System Software Design Document  
(DRAFT 0.7)

Analog Devices Inc.

Contents

[Version 5](#_Toc419179)

[Definitions 6](#_Toc419180)

[Part I: Framework 7](#_Toc419181)

[What is an App? 8](#_Toc419182)

[Messaging: Internal and External 8](#_Toc419183)

[Configuring the Watch at Run-Time 9](#_Toc419184)

[Part II: Apps 10](#_Toc419185)

[PPG App 11](#_Toc419186)

[Continuous vs non-continuous PPG 12](#_Toc419187)

[Configs: ADPD DCFG, ADXL DCFG and PPG LCFG 12](#_Toc419188)

[Additional LCFGs 12](#_Toc419189)

[PPG M2M2 API 13](#_Toc419190)

[M2M2 Commands 13](#_Toc419191)

[Usage 14](#_Toc419192)

[M2M2 Data Events 14](#_Toc419193)

[M2M2 Notifications Events 15](#_Toc419194)

[DMA and Non-DMA Mode 15](#_Toc419195)

[Bio-Electric App 17](#_Toc419196)

[M2M2 Commands 17](#_Toc419197)

[Start 17](#_Toc419198)

[Stop 17](#_Toc419199)

[Subscribe 17](#_Toc419200)

[Unsubscribe 17](#_Toc419201)

[SetMode 17](#_Toc419202)

[ReadAD5940() 17](#_Toc419203)

[WriteAD5940() 17](#_Toc419204)

[LCFGRegRead() 17](#_Toc419205)

[LCFGRegWrite() 17](#_Toc419206)

[Low-Touch App 19](#_Toc419207)

[Wrist Detection 19](#_Toc419208)

[Processing the Generic Config Block 20](#_Toc419209)

[Logging App 21](#_Toc419210)

[File System 21](#_Toc419211)

[Compression by Reduction of Repetitive Data 21](#_Toc419212)

[File Download 24](#_Toc419213)

[User Interface App 25](#_Toc419214)

[Customization 27](#_Toc419215)

[User Interface 27](#_Toc419216)

[Custom Bio-Medical App 27](#_Toc419217)

[M2M2 Commands and Streams 28](#_Toc419218)

[New or Changed M2M2 Commands 28](#_Toc419219)

[Time Stamps 28](#_Toc419220)

[Part III: System Topics 29](#_Toc419221)

[Communications Interfaces 30](#_Toc419222)

[Universal Serial Bus 2.0 Full Speed 30](#_Toc419223)

[DFU Class 30](#_Toc419224)

[USB CDC ACM Class 30](#_Toc419225)

[Custom USB Class 30](#_Toc419226)

[USB 2.0 Full Speed Interface Definition 31](#_Toc419227)

[Packets Structure 31](#_Toc419228)

[Bluetooth Low Energy 31](#_Toc419229)

[nRF52840 Flash 33](#_Toc419230)

[Layout 33](#_Toc419231)

[Device ConfigurationBlock 33](#_Toc419232)

[Programmatic Access to the Device Configuration Block 35](#_Toc419233)

[Case 1 Update via M2M2 API 35](#_Toc419234)

[Case 2 Update via OTA Firmware Update 36](#_Toc419235)

[Case 3 Read-Only Programmatic Firmware Access of DCB 36](#_Toc419236)

[App Configuration Blocks 36](#_Toc419237)

[User Blocks 36](#_Toc419238)

[Other 37](#_Toc419239)

[File System 37](#_Toc419240)

[Secure Booting 37](#_Toc419241)

[Secure Updating 37](#_Toc419242)

[RTOS 37](#_Toc419243)

[Toolchain 37](#_Toc419244)

[Unit Testing 38](#_Toc419245)

[BLE Dongle 38](#_Toc419246)

[Log Download Tool 38](#_Toc419247)

[Device Configuration Block Generator and Push Tool 38](#_Toc419248)

[Serial Number 39](#_Toc419249)

Table of Figures

[Figure 1 Generalized App 7](#_Toc529174837)

[Figure 2 Sensor App Data Flow 8](#_Toc529174838)

[Figure 3 Anatomy of a Sensor App 8](#_Toc529174839)

[Figure 4 Example of a Default Sensor Configuration 9](#_Toc529174840)

[Figure 5 Example of a Customer Sensor Configuration 9](#_Toc529174841)

[Figure 6 Algorithm App Data Flow 10](#_Toc529174842)

[Figure 7 Example of a Default Bio-medical Configuration 11](#_Toc529174843)

[Figure 8 Example of a Customer Bio-medical Configuration 11](#_Toc529174844)

[Figure 9 M2M2 Packet Format 12](#_Toc529174845)

[Figure 10 Routing Data between Apps 12](#_Toc529174846)

[Figure 11 System Bio-medical and Sensor Software Stack 15](#_Toc529174847)

[Figure 12 ADXL Sensor App 16](#_Toc529174848)

[Figure 13 PPG Bio-Medical App 16](#_Toc529174849)

[Figure 14 ECG Bio-Medical App 17](#_Toc529174850)

[Figure 15 Pedometer App 17](#_Toc529174851)

[Figure 16 PPG Stack 18](#_Toc529174852)

[Figure 17 ECG and EDA Stack 20](#_Toc529174853)

[Figure 18 Pedometer Stack 21](#_Toc529174854)

[Figure 19 Person Detection App Processing the Generic Config Block 23](#_Toc529174855)

[Figure 19 User Interface 29](#_Toc529174856)

[Figure 20 BLE Central and Peripheral Use Cases 34](#_Toc529174857)

[Figure 21 Device Configuration Block 36](#_Toc529174858)

# Version

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Name(s) | Notes |
| 0.3 | 2018/09/19 | Ajeet Mall  Andre Straker  Punith Kumar  Rajesh V | Major overhaul to cover the Analog Wearable Framework |

