Workshop documentation

01-2020

Prof. Sivan Toledo

Students: Loren Papiashvili, Ron Belkin, Ron Lakunishok, Mark Klein

'Distress Board' project:

Our project implements a product consisting of the CC1350 Texas Instruments board, in combination with the BMI160 Bosch accelerometer, with a designated application to supply the user with a comfortable UI.

The project keeps track of the user's well-being both reactively and proactively.

From a reactive standpoint, the board has two buttons – one on each side, once one of these buttons are pressed, the designated app enters a distress state, allowing the user to either react to it (cancel) or wait some time and then send a distress signal via SMS messaging.

From a proactive point of view, the board has a fall detection feature, allowing it to notify the app once the user has (allegedly) fallen, in order to let the app enter a distress state and query the user if everything is alright.

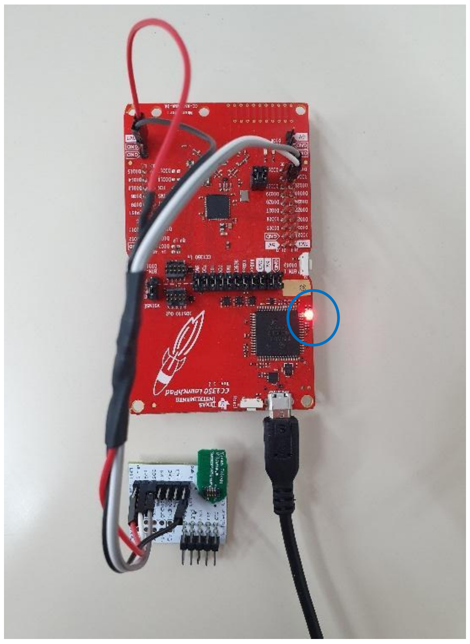
Another proactive feature is the stationary/idle position time management feature, in which the board monitors the movements of the user, to decide whether the user is in a relatively stationary/idle position.

If this is true over a customizable (default is 30 minutes) period of time, the board notifies the app and prompts it to enter a distress state.

<u>Usage:</u>

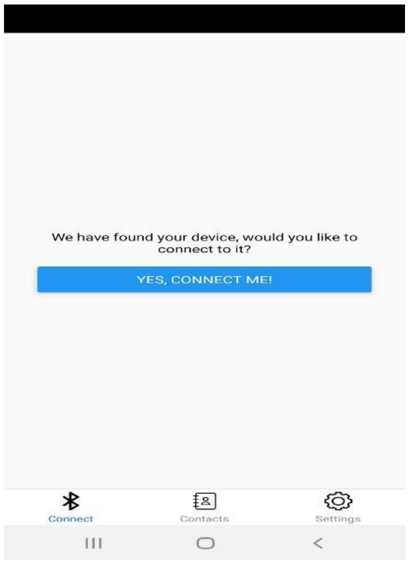
In order to use the app and provide the best user experience, we specify the following steps:

1. Make sure the hardware is connected to a power source. (Once connected, the board should have LED lights [one red/one green/both] light up) – *Illustration*:



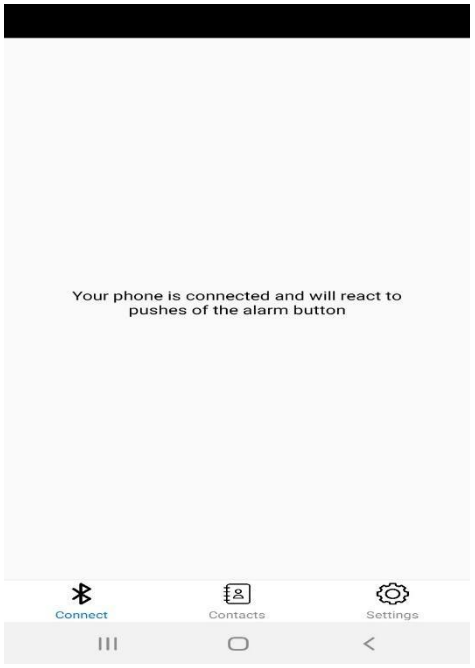
2. Open the designated app and wait as it scans for your device.

