5-1-2016

Integrating Bluetooth Low Energy Peripherals with the Amulet

Anna J. Knowles
Dartmouth College

Integrating Bluetooth Low Energy Peripherals with the Amulet

Anna J. Knowles
Senior Thesis
Advised by Professor David Kotz
Dartmouth Computer Science Technical Report TR2016-807

Abstract

The Amulet is a health monitor, similar in size and shape to a smartwatch but specifically designed to have a longer battery life and handle data securely. It is equipped with a Bluetooth Low Energy (BLE) radio in order to receive data from BLE-enabled sensors and transmit data to smartphones, but the full implementation of BLE communication on the Amulet is still a work in progress. This thesis describes architectural changes that improve the Amulet's ability to receive data from a variety of BLE-enabled sensors and make it easier for developers to integrate new BLE-enabled sensors with the Amulet by introducing support for connecting to multiple sensors at the same time, rewriting the radio code to be more generic, and exposing BLE functionality to the AmuletOS. We discuss the relevant parts of the AmuletOS and the BLE protocol as background, describe the current structure of BLE communications on the Amulet, and document the proposed changes to create a system for easily integrating new BLE-enabled sensors and handling connections to multiple sensors simultaneously.

1 Introduction

The Amulet is a wrist-worn health monitor, similar in size and shape to a smartwatch and equipped with a built-in clock, accelerometer, light sensor, microphone, and temperature sensor [1]. Amulet was designed in a research collaboration between Dartmouth College and Clemson University. It is intended for continuous monitoring of personal health statistics and environmental variables, useful either for research studies or to empower individuals to monitor their own health while keeping their data secure. The built-in sensors are insufficient for the range of data that a researcher might want to collect, so the Amulet is also equipped with a Bluetooth Low Energy (BLE) radio that can connect it to BLE-enabled sensors and enable it to upload data to a smartphone or server. Amulet is designed to have low-power demands, a correspondingly long battery life, and to be extensible by external developers [2]. The architecture and software are designed to make it easy for developers to write new event-based applications using the existing API calls, however, hardware is less easily modified. It is therefore very difficult to

quickly extend the sensing capabilities of the Amulet by adding new internal sensors. Luckily, the BLE radio extends the Amulet's data collection abilities without hardware modifications, as it can be connected to a variety of BLE-enabled sensors that produce different types and formats of data simply by making software changes.

The current approach for BLE connections in AmuletOS is hard wired and not extensible, tailored specifically to a Heart Rate Monitor that provides data as defined by the BLE standard. It works correctly and efficiently for receiving heart-rate measurements but cannot be easily modified to receive other data. To add a another type of sensor – one that produces data in a different format and with a different length – a developer would have to modify code on the two microcontrollers that constitute the Amulet's hardware. Much of the code implementing the BLE functionality on the radio would need to be rewritten, as it is specific to the Heart Rate Monitor functionality and formatting; however, simply substituting the formatting of the new sensor type would recreate the problem of non-extensibility in the radio code. The AmuletOS assumes that the only type of BLE sensor is a heart-rate monitor, and the current core and radio code require substantial restructuring to allow the Amulet to support two different types of sensors in the same firmware image. Amulet will be more useful if it is easy to extend the functionality of the device by integrating new BLE sensors, a given firmware image can contain the formatting and connection information for more than one type of sensor, and multiple sensors can be connected at the same time. Easy, reliable BLE communication is essential to the vision of Amulet as a useful, extensible platform for future research studies or for wide usage [2].

This thesis describes our redesign of the AmuletOS BLE architecture to support new and different external sensors and make it easier for future developers to integrate different types of BLE sensors with AmuletOS.

2 Background

The following section provides a brief overview of the features of the Amulet platform and the Bluetooth Low Energy protocol that are relevant to the integration of BLE sensors with the Amulet.

2.1 Amulet

The Amulet is implemented on two microcontrollers: an MSP430 – the application processor for primary computation – and an nRF51822 M0 microcontroller that contains and controls the radio [1]. For discussing BLE communications, we find it useful to think of the Amulet in three layers: the application layer, the core, and the radio as shown in Figure **??**. The applications and core AmuletOS both run on the application processor, while the radio controls run on the M0. Communication between the MSP430 and the M0 occurs over the **SPI!** (**SPI!**) bus.

Applications: Amulet applications are all event driven and expressed as a **FSM!** (**FSM!**) in the QM framework. A typical application transitions between a well-defined set of states in response to signals generated by the core upon user input (by button or capacitive touch slider), timer interrupts, or the arrival of data from a sensor. Each

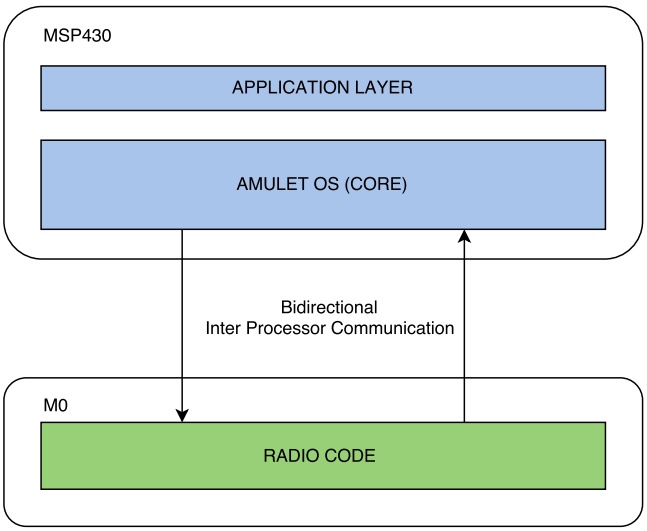

Figure 1: The high-level architecture of the Amulet

state may contain some logic or calls to AmuletOS functions written in a restricted version of C that does not allow the use of pointers and executed upon entry or exit. The resulting graphical representation of the state machine is translated into standard C by the compiler. An example blood-pressure application written in QM is shown in Figure **??**. The **FSM!** structure makes it easier to reason about the correctness of applications and, as long as the application is written correctly, prevents them from getting into unrecoverable states.