# Definitions

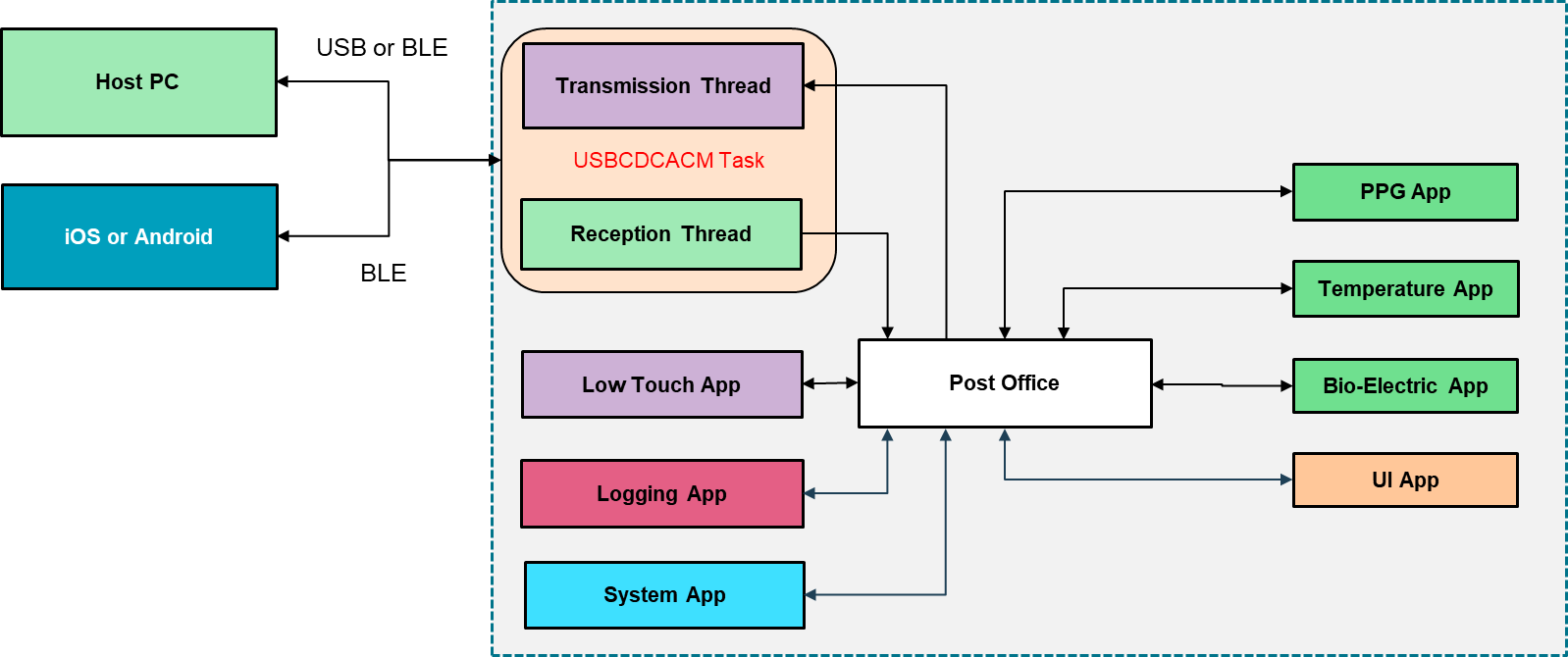
Customer – A customer of Analog Devices. The customer will deploy the watch to its end users.

End user – The final user of the watch. These are customers of Analog Devices’ customers.

DCFG – Device Configuration File (DCFG)

LCFG – Library Configuration File (LCFG)

Part I: Framework



## What is a Sensor Module?

In the context of the framework a sensor module is a self-contained RTOS task that exposes an M2M2 API. A sensor module can process commands and return a mandatory response – this is referred to a synchronous command-response pair. The sensor module can generate asynchronous data events and notification events.

### Messaging: Internal and External

The firmware uses a messaging protocol dubbed M2M2 (Machine to Machine v2). The M2M2 packet format is based on the UDP protocol packet.

**[Source Address, 16 bits][Destination Address, 16 bits][Packet Length, 16 bits][Checksum, 16 bits][Payload, n bytes]**

Source Address: the address of the sensor module that is sending the packet.

Destination Address: the address of the sensor module that will receive the packet.

Length: Size of message package in bytes = number of bytes for header (8) + number of bytes of payload

Checksum: check sum of entire packet with checksum set to 0. If the checksum is not used, then the value will be 0.

The address values are fixed at compile time. Where each sensor module has a unique address assigned by developers. Where the command-response pairs for a specific app will be assigned an address. Notification and data events have their own address. Therefore, a single sensor module will have one address for command-response pairs. And 0+ data event addresses. And 0+ notification addresses. To receive notification events or data events a sink sensor module must subscribe to the applicable event in the source senor module.

M2M2 allows for synchronous command-response pairs, asynchronous notification events and asynchronous data events.

Sensor modules expose an API that processes M2M2 command-response pairs, generates notification events and data events.

M2M2 relies on the *Post Office* to route messages internally and to external ports for sending to external sensor modules.

## Configuring the Watch at Run-Time

The use case of the watch will vary from customer to customer. While one customer may want to have PPG and ECG for 5000 end-users, another may want to collect accelerometer data and EDA only.

Therefore, customers must be able to set [configurations for the device](#_Device_Configuration_Block) before distribution to users. In addition, once distributed, customers may want to push a field update to end-users.

To enable this feature, we must have:

1. A configuration block that is readable by the firmware at start-up for configuring the system to place it into a known state.
2. A way to generate the configuration block.
3. A way to push the configuration block from the PC (via USB) and from the mobile device via BLE.

On 1) A [Device Configuration Block (DCB)](#_Device_Configuration_Block) will be added. This block should contain the necessary start up commands, device configuration and sensor module configurations.

On 2) A host/PC side tool is needed for this. It should be easy to use. As well save the configuration for future editing and saved for use by 3.

On 3) This can be same host/PC tool operating in headless mode.

Part II: Sensor Modules

# ADPD Sensor Module

This module will provide optical sensor data from ADPD4002 optical device. The module configures the watch with preference given to the user-defined configuration available in NOR Flash of the watch over the configuration within the firmware.

## Config: ADPD DCFG

As is the case for the gen3 software the ADPD will have the DCFG in the same format.

User specified CFGs will be stored in the Device Configuration Block area of the flash.

Loading a DCFG must happen in the following order:

1. Use the global LoadCFG(CFG\_index) –
   1. if a CFG is returned, then this CFG is used. Skip to step 3
   2. If a CFG is not found, then skip to step 2.
2. Load the CFG applicable in local memory
3. Configuration load end

## Configs: ADPD LCFG

The ADPD will have an LCFG as parameter values.

