loss of consciousness which also provides medical personnel with collision data. This report will review the project's problem formulation, solution design, solution implementation, and future work.

PROBLEM FORMULATION

The goal of this project was to create a system capable of detecting and recording an impact and notifying an emergency contact if the user is unresponsive. The system is composed of both a physical device and a mobile application.

The hardware peripheral requirements included needing an accelerometer that could measure acceleration values up to 200 Gs with a low power draw so that a wider range of impacts could be detected. A development board with Bluetooth Low Energy (BLE) connectivity was also required so that we could securely send data between the hardware and the mobile application. A portable battery is also needed so that the entire system could be powered while also being able to move around with the user.

On the software side, we had to figure out the most efficient method of receiving data from the hardware device. While we knew we wanted to use BLE as the form of communicating data between software and hardware, we had to determine if we wanted the hardware peripheral to ping the mobile application whenever data was to be sent, or vice versa. We also had to decide what operating system and framework would best suit our needs for this project, as each framework has its own advantages and disadvantages.

In conjunction with the hardware, the mobile application should only seek data from the hardware peripheral when the user indicates to do so. Another approach would be for the hardware peripheral to start tracking data whenever the mobile application pings the peripheral to do so.

Because we want to send a text message to the user's emergency contact if the user becomes unresponsive, we also needed to solve the issue of being able to automate and send a text message via the mobile device whenever required to do so.

The user interface and user experience of the mobile application are also important factors, as it needs to have a clear workflow so that the user is not confused and overwhelmed when using the application. Since the use case of our product applied to many different fields, we wanted to give the user the option to set their own acceleration threshold, based on what activity they were performing. For example, an individual skiing may want to set a higher acceleration threshold to detect more severe impacts as opposed to an individual going for a bike ride. Users should be able to choose their activity and assign a specific acceleration threshold value to that activity.

We also want to allow users to view their history. Users would be able to see their previous activity runs and get a detailed view of the acceleration data among other statistics from that particular activity run. With this, we needed to preserve the data and state of the mobile application every time a user opened or closed the application.

The overarching goal of the mobile application was to give the user full control of what activity they are embarking on while selecting a certain acceleration threshold associated with that particular activity run, while also communicating with the hardware peripheral whenever an impact crossing the acceleration threshold is detected. The mobile application should also send out a message if a high enough impact is detected and the user does not respond to the alert in a timely manner.

CONCEPTUALIZATION

HARDWARE

The Crash Buddy will consist of a helmet with accelerometers embedded within it. These accelerometers will all be connected to a breakout board containing an ESP32 microcontroller, and data can be stored on SPI flash memory. Power for the system will come from a battery bank that will connect to the breakout board.

In order to collect appropriate data to determine concussions, the accelerometers need to detect motion with at least 100gs of acceleration in three dimensions. The accelerometers the Crash Buddy will use can detect acceleration at ±100g, ±200g, and ±400g, which will be adequate for the helmet's purpose. The acceleration measurement range can be changed by changing the bits. The accelerometers collect data by using a low-noise capacitive amplifier that converts the capacitive unbalancing MEMS sensor into an analog voltage that can be accessed by the user through an analog-to-digital converter. The accelerometers transmit data by using an SPI serial interface.

The breakout board containing the ESP32 microcontroller includes an integrated 802.11b/g/n HT40 wi-fi transceiver and integrated BLE support. It also has SD-card interface support and a LiPo charger. These features are necessary for the project and having them contained on one breakout board simplifies the electronic system that needs to be built. The ESP32 itself is a dual-core ultra-low power MCU. It has an SPI serial interface, so it can communicate with the accelerometers.

The IC Flash memory is another component in the electronic system that uses the SPI serial interface. Data can be transferred serially into and out of the device. The memory has

built-in memory protection that begins when the device is powered off. A powered-off device will not react to external signals.

The last part of the hardware is the PCB, which will be custom designed so that it will take as little space as possible and also safely contain components. A PCB minimizes environmental effects on components and ensures that reliable electric connections are made between parts.

FIRMWARE

Initialization

When the device is turned on by enabling the power bank, the device will begin to initialize. It will initialize and format the flash memory, initialize the accelerometer in the proper range and transmission mode, and initialize the Bluetooth module in Bluetooth Low Energy (BLE) mode. If everything is initialized successfully, the device begin a BLE server and will await a client connection.

Pairing

Following initialization, the device will begin to advertise as a BLE Generic Attribute Profile Server (GATTS), signaling that the device is available. The device's BLE module, a technology used to establish wireless connections with reduced power consumption than traditional Bluetooth, will wait until a connection with an iPhone running the Crash Buddy mobile application is established. Once a connection is established, the device will begin monitoring for impacts.