Applications only interact with the AmuletOS through the API functions defined in the Amulet.h header file. An XML file is used to explicitly specify the functions in Amulet.h that each application can use. Applications are assumed to be potentially malicious and are not given direct control of the radio or access to other energy expensive functionality to prevent them from running down the battery or monopolizing the radio. The Amulet compilation process includes static analysis to check that applications are only using permitted API calls and are not accessing memory outside of their alloted block, while the QM translator ensures that they have separate namespaces for all variables and states by appending the application ID to all function and variable names [**?**].

To get data from a sensor, applications "subscribe," that is, they ask to receive an event from the core whenever data arrives from that sensor. To actually receive the data an application must call an AmuletOS function to retrieve it, as the restriction on using pointers means that it cannot be included in the event.

Core: The core manages applications and sensors and handles all of the hardware and timer interrupts. It serves the same role as an operating system on a fully fledged computer – hence the name "AmuletOS". The core keeps track of application subscriptions to sensors in a queue and sends events to subscribed applications when data from

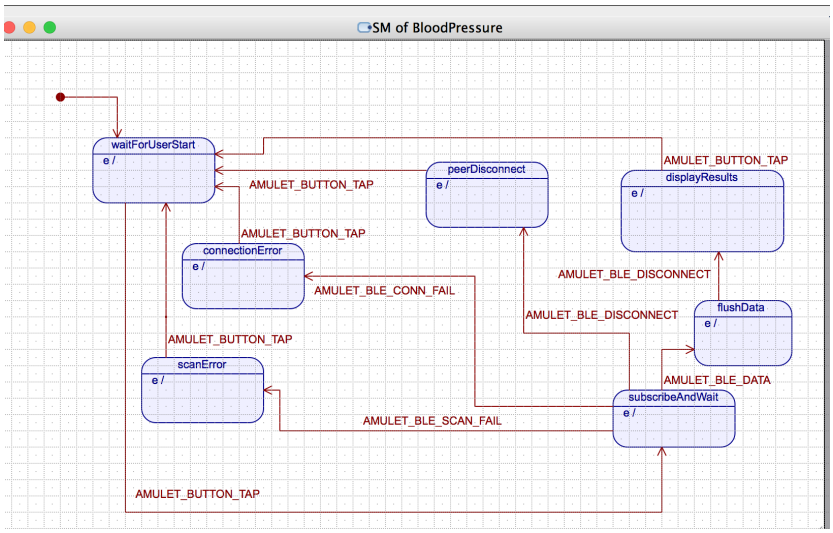

Figure 2: An example application to receive and display blood pressure on the Amulet

a sensor arrives or an application timer expires. It responds to requests by applications to retrieve data, display text on the screen, and write values to the SD card, and runs applications when events arrive for them or a button is pressed. For applications that get data from BLE sensors, the core serves as the mediator between the application and the radio. The core code is written in standard C without restrictions, but most of the code base only uses statically allocated memory for the sake of speed and to avoid memory leaks.

Radio: The BLE stack is implemented on the M0 microcontroller, which is responsible for all Bluetooth communication. Communications between the radio board and the application processor happen over Inter-Process Communication (IPC) on the **SPI!** bus, which serves as a physical link between the MSP430 and the M0. The IPC protocol is custom-defined for the AmuletOS, with a header format and field-values defined in the core code.

In the first implementation of the AmuletOS and radio drivers, the M0 was the entry point for all sensor data and application subscriptions were created in the core and then mirrored on the M0. All subscription data was sent over IPC to achieve this mirroring, but as sending data over the SPI bus is slow and energy-intensive the current iteration of the Amulet hardware and architecture only stores subscriptions in the core, on the MSP430. A recent hardware upgrade introduced a new version of the M0 to the Amulet, and the full IPC protocol and bidirectional communication between the core and M0 have not yet been re-established since that change, but sending unwrapped data from the M0 to the core over the SPI bus still works. In this newer version of the hardware the M0 does not receive all of the internal sensor data and is dedicated to the BLE stack and the radio controls.

2.2 Bluetooth Low Energy

Bluetooth Low Energy, also known as Bluetooth Smart or Bluetooth Version 4.0+, is a new version of the standard Bluetooth Protocol designed specifically for low-power devices [**?**].

BLE is a connection-oriented protocol; the connection established between two devices. Each device takes one of two distinct roles: A sensor is a *peripheral*. It is a data provider and can only sustain one BLE connection at a time. The Amulet plays the role of a *central*.[1] A BLE central can create and maintain connections to multiple peripherals at the same time [**?**].

To create a connection, a central (the Amulet) initiates a scan to discover peripherals (sensors) in the area, receiving advertising packets from any peripherals that are actively broadcasting *advertisements*. The advertisement format is decreed by the BLE standard **GAP!** (**GAP!**) and provides some basic information about the device and its functionality (see Figure **??**). Advertisements are limited to 31 bytes of advertising data [**?**]. If the initial advertising packet indicates that the peripheral will provide a scan response, the central can request more information about the peripheral without connecting. This scan response is in the same format as the advertising packet but is 255 bytes long [**?**].

After collecting advertisements the central selects a device to connect to, either by filtering using the information in the advertising packet or by allowing a user to choose the peripheral from a list. It sends a request to connect using the **BD!** (**BD!**) address of the device, which is a six-byte hex string that is similar to the MAC address of a networked device.[2] BLE supports several methods for exchanging a shared secret and performing security handshaking at the beginning of a connection, but for devices without a method of input or complex output (like most sensors) the BLE protocol specifies a "Just Works" method of authentication, which simply establishes a connection without a securely shared secret [**?**]. Once the connection is established, the peripheral stops sending advertising packets and maintains an exclusive connection with the central.

