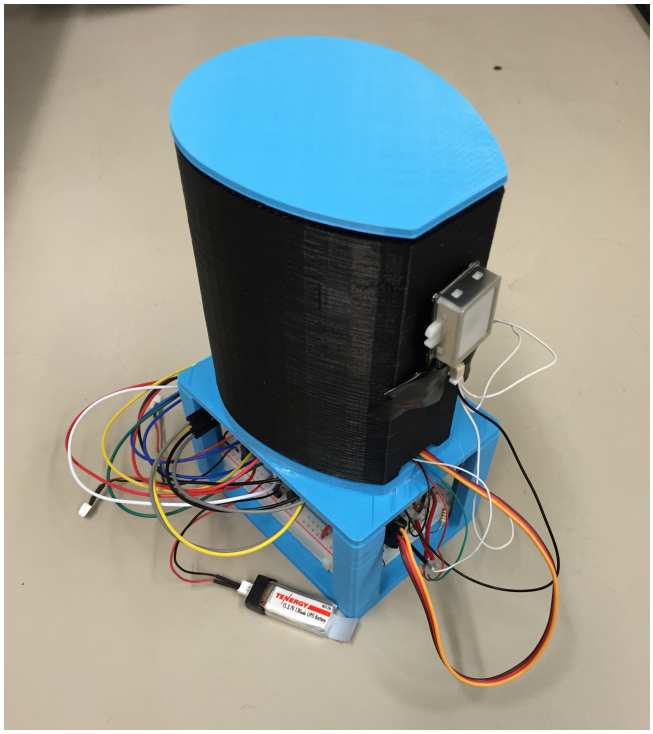
SmartPills
Project Report

EECS 149/249A
Fall 2014

Joshua Goldberg
Gil Lederman
Vicenc Rubies Royo
Parsa Mahmoudieh



Project Introduction

The goal of our project was to create a smart pill bottle that would enhance the user experience surrounding medication adherence. The pill bottle prototype that we developed is capable of measuring the number of pills left in the bottle and it automatically unlocks and opens when the prescribed individual touches their finger to the fingerprint scanner. It also communicates with an Android app to display to the user the number of pills remaining as well as other useful data.

Embedded System Architecture

Introduction

The scope of the mbed part of the architecture includes controller logic, sampling and driving the GT511C3 (fingerprint scanner), interfacing with the ADXL345 ("smart" accelerometer), measuring the load cell, commanding the micro-servo, and communicating with the Android device.

Choice of high-level architecture and MBED-RTOS

Our initial task was to choose an architecture that will allow us to concurrently perform the various tasks involved. One of the main constraints was the implementation of the communication with an android device through the BLE (effectively, though the UART). Rather than deal with asynchronous non-blocking two-sided communication through the use of UART interrupts and some buffers (or, alternatively, through periodic polling as part as a main controller loop), we chose to use a simpler approach, reading/writing the UART in a blocking manner. Of course, this is only a reasonable solution if we are using threads, and so we chose to use the mbed-rtos, which schedules threads for us, and in addition provides safe communication mechanisms between them in the form of queues (even useable from interrupt context!). This allows us to completely avoid using any other explicit locking mechanisms, and make do with a single shared variable that is only written to from two possible ISRs. There are other "shared variables" of course, but they are all initialized by the first thread on startup, and are henceforth read-only.

High-level architecture

While the system has a few auxiliary threads, there are a total of three "active objects", that is, threads that are encapsulated in a subclass of a class called 'FsmThread', each with its own input queue and an internal state, and a common implementation of FsmThread::run() method that blocks on the input queue until there is a message, and then proceeds to dispatch it to a virtual method fsm_handler, which each individual FSM subclass implements on its own. These are:

ControllerFsm

This is the main state machine, implementing the FSM chart seen in Figure 1 (not including commands from the Android, which do not move the Controller to a new state under normal operation):