Data Storage

Once a successful connection is established, the device will begin saving acceleration data in pairs containing the sampled acceleration value and the time of sampling. The data will

be stored in a cyclical ring buffer, which will overwrite old data in favor of new data so that the necessary memory can be allocated statically. This method assures that the device will efficiently use a set memory size that can be expanded or contracted based on the needs of the device.

Data Transmission

Once the application has paired to the device and a crash has occurred, crash data will be accessible over BLE via a GATTS running on the device. A GATTS is a common technique used by BLE devices to communicate data by placing it in the GATTS database, from which a GATTS client can read data via BLE. The iPhone application will be acting as the GATTS client in this implementation. When the device detects a collision has occurred, it will move all the relevant data into the GATTS database and signal to the iPhone application that new collision data is available.

Data Analysis

Finding collisions within the data involves monitoring if an acceleration threshold is exceeded. When a threshold is exceeded, the device will continue to record some additional post-crash data before making all data relevant to the crash available to the Crash Buddy app.

SOFTWARE

Mobile Application Usage

Alongside our physical device, we are incorporating a mobile application that will serve as the main product interface for the user.

At a high level, a user will be able to start tracking within the app. The collision detection threshold will be communicated to the peripheral and if a collision occurs a 4-5 second window of acceleration data will be shared from the peripheral to the phone. The data will be logged to the phone and relevant statistics will be generated. The app will then send a ping to the user

explaining that their emergency contacts will be notified if the user does not interact with the app within an allotted amount of time after the collision has occurred. At their convenience, the user will be able to end tracking, stopping the data collection. The application will additionally allow users to adjust settings such as emergency contact information and crash threshold information. All information will be stored locally on the phone and will be persistent between app closures. A high level flowchart can be found in Appendix B.

Mobile Application Framework

We decided between three potential mobile application platforms - Android, React Native, and iOS. When making a decision on which platform to choose, we focused on a set of specific criteria - community support for each platform, Bluetooth library usability, and existing knowledge within our team.

Choosing a platform with large community support was important because there would undoubtedly be instances where we run into bugs with our implementation and will need to search for information on the internet. Having a framework that has a lot of support makes the process of getting the solution easier since it is likely that the problems we encounter have already been dealt with.

One of the more foreseeable difficulties that we believed we would face was utilizing the Bluetooth library within the framework we decided upon. The Bluetooth library would be the trickiest to implement as it would interface with the firmware of our physical device, and given that our team did not have a lot of experience with Bluetooth network implementation, we aimed to choose a framework with an easier-to-handle Bluetooth interface.

When discussing Android development, most of our team members were not familiar with the framework. Even though it is a widely supported framework, our existing knowledge was not extensive enough to justify utilizing the Android framework.

React Native was another potential framework we discussed since it utilizes languages such as Java that our team members are familiar with from coursework. React Native is also a heavily used framework, and there is a substantial amount of support from the community for the framework. However, when we were looking into the Bluetooth libraries that are supported in React Native, we found that there were multiple libraries to choose from. Given our time frame, we would prefer to stick with just one clear-cut Bluetooth library, and because there are multiple options to choose from within the React Native environment, we felt as if there was uncertainty when it comes to Bluetooth packages in the React Native framework. If we were to choose React Native as our framework and come to realize that the Bluetooth package we are using is not working as expected, we may fall behind on our goals if we spend too much time debugging the issue/switch and try a different Bluetooth library.

With regard to iOS development, our team members were more familiar with the iOS framework compared to Android and React Native frameworks. We found that there is a singular Bluetooth library that is used within the iOS framework. Having one library meant that if we ran into issues, the problems would likely pertain to the Bluetooth library we were using instead of multiple different libraries found in the React Native framework. Because iOS development is popular and mainstream, there is a lot of community support and feedback on the internet that we can refer to if we run into technical issues.

To sum up, we agreed upon using the iOS framework because of the large community support, clear-cut Bluetooth library, and general existing knowledge our team already has regarding iOS development.

Mobile Application Frontend Design

The frontend is split into three pages: homepage, settings, and logs. The navigation between each page will be built into the homepage and the top navigation bar. Mock-ups can be seen in Figure 1.