ADPD will have default LCFG. This default LCFG will be hardcoded in the source code – but must be easy to update – in other words, store them in appropriate include files.

User specified CFGs will be stored in the Device Config Block area of the flash.

LCFG should have an additional volatile storage in memory.

Loading a LCFG must happen in the following order:

1. Use the global LoadCFG(CFG\_index) –
   1. if an LCFG is returned then this CFG is used. Skip to step 3.
   2. If an LCFG is not found, then skip to step 2.
2. Load the CFG applicable in local memory
3. Apply the LCFG in volatile memory

|  |  |  |
| --- | --- | --- |
| **LCFG parameter** | **Description** | **Value** |
| ClockCal | Calibrate the clock for better accuracy | 0 : Clock calibration to be skipped  1 : Clock calibration to be performed |
| SlotConfig | Select slot, channel and dark/signal for the ADPD device | 4:0 – Slot selection  5: 0 – Channel 1, 1 – Channel 2  8:6 – Signal Data size  11:9 – Dark Data Size |

## ADPD M2M2 API

### M2M2 Commands

* Like all M2M2 commands these commands are synchronous with a command and response phase
* Each response should indicate the result of the command if successful, else an error if unsuccessful

LoadCFG

* Load ADPD DCFG.
* The device should have an ADPD DCFG hard coded in source.
* LoadCFG()
  + Try to load the CFG from NOR Flash using the **global** LoadCFG() call.   
    If it’s unsuccessful, then the hard coded CFG in firmware is used.

ClockCal

* Perform calibration of the ADPD device

Start

* Start the ADPD based on the DCFG selection
* Start should take no parameters

Stop

* Stops the ADPD sensor module

Subscribe

* Another app requests to start receiving data events from the ADPD Module.
* The ID of the data event(s) must be specified.

Unsubscribe

* Another app requests to stop receiving data events from the ADPD Module.
* The ID of the data event(s) must be specified.

RegWrite

* Write a series of register address-data pairs contained in the payload to the ADPD device.

RegRead

* Read a series of register address values contained in the payload from the ADPD device.

#### Usage

Example usage:

1. Subscribe
2. LoadCFG
3. ClockCalibration
4. RegWrite/RegRead
5. Start
6. …
7. Stop
8. Unsubscribe

# ADXL Sensor Module

This module will provide optical sensor from ADXL362 accelerometer device. The module configures the watch with preference given to the user-defined configuration available in NOR Flash of the watch over the configuration within the firmware.

## Config: ADXL DCFG

As is the case for the gen3 software the ADXL will have the DCFGs in the same format.

User specified CFGs will be stored in the Device Config Block area of the non-volatile memory.

Loading a DCFG must happen in the following order:

1. Use the global LoadCFG(CFG\_index) –
   1. if a CFG is returned then this CFG is used. Skip to step 3.
   2. If a CFG is not found, then skip to step 2.
2. Load the CFG applicable in local memory
3. Configuration load end

## ADXL M2M2 API

### M2M2 Commands

* Like all M2M2 commands these commands are synchronous with a command and response phase
* Each response should indicate the result of the command if successful, else an error

LoadCFG

* Load ADXL DCFG.
* The device should have a default ADXL DCFG hard coded in source.
* LoadCFG()
  + Try to load the CFG from NOR Flash using the **global** LoadCFG() call.   
    If it’s unsuccessful, then the hard coded CFG in firmware is used.

Start

* Start the ADXL based on the DCFG selection
* Start should take no parameters

Stop

* Stops the ADXL sensor module

Subscribe

* Another app requests to start receiving data events from the ADXL Module.
* The ID of the data event(s) must be specified.

Unsubscribe

* Another app requests to stop receiving data events from the ADXL Module.
* The ID of the data event(s) must be specified.

RegWrite

* Write a series of register address-data pairs contained in the payload to the ADXL device.

RegRead

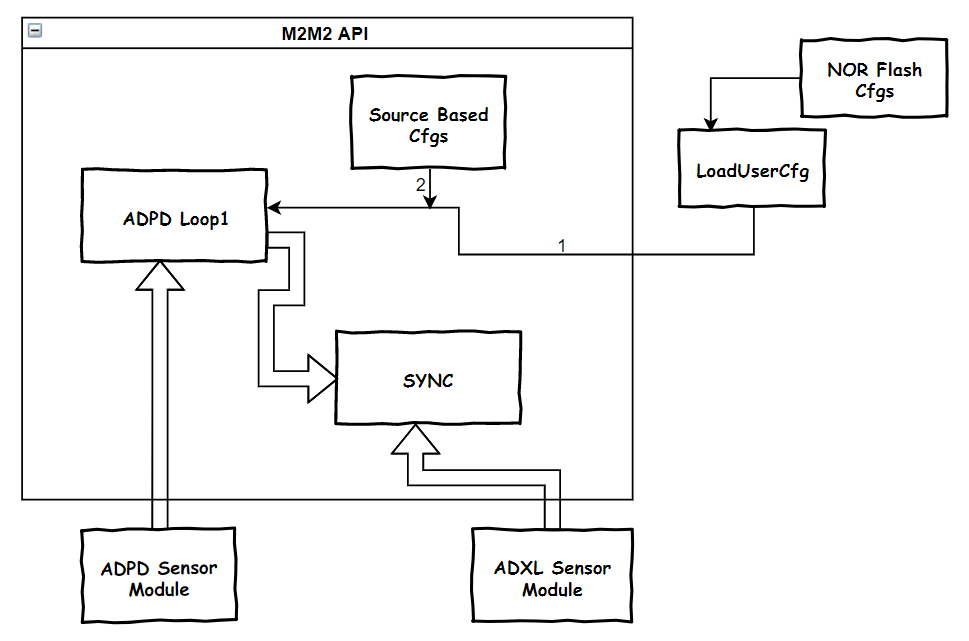
* Read a series of register address values contained in the payload from the ADXL device.

#### Usage

Example usage:

1. Subscribe
2. LoadCFG
3. RegWrite/RegRead
4. Start
5. …
6. Stop
7. Unsubscribe

# PPG App



## Continuous vs non-continuous PPG

In the gen3 firmware, once started PPG is always on. The next generation firmware will support duty cycle-based PPG. In this case PPG can be off for a specified number of seconds, then on for a specified number of seconds.

