University of Zurich[UZH]

SOFTWARE PROJECT – Communication Systems Group, Prof. Dr. Burkhard Stiller

# modum.io Software on the CC2650 SensorTag

## Portability, Security and General Architecture

*Andreas Knecht*
*Zurich, Switzerland*
*Student ID: 11-916-152*

Supervisor: Thomas Bocek
Date of Submission: February 1, 2017

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Software Project
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

# Chapter 1

# Introduction

The idea behind modum.io [12] is to combine blockchain and IoT technology to improve the supply chain processes in the distribution of medicinal products. One application is the recording of temperature measurements during shipments in order to discover temperature deviations that may be harmful to the medicine, as mandated by GDP regulations. The current hardware platform capturing those measurements is a prototype board by Texas Instruments (TI). It has shortcomings such as high cost, limited security and inadequate water and dust resistance. Thus, it will be replaced in the near future with a custom developed hardware platform running new software. This document describes the current hardware's software structure, security requirements for the new system and portability considerations to port the current sensor software to the new hardware platform.

# Chapter 2

# Sensor Software Architecture and Project Structure

The software is split in two projects: the SensorTag (App) project and the SensorTagStack (Stack) project. The flash memory where the program bytecode resides, as well as RAM are divided into two parts and each project will only use its assigned part of the memory. The division is managed by a tool in the TI development toolchain called Boundary Tool. This is described in detail in the Software Developer Guide [1]. The stack project implements lower layers of the firmware and Bluetooth stack and doesn't have to be changed for most use cases of the SensorTag. Thus, a reason for the split may be to achieve better compilation times by splitting off parts of the firmware that don't normally have to be changed often. The sensor software is based on TI-RTOS, a real time operating system with pre-emptive task scheduling.

## 2.1 General Project Layout

The modum.io [12] projects app & stack are based on the default TI demo projects (app & stack) for the SensorTag of the SimpleLink BLE stack version 2.1.1.44627. These projects make use of all the sensors on the SensorTag board. Unfortunately, these projects are set up to not include any code files (*.c nor *.h files) locally, but to virtually link to all the required files located in the TI-RTOS install location. The virtual folder structure of the project differs considerably from the folder structure in the TI-RTOS install location where the files are actually located. Thus, it is not a viable option to copy an entire directory tree from the TI-RTOS install location to the local project directory. Instead, the modum.io projects only replace those virtual folders and their contents with local copies where files need to be modified and therefore be put under Git version control. Additionally, some included files that can be found via the include search path are completely missing from the projects' virtual file trees.

The development environment (Figure 2.1) is based on Eclipse [14]. Figures 2.2 and