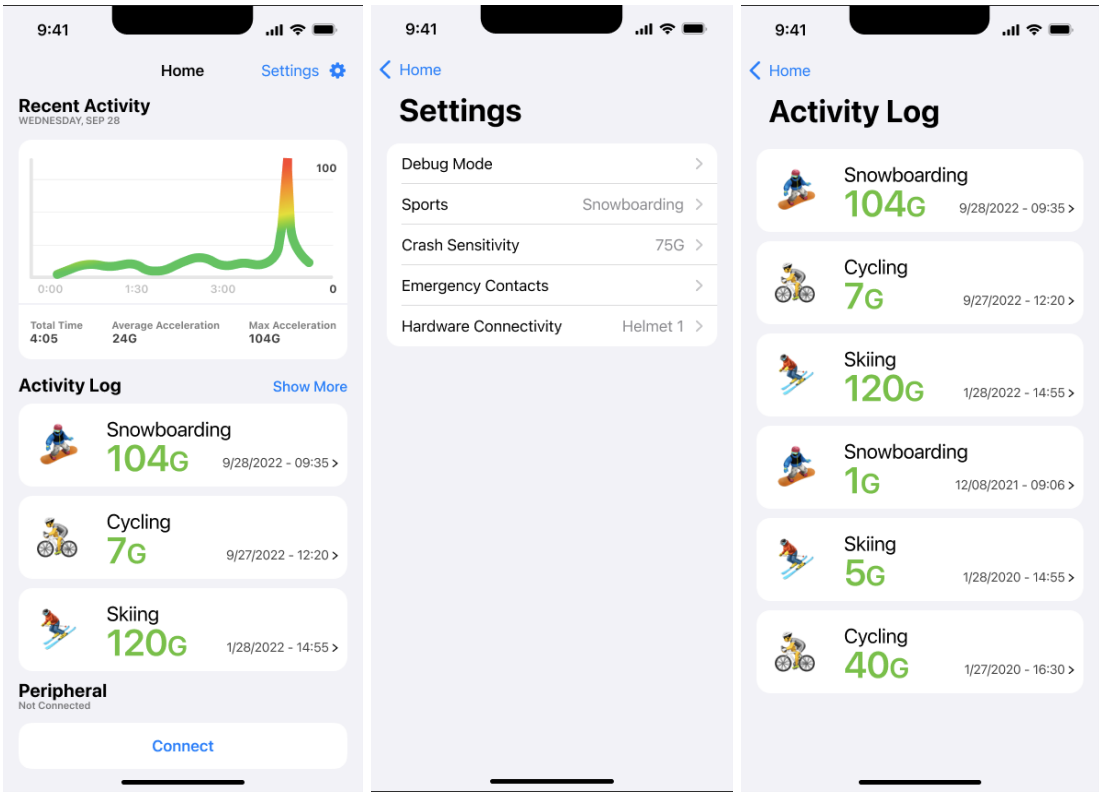


Figure 1: Primary Page Mockup

We planned to use the built in Swift UI components. This allowed for a quick and efficient development cycle as we did not have to build our own custom components from scratch. We tried to reuse as many components across the pages as possible to keep a consistent

theme. The general design across each page will remain consistent to foster a cohesive user experience.

The home page displays basic information like the status of the helmet, current/most recent ride, as well as quick settings access. Each component will be able to be used as a shortcut to the other pages. Clicking on the status of the helmet will direct to the “record run” page, clicking the most recent ride will bring up the logs page, and clicking on the quick settings will link to the settings page. The settings page will have a standard list of categories grouping individual settings and allow the user to alter things like the crash threshold or emergency contact. The logs page will display graphs of recorded runs and important values.

IMPLEMENTATION

HARDWARE

The final hardware implementation did not significantly deviate from the conceptualized hardware. The key components of accelerometers, a microcontroller, flash memory, and a portable battery bank were present in the final implementation. One difference between the conceptualized hardware and the finalized hardware is the lack of the custom PCB. Although a PCB was designed, it was never implemented due to the accelerometers already coming installed on a compact evaluation board that made the PCB redundant.

One significant challenge with the hardware was finding an accelerometer that could measure accelerations in the desired range and also communicate with the microcontroller. Initially, ADXL372 accelerometers were considered because the datasheet indicated that they could detect accelerations in the desired range of -200 g to 200 g [4], and had I2C and SPI protocol compatibility, which would be necessary for the microcontroller to read the data. The first issue with the accelerometers came when they first arrived. They were too small to have lead wires soldered to them, so new ones that came attached to an evaluation board were ordered. These boards were appropriately sized, but we had difficulty getting the accelerometer to interface with the microcontroller. There was also a lack of documentation on the ADXL372, so it was ultimately decided to use a newer accelerometer. Like the ADXL372, the ADXL375 can detect acceleration from -200 g to 200 g [5], is compatible with I2C and SPI protocols, and has documentation about how to connect it to various microcontrollers. During testing, we found that the ADXL375 accelerometer successfully transmitted accurate data to the microcontroller, so it was used in the final implementation. One change made from the conceptualization in the final implementation was reducing the number of accelerometers. Initially, we planned on using three

accelerometers, but we decided to only use one because the three accelerometers would all be providing the same acceleration data, making it redundant. Using one accelerometer would simplify the wiring and decrease the power consumption.