Definition

* Continuous PPG: PPG always on
* Non-continuous PPG: PPG on for a specific period time, then off for a specific period;

An LCFG entry will be used to control this feature

* 16-bit ON time in seconds
* 16-bit OFF time in seconds
* tOFF time == 0x0000 and tON time > 0x0000 => Continuous PPG
* Else => Non-continuous PPG

This feature should be managed on RTOS timer.

There are two cases:

1. The PPG app is used for loop 1 and loop 2 optimization.
2. The PPG app is not used for any sort of optimization.

Case 1 is understood.

Case 2 is new. In this case the customer will perform their own optimizations but can still use the Analog method of PPG duty cycle. In which case, there is no need for the PPG task to receive ADPD and ADXL data – the PPG does not have to subscribe

## Configs: PPG LCFG

The PPG will have an LCFG in the same format – however the LCFG will have new entries due to new features and may require changes in entries due to the ADPD400x.

PPG will have default LCFG. This default LCFG will be hardcoded in the source code – but must be easy to update – in other words, store them in appropriate include files.

User specified CFGs will be stored in the Device Config Block area of the flash.

LCFG should have an additional volatile storage in memory.

Loading a DCFG must happen in the following order:

1. Use the global LoadCFG(CFG\_index) –
   1. if an LCFG is returned then this CFG is used. Skip to step 3.
   2. If an LCFG is not found, then skip to step 2.
2. Load the CFG applicable in local memory
3. Apply the LCFG in volatile memory

### Additional LCFGs

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Definition** | **Mode**  **8-bits** | **PPG**  **8-bits** | **Output Sample Rate**  **16-bits** | **ADPD Duty Cycle**  **32-bits**  **0-15: tON - On time in seconds**  **16-31: tOFF - Off time in seconds** |
| ADPD Off, ADXL Off | 0 | N/A | N/A | N/A |
| ADPD Off ADXL On | 1 | N/A | ADXL Sample Rate | N/A |
| ADPD On, ADXL Off | 2 | 0 – None  1 – Loop 1  2 – Loop 1 and 2 | ADPD Sample Rate | ADPD on for tON seconds  ADPD off for tOFF seconds |
| ADPD On, ADXL On | 3 | 0 – None  1 – Loop 1  2 – Loop 1 and 2 | Common Sample Rate (Hardware sync -> ADPD driving ADXL) | ADPD on for tON seconds  ADPD off for tOFF seconds |

## PPG M2M2 API

### M2M2 Commands

* Like all M2M2 commands these commands are synchronous with a command and response phase
* Each response should indicate the result of the command if successful, else an error if unsuccessful

LoadCFG

* Load PPG LCFG.
* The device should have a PPG LCFG hard coded in source.
* LoadCFG()
  + Try to load the CFG from flash using the **global** GLoadCFG() call.   
    If it’s unsuccessful, then the hard coded CFG is used.
  + Finally, apply the volatile MLCFG – where these values would be written via PPGWrite() prior to issuing the start command.

Start

* Start the PPG based on the LCFG selection
* Start should take no parameters – all settings must be made available via the PPG LCFG.

Stop

* Stops the PPG app

Subscribe

* Another app requests to start receiving data events from the PPG App.
* The ID of the data event(s) must be specified.

Unsubscribe

* Another app requests to stop receiving data events from the PPG App.
* The ID of the data event(s) must be specified.

LCFGWrite

* Write a series of LCFG register address-data pairs contained in the payload to the PPG app.

LCFGRead

* Read a series of LCFG register values contained in the payload from the PPG app.

#### Usage

Example usage:

1. Subscribe
2. LoadCFG
3. LCFGWrite/LCFGRead
4. Start
5. …
6. Stop
7. Unsubscribe

### M2M2 Data Events

The PPG Module M2M2 packets will differ in nextgen firmware when compared to the gen3 firmware.

The driving factors are:

* ADPD400x support
  + Channels 1 and 2
  + Slots A, B, C, D, E, F, G, H, I, J, K, L
* PPG modes
  + ADPD, ADXL
  + ADPD only
  + ADXL only

External Subscribers only??

|  |  |  |
| --- | --- | --- |
| Section | Size | Name |
| Header | 2 | Source |
|  | 2 | Destination |
|  | 2 | Checksum |
|  | 2 | Length |
| Payload | 1 | Command |
|  | 1 | Status |
|  | 2 | Sequence number |
|  | 4 | Data header  15: 0 – ADPD  14: 0 - ADPD Channel 1; 1 - ADPD Channel 2  10-13: ADPD – Slot #; 0-A, 1-B, 2-C, 3-D, 4-E, 5-F, 6-G, 7-H, 8-I, 9-J, 10-K, 11-L  8-9: Data size; 0 – 1 byte; 1 – 2 bytes; 2 – 3 bytes; 3 – 4 bytes  0-7: # of data samples  15: 1 – ADXL  8:14: Reserved  0-7: # of data samples |
|  | NN | Samples – timestamps(s) and data |
|  | … | Additional Data Header |
|  | … | Additional Data Samples |

### M2M2 Notifications Events

Xxxxxxx

### DMA and Non-DMA Mode

The PPG App contains two specific modes:

* Tuning mode
  + The loop 1 and loop 2 algorithms are expected to actively tune the ADPD therefore data must be processed in near real-time.
  + This *may* require the use of non-DMA mode. However, this is to be determined.
* Data Collection mode
  + In data collection mode, either for logging, or passing to algorithm. In this mode, for power optimization, DMA will be used to automatically pull data from the system, only processing the data when enough has been collected.

Therefore, it is necessary to allow the system to switch between DMA and non-DMA mode when required.

TODO: how to use DMA in the system; how big are the memory blocks; what does the M2M2 packet look like; using DMA when sending sample externally via M2M2;

# AD5940 Sensor Module

The AD5940 module will be composed of the gen3 firmware AD5940 module, the ECG module (minus algo), EDA Module and BCM Module.

The SenosrAD5940Application.c can be folded back to reduce the amount of M2M2 commands moving around in the system.

The regular M2M2 commands will be START/STOP/SUBSCRIBE/UNSUBSCRIBE.