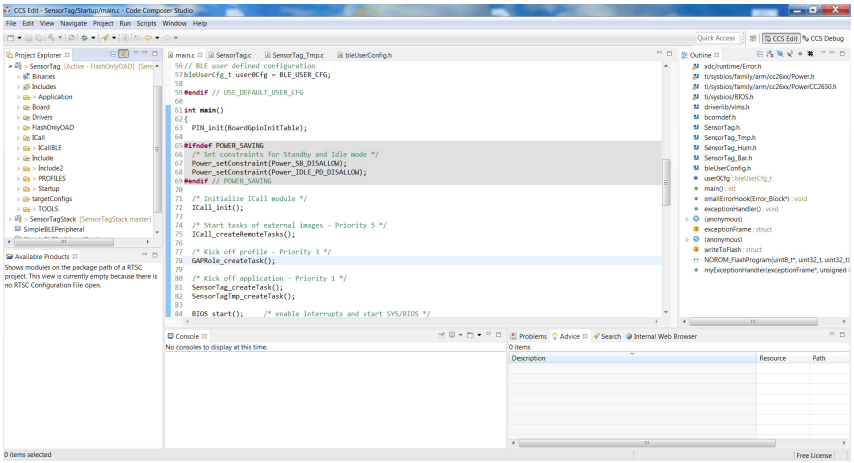

Figure 2.1: TI Development Environment

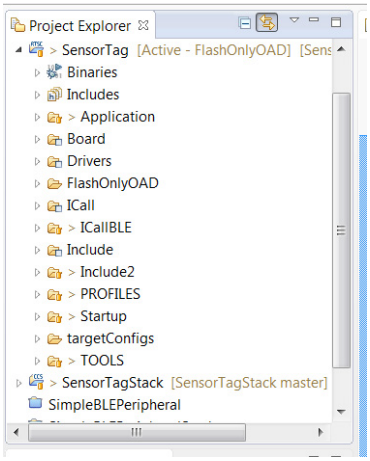


Figure 2.2: App Project Folder Structure

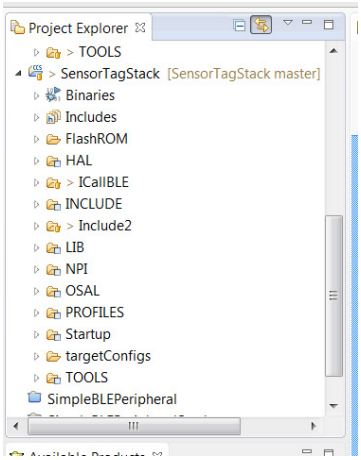


Figure 2.3: Stack Project Folder Structure

2.3 show the app and stack projects respectively. All folders with a white square in the lower right corner are virtual project folders.

This is a description of only those folders and files of the app & stack project that are relevant for the modum.io software customizations.

### 2.1.1 App Project Folder Structure

**Application** Contains the "application" code for all the sensors. Some of these sensor applications run on their own task in the RTOS, others run on the main task. "SensorTag.c" and "SensorTag.h" correspond with the main task in the RTOS. "SensorTag_Tmp.c" and "SensorTag_Tmp.h" correspond with the temperature task in the RTOS. All the files in here that don't correspond to the main task nor temperature sensor were aggressively cropped to only retain code that disables the respective sensor on sensor boot. "SensorTag.c" also contains lots of initialization code of the Bluetooth API, such as setting the advertising data packets, advertising interval, transmission power, timing constraints and other Bluetooth related parameters, as well as launching the Bluetooth functions in the stack.

**ICallBLE** Contains code to call functions in the stack project and retrieve the result. This was extended by functions to write, read and erase flash memory, which, at the time of writing these functions, was implemented in the stack project. Some implementations of flash memory access exists in the app project, but were never tested for suitability of storing measurements after using the implementation in the stack project proved successful. The SimpleLink BLE stack v2.2.1.18 has additional implementations of flash memory access in the app project that were tested suitable and effective, thus requiring no modification of the message passing functionality between the app and stack project.

**Include2** Is an additional include directory containing a local copy of the file "hci_ext.h" which is included by code in the "ICallBLE" directory, but is not present in the project's virtual folder "Include". It contains constants for message types to be passed between the app and stack project. Additional constants were added to represent the flash memory operations. Instead of adding local copies of all files in "Include", the file was copied to the new directory "Include2", which is added to the project settings include search path at the top to make sure that this local copy of the file is used instead of the one in the TI-RTOS install location. Additionally, this directory contains the file "hiddev.h" required by the battery level profile, which is not findable in the include search path of the original SensorTag project.

**PROFILES** Contains the Bluetooth service and characteristic definitions (profiles) that are exposed to the Bluetooth API, as well as code to handle reads and writes to those characteristics. For the temperature service the main file is "irtempservice.c". Each characteristic definition, as well as each characteristic's read and write callbacks are split off into a separate header file for each characteristic. The profile definition contains the characteristic UUID, characteristic access permissions and, optionally, the memory location where the characteristic's value is stored. This value is not

used by the Bluetooth API for anything other than returning this pointer in the characteristic's read and write callback. Since the characteristic that is being read or written is known inside the callback, it is generally clearer to use the characteristic's value field instead of the pointer passed into the callback function via argument. The "irtempservice.c" file also contains functions that are exposed to the temperature application task so that it can interact with the profile. All files in here that were part of the default SensorTag project, but not part of the temperature service or compulsory services as defined by the Bluetooth standard [1, 2], were completely removed. A battery service was added.