The microcontroller used was the Adafruit Feather HUZZAH ESP32 breakout board, although other microcontrollers were ordered in case of a malfunction. As mentioned in the conceptualization, the microcontroller's integrated BLE support and low power consumption made it ideal for a wearable product. The board uses I2C and SPI protocols for communication, which the accelerometer also uses. Unlike the accelerometer, there were no significant issues with the microcontroller and no changes were made from the conceptualization stage.

Finally, a 3D-printed base was designed to house the entire system. The base was designed with a curved bottom so that it can fit most helmet curvatures, and had a piece of VHB tape to attach to the top of the helmet. The microcontroller and accelerometer are affixed to the base with nylon screws and washers to prevent any potential electrical interference. The base with the microcontroller and accelerometer attached to it are shown in the figure below.

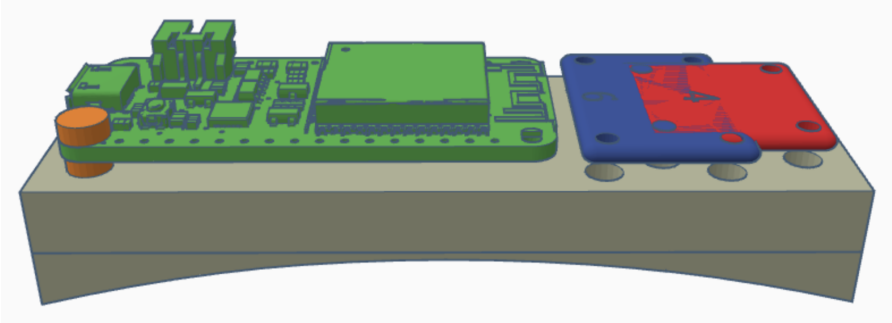


Figure 2: Final Crash Buddy Peripheral CAD

FIRMWARE

The firmware for the peripheral device was implemented using the Arduino framework. The implementation utilized the “ESP32 BLE Arduino” library for BLE operations and the “Adafruit_ADLX375” library for sensor communication.

The workflow of the peripheral has three distinct stages: Initialization, waiting on client actions, and impact detection. The high-level workflow of the peripheral goes as follows:

1. Initialize I2C bus
2. Initialize ADXL375 sensor
3. Initialize BLE module
4. Configure BLE module as GATTS
5. Begin advertising GATTS over BLE
6. Wait for a client to connect to the GATTS
7. Wait for a connected client to set an impact detection threshold
8. Begin monitoring for impacts using the given threshold

In the initialization stage, the I2C bus, ADXL375 and BLE module are enabled and configured. The ADXL375 is configured in I2C mode with a sampling frequency of 100 Hz and a full range of +/- 200 g’s. The BLE module is configured to use the GATTS protocol under the name “ESP32” and the UUID “09980000-1280-49A1-BACF-96520962E66”.

The GATTS protocol works by making data available in a series of tables called “Services”. Each service contains a series of elements called “Characteristics”. A connected client can request to write, read or be notified of any changes to a specific characteristic within a specific service. The GATTS scheme on the peripheral uses 1 service containing 25 characteristics. The first 9 characteristics contain different attributes of the peripheral, such as the

threshold value, the device status, and information regarding the size and availability of any new impact data. The accelerometer data is distributed across the remaining 16 characteristics. Since each individual characteristic is limited in size to 516 bytes and each accelerometer data point is 6 bytes wide, a single characteristic can hold 86 datapoints. In order to send the desired 20 seconds of data for each threshold-exceeding impact, 2000 data points total need to be sent using the GATTS protocol. To do this, the data is divided into 16 sequential chunks of 86 data points and placed into 16 sequential characteristics. In order to access all 2000 data points, a client reads each characteristic sequentially then concatenates the data. See Appendix C for a data sheet describing our GATTS protocol implementation.

Once a client device is connected to the server, the peripheral waits for the client to set their desired threshold for impact detection. Following the impact threshold being set, the peripheral will begin reading the sensor values into a 2000 data point wide ring buffer data structure.

If an impact exceeding the given threshold is detected, the device will continue to record an additional 1000 data points after the impact. Following the recording of the additional data points, the ring buffer will hold 999 data points preceding the initial impact, 1 data point at the initial impact, and 1000 data points following the initial impact. This totals to about 20 seconds of data.

Once the desired data around an impact event has been gathered, the values in the ring buffer are divided into 16 chunks containing 86 data points each. These chunks are then sequentially loaded into the corresponding characteristics. Once all the data chunks are loaded to their respective characteristics, a characteristic describing if new data is available is updated.