The additional configurations such as setting the ODR, setting the excitation frequency, setting the dynamic scale, data rate and DFT number will be part of LCFG

### M2M2 Commands

#### 

LoadCFG

#### Start

* Start the AD5940 sensor module based on the LCFG
* It will not take any parameters and all settings will be from its LCFG

#### Stop

* Stop the AD5940 Sensor Module

#### Subscribe

* Another module which might process this raw data requests to start receiving data events from the AD5940 sensor module
* The ID of the data event(s) should be specified

#### Unsubscribe

* Another module which has been processing this raw data requests to stop receiving data events from the AD5940 Sensor Module
* The ID of the data event(s) should be specified

#### SetMode

* Set the mode as to whether ECG or EDA or BCM stream should be output

#### RegRead

* Read the AD5940 32-bit registers specific to the AD5940 sensor module

#### RegWrite

* Write the AD5940 32-bit registers specific to the AD5940 sensor module

#### LCFGRead

* Read a series of LCFG register values contained in the payload from the AD5940 sensor module

#### LCFGWrite

* Write a series of LCFG register address-data pairs contained in the payload to the AD5940 sensor module

Example usage:

1. LoadCFG()
2. Subscribe
3. Start
4. …
5. Stop
6. Unsubscribe

Optional

1. RegRead/RegWrite
2. LCFGRead/LCGWrite

### M2M2 Data Events

|  |  |  |
| --- | --- | --- |
| Section | Size | Name |
| Header | 2 | Source |
|  | 2 | Destination |
|  | 2 | Checksum |
|  | 2 | Length |
| Payload | 1 | Command |
|  | 1 | Status |
|  | 2 | Sequence number |
|  | 1 | Data Header  Bit 1-0: 00 – ECG 01 – EDA 10 – BCM 11 – Invalid  Note that only one of the above 3 steam can be active at a time |
|  | NN | Samples – timestamps(s) and data |
|  | … | Additional Data Header |
|  | … | Additional Data Samples |

# Low-Touch App

The Detection App detects whether the watch is on a person. If the watch is on a person, then the Detection App will drive the system to a defined point as determined by configurations set by the Customer.

To do so the Detection App has two major components, wrist detect and device configuration processing.

The App uses the AD7156 for person detection. The configuration of the AD7156 will be available via a default configuration and a customer configuration – such that the device can be tuned to specific customer requirements.

On detection of a person the Detection app has an additional crucial role.

The UI App must automatically process the UI configuration block in the Device ConfigurationBlock.

## Wrist Detection

The AD7156 can be used to detect whether the watch is on or off the user’s wrist. With this the watch can automatically move between active and inactive states, where:

* Active state: the defined application is active (the requisite sensors and bio-medical applications are active, and the system is logging the necessary data), where the defined applications is defined by the g*eneric config block*.
* Inactive state: the defined application is not active - the requisite sensors and bio-medical applications are **not** active, and the system is **not** logging the necessary data.

Moving between the active and inactive states much be logged into the log file such that:

* Start of inactive state is logged
* Start of active state is logged

Wrist detection for on and off states must have a settling time that is configurable.

Therefore, the following must be a part of the system app:

* It encompasses the ADPD7156 device driver.
* Allows configuration of the AD7156
* Set the wrist detection on and off settling time
* Stream (asynchronous event) for wrist on and off events.

## Processing the Generic Config Block

The Generic Config block is composed of M2M2 commands. The System App processes each M2M2 command. By doing so the System App will move the system to a fully active state – the requisite sensors and bio-medical applications are active, and the system is logging the necessary data.

M2M2 commands are packed into the Generic Config block. Each command can be pulled out using the length field.

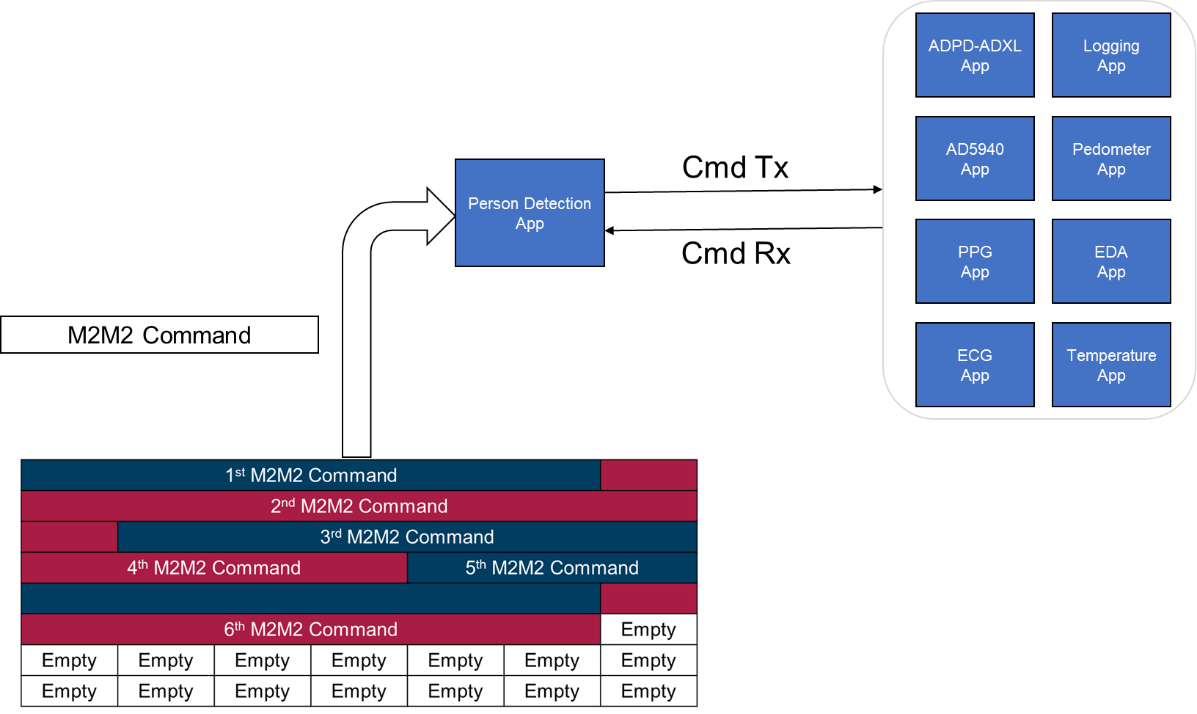


Figure 19 Person Detection App Processing the Generic Config Block

# Logging App

## File System

Given the requirement for USB MSC the file system type will very likely be FAT16. LFN support is not necessarily needed. File names can have the following 8.3 format.