The BLE standard defines official Generic Attribute Profile (GATT) profiles and services for various types of data that a BLE-enabled device might provide. The GATT profile specifies the services that a device offers and thus the functionality that the device must provide; *characteristics* within the service, identified by *handles*, define the format in which the device delivers data [**?**]. For example a BLE-enabled blood pressure monitor would implement the BLE "Blood Pressure Profile," which encompasses the BLE "Blood Pressure Service." This guarantees that the device will send indications with the blood pressure reading in a certain format, allow reading and writing of characteristics, and may send additional notifications if the optional intermediate cuff pressure characteristic is enabled [**?**]. For most GATT servers, the client can either read values directly or request a message whenever a value changes by turning on *notifications* or *indications*. Blood-pressure monitors and heart-rate monitors both operate on the paradigm of the client enabling notifications (or indications) and then

[1] If it was sending data to a smartphone it would be the peripheral

[2] We assume all peripherals the Amulet connects to will have public **BD!** addresses that are unchanging for the lifetime of the device, or a static random address that is the same for the duration of the connection. The BLE Generic Access Profile also allows for random device addresses that change during a connection as a security feature [**?**].

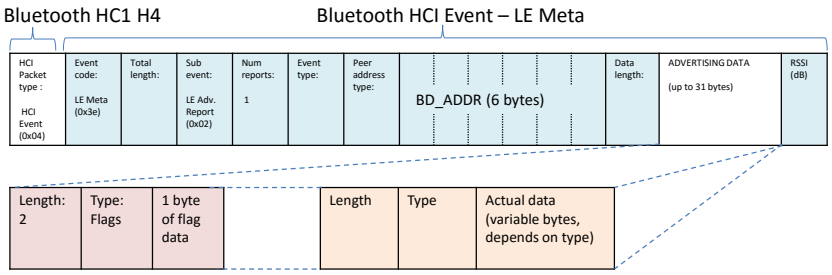

Figure 3: The format of a BLE advertising packet, derived from observation of traffic in Sudikoff with Wireshark

waiting for the server to notify (or indicate) the client with the new data. This behavior is specified in the GATT Service definition. Notifications and indications essentially perform the same function by allowing the sensor to push data to the client. The only difference is that the indication requires an explicit acknowledgement from the Bluetooth stack on the client and only one indication can be sent per connection interval because of this requirement of explicit acknowledgement [**?**]. BLE has a low energy impact because the central and peripheral agree on a connection interval – between 7.5ms and 4s – when they will exchange data, which allows them to sleep in between the transmissions [**?**]. Multiple notifications can be sent in the same connection interval (the nRF51822 controller, for example, can send six data packets per connection interval) and they require no work on the client side at the application level when receiving [**?**]. Notifications are acknowledged in the link layer, so there should not be any data loss [**?**].

3 Related work

The Amulet group has published several papers on the device to date. Sorber et. al. propose the Amullet and describe the goals of the project – security, privacy, and usuability – in a 2012 paper titled "An Amulet for Trustworth Wearable mHealth" [2]. They also state the goal of "allowing App developers to focus on the health-related logic and data processing while the Amulet system automates many security and networking tasks" [2], which informs the "easily extensible" goal of our current project. A paper written by Molina-Markham et. al. in 2014 – "Amulet: A secure architecture for mHealth applications for low-power wearable devices" – more explicitly describes the architecture and architectural security features of the device and expands on the goal of customisability and the ability to install third party applications [**?**]. This paper introduces the two-tiered architecture of the Amulet and describes the advantages of having applications as finite-state machines, and the mechanism of application isolation by limiting the features of C and static analysis at compile time [**?**]. Neither of these papers has addressed the Bluetooth communications of the Amulet in any depth.

In a 2016 paper titled "Beetle: Flexible Communication for Bluetooth Low Energy," Levy et. al. describe "Beetle," a "hardware interface that virtualizes peripherals at the application layer" and allows multiple applications on the central to receive data from the same peripheral, implements access control for operations and values on the peripheral, and enables peripheral to peripheral communication through a gateway [**?**]. The paper provides useful background information on BLE and the way existing operating systems present BLE to applications. They authors present example implementations on Android and Linux that insert a "Beetle" layer in between the BLE controls and existing applications to intercept the BLE packets and expose only the desired functionality to the application by presenting it with a virtual peripheral. The current implementation of AmuletOS and the radio, which hide the internals of the BLE connection to the application and allow two applications to receive data from the same sensor, is not entirely unlike this approach. Although the AmuletOS is an embedded operating system and the Beetle layer is designed for a full-fledged OS, the ideas about access control and sharing sensors presented in this paper are highly relevant to the problem of integrating BLE sensors with AmuletOS and suggest interesting directions for future development.

To explore BLE transmissions and structure we used a model UA-651 BLE Blood Pressure Monitor manufactured by A&D Medical [**?**]. A&D engineering provided an example Android application for connecting to their Blood Pressure Monitor in their Github repository [**?**] that we referenced for hints on how to connect to the device, and provided an example of the Android callback structure for handling BLE connections. The Linux utitility Gatttool, which is included in the bluez package, was also invaluable in discovering device characteristics and how the BLE protocol worked. The simple set of commands that it exposes to the user at the command line served as the inspiration for the collection of BLE commands that are elevated to the AmuletOS in the proposed implementation.

4 Current Implementation

The current version of Amulet is capable of BLE communication but the implementation is limited and inflexible: Amulet BLE only works with heart-rate monitors that implement the BLE-defined Heart Rate Profile and the radio code is inextricably bound to the sensor type and data formatting of the sensor.