This signals the client that a crash has occurred and new data is available. Impact detection resumes until this process is repeated at the next impact detected.

SOFTWARE

For our home page, we have a graph of acceleration data from the latest run, a log of the most recent activity runs, a navigation link to a settings page, and a button to start tracking a new activity run given that the hardware device is connected via Bluetooth.

The activity log takes the user to a new page where it lists all the previous activity runs, and the user has the ability to click into any individual activity and view the graph of acceleration data, location of occurrence, total time of the run, average and max acceleration of that respective activity run.

When the hardware peripheral is not connected, the user is not able to start and track a new activity run. When the hardware peripheral is connected, the user is able to start a new run and is prompted by selecting an activity profile, which is made up of a sport and acceleration threshold in Gs. The user is then able to select an emergency contact. Afterward, they can start tracking a new activity run.

On the settings page, users have the ability to add, read, edit, and delete a new activity profile or emergency contact. For an activity profile, users have the ability to add, read, edit, or delete a new sport and acceleration threshold. The same functionality applies to emergency contacts, as users are able to fill out contact information including name, phone number, address, and relationship to the user.

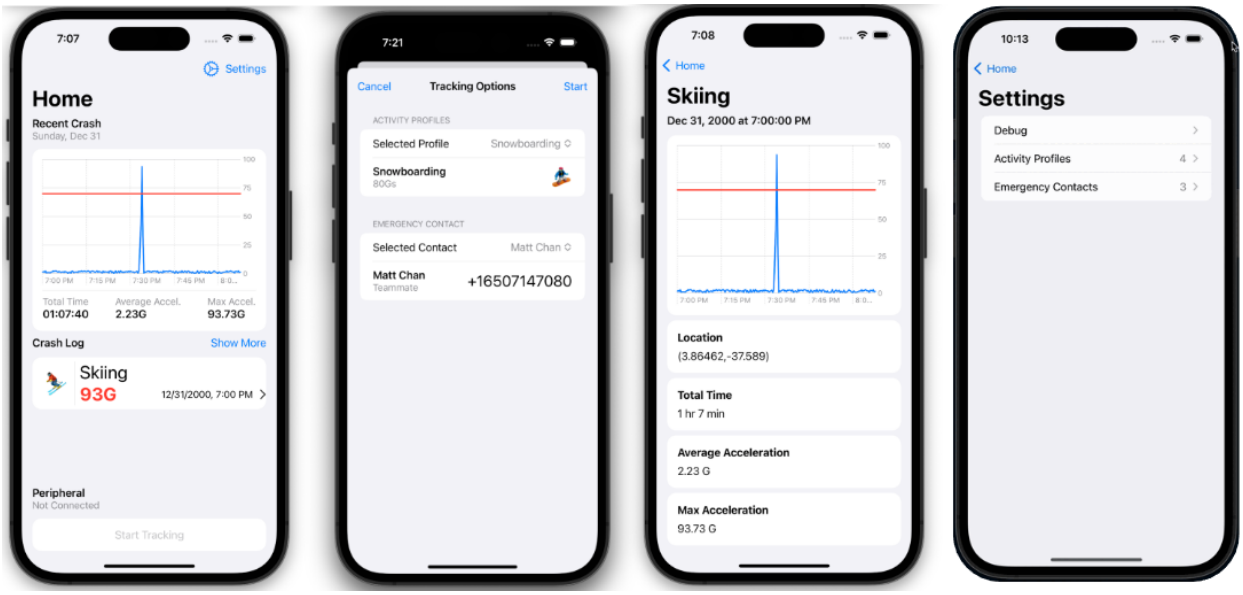


Figure 3: Views of Home Page, "Start Tracking" Pop up alert, Individual Crash Log Activity Page, Settings Page (left to right)

Once a crash is detected, the user will be prompted to cancel an automated text message to their selected emergency contact. If not canceled within thirty seconds, the application will send a text message with information regarding the crash including the geographic coordinates of the user and the activity type. The automated text message workflow consists of having a locally hosted python server on Flask. This is relayed to Twilio, which is the service that sends the text message. Ngrok is used as a proxy to the local host, which removes the need for a hardwire connection from our mobile device to a server within our current configuration. We are using Swift CoreLocation to record the user's location, and Alamofire is used to make a HTTP request from the application with information including the contact's phone number and user's location.