**Startup** Contains the "main.c" file which includes the startup routine of the sensor software. The barometer and humidity sensors are the other two sensors besides temperature that are implemented to run on their own task in the RTOS. The start of those tasks was removed from the startup routine because these sensors are not used in the modum.io system. In addition, an exception handler that turns on the red LED and writes some debug information to flash memory was added to this file.

**TOOLS** This folder and its contents were replaced with local copies in order to locally modify the file "appBle.cfg" in order to point the default exception handler to the custom one implemented in "main.c". This has a side-effect that the files "ccsLinkerDefines.cmd" and "ccsCompilerDefines.bcfg" no longer point to the copy in the TI-RTOS install location, and thus, modifications of those files by the Boundary Tool in the stack project need to be manually applied to the local copies of these files in the app project. The file "cc26xx_ble_app_oad.cmd" contains instructions for the linker on where in flash memory and RAM to place parts of the app, based on calculations from the boundary memory address between the app and stack as derived by the Boundary Tool. A mistake in this file results in the app project using one flash page less than it theoretically could, see [13], which is utilized by the modum.io software to store measurements in the unused flash page.

### 2.1.2 Stack Project Folder Structure

**ICallBLE** Contains code to receive the function call messages from the app project and post the result. This was extended by functions to write, read and erase flash memory as detailed in the section of the same name for the app project. The file "MeasurementStore.h" was added to implement the storage of the measurements.

**Include2** Is an additional include directory containing a local copy of the file "hci_ext.h" as detailed in the section of the same name for the app project.

**TOOLS** Is left as a virtual folder, contains the files "ccsLinkerDefines.cmd" and "ccsCompilerDefines.bcfg", which are automatically modified by the Boundary Tool during the build of the stack project to contain defines that represent the memory addresses of the boundary between the app and stack software in flash memory, as well as RAM.

## 2.2 Tasks

The app project runs on two tasks: the main and the temperature task. The stack projects consists of (multiple) separate tasks. ICall is a message passing interface that is used to enable synchronization and message passing between those tasks. ICall is explained in detail in the TI Software Developer Guide [1].

### 2.2.1 Tasks With Respect to Bluetooth Operations

The Bluetooth profiles' read and write callbacks are not executed on either the main nor temperature task. Instead, callbacks for Bluetooth profile reads and writes are registered with the GATT server application, which is implemented in the stack, and these callbacks are called from the GATT server task managed by the stack. This task runs with a high priority because Bluetooth radio communication has high time constraints for responding to packets. This makes it important that the callbacks of the Bluetooth profiles return quickly, lest the Bluetooth connection be terminated. Thus, any expensive operations, such as reading measurements from flash need to be executed on one of the lower priority tasks of the app project. To achieve this, ICall is used to post a message to the main task, which will then read the measurements when it pops the respective message of the message stack.

### 2.2.2 App Main Task

The main task wakes to pop messages off of the ICall message stack. This includes the request by the measurement Bluetooth characteristic to read measurements from flash memory. Everytime a characteristic is changed, a message is put on the stack as well, so that the main task can perform appropriate actions, such as disabling the temperature task when the recording characteristic has been set to disable. In addition, the main task is programmed to wake every second and blink the green LED when the sensor is advertising and the red LED when it is recording.

### 2.2.3 App Temperature Task

The temperature task wakes up periodically (the period is configurable via the period characteristic of the sensor API exposed to the client) and acquires and stores one measurement in flash memory. When the sensor recording is disabled, this task is disabled, thus, stopping the periodic recording of measurements.

# Chapter 3

# Security

Security implementation is lacking in the current hardware platform's software. Requirements for the new hardware platform and its software are gathered and implementation approaches discussed.