Continuing to add sensors in the same way would have required re-writing code for the radio every time a new data format was introduced and making changes throughout the AmuletOS core code, requiring a developer to fully understand both parts of the system and reprogram both microcontrollers every time they wanted to add a new sensor. Furthermore, the current case design of the Amulet intentionally makes it difficult to access the programming port of the M0.

In the current implementation of BLE and the radio (Figure **??**), when an application wants to get data from a heart-rate monitor, it uses an API call to "subscribe" to the sensor. The core stores that subscription in a queue and turns on the radio if it is not already on. On the old hardware the core then bundled the subscription information in an IPC header and sent it to the radio (M0), which kept a mirrored copy of the subscriptions. The current version does not mirror subscriptions on the M0 and turning on the radio

immediately starts a scan for compatible sensors. The scan results are filtered on the M0 until a peripheral that implements the Bluetooth GATT Heart Rate Profile is discovered, at which point the radio attempts a connection. The radio automatically enables notifications from the GATT server on the peripheral by writing to a specific characteristic. When a notification arrives, the radio extracts the data and sends it to the core on the **SPI!** bus. With the old hardware, the radio wrapped it in the IPC header, but the full IPC protocol has not been reimplemented. The core receives the data, then steps through the subscription queue and send a "data ready" event to any application that had registered a subscription to the Heart Rate Monitor. The application then calls a `getHeartrateData` function to get the sensor data.

In the original model (with subscriptions mirrored on the M0) when an application unsubscribed from a sensor, the core simply marked its copy of the subscription as disabled in order to avoid sending an expensive IPC message. Subscriptions were actually deleted when they "expired," a functionality that has since been deprecated. The current implementation does not actually delete the subscription when an appliction unsubscribes, even though it is safe to do so. With the current applications the Amulet is not processing enough subscribe and unsubscribe events to be in any danger of overflowing the subscription queue, so this method is functional if not scaleable.

In the current implementation, the only control that the core has over the radio is the ability to turn it on or off, and turning on the radio immediately initiates a scan for heart-rate monitors. The only data that flows back from the radio to the AmuletOS is the heart-rate readings, sent over the SPI bus without headers or other metadata. The radio only connects to heart rate monitors that implement the GATT Heart Rate Service.

5 Proposed Model

Our proposed redesign of BLE integration with the AmuletOS is not a radical re-architecting of the flow of control and data through the system, but a careful restructruturing of the existing BLE functionality for better extensibility. The most significant change is that the sensor-type-specific code is removed from the M0 and AmuletOS is given finer control over the BLE radio and more error signals are added for better error handling in applications. The proposed flow of data and commands are shown in Figure **??**. The primary differences from Figure **??** are the changes in commands and information sent from the core to the M0 when initiating a connection, the generic radio code, the look-up of a handler to deal with received data, and the generic `AmuletGetData` function. The generic radio means that future developers do not need to do anything with the radio code and the formatting of the data and sensor-specific connection values can be defined in the AmuletOS. The proposed model also enables the AmuletOS to keep track of subscriptions to different instances of the same type of BLE sensor and allows an application to own subscriptions to multiple BLE sensors.

There are two main assumptions made in the conception and description of this model: 1) Inter-Processor Communication will be re-instated with a header that includes certain fields and flags, and 2) the radio code can be modified to perform the desired operations in a generic manner that works with most sensors. The first is an implementation and debugging task, but the second is less certain, as the manufacturers of BLE

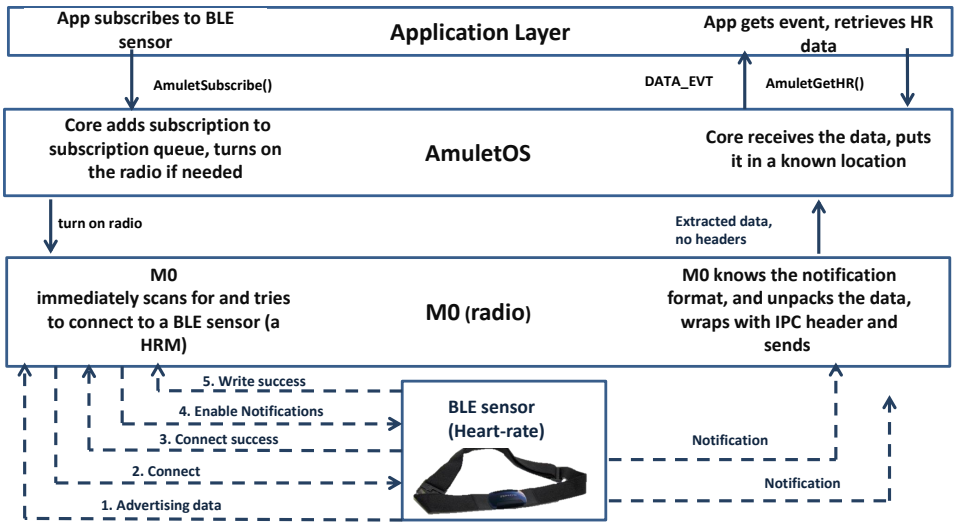

Figure 4: The current data flow, and interaction with the BLE heart rate monitor. Note that the core can only turn on the radio, and data is sent over IPC without headers. The function for retrieving data is heart-rate specific