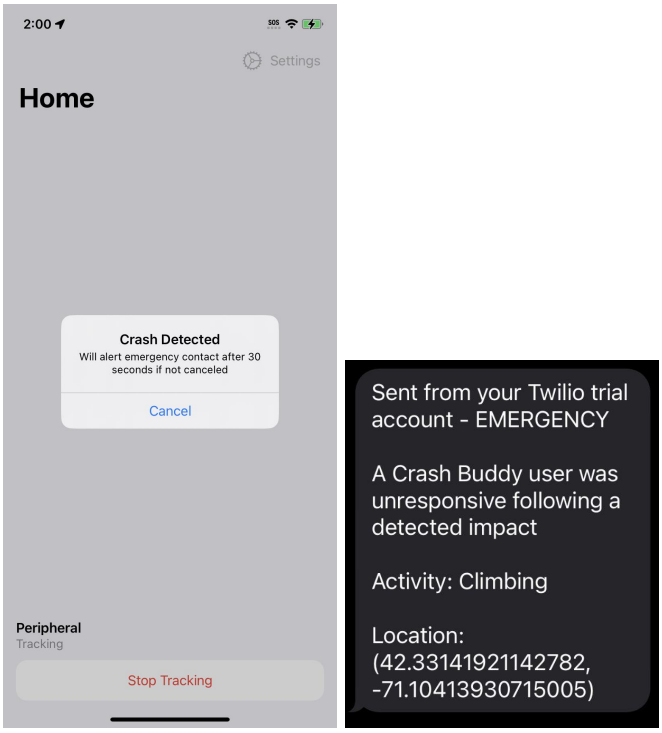


Figure: Crash Detection Alert and Example Automated Text Message

In order to save and retain the mobile application whenever it is opened/closed, we utilized persistent data within the mobile application to locally save all the data to the device itself.

COST ANALYSIS

The following table outlines the final cost of producing our device. Evidently, we were successfully able to remain under our allotted budget, which serves as a clear indication that the device could be viable in the consumer market.

Item	Quantity	Total Price
Portable Battery	1	$10
Adafruit ESP32 Feather	1	$25
ADXL375 Accelerometer	1	$13.66
3D Printed Base	1	$25.43
Heat Set Inserts	6 (Pack of 10)	$6.75
Nylon Spacer	7	$7.84
Nylon Screw	6 (Pack of 100)	$7.97
		~ $96.65

FUTURE WORK

To further refine our project on the software side, we would move the local mobile application storage to a remote database. Additionally, the software backend would be hosted on the cloud instead of locally, so that any device would be able to connect to our service without the need to be physically connected to the server. With the database set up, we would be able to further develop our Crash Buddy User Network and store user account information to further refine the customer experience.

As for hardware, we would further improve the usability and safety of the product. We would add peripheral buttons to control the power, bluetooth pairing, and starting/stopping crash detection. We would also add additional memory and an integrated battery to improve the performance and practicality. After all these are added, we would add an encasing to elegantly enclose the internals and integrate it with the helmet. We would also further test the product's software and hardware features to make additional improvements.

CONCLUSION

This report outlines our methodology for constructing a system for helmets capable of detecting collisions and engaging with its user to ensure their safety. By utilizing accelerometers and the user's mobile device, something which nearly everyone in modern society has, we were able to reduce the risk on athletes in the case of a collision. Not only does the device save the collision data, which can be used for further analysis, but it also acts as a failsafe when the user becomes unresponsive. This product makes sense to build 'today' as sensors, phones, and small efficient batteries have never been so cheap. As previously mentioned, this reduces the total cost of building such a device, which in turn makes it viable for the consumer market. In sum, we are simply striving to create a better and safer society.

REFERENCES

[1] A. H. Ropper and Author AffiliationsFrom the Department of Neurology, “Concussion: Nejm,” *New England Journal of Medicine*, 26-Apr-2007. [Online]. Available: https://www.nejm.org/doi/full/10.1056/nejmcp064645?casa_token=vFPsSqYlsdkAAAAA%3A0v6MRMW0VxzonactJxI5_8zzoUznRiqFa4l9BB3mg-YbdG0HNrkrAjWtyi8vbXWneriIKmEA8m2RDw. [Accessed: 15-Dec-2022].

[2] Willer, B., Leddy, J.J. Management of concussion and post-concussion syndrome. *Curr Treat Options Neurol* 8, 415–426 (2006). https://doi.org/10.1007/s11940-006-0031-9