## 3.1 Requirements

Two security requirements have been defined for the sensor. First, measurements have to be signed on the sensor with a private key whose corresponding public key is signed by modum.io. Each sensor has a different private key. The private key should be very hard to retrieve by an attacker, including potentially using a special crypto chip for key stoage in the future. The algorithm to be used for this is the Elliptic Curve Digital Signature algorithm (ECDSA) [4, 7].

Secondly, there needs to be role management on the sensor, distinguishing anybody, authorized users and admins. Only authorized users shall be able to read and write most characteristics of the sensor, such as the enable characteristic that enables and disables the recording. Only admins shall be able to write calibration coefficients to the sensor.

Tables 3.1 and 3.2 contains a risk analysis of modum.io's system with respect to the sensor. The probability and impact are rated on a scale from 1 to 10. Thus, risk is assessed on a scale of 1 to 100 ($10 \times 10$). All risks with solution category "Software" are covered by above requirements. "Physical" risk mitigation solutions entail packing the sensor in a tamper-resistant case. "Process" risk mitigation solutions include regular recalls of sensors by modum for physical checking of sensor integrity, training of users and more.

### 3.1.1 Need for Encryption

In order to enable role management, connections with the sensor need to be encrypted (lest every single read or write request would need to be signed by the user). After an

| Description | Probability | Impact | Risk (Probability × Impact) | Solution Approach | Notes |
|---|---|---|---|---|---|
| Flashing of firmware | | | | | |
| Malicious via USB | 6 | 7 | 42 | Physical | |
| Malicious over-the-air | 7 | 7 | 49 | Software | |
| Accidental | 2 | 4 | 8 | Process | |
| Physical tampering with sensor board | | | | | |
| Destroy memory | 7 | 2 | 14 | Physical | |
| Replace temperature sensor | 1 | 6 | 6 | Physical | |
| Destroy sensor | 7 | 2 | 14 | Physical | |
| Separate sensor from shipment (accidentally) | | | | | |
| Forget packing sensor | 10 | 1 | 10 | Process | |
| Pack sensor separate from parcel in different temperature environment | 8 | 1 | 8 | Process | |
| Swap sensors | 10 | 3 | 30 | Software | |
| Lose sensors | 10 | 1 | 10 | Process | |
| Separate sensor from shipment (maliciously) | | | | | Requires cooperation of sender & receiver |
| Pack sensor separate from parcel in different temperature environment | 7 | 6 | 42 | Process | |
| Sender doesn't pack sensors, but reads them out locally | 3 | 5 | 15 | Software | |
| Swap sensors between shipments | 2 | 3 | 6 | Software | |
| Unauthorized readout of measurements (USB) | 6 | 1 | 6 | Physical | |
| Unauthorized readout of other characteristic (USB) | 2 | 1 | 2 | Physical | |
| Readout of sensor private key | 9 | 8 | 72 | Physical | |
| Side channel attacks against sensor crypto | 1 | 1 | 1 | Out of scope | |
| Unauthorized device poses as sensor | 2 | 4 | 8 | Software | |
| Clone MAC address of authentic sensor | 2 | 4 | 8 | Software | |

Table 3.1: Risk Analysis

| Description | Probability | Impact | Risk (Probability × Impact) | Solution Approach | Notes |
|---|---|---|---|---|---|
| Manipulation of temperature data | | | | | |
| Inserting / deleting / reading out temperature measurements | 3 | 5 | 15 | Physical | |
| Reading / writing of calibration coefficients or other important parameters | 2 | 4 | 8 | Physical | |
| Battery empty | 10 | 3 | 30 | Software & Process | |
| Unauthorized removal of battery | 7 | 2 | 14 | Physical | |
| Common damage to sensor (water, extreme temperatures, wear and tear) | 9 | 3 | 27 | Physical | |
| Theft of sensor | 2 | 9 | 18 | Process | |
| Unauthentic sensors disguised as modum devices | 2 | 9 | 18 | Software | |
| Disposal of sensors | 5 | 2 | 10 | Process | |
| Replay attack toward sensor | 1 | 2 | 2 | Software | |
| Replay attack toward app | 6 | 7 | 42 | Software | Include measurement timestamp in signed data |
| Send data to app that don't stem from valid sensor | 6 | 7 | 42 | Software | Only accept data signed by sensor |
| Bluetooth eavesdropping | 5 | 3 | 15 | BT 4.2 | |
| Breaking Bluetooth encryption (v4.1) | 5 | 3 | 15 | BT 4.2 | |
| Reading of measurements OTA | 10 | 2 | 20 | Software | |
| Reading/writing of important characteristics, such as calibration coefficients, OTA | 10 | 4 | 40 | Software | |
| Bluetooth jamming | 8 | 2 | 16 | | |