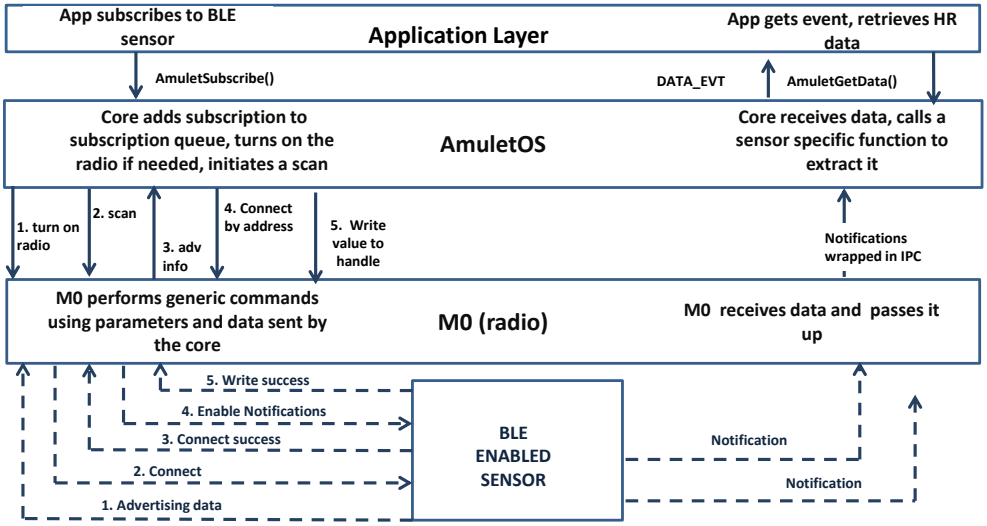

Figure 5: The new interaction with a BLE sensor, with the core playing a more active role. Note the increased communication between the AmuletOS and the radio, especially when establishing a connection.

sensors all seem to interpret the standard differently and the interoperability of a central and a peripheral is not guaranteed out of the box.

5.1 At the application level

The way an application gets data from a BLE-enabled sensor does not change dramatically in the proposed design. An application still subscribes, but specifies the type of BLE sensor that it is requesting data from as an argument. To accommodate multiple sensors, if the application subscribes to more than one BLE sensor it must also specify a unique integer identifier for each of them in the range of 0 to `MAX_SENSORS`, which is a constant value that must be less than 255. If it has only one subscription the identifier must default 0. This identifier is passed to the subscription function provided by the AmuletOS and is stored by the core, giving the application and the core a shared reference for that individual sensor.

Applications should assume that all of their requests to the AmuletOS succeed unless they explicitly receive a failure event.[3] Thus, after subscribing, an application should proceed directly to state of waiting for data. The proposed changes define several new events to inform the application of BLE scan or connection failures, as many connection or scanning problems – specifically, the desired sensor not being present, in range, or advertising – can be fixed by simple user intervention (turning the device on).

To retrieve data, the application calls a generic function, passing in sufficient information for the core to identify the subscription and an array for the core to fill with data. When applications are translated from a graphical finite-state machine in QM into C, arrays are replaced by structs that contain a pointer to values and the length of the array to prevent out-of-bounds errors and to get around the "no pointers" restriction [**?**]. The core will only return as much data as will fit in the array. Both the application and the core should know the length and type of the expected data from a given sensor, but the core determines how much data is actually returned and will not overflow the given array, preventing buffer overflow errors.

5.2 At the core level

The proposed changes expose the major actions of establishing a BLE connection and sending and receiving data to the core, allowing the AmuletOS to ask the radio to perform the essential functions needed to connect to a BLE sensor and request notifictions or indications from it. To enact the changes, the core stores more identifiers and associations for each connection than it does in the current implementation, as each application can have multiple sensors, and those "multiple sensors" could be separate subscriptions to different instances of the same type of sensor.

5.2.1 Subscriptions and connecting to a sensor

The logic of sensor subscription in the current model assumes that any application that subscribes to a heart-rate monitor will receive the readings from the single heart-rate

[3] In the AmuletOS, these events are defined in an enum. All of them are appended with _SIG

```
typedef struct SubscriptionQueue Subscription;

struct SubscriptionQueue {
    uint16_t subscriptionID; // internal to core
    uint8_t principal; // the app that is subscribed
    uint8_t service; // e.g. timer, BLE
    uint8_t sensorType; // added for BLE
    uint8_t appSensorID; // core and app share
    uint8_t bdAddr[BD_ADDR_BYTES]; // 6 bytes - core and radio share
    bool disabled;
    bool continuous; // default true
    uint8_t * data; // pointer to data for sensor
    Subscription * next;
};
```

Figure 6: The struct used to contain the subscription information in the AmuletOS

monitor to which the Amulet is connected, if a connection can be made. As Levy et. al. pointed out this implementation is convenient for allowing one application to display the heart rate and another to store it [**?**], for example, but does not fulfill the hypothetical case of using the Amulet to collect data from two accelerometers, one attached to each of a wearer's ankles, to measure gait. In the two-accelerometer example, having the subscription logic default to getting data from an existing sensor would result in data being gathered from only one accelerometer, defeating the point of the experiment. Thus the proposed implementation requires the application to specify a unique byte identifier for each sensor it wants to subscribe to and send that value to the AmuletOS when asking for a subscription. The AmuletOS stores it as `appSensorID` in the subscription struct, shown in Figure **??**, and it is used by AmuletOS and the application to ensure that the two layers are referring to the same sensor when exchanging requests and events. The subscription logic defaults to assuming that when an application asks for a subscription it wants a new, individual sensor. This is more like the implementation of BLE on existing operating systems like Android and Linux, which limit applications so that only one can receive data from a given BLE connection [**?**], but allows for multiple instances of the same sensor.

When an application subscribes to a sensor, the AmuletOS creates a new entry in the subscription queue, leaving the **BD!** address field zeroed out. It then asks the M0 to initiate a scan by sending an IPC message to the radio with any scanning parameters that are specific to the sensor type. The core only allows one scan to run at a time and sends a scan failure event to any application that attempts to request a scan while another scan is in progress. It is the application's responsibility to handle the event correctly and either retry or display a message to the user. The radio sends found advertisements back over IPC to the core, which passes them to the sensor-type-specific scan filter function that parses the advertisement and returns a boolean. When an advertisement from the desired type of device is found, the core adds that BD address to the subscription entry. We chose the **BD!** address as shared reference for the radio and the core to identify a connection to a sensor because it is (most likely) unique and can be used by the core to re-initiate a connection in the future, potentially without scanning again.