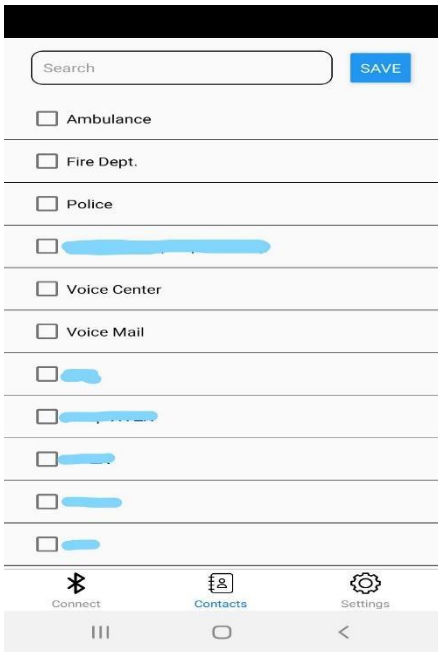
3. Once a device was found, a message will be displayed, press it to connect to the device. – *Illustration:*



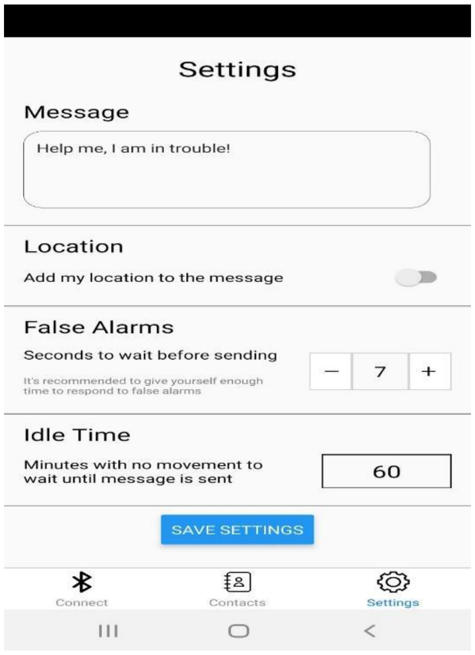
We have found your device, would you like to connect to it?

4. Now, the app and the board are connected, you may use the board as described earlier, and customize the contacts that will receive your distress signal, the message that will be sent, and much more – through the *contacts* and *settings* tabs. – *Illustration:*


Your phone is connected and will react to pushes of the alarm button


Search

- ☐ Ambulance
- ☐ Fire Dept.
- ☐ Police
- ☐ [redacted]
- ☐ Voice Center
- ☐ Voice Mail
- ☐ [redacted]
- ☐ [redacted]
- ☐ [redacted]
- ☐ [redacted]
- ☐ [redacted]


Settings

Message

Help me, I am in trouble!

Location

Add my location to the message

False Alarms

Seconds to wait before sending: 7

It's recommended to give yourself enough time to respond to false alarms

Idle Time

Minutes with no movement to wait until message is sent: 60

Motivations:

At first glance, the project might seem redundant to some, especially in the era of smartphones and smartwatches, however, we would like to argue that it fits in perfectly as a nice addition to the existing IOT environment.

The project's use cases are limited only by imagination, here we will specify the ones the project was most designated for.

The distress button, which if pressed leads to the distress signal being sent as an SMS, optionally including location (customizable), can be used in every scenario where an attacker tries to threaten/harm the user carrying our project. The device we designed is to be worn on an accessible location such as the waist, so that it can be easily reached during a physical assault.

When designing the project, the situations we had in mind are such where the victim cannot, or does not have enough time, to call for help using their cellphone – such situations include sexual assault / rape, physical beating / planned ambush / stabbing, robbery – these usually occur unexpectedly, and the victim is very limited in their movements, a simple, minor action as a button press on a reachable area is the best chance the victim has in quickly getting help and deterring the attacker.