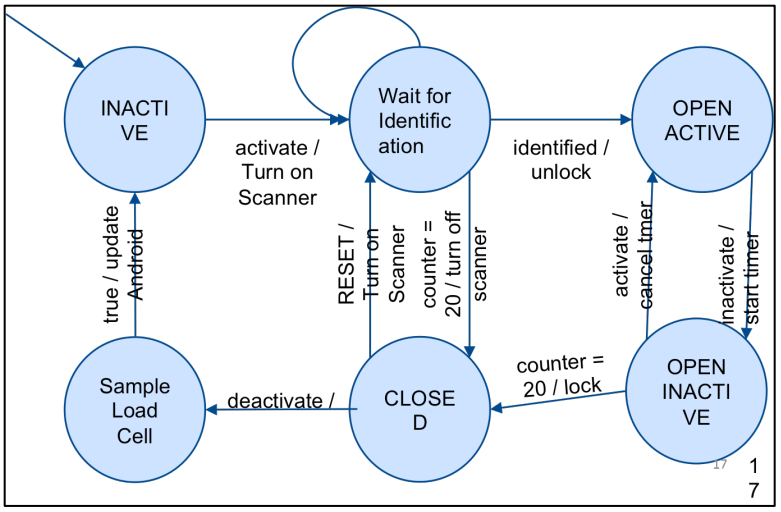


Figure 1. Deterministic Finite State Machine

The important observation is that the software only samples the load cell (effectively detecting the number of pills) some time after the bottle has become "inactive", that is, the accelerometer reports it has stopped moving and is not tilted.

AndroidFsm

The Android FsmThread is relatively simple. All it does is wait for events from the other FsmThreads (controller, scanner, or some admin debug provider thread), and when it receives one, it translates it into a matching Android-MBED protocol message (See Android communication section) and sends it.
We will mention here that there is an extra thread, not an FsmThread (since it has no input queue and listens to no "internal" events), called AndroidSerial, which does a blocking read on the BLE UART. When it received a valid message from the Android, it translates it to a matching FSM event, and injects a message into the system, basically pushing a message into either ControllerFsm or ScannerFsm queues.

ScannerFsm

This is the Fingerprint scanner FSM. It is mostly superfluous; most of its functions might as well have been implemented in the ControllerFsm in a non-blocking manner. It has only one truly blocking procedure, enrolling a new fingerprint (we've made the identifying process event-based, with the help of a periodic timer, in order to allow for cancellation in the middle of the identification process), and we dedicate a thread to it mostly for "legacy" reasons, its not that complicated to implement the enrollment asynchronously and forego the thread here.
We've used a very useful mbed library for interfacing with the GT511C3, originally created by Toshihisa T.

Other "passive" components

Here we list objects that interface with various scanners and actuators, but are not "active", have no thread context of themselves, and their methods are strictly non-blocking.

LoadCell

The LoadCell object is used to sample the Load Cell, save a history of sampled values, and saves calibration values. To solve the problem of counting how many pills are in the container, we had to first solve the problem of accurately measuring the weight inside the container. Once that was determined, we could figure out the number of pills simply by dividing the weight of all the pills inside the container by the weight of each pill and then rounding. Since the voltage outputted from the load cell amplifier into the ADC port of the mbed varied because of noise, we averaged 50,000 samples for each measurement. We then measured the voltage for different weights and verified that the load cell reacted linearly to weight in the range we were interested in.
After measuring the sensitivity of the load cell to each pill, which did not change, we had to find the bias, which varied a lot between adjustments of our physical setup. We calibrate the pill bottle and record this bias the first time it is turned on. In order to calibrate it the user has to set the bottle empty on a flat table and press calibrate for the mbed to record the voltage bias on the load cell with zero pills. After this measurement is taken, we were able to count the number of pills in the bottle for great accuracy.

ServoLock

The ServoLock object simply sets a PWM value for the pin connected to the micro-servo, a relatively quick operation. A library by Simon Ford was used to drive the PWM pin.

AdxlModule

This module manages the ADXL345 accelerometer. The ADXL345 is a very sophisticated accelerometer. It features a built-in controller that can be programmed to detect activity, inactivity (with a properly defined time period), free fall, and in addition provide outgoing interrupts for these events.
We have been able to limit all ADXL345 handling to a one-time register setup on startup. From there, the ADXL345 provides us with two different interrupt lines, detecting activity and inactivity (for a given time period) in an interleaving fashion – that is, the interrupts are never enabled together, an ACTIVE interrupt must be present between every two INACTIVE interrupts.
We originally used these two interrupts to feed events into the system (mbed-rtos queues can be used from within ISRs), but later switched to a different approach. We re-discovered the fact that edge conditions tend to be fragile and prone to deadlocking. Instead of injecting a single ACTIVE/ INACTIVE event, we used the respective ISRs to set a shared variable AdxlModule::mIsActive, and added a periodic ticker interrupt to the ControllerFsm, which explicitly checked the shared variable to detect and execute ACTIVE/INACTIVE transitions.
We used a library here which mostly took care of the SPI interface, created by Aaron Berk.