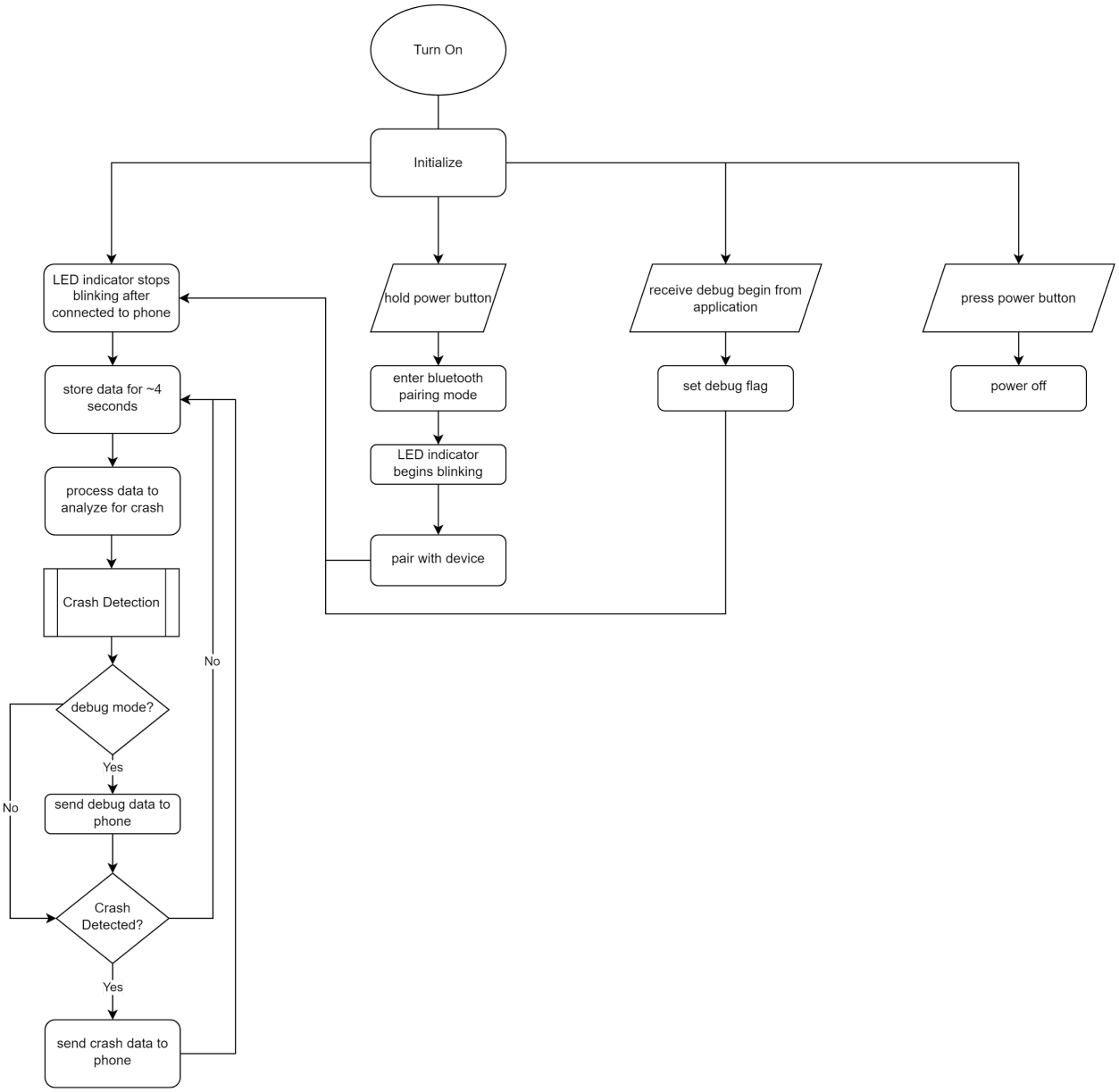
[3] S. Kockara, T. Halic, K. Iqbal, C. Bayrak and R. Rowe, "Collision detection: A survey," *2007 IEEE International Conference on Systems, Man and Cybernetics*, 2007, pp. 4046-4051, doi: 10.1109/ICSMC.2007.4414258.

[4] Analog Devices, “Digital MEMS accelerometer data sheet ADXL375 - Analog Devices,” *Analog Devices*, 2014. [Online]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL375.PDF. [Accessed: 15-Dec-2022].