Table 3.2: Risk Analysis (continued)

encrypted connection has been established, the user authenticates to the sensor, after which sensor operations are authorized according to the user's role until the connection is terminated.

## 3.2 Evaluation of the Options Provided by Bluetooth LE

Bluetooth LE already offers encryption and authentication. Below is a discussion of its security and applicability for this project.

### 3.2.1 Authentication

Authentication of Bluetooth LE connections can use multiple modes, as defined in the Bluetooth standard [2]. Since the sensor doesn't have a display nor keypad, however, the only readily available mode is JUSTWORKS, which is basically an anonymous encrypted channel without authentication. Thus, authentication cannot be achieved with Bluetooth's own methods. Since the sensor has a unique private key, however, as well as the modum.io public key, authentication can be implemented via checking of signatures.

The sensor can authenticate to the user by signing something with it's private key, which is already stored on the sensor. Modum can authenticate a user by something like signing a challenge generated by the sensor together with the intended role for the user with the modum.io private key.

Authentication on a previously established anonymous encrypted channel requires a simultaneous check that the users are really talking on the same channel and not two separate channels with a man-in-the-middle attacker between.

This can be achieved by including the session key used for channel encryption in the authentication procedure, such as signing (a hash of) the session key and transmitting the signature to the other party, thus allowing the other party to simultaneously verify user authenticity (by verifying the signature) and the absence of a man-in-the-middle (by verifying that the signature is actually a signature of the session key of the encrypted channel).

### 3.2.2 Encryption

Bluetooth Low Energy up to version 4.1 uses a very weak key-exchange protocol that allows an attacker to easily derive the session key if he is able to eavesdrop the initial establishment of the encrypted connection. If he missed to eavesdrop the connection establishment, he can inject a packet to force a key renegotiation, allowing him to eavesdrop this new key exchange [3].

Since the Bluetooth LE encryption up to version 4.1 allows an attacker to easily eavesdrop and mount man-in-the-middle attacks on the encrypted channel, only Bluetooth LE version 4.2 can be used for this use case. Bluetooth LE 4.2 uses a state-of-the-art key exchange using the Elliptic Curve Diffie–Hellman algorithm (ECDH) [4, 5]. To encrypt connections Bluetooth LE uses the state-of-the-art Advanced Encryption Standard (AES) [11] since multiple versions.

## 3.3 Implementation in modum.io's System

Since the session key seems impossible to retrieve via the Android Bluetooth API, two alternative approaches need to be considered [8, 9, 10]. One option is for the Android client to generate a random key and encrypt it with the sensor's public key using the Elliptic Curve Integrated Encryption Scheme algorithm (ECEIS) [4, 6], transmit it to the sensor, after which the sensor and client establish a Bluetooth LE encrypted connection using the key generated by the client as an out-of-band secret for Bluetooth's out-of-band authentication mode. After that, authentication of the user to the sensor still needs to be performed, but a shared secret is available to ensure that there is no man-in-the-middle.

The other option is for the client to generate a random ECDH [4, 5] keypair and writing the public key to the sensor, as well as reading the sensor's public key. Then they can establish a shared secret using ECDH, which they can then use as an out-of-band secret for Bluetooth's out-of-band authentication mode. After that, again, authentication of the user to the sensor still needs to be performed, but a shared secret is available to ensure that there is no man-in-the-middle.