Android Architecture and Communication

Introduction

It was necessary to write an Android application in order to enable communication between the user and the bottle. Some of the important features that needed to be part of the application included a way for the user to enroll his/her fingerprint, a way to set the pill weight, requests to (re)calibrate the bottle, etc. To achieve all of this, three things were necessary: first, to find a way to enable communication between the mbed and Android, second, to design a communication standard for the messages sent to and from the Android and the mbed, and, finally, to create a user friendly UI for the Android application.

Enabling Bluetooth Communication

In order for two devices to communicate over Bluetooth two basic things must happen: first, the devices must pair and then they must connect. To achieve this, the "Bluetooth Chat" sample code provided in the Android Eclipse IDE was modified. This code was already functional, so most of the features that we needed for communication were already present. In summary, the program uses threads that take care of all the steps from pairing, all the way to sending and receiving data. For the purposes of our project the "ConnectedThread" was the only thread that needed modification. This thread takes care of sending arrays of bytes from the Android to the mbed and continuously listens for incoming data by keeping track of an inputStream associated with the Bluetooth socket. Once data arrives it is then passed onto a buffer, which is in turn used to retrieve the information and read messages.

From the mbed side, Bluetooth communication is very similar, albeit easier. Given that the BlueSMiRF Bluetooth module is accessed physically via UART from the microcontroller, there is no need for the pairing or connecting steps. By simply interconnecting the Rx and Tx lines of the Bluetooth to the Tx and Rx lines of the mbed, respectively, one can regard the connection as being equivalent to the usual serial connection between the computer and the mbed via USB.

Finally, the Bluetooth module can be in one of two modes: command mode and data mode. In command mode, inputs to the module are treated as commands that serve to modify configuration parameters such as baud rate (9600 for our project), Pin number, Power modes etc. In Data mode, the BlueSMiRF simply acts as a transparent data gateway for information exchange. In the context of our project, command mode was only used to set the baud rate, and data mode was used virtually all of the time.

Communication Standards

An important feature for the project was the necessity to read and write messages in order to exchange information between devices. More important was the necessity to establish a standard for those messages. In that regard, messages were encoded as a series of bytes representing a header, a type, a length and a payload. The header bytes are merely used so that each device can discern between incoming messages and other incoming data. The type bytes simply hold information regarding what type of message is being sent or received. The length bytes contain the length (in bytes) of the payload, and the payload bytes (always a multiple of 4) contain the encoded information, which can only be decoded properly if the associated message type is known. Table 1 contains the messages, their structure, and function.

Message	Header	Type	Length	Payload	Use
Request Enroll Message	0xCCCCCCCC	0x00000000	0x00000000	N/A	Request from the Android to the mbed to register a new fingerprint
Request Calibrate Message	0xCCCCCCCC	0x00000001	0x00000000	N/A	Request from the Android to the mbed to calibrate the bottle
Set Weight Message	0xCCCCCCCC	0x00000002	0x00000004	0xFFFF-0x0000	Sets the weight of a single pill in milligrams
Request History Message	0xCCCCCCCC	0x00000003	0x00000000	N/A	Requests an update of the pill number history from the bottle
Acknowledge Message	0xCCCCCCCC	0x00000004	0x00000000	N/A	Acknowledges an event
Request Remove Finger	0xCCCCCCCC	0x00000005	0x00000000	N/A	Requests the user to remove the finger from the scanner
Request Press Finger	0xCCCCCCCC	0x00000006	0x00000000	N/A	Requests the user to press their finger against the scanner
History Message	0xCCCCCCCC	0x00000007	0xFFFFFFFF-0x00000008	(Length/8) packet/s;	The message contains packets. Each packet contains a pill number and a timestamp. The timestamp corresponds to a time when a change in number of pills was detected.
New Sample Message	0xCCCCCCCC	0x00000008	0x00000004	0xFFFF-0x0000	Message containing the number of pills in the bottle.
Request Unlock	0xCCCCCCCC	0x00000011	0x00000000	N/A	Requests to unlock the bottle

Table 1. Messages and their information

The Android application can receive History messages but the code needed for interpreting the payload, although straightforward, has not yet been implemented due to time constraints.