After finding a desirable advertisement, the core immediately sends the radio a request to connect by **BD!** address, followed by a request to write the necessary value to turn on notifications or indications. The core assumes that the commands it issues to the radio succeed unless it receives an error message from the M0. Assuming success instead of requiring explicit notification of success avoids unnecessary energy and time-intensive IPC communication.

5.2.2 Receiving data from a sensor

When the core receives notification or indication data from the radio over IPC, it passes it to a sensor-type-specific function to be unpacked. For example, for a blood-pressure sensor, the unpacking function extracts the data from the format specified by the GATT Blood Pressure Measurement Characteristic and writes it to a statically allocated buffer that serves as a section of FIFO memory for data. It returns a pointer to the first byte of the data, which the core places in the subscription that that sensor is associated with before notifying the subscribed application that data is ready for retrieval.

The core does not actually guarantee that the data will still be there when the application attempts to retrieve it: if the application is slow and the core has received another piece of data from the same sensor, the pointer will point to the first byte of the newest data. This is an intentional design choice so that an application only retrieves the most current value from a sensor. If the application is particularly slow in asking to retrieve the data and the core is receiving large quantities of data from other sensors, it is also possible that the data could be overwritten with data from some other sensor if the statically allocated buffer is not large enough. We judged this to be a reasonable risk to take, as the Amulet is meant to be a fast, responsive, real-time system. We chose to use the FIFO data queue instead of simply allocating a static data buffer for each type of sensor because we wanted to allow for the possibility of the AmuletOS receiving data from multiple sensors of the same type at almost the same time and be able to serve the subscribing applications correct data rather than a duplicated reading from one sensor. We chose a simple FIFO memory model with a risk of incorrect data instead of using dynamic memory allocation to always provide the correct data because it can be managed with a few lines of code, static allocation of memory at compile time is faster than dynamic execution during runtime, and there is no risk of memory leaks or need to free data before it gets overwritten.

When an application indicates it no longer wishes to receive data by unsubscribing from a sensor, the subscription entry can safely be deleted.

5.2.3 Defining sensor types

Another key component of the proposed changes is the definition of "sensor classes" in the core. C is not an object-oriented language but solving the problem of defining a sensor type with functions and variables would be trivial in an object oriented language with polymorphism. As it is, we think of adding the information for a new sensor as implementing an interface (just like Java): the developer must create new .c and .h files, define several constants, and write three functions, each of which must have a specific signature. Because we are working in C, however, all functions and variables must have

```
typedef bool (* filter_func) (ble_adv_data_t * scan_result);
typedef uint8_t * (* data_func) (ble_notification_t * data,
  uint8_t datalen);

typedef struct ble_sensor_info {
  uint8_t type;   //for core and application use
  uint8_t scan_seconds;  // parameter for scan
  filter_func scan_result_filter;
  uint16_t byte_data_len; // length of data to pass up to app, in bytes
  uint8_t handle_to_set;  // handle to write to turn on notifications/
    indications
  uint8_t value_to_write;  // notifications or indications
  data_func data_unpacker; // data handler
} ble_sensor_info_t;
```

Figure 7: The struct that contains the important information about a "sensor class", and the typedefs of the function pointers, which specify the required function signatures.

unique names. We suggest that future developers adhere to the convention of appending all of the function names with the internal sensor type identifier. Two of the functions are the previously-mentioned "unpacking" function and the scan-result filter. The third simply collects all of the constants and function pointers to the other two together in the struct shown in Figure **??**, which is our rudimentary equivalent of a (Java) class. The sensor-adder must call this third function from the `buildSensorTable` function that is run when the core first tries to reference the sensor table and constructs a table of sensor-entries that the core can easily reference and use to look up constants and function pointers.

A given sensor type can be included in a firmware image for the Amulet by editing the includes in the `ble_core_sensors.h` file.

5.3 In the radio code

In the proposed redesign, the radio is fully generic and should require no changes to add a new sensor. It should perform the following behaviors, if possible, when requested to do so by the core, using the parameters that the core sends.

- Scan for advertisements and send responses to the core. Send an error to the core if the scan times out without the core sending a request to connect.
- Connect by **BD!** address. Send an error to the core if it fails.
- Write a value to a characteristic handle.
- Read a value from a characteristic handle.
- Disconnect from a sensor.

A peripheral-initiated disconnect or the arrival of notification or indication data should also be recognized and sent to the core in the appropriate format and the radio

should handle the security handshaking and connection parameter negotiation without input from the core.

The write and read are not expected to report error values (yet) because the developer adding the sensor should have tested the handles, values, and permissions on another system for setting notifications and indications, and being able to read or write other values is not crucial for the correct functioning of a sensor in the style of a heart-rate or blood-pressure monitor.

The radio must also keep a table associating the initial **BD!** address with the internal connection identifier it uses, so that it can include the **BD!** address in any related IPC message that it sends to the core after establishing a connection, as the notification or indication data does not usually include the **BD!** address of the source.

5.4 Continuing the Amulet's goals of energy and security

The proposed implementation will likely consume more power than the current one, as it requires more energy-intensive IPC communications. We believe that the increased utility of the Amulet will make it a worthwhile trade-off, but it would be useful to quantify the decrease in battery life to evaluate how useful shortenting the scan interval or implementing scan filtering on the radio would be. In the proposed design the AmuletOS uses as few IPC transmissions as possible by assuming that all commands sent to the M0 receive unless it is specifically told otherwise.