The filename format, assuming MMDDTIME.log.

* MM – Month; ex: Jan – 01; Dec – 12
* DD – Day; ex: 5th – 05; 31st – 31
* TIME – Currently associated with the time such that, HH:MM:SS is written as a hex value where the lower 16-bit are used. For example 15:59:59, becomes 15595910 or 2613716, but dropping the last digit such that 26137 is used.

Examples:

* A log taken on 2018 December 21st at 13:45:50 (13455010 == 20D9616 -> 20D9) would have a file name of:
  + 122120D9.log

Conflicts can be handled by incrementing numeric value of the filename by one until there is no conflict. For example, if 122120D9.log is already on the file system, then the file name 122120DA.log is tried, etc..

The logging app in the framework functions by subscribing to the stream generators. As streams arrive at the logging app each stream is saved to the log file in the order in which they arrive. In general streams are saved as is. However, in the section Compression by Reduction of Repetitive Data, we discuss the introduction of a compression scheme to reduce redundant repetitive data.

The file system does insert some details into the log file:

* At the start of the log file the following information is inserted:
  + Participant information
  + Current date, time and time zone
  + Board version indicating the ADPD, ADXL and AD5940 versions
  + Firmware version information and Git build details
  + PPG algorithm version
  + Pedometer algorithm version
  + ECG algorithm version

## Compression by Reduction of Repetitive Data

Right now, we save streams as-is to the flash. A large waste are the repetitive headers for streaming packets. In particular, the source, destination and length are the same – therefore these can be compressed via a lookup table.

For example, for the sync PPG stream, is repeated for every stream:

C4 05 C5 01 00 48 00 00

Assuming we use the checksum as planned. Then the first 6 bytes are repeated. Over the course of a day of logging this adds up quickly.

We can introduce a lookup table. For example, where: C4 05 C5 01 00 48 is reduced to 00. The next look up entry will be 01, etc.

We can go from 00 to FE for 255 most used commands. If there are more, we can send FF followed by 00 to FE. If more than 512 commands are there, we can use 00 to FD and use FF and FE for extension of commands. Since the commands for the FS destination is way less than 255 (40 to be precise) and the length for each stream is fixed, we will assign just one byte for each header (source, destination and the length)

Similarly, we can use this method for data packets. For example, the probability of HR being more than 126 is less, so we can send one bytes and if it’s more than 126, we can send two bytes. We can do the header part in the first phase which will be very easy to implement.

In which case

**C4 05 C5 01 00 48** 00 00

Is reduced to

**00** 00 00

So we save 5 bytes per data packets.

The lookup table will be created and stored for each header info. The entry will be such that we have, for example:

**00** **C4 05 C5 01 00 48**

**01** **C4 00 C5 01 00 2E**

**--------------------------**

--------------------------

--------------------------

**28 C4 07 C5 01 00 3C**

The order of the header info in the lookup table doesn’t matter since each header is encoded with 1 byte. However, the header corresponding to the most probable stream should be put in the beginning so that the searching in the lookup table will be fast. Also, the encoded number in the look up table needn’t be present. We will compare the header info with the data stored in the look up table using a *for* loop. Once it matches, we will exit the *for* loop and assign the code which will be same as for loop counter where it existed the loop. Other methods can also be used to assign the code. Since the destination address and the length is fixed for each stream, just comparing the source address (put only the source address in the lookup table) is enough to get the encoded data. The function fs\_hal\_write\_packet\_stream() in file\_system\_utils.c file is the best place to encode (compress) the header info. The decoding process will use the same lookup table to decode the header info. The lookup table is shown below for reference:

const uint16\_t LUT\_m2m2\_header[40] =

{

0xC000,

0xC001,

0xC101,

0xC102,

0xC103,

---------

---------

---------

0xC805,

}

Let’s say we have the following:

**C4 05 C5 01 00 48** 00 00 28 00 7D 04 82 2B 34 46 38 29 34 46 82 02 82 02 81 02 81 02 82 02 82 02 5D 97 01 00 C7 97 01 00 3B 98 01 00 7F 98 01 00 63 00 63 00 6B 00 5F 00 28 00 2B 00 27 00 2C 00 37 01 3C 01 38 01 39 01

**C4 00 C5 01 00 2E** 00 00 28 43 2E 00 58 B9 19 46 0F 00 00 00 00 00 00 00 00 00 3F 30 05 63 0F 0C 64 00 35 1C 00 00 BB AA 01 00 00 00 00 00

**C4 05 C5 01 00 48** 00 00 28 00 7E 04 88 35 34 46 3E 33 34 46 81 02 82 02 82 02 82 02 81 02 82 02 AF 98 01 00 53 98 01 00 E9 97 01 00 3A 97 01 00 5F 00 67 00 5C 00 66 00 2C 00 27 00 2B 00 29 00 3B 01 38 01 3B 01 38 01

**C4 05 C5 01 00 48** 00 00 28 00 7F 04 8E 3F 34 46 44 3D 34 46 82 02 81 02 82 02 82 02 81 02 82 02 6A 96 01 00 8E 95 01 00 3B 95 01 00 CF 94 01 00 5F 00 66 00 5F 00 5C 00 27 00 2E 00 29 00 2B 00 3F 01 3B 01 39 01 38 01

**C4 00 C5 01 00 2E** 00 00 28 43 30 00 49 F1 19 46 07 00 00 00 00 00 00 00 00 00 01 00 2D 3C 01 00 44 39 BC F0 3F 30 05 63 0F 0C 64 00 35 1C

This will be stored in the NAND flash as follows:

**00** 00 00 28 00 7D 04 82 2B 34 46 38 29 34 46 82 02 82 02 81 02 81 02 82 02 82 02 5D 97 01 00 C7 97 01 00 3B 98 01 00 7F 98 01 00 63 00 63 00 6B 00 5F 00 28 00 2B 00 27 00 2C 00 37 01 3C 01 38 01 39 01