Both approaches have the drawback that the sensor needs to have an implementation of an additional elliptic curve cryptography algorithm (either ECIES or ECDH) [4, 6, 5], in addition to ECDSA [7], wich possibly results in higher crypto chip costs or larger bytecode size if the crypto is done in software.

# Chapter 4

# Porting Outlook & Considerations

The new sensor will likely have a much simpler operating system that doesn't use preemptive task scheduling, in order to save power and extend battery life. This is a factor making the current sensor software architecture potentially different to the new sensor software architecture.

The read and write handlers of all the characteristics are well structured. This could be beneficial for portability, although it has to be noted that the reading and writing of most characteristics is not complex part of the current sensor software, except for those involving measurements, as well as the the work done on the temperature task in order to handle the measurements. Each separately, all sensor functions aren't complex, but they affect widely different tasks, parts of the operating system and stages of execution, making an estimation of ease of portability to a different operating system with a different task scheduling approach difficult.

The flash memory drivers and memory layout will almost certainly be different on the new sensor, requiring a new analysis of the best approach to store measurements.

The driver of the temperature sensor and flash memory will be different on the new sensor, however it can be expected that temperature sensor should expose a simple interface (just reading temperatures) and, thus, it should not be difficult to port the code acquiring measurements. The lack of other sensors on the new board will make many sensor interactions, such as disabling unused sensors, unneccessary.

The Bluetooth stack of the new sensor will need to be fully Bluetooth 4.2 compliant, including the Secure Connections feature, so that ECDH [4, 5] is used for the key-exchange instead of the weak algorithm analyzed in [3].

The requirement of over-the-air flashing of new firmware is something that is not implemented in the current firmware and will have to be developed from scratch, as well as digital signature checking of new firmware to be flashed for security reasons.

# Bibliography


[1] SimpleLink Bluetooth low energy CC2640 wireless MCU Software Developer's Guide For BLE-Stack Version: 2.1.0.

[2] Bluetooth Specification Version 4.2.

[3] Mike Ryan, iSEC Partners: Bluetooth: With Low Energy Comes Low Security, https://lacklustre.net/bluetooth/Ryan_Bluetooth_Low_Energy_USENIX_WOOT.pdf, 26.1.2017.

[4] Elliptic Curve Cryptography, https://en.wikipedia.org/wiki/Elliptic_curve_cryptography, 31.1.2017.

[5] Elliptic Curve Diffie–Hellman, https://en.wikipedia.org/wiki/Elliptic_curve_Diffie-Hellman, 31.1.2017.

[6] Integrated Encryption Standard, https://en.wikipedia.org/wiki/Integrated_Encryption_Scheme, 31.1.2017.

[7] Elliptic Curve Digital Signature Algorithm, https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm, 31.1.2017.

[8] Android, https://www.android.com/, 31.1.2017.

[9] Android Developer Reference android.bluetooth, https://developer.android.com/reference/android/bluetooth/package-summary.html, 31.1.2017.

[10] Stackoverflow: Android Query Bluetooth LE Secure Connections Long-Term Key, https://stackoverflow.com/questions/41466005/android-query-bluetooth-le-secure-connections-long-term-key, 31.1.2017.

[11] Advanced Encryption Standard, https://en.wikipedia.org/wiki/Advanced_Encryption_Standard, 31.1.2017.

[12] Modum, https://modum.io, 31.1.2017.

[13] TI E2E Community: Why does the CC2640 OAD linker script appear to waste a page of flash?, https://e2e.ti.com/support/wireless_connectivity/bluetooth_low_energy/f/538/t/499930, 1.2.2017.

[14] Eclipse, https://eclipse.org, 1.2.2017.

# List of Figures

2.1 TI Development Environment . . . . . 4
2.2 App Project Folder Structure . . . . . 4
2.3 Stack Project Folder Structure . . . . . 4

# List of Tables

3.1 Risk Analysis . . . . . 10

3.2 Risk Analysis (continued) . . . . . 11