The fall detection identification, a useful feature, especially for the elderly and for those suffering from epileptic seizures. Counting on the people near us (physically), if there are any, is not an adequate solution to the problem, especially since most of the time we are not surrounded by people aware of a certain condition we might have. These populations require someone (or something) to intervene and call for help in their behalf.

These are the fundamental scenarios we thought about when implementing this feature, yet many more can be found relevant. For instance, when a pedestrian is involved in a road accident, they will inevitably fall, leading to a distress signal being sent. This is of utmost importance when discussing hit-and-run accidents, which unfortunately are not that uncommon (see this report for further detail).

The stationary/idle position time management – as students, we are no strangers to feelings of depression and sadness, or long study sessions with little to none water/food/bathroom breaks. From what we've heard, this is especially true for students taking pharmaceuticals to improve their concentration.

What these situations have in common is that we need someone to pull us out from them, in the name of our mental and physical well-being.

However, each person has his/her own limit, therefore the number of minutes allowed for being in a relatively stationary/idle position is customizable.

More scenarios can be thought of where this feature is useful. For example, consider a person having some disability, or an elderly with limited motoric capabilities, these groups can sometimes get 'stuck' in a position they cannot get out of. This can happen in the bathroom, where they have trouble getting out of the bathtub, or getting up after visiting the restroom. In such events, we would like a person of interest, maybe a caretaker or a family member, to be notified.

(This is not the primary goal of the feature, and it might seem odd how our project servers this purpose, since a person in such a situation can just click the distress button, or not wear it at all as it might be inconvenient, yet there is the option to put the product away while taking a bath or visiting the restroom, and it will serve as a timer after which an SMS is sent to a preselected list of contacts)

Implementation:

To develop the **board** functionality, we first had to have BLE communications available. For this purpose, we looked at the *simple_peripheral* example supplied by TI (Texas Instruments, the company that produces the boards).

This example gave us the basis for BLE communications and exposed two very convenient features for us to use.

The first feature was the characteristics. *simple_peripheral* comes with 4 characteristics, some of which are readable, some writable, some both, and one notifiable characteristic – meaning that any change to it yields a notification being sent to whoever is monitoring it.
To signal a distress state, the board changes the value of characteristic number 4, the one that is notifiable.

These characteristics can be thought of as key, value pairs where the key is some UUID. To learn these UUIDs and experiment with reading/writing to them we used the *BLE Scanner* app (we used it on android devices). This app was the main software verification tool we used until our app was finished. (more on the app later)

The second feature was the periodic event function. *simple_peripheral* comes with a function that occurs every number of milliseconds, this number can be changed in the code, in our current version that number is 1000, meaning that the function code is executed about every second.

Considering that our project is about constantly measuring the movement and checking the well-being of the user, the periodic function was very natural for us to use.

Another example project from TI that we used was the *pinShutdown* project, which incorporated button press handling code. This project had the code we needed as part of its main task, so we had to pick out the parts that we needed by going through the project and studying it carefully, similar to what was done with *simple_peripheral*.

[*we are about to discuss some electrical/physical principles related to the hardware, this is only our understanding of them, please excuse us if there are some inaccuracies*]

What happens is that when a button is pressed, a clock is activated for a certain period, after this period we measure voltage at a point close to the button and if that voltage is zero, we identify this chain of events as a button press.

Why was the clock activated? This was done as part of a principle called ‘debounce period’. Apparently, when the button is pressed, many signals are being sent, so we want to wait for things to settle down and stabilize in order to reliably sample a press of the button.

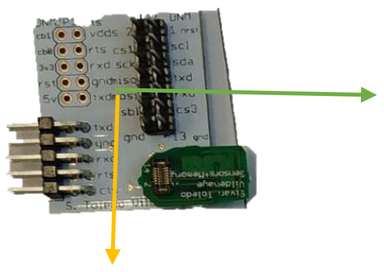
Once the press is identified, the function *CreatePanic* is called, which changes the value of some variable. This variable is inspected in every execution of the periodic function and if a change was noticed, the value of characteristic 4 is changed, and hence a notification is sent to whoever is monitoring this characteristic.