**01** 00 00 28 43 2E 00 58 B9 19 46 0F 00 00 00 00 00 00 00 00 00 3F 30 05 63 0F 0C 64 00 35 1C 00 00 BB AA 01 00 00 00 00 00

**00** 00 00 28 00 7E 04 88 35 34 46 3E 33 34 46 81 02 82 02 82 02 82 02 81 02 82 02 AF 98 01 00 53 98 01 00 E9 97 01 00 3A 97 01 00 5F 00 67 00 5C 00 66 00 2C 00 27 00 2B 00 29 00 3B 01 38 01 3B 01 38 01

**00** 00 00 28 00 7F 04 8E 3F 34 46 44 3D 34 46 82 02 81 02 82 02 82 02 81 02 82 02 6A 96 01 00 8E 95 01 00 3B 95 01 00 CF 94 01 00 5F 00 66 00 5F 00 5C 00 27 00 2E 00 29 00 2B 00 3F 01 3B 01 39 01 38 01

**01** 00 00 28 43 30 00 49 F1 19 46 07 00 00 00 00 00 00 00 00 00 01 00 2D 3C 01 00 44 39 BC F0 3F 30 05 63 0F 0C 64 00 35 1C

## File Download

1. A block of 526 bytes is formed in firmware as shown below

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Src | Dest | Length of packet | Checksum | Cmd | Status | Length of stream | Stream packet | CRC |
| 2 | 2 | 2 | 2 | 1 | 1 | 2 | Maximum 512 bytes | 2 |

Where length of packet is 526 bytes

Checksum is used as a counter starting from 0,1,2----65535 and then the tool has to keep track of it and send again 0,1,2 at roll over.

Length of stream which is maximum of 512 packets

CRC – of all 524 bytes

1. The tool/app does a download of the whole file as blocks of 526 bytes into a file, each time comparing a calculated CRC with that in the above block. It marks those blocks which have an unmatched CRC.
2. Now it loops over the blocks and for each block and if it a good block copes it into a second file and if it is with different calculated CRC from that in block, it
   1. Will ask the firmware for that block. The firmware opens the file and (based on a filename in the REQ pkt and gets the particular file. The tool again does the CRC calculation and if it matches copies that to the second file, else, does the retry of this step couple of times.
   2. If it does not get the good block, it asks user to download again

Once the good file is obtained, JSON conversion happens at which point, we can use the parsing of the compressed byte pattern and replace it with a table lookup 6-byte value corresponding to it.

# User Interface App

Possible UI libraries:

* <https://ugfx.io/index>
* <https://littlevgl.com/>
* Roll your own

Fonts supported must include Asian and Western type faces.

User input:

* The graphical presentation on the screen
* Device buttons, where
  + Upper (navigation) button is used to move between GUI screens.
  + Lower (action) button is used to act when needed.
  + Simultaneous pressing of both buttons will do a hardware restart of the device.
  + Long press of the top button turns the device on.
* Touch screen, where
  + Swipe left (navigation) is used to move between GUI screens.
  + Clicking the screen (action) is used to take action, when needed.

Display behaviour:

* Touching the device screen or any of the buttons will turn the display on,
* The display should have a configurable time out before it turns off again.

Assets such as fonts and images can be bundled with the application code as constants such that do not consume RAM.

The UI App must automatically process the UI configuration block in the Device ConfigurationBlock.



Figure 20 User Interface

# M2M2 Commands and Streams

See the accompanying Excel documents for the M2M2 commands.

## New or Changed M2M2 Commands

…

## Time Stamps

In each log, there will be a DateTimeInfo field indicating the data-time of starting this log. This must be converted to seconds elapsed from Jan 1, 2000. And multiplied by 32,000 to convert to ticks. To this result, each TS value from stream must be added. This will give the ticks of each sample. This can be converted to UTC time.

Part III: System Topics

# Communications Interfaces

The basic communication interfaces are USB and BLE.

The user can use either the BLE or USB interface which can be accessible by the user. This will prevent multiple, possibly conflicting commands being issued.

If the USB is plugged in, then the USB interface should become active and the BLE interface should become inactive.

If the USB is unplugged, then the USB interface should become inactive and the BLE interface should become active.

## Universal Serial Bus 2.0 Full Speed

The USB device will a composite device exposing at least the following two interfaces:

* DFU Interface
* Custom Interface

### DFU Class

This class will be used to update the device firmware over USB from the host to the device. This is owned by Fenda.