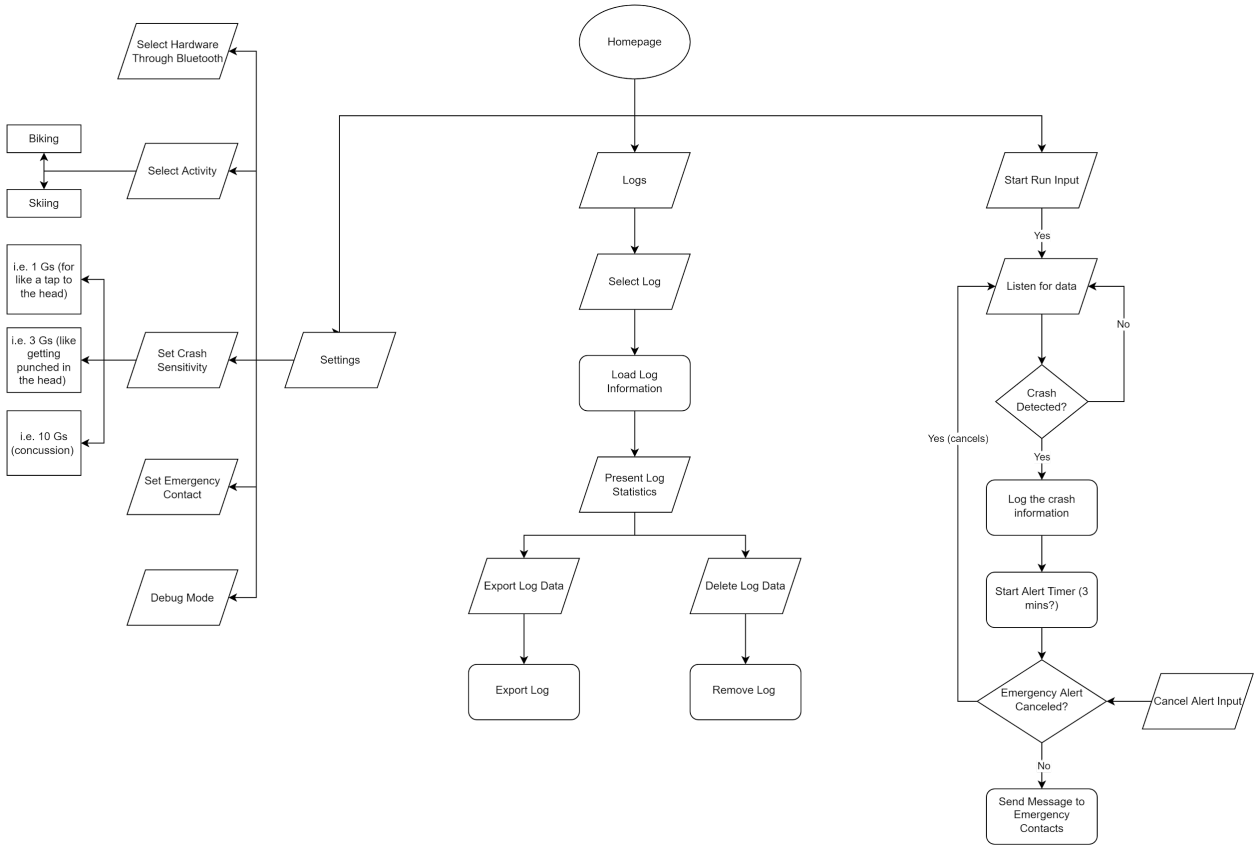
[5] Analog, “ADXL375,” *Datasheet and Product Info | Analog Devices*, 23-Aug-2018. [Online]. Available: https://www.analog.com/en/products/adxl375.html#product-overview. [Accessed: 15-Dec-2022].

APPENDICES

APPENDIX A (Hardware Block Diagram)



APPENDIX B (Software Block Diagram)



APPENDIX C (BLE GATT TABLE LAYOUT)

BLE GATTS PROTOCOL:

GATT IDs:

- From MSB -> LSB
  - 16 Byte ID = 2 Bytes Base UUID + 2 Bytes Characteristic UUID + 12 Bytes Device UUID

> 0998XXXX-1280-49A1-BACF-965209262E66

  - where XXXX is the CHAR ID

GATT TABLE LAYOUT:

MEANING	CHAR ID	DATA TYPE	PERMISSIONS	DESCRIPTION
STATUS	0x0001	bitmap - 32 bits	READ NOTIFY	0th = accelerometer connected
SET_THRESHHOLD	0x0003	uint32_t	READ WRITE	Allows client to set an acceleration threshhold for determining
DATA_AVAILABLE	0x0004	uint32_t	READ NOTIFY	Incremented each time a new set of crash data is available
CHAR_AVAILABLE	0x0005	uint32_t	READ	total # of data characteristics available to be read
MAX_CRASH_DATA_CHAR_SIZE	0x0006	uint32_t	READ NOTIFY	Max # of data_points per CRASH_DATA characteristic (see note below)
SET_ENABLE_DEBUG	0x0007	uint8_t	READ WRITE	Set to non-zero value to enable DEBUG MODE
DATA_SIZE	0x0008	uint32_t	READ NOTIFY	Total # of data points available to be read
if a crash occurred				
CRASH_DATA	0x0010-0x0020	struct data_point[MAX_CRASH_DATA_CHAR_SIZE]	READ	Accelerometer data from crash in chunks of at most MAX_CRASH_DATA_CHAR_SIZE data_points (see note below)

```
struct data_point { // total size = 6 bytes
    int16_t value; // value recorded by accelerometer
    int32_t time;  // clock time, not real time
}
```