Our project also includes an accelerometer, specifically the BMI160 from Bosch.

It communicates with the board through I2C protocol. This required us to connect the SDA and SCL pins connected to the sensor to the corresponding pins on the board. To understand what pins are configured as SDA/SCL we looked at the configuration of the *i2ctmp* project, another example from TI that illustrates basic I2C communication with a temperature sensor.

The BMI160 comes with a driver which exposes various functions to configure, initialize and read data from the sensor. However, to use this driver we had to implement some functions on our own: reading/writing data to a register in the sensor over I2C and delaying. More on these in the *obstacles* section.

Once we handled the I2C infrastructure, we could read acceleration data from the sensor, this is how we identified falls and idle/stationary periods.



The green arrow represents the x axis and the yellow arrow represents the y axis.

The z axis is the one pointing 'inwards', or 'through', the sensor.

The user wears the device with the y axis pointing down, as in the image above. (this matter was discussed shortly in the project presentation)

<u>Notation:</u> acceleration in axis x is x_a. Correspondingly, define y_a, z_a.
(acceleration is measured in $\frac{m}{s^2}$)

To identify a fall, we require $z_a \geq 9 \,||\, x_a \geq 9$

This captures falls where the person who fell lies on their side, and the ones where the person lies on their back/front.

If the person switches between lying on their side and lying on their back/front – another notification is **not** sent, this is a feature rather than a bug.

However, once a person gets back up ($y_a \geq 8$), a notification will be sent if he falls again.

An idle state was defined as:

$$lying\ down\ on\ back||lying\ down\ on\ side||sitting\ at\ an\ angle||sitting\ up\ straight$$

Where:

$lying\ down\ on\ back$	is	$z_a \geq 9\ \&\&\ x_a \leq 1$
$lying\ down\ on\ side$	is	$x_a \geq 9\ \&\&\ z_a \leq 1$
$sitting\ at\ an\ angle$	is	$y_a \leq 8\ \&\&\ z_a \geq 2$
$sitting\ up\ straight$	is	$y_a \geq 9\ \&\&\ z_a \leq 1$

We check whether we are in an idle state each time the periodic function is executed (meaning each second, in the current version of our code), if so, we increase a counter, otherwise, we assign zero to the counter.

If the counter becomes greater or equal to the number of allowed idle minutes times 60 (the number of allowed idle seconds essentially), we call *CreatePanic*.

In addition, each period we check whether the value of characteristic 1 has changed, to see if the user updated the allowed num of idle minutes in the app.

Let’s talk about the **app** implementation.

The app was built using **React Native**, an open source framework developed by Facebook, which allows you to build native Android and iOS apps using (mostly) Javascript code, with a single codebase that is derived from the **React** library that already exists for web browsers.

React Native comes with a lot of useful built-in modules (mainly UI related ones). However, those are really basic, and in order to customize the functionality, we had to rely a lot on external libraries to do so. Luckily, the React Native ecosystem is rich in those resources, and we could easily find custom modules that suited our goals almost perfectly.

The main one would be a library called **react-native-ble-plx**, which is a set of tools that enables you to manipulate the native BLE capabilities of your phone with a fairly simple set of APIs and abstractions.

Using this library, we have created a **BLE service**, which is, in essence, a class that represents the module and abstracts it even further, simplifying the operations we have to perform to deal with BLE. We exported functions such as one that scans for our board (we used its name to identify it for simplicity, but we could have used any other property such as UUID), that monitors the aforementioned characteristics and fires distress events on demand.

Storing the contacts and settings selected by the user was rather simple too, using the built in **AsyncStorage** module of React Native, as well as a library called **react-native-contacts** that allows you to access your phone’s contacts list and get them in JSON format.