The proposed changes to the AmuletOS APIs available to applications and to the subscription process maintain the security tenet of restricting applications' access to important or potentially harmful operations [**?**]. The subscription logic passes a failure back up to the application if the scan fails instead of retrying endlessly, and further logic could be implemented to prevent an application from retrying too frequently and monopolizing resources. The core's position in between the applications and the radio, and by extension the connected sensors, allows it to control the applications' access to data and services from the sensors. The proposed changes do not currently allow an application to write a value to a handle, or to read one. This limitation of functionality provides security. The AmuletOS could potentially implement more rules for access control on a per sensor-type or per application basis, moving towards the fine grained access control described in the paper on the "Beetle" system [**?**].

6 Limitations

The main limitation of the proposed changes is that the generic BLE radio has not yet been written and IPC has not been reimplemented on the new hardware. We developed proof-of-concept code for the proposed changes and confirmed that it works down to the IPC level: it formats messages and queues them up to be sent but the functionality to send them has not been restored. We tested the changes in receiving data, which start in the `ProcessIPC` function by inserting fake data, but we cannot be certain the design works until the radio implementation can be tested.

The memory available on the Amulet limits the number of sensor type definitions that can be included in a given firmware image. The current prototype has 128K FRAM

(which can be used for both ROM and RAM), and the proof-of-concept code takes about . The speed of the **SPI!** bus might also limit the amount of data that that the Amulet could receive and the number of commands it could issue, which would in turn limit the number of sensors that the Amulet could be connected to at any one time. There does not appear to be a specific limit set by the BLE standard on the number of peripherals that can be connected to a central. The limit is probably dependent on the specific implementation of the BLE stack. The current heart-rate receiving central on the radio can connect with three separate heart-rate sensors and receive data from all of them.

The AmuletOS does not guarantee that the the application will get the correct data: there is a chance that if the application takes too long to retrieve the data or the AmuletOS is overwhelmed with an influx of data, the application may get an incorrect reading when it attempts to retrieve data

The subscription process does not currently allow for the case where an application does not need to connect to a new sensor but would be satisfied by collecting data from an already connected sensor of the appropriate type. The value of allowing two different applications to receive data from the same sensor is emphasized in the paper on "Beetle," in which the authors give the example of having one application display heart rate and another log it [**?**].

The current set of values that a developer needs to specify to add a sensor are based on assumptions about the way the sensor provides data and limit reusability of the sensor definitions. The proposed changes require the developer to know the value of the handle to write to enable notifications or indications. The handle can be found by reading the **CCCD!** (**CCCD!**) on the device. Hardcoding the handle value works for connecting to peripherals of the same model from the same manufacturer, but means that a sensor from a different manufacturer or a different model that implements the same GATT server might need a different specification file to connect to it, even though the function for unpacking data might be the same.

The "sensor class" definition of the proposed changes only supports either notifications or indications for a sensor, not both. For example, a device implementing the GATT Blood Pressure profile may provide optional notifications with intermediate cuff pressure and must provide indications with the final reading. In the proposed design (as in the original BLE structure), the Amulet can only receive the notifications or the indications, even if the peripheral provides both. It would be simple to add another two fields to the sensor definition entry so that a sensor could send the Amulet both notifications and indications, but allowing a variable upper bound on the number of notifications and indications would be trickier.

The proposed changes, and the proof-of-concept code, send a "data is here" event to every subscribed application every time a notification or indication is received from the radio. For some sensors, however, it might be more appropriate for the AmuletOS to average the data and only notify the application every nth time data was received. This could be achieved by pushing the responsibility for notifiying the application into the data handler function specified in the sensor type definition, which would mean that sensors could have completely different patterns for sending data events to applications depending on their type.

7 Evaluation

How do we measure the success of these proposed changes? The first question is whether the new code works to get data from a BLE-enabled sensor to the Amulet, and whether it does so reliably? In other words, does it provide basic functionality by scanning, filtering, connecting, enabling notifications or indications, receiving the data, and then disconnecting without entering an unrecoverable state?

We unit tested functions and data structures during development to check for correct atomic behavior, but full integration tests with an actual BLE sensor are not yet possible, as the reinstatement of IPC and implementation of generic central functionality on the M0 has not been completed. As a substitute, we tested the flow of errors and data through the AmuletOS up to the applications by manufacturing hand-written IPC messages on the MSP430 in the `onIdle` loop, which runs whenever the AmuletOS is not handling interupts. We passed the test data into the `ProcessIPC` function, which is the expected entry point for an IPC message into the logic of the core, and successfully showed that with correctly formatted data and IPC headers the example blood-pressure application and the error handling works as it is supposed to: the errors and data are passed up to the correct application.

Future full integration testing should:

- test repeated connecting, disconnecting, and reconnecting to the same device
- test connecting to two different sensors, with two different applications receiving data (i.e. have a heart-rate sensor connected and collecting data, then connect to and collect data from a blood-pressure sensor as well).
- test one application subscribing to two sensors, which would require writing a new application.
- test error behaviors by scanning in an area without a device of the desired type.

Performance testing to evaluate the following would also be valuable:

- test sensors that send at a high data rate with a short connection interval to see how fast the system responds and measure throughput.
- measure latency from sensor to application. We expect the IPC transmissions over the SPI bus and the actual BLE transmission to be the slow portions of the system
- measure impact of exposing BLE commands to the core on battery life by comparing the current implementation (sensor specific) with the proposed implementation (generic)
- experiment with the effects of constants and the sizes of arrays and queues on performance, and fine tune the values and queue and array sizes.

The more difficult thing to measure is how "easy" it is for a developer to add a new BLE sensor. In the proposed model, the developer only has to make changes in the

AmuletOS rather than in the radio code, which we consider significantly easier: the radio deals with the specific parameters of the BLE connection and is tricky to program, while code on the AmuletOS is at a higher level of abstraction and not as finicky. To add a new type of sensor with a different data format in the proposed model, a developer must create a new pair of .c and .h files, write three functions that conform to pre-defined signatures, define four constants, and edit an existing pair of .c and .h files to inform the AmuletOS of the existence of the sensor by calling one of the functions that they have written and including their new files.