Each message is in fact a child of the parent class MessageH.java, which contains all the fields (header, type, length and payload) and methods necessary for converting a message into an array of bytes or to translate an array of bytes into a message. Using the MessageH class made it very easy to add additional messages.

User Interface

For the user interface a simple scheme was used using the Android ViewFlipper layout. This Android layout allows switching between screens containing different information, buttons, etc. When the user first enters the application he is shown a screen containing three buttons: Connect, Help and About. For the project, only the "Connect" button was implemented since it was the only crucial one. Once the user presses "Connect" the program starts an Intent to try to connect to the bottle via Bluetooth. Once the bottle has been connected, the user is

prompted to a new screen containing buttons and a text view. The text view updates the user on the number of pills in the bottle (while connected) as pills are added or removed. The buttons on the screen are used to calibrate the bottle, set the weight of the pills, get the history of pill usage over time, unlock the bottle, disconnect the application from the phone, and enroll a fingerprint to the scanner. In particular, this last button prompts to another set of screens that guide the user through the process of enrolling his or her fingerprint.

Hardware Design

Electrical Design

The load cell that we bought is effectively a Wheatstone bridge of strain gauges that has two leads with a linearly increasing differential voltage. An instrumentation amplifier was designed and implemented between the load cell and the ADC of the mbed in order to amplify this differential voltage by a factor of 300 to vary between 0 and 3.3V by 48 millivolts per gram. This equates to a variation of about 45 ADC units per pill, which is plenty in order to measure individual pills.

Mechanical Design

SolidWorks was used to develop CAD models for the bottle. These models were then 3D printed and assembled together into the final prototype. We developed an outer cylindrical shell that holds a micro-servo attached to a lid, as well as the fingerprint sensor. The inner cylinder holds the pills and is attached to the floating end of the load cell. The other end of the load cell is fixed to the base of the assembly. This whole structure was then attached to the top of two plates, separated by four pillars. Between the two plates is where all the circuitry resides. See Figure 2 for the entire CAD model of the design.

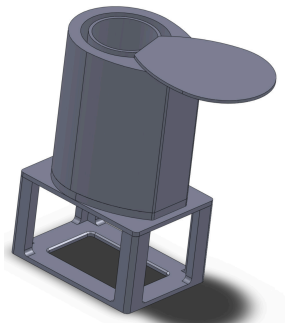


Figure 2. CAD Model of SmartPills

Sources of Error

One major source of error in counting the number of pills was that pills coming from the same pill bottle do not all weigh the same. We found this by measuring the pills with a precise balance and observing a difference in their weights. There was also a lot of parasitic capacitance and inductance in our module, which changed our voltage measurements and also sometimes caused multiple components to malfunction. This was due to the whole pill bottle module being very compact. There was one small breadboard and many wires connected to the sensors, batteries, mbed, actuators, and bluetooth module that were cramped and looped together without being shielded.
Other minor sources of error include temperature variations which could change the sensitivity and or bias of the load cell. Also, attempting to take measurements of the load cell on a non-level table is another source of error. We also found that the fingerprint sensor did not reliably identify the user's finger with every press.

Conclusion

We have learned many things from this project. We discovered that components working separately does not guarantee that they will still work once they are all integrated together. We also found that noise sources are a very big issue especially for analog measurements and they must be dealt with in the correct manner to avoid failure. We also found proper and formal modeling of the system very helpful in our project for achieving a working bug free system. Formal modeling syntax was also very useful when communicating amongst the team as we designed our system because it was a standard vernacular that we all understood. For future iterations of our project we would have better wiring and shielding to cut down on system noise. We would also use a more reliable fingerprint sensor and we would design a PCB for all of the components in order to make a smaller bottle. There is also much work that can be done to the user interface of the Android app to enhance the user experience when using SmartPills.

All code available at:
https://github.com/lederg/ibottle

Video available at:
http://youtu.be/ZpQqOkrph1s