Most of the UI was made by us, with a few small exceptions where we used premade UI elements from external sources and **react-navigation** to make the bottom tabs

<u>Obstacles:</u>

Board

- At first, we didn't have any clue as to what we should do and how to begin working with the board, even defining objectives was difficult
- We found *simple_peripheral* online, but it was Professor Sivan Toledo's guidance that helped us understand how to begin working. He introduced us to the BLE scanner app and explained how to work with Code Composer Studio, we also received an explanation for the difference between *simple_peripheral_app* and *simple_peripheral_stack*
- When trying to write a handler for a button press, we had to understand exactly what parts from *pinShutdown* were relevant, this required that we investigate unfamiliar topics like the 'debounce period' concept
- We searched online for help regarding the hardware and software aspects that we had to consider in order to fully understand the *pinShutdown* example and were first introduced to the TI forums where we found answers to some of the questions we had
- When we were done with the button press functionality (the board and app development are independent once we reached the convention that a distress state is identified by a change to characteristic 4), we approached Sivan to get an accelerometer. Sivan gave us an accelerometer and soldered it to a panel with the pins we needed for I2C communications.

Beginning to work with the accelerometer was difficult – we didn't even know how to connect it to the board.

- Following Sivan's suggestion, we searched for the configuration of SDA/SCL pins and understood how to physically connect the sensor to the board.
- We had to remind ourselves of the way I2C works, understand how TI's I2C API functions, and how to read/write to registers in the sensor.
- We watched a couple of videos about I2C on *youtube* and made sure the same principles apply for the sensor by visiting its datasheet. We looked up TI's I2C API for some basic examples on how to read/write over I2C. However, the API interface does not support reading/writing to a specific register, so we turned back to the datasheet to find out that the first byte of the I2C transaction after the slave address and R/W bit (we learned how to configure these from the driver documentation) is the register address.

App

- *At first, we did not know how to make our phone interact with the board. We were certain that the Blutetooth capabilities could be used, but looking at React Native's base documentation yielded little to no results. So we turned to npmjs.com to look for an external resource. Soon enough, we found react-native-ble-plx, we started reviewing its documentation and the abstractions it provided were crystal clear and simple to operate.*
- *Afterwards, we needed some reliable way to send SMS. Again, we turned to npm to see whether we can find something that does that, but all the libraries we found required user interaction to deliver the messages, which completely defeated our goals. We looked for alternative solutions to tackle the problem. React Native provides the capability to write modules in the native language of the phone's OS (in our case, Java for Android). After some research, we found that Android has a module that directly sent SMS, so we used it and found out how to integrate that module in our Javascript code, ending up with a native Java module built by us, which allows us to send SMS messages without interaction from the user*
- *Location details were a bit of a thinker. Although React Native has a module that extracts location data from your phone, it merely provides you the coordinates, but no way to represent them in a message (raw coordinates are not human readable). Google Maps came to our aid, with a simple API that can receive those coordinates and locate them on the world map. So we send the url of the API with the coordinates we get from React Native.*

Resources:

Board

- First, we would like to thank Professor Sivan Toledo for agreeing to meet with us, explain based on his previous knowledge, and overall maintaining his patience as he guided us through beginning each major step in the project.
- We would also like to note the TI forums, where we found answers to many of our questions and code examples or principle explanations that helped us a great deal.
- TI's example projects were essential to understanding what features are available and how they're used
- BMI160 datasheet and driver documentation – the datasheet explains fairly well what features and capabilities the sensor supports, it also has a lot of information about communicating with the sensor over I2C, the driver is written in an organized manner and each struct/function are documented in an easily understandable way

App

- *React Native - Facebook's framework for Native apps written in JS*
- *react-native-ble-plx - A toolset for handling BLE operations for React Native*
- *react-native-contacts - A library to extract contact data from your phone*
- *react-native-push-notification - A library we used to make local distress notifications*
- *react-navigation - A library we used to build the bottom tabs of the app, navigating between different screens*