They must know whether their sensor provides notifications or indications, which characteristic handle to write to to turn those on, and the format of the data that is delivered in a notification or indication. Of the three functions to write, one simply collects all of the values and other functions and packages them into a struct. The second handles notification or indication data, and in most cases will simply be responsible for extracting data and copying it into the FIFO memory buffer. The third is likely to be the most complicated, as it must parse advertisement packets to determine whether the peripheral that sent the advertisement is of the desired type. Advertisments are in a standard format that makes parsing fairly simple (see Figure **??**), so even this should not be too difficult, especially if the developer has been able to observe the byte composition of an advertisement from their device by using a sniffing tool such as Wireshark.

One way to determine the ease of adding a new sensor would be getting a developer to actually add a new sensor, evaluating how long it took them, and interviewing them about their experience of the process. It would be interesting to do a study with at least three to five new developers, and have them all work on adding a sensor that had already been successfully added and tested. The developers would be provided with a copy of the AmuletOS (without the target sensor added), and documentation explaining how to do the task. They would be set free to attempt it for a set amount of time, then evaluated on how close they got to completeing the addition and interviewed afterwards on their experience. In selecting the potential developers for such an evaluation, we would need to consider the target writer of Amulet applications and who is likely desiring to add sensors. At a minimum, all subjects would need to be proficient in C. A simpler approach would be to have someone new to the Amulet group add a sensor that has not yet been added and tested and perform the same interview process with them.

8 Future Improvements and Projects

One related future project would be the implementation of simplistic pairing between the Amulet and a sensor. The Nordic Soft Device that provides much of the BLE radio functionality has its own form of pairing, but pairing implemented in the AmuletOS would be easier to use. It would be convenient to have the Amulet remember the **BD!** address of the device that it has previously connected to, to ensure that the Amulet reaches the heart-rate monitor or blood-pressure sensor its user actually wearing. BLE devices only advertise when they are not connected, so the chances of inadvertently connecting to another person's device are slim in real-world use, but as the Amulet is more likely to be used in research studies and may have multiple people connecting and reconnecting to sensors in the same room, having a pairing mechanism would be

useful. A primitive form of pairing could be implemented by editing the subscribing and unsubscribing logic. Instead of removing subscription entries when either the application or the peripheral initiates a disconnection, the AmuletOS could simply mark the entry as disabled, storing the pairing of **BD!** address, application, and sensor type. The subscription function signature could be changed to include a "try to reconnect" flag. When the application asked for a subscription again, if the core found a disabled subscription entry for that application with the appropriate sensor type it could simply try to connect to the stored **BD!** address directly, skipping the energy intensive scanning step and the IPC transfers required to send scan results to the core. If it did not find a stored subscription, it could create a new one and follow the usual scanning procedure. As mentioned in the limitations, the subscription functionality could also be improved by adding an option for an application to receive data from an already connected sensor, if available, rather than looking for a new individual sensor every time.

Another related project would be the development of a pairing app that would allow a user to select the device to connect to from a list returned by the scan, rather than letting the AmuletOS connect to the first sensor that matches the scan filter requirements. This is challenging because of the limited display space and might require as-yet-unimplemented inter-app communication.

The AmuletOS iterates through the subscription queue frequently to look for subscriptions that match certain parameters. As the Amulet is loaded with more applications and connected to more sensors, it might be worth replacing the queue with another data structure that allows for faster access.

The new design filters scan results in the core, which requires all advertising packets the radio receives to be sent to AmuletOS over IPC. An energy efficient modification would send some filter parameters to the radio with the scan instruction so that the M0 could do some initial filtering or exclude certain types from the scan. Not only would this save on IPC communications, but the MSP430 could be put in a low power idle state while the radio processed the scan data, only waking up when the results were sent.

Another useful feature to implement would be dynamic discovery of the handle to write to turn on notifications or indications. We would also like to implement automatic inclusion the the sensor type source code in a firmware image depending on the dependencies of the application. We could also implement more access control on a per sensor type basis, or limit the access of certain applications to some APIs [**?**]

9 Summary

This thesis describes the proposed redesign of the AmuletOS to integrate multiple types of Bluetooth Low Energy Sensors with the AmuletOS. The key changes proposed are rewriting the radio to be more generic and permitting the AmuletOS finer control of the BLE states. The proposed redesign tracks which application is connected to which sensor and ensures that the application gets data from the correct sensor even if there are multiple sensors of the same type connected and sending data simultaneously.

We discuss the limitations of the proposed changes, outline a plan for more in-depth evaluation once the generic radio is implemented, and describe some ideas for related future work.

Acknowledgements

Thank you to Professor David Kotz for advising my thesis, and to Professor Zhou and Professor Campbell for serving on the committee for my presentation and providing valuable feedback.

Thanks to Tianlong Yun, Ron Peterson and Josiah Hester for answering lots of my questions, and to everyone involved in the Amulet project for being very welcoming and helpful.

This research results from a research program at the Institute for Security, Technology, and Society at Dartmouth College, supported by the National Science Foundation under award number CNS-1329686. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the sponsors.

References

[1] Andres Molina-Markham, Ronald A. Peterson, Joseph Skinner, Ryan J. Halter, Jacob Sorber, and David Kotz. Poster: Enabling computational jewelry for mhealth applications. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 374–375, New York, NY, USA, 2014. ACM.

[2] Jacob Sorber, Minho Shin, Ronald Peterson, Cory Cornelius, Shrirang Mare, Aarathi Prasad, Zachary Marois, Emma Smithayer, and David Kotz. An Amulet for trustworthy wearable mHealth. In *Workshop on Mobile Computing Systems and Applications (HotMobile)*, pages 7:1–7:6, February 2012.