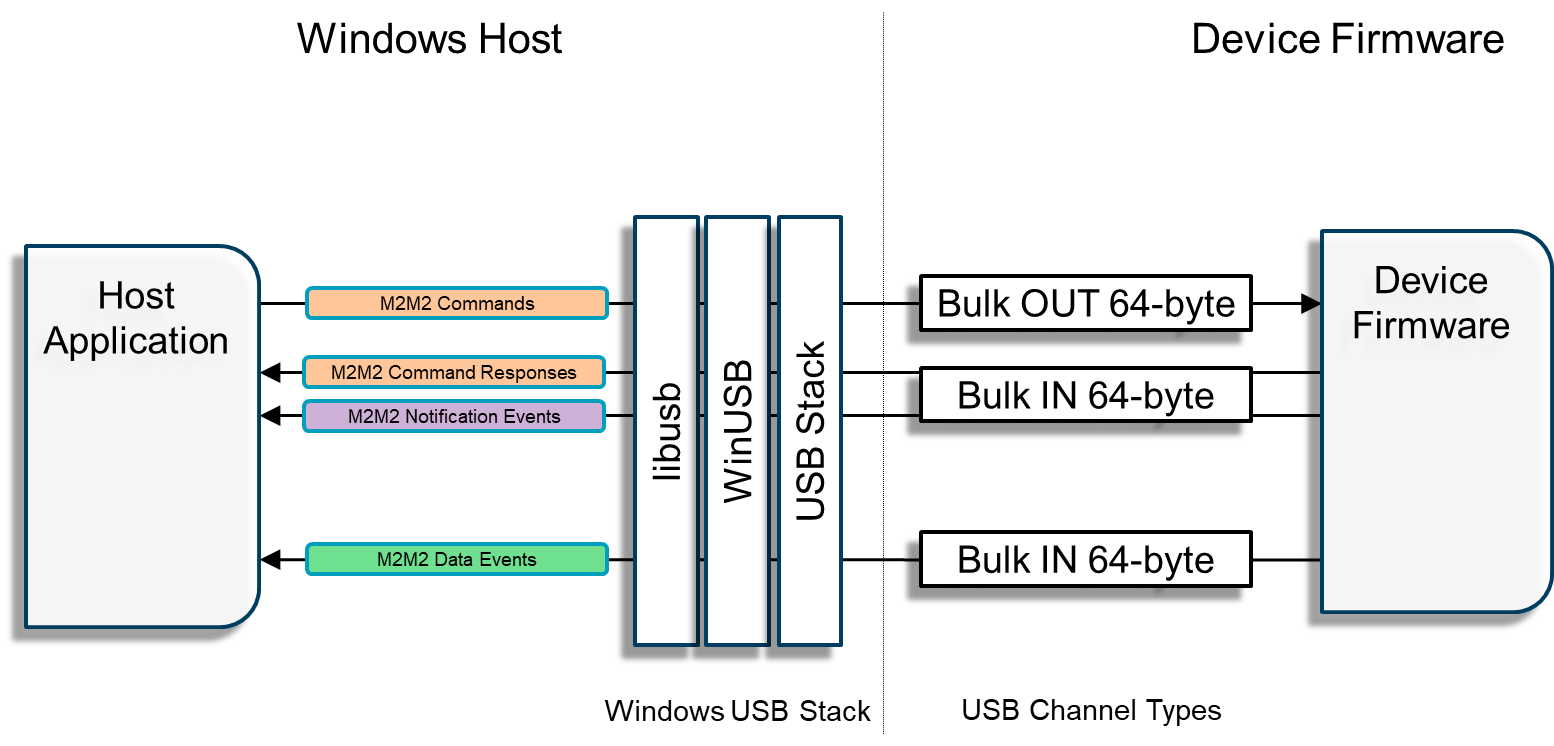
### 

### Custom USB Class

The ADI USB custom class between the watch and the host computer will serve the purpose:

* M2M2 command-response pairs and notification events; M2M2 packets are initiated from the host on the TX pipe, the firmware responds on the RX pipe. Where the TX pipe is a 64-byte BULK OUT channel and the RX pipe is a 64-byte BULK IN USB channel. Note, if RAM is an issue, these bulk packets can be reduced to 32-byte for both the IN and OUT packets.
* M2M2 asynchronous data event; M2M2 asynchronous data events originate from the watch. This bulk data is composed of sample mode data from specific modules or apps. In addition, logs are download over this 64-byte BULK IN USB channel.

This separation of command-response pairs and asynchronous data events will provide the host machine with a straight forward method of prioritization.



It is desired to have a data transfer rate of at least 6Mbps.

To avoid writing a driver for Windows WinUSB can be used. WinUSB relies on user space Windows API to receive data from the device (<https://docs.microsoft.com/en-us/windows-hardware/drivers/usbcon/using-winusb-api-to-communicate-with-a-usb-device>). Using WinUSB on the PC. Note, a WinUSB required a signed certificate – this certificate will have to be obtained for the production model. However, for initial development work Zadig can be used to install the driver: <https://zadig.akeo.ie/> (source: <https://github.com/madwizard-thomas/winusbnet/issues/20> ).

#### VID and PID

The Vendor ID can be the standard Analog Devices VID: 0x0456

The Device ID is TBD: TODO – It is unknown who in ADI manages DID assignment.

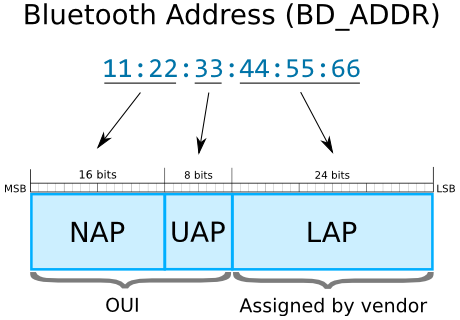
<https://devicehunt.com/view/type/usb/vendor/0456>

## Bluetooth Low Energy

* Battery Level state (characteristic: 0x2A19)
  + <https://www.bluetooth.com/specifications/gatt/viewer?attributeXmlFile=org.bluetooth.characteristic.battery_level.xml>
* BLE UART
  + UART/Serial Port Emulation over BLE
  + <https://www.nordicsemi.com/DocLib/Content/SDK_Doc/nRF5_SDK/v15-2-0/ble_sdk_app_nus_eval>

### BLE Mac Address

Ref: <https://macaddresschanger.com/what-is-bluetooth-address-BD_ADDR>



Analog Devices has two defined OUI (Ref: <https://www.adminsub.net/mac-address-finder/analog>)

Analog Devices OUI - TBD

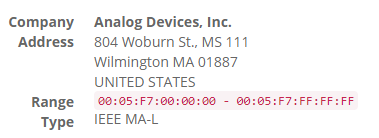
#### Option 00:E0:22

<https://www.macvendorlookup.com/search/00:E0:22:00:00:00>. If this still indeed does belong to ADI, this is likely open for use.



#### Option 00:05:F7

<https://www.macvendorlookup.com/search/00:05:F7:00:00:00>. It is unknown who in ADI owns this. This potentially goes back years.



# nRF52840 Flash

## Layout

The detailed layout of the flash is left to Fenda given the requirement of secure OTA.

The nRF52840 has 1MB of flash. For redundancy the flash should be divided as recommended by Nordic in the latest SDK documentation. Dual-bank layout is preferred, but understandably, the size of application and other aspects of the data stored on the flash will play a role in selecting single-bank vs dual-bank layout.

https://www.nordicsemi.com/DocLib/Content/SDK\_Doc/nRF5\_SDK/v15-2-0/lib\_bootloader\_dfu\_banks

The Application can be defined as

* Program binary image

## Device ConfigurationBlock

Each deployment will likely require a unique configuration that is selected by the researchers such that the end users will receive the watches ready to go when turned on.

For example, a deployment may require PPG and EDA only, while another deployment may require PPG, while a third may require PPG, ECG and Pedometer.

In addition, the sensors may be tuned for each specific use case.

Therefore, it is important that for a specific deployment the researchers be able to customize a group of watches as they see fit.

To do this we introduce the idea of a customizable start-up configuration.

In our device the API is essentially defined by M2M2 commands. Therefore, the configuration is simply a series of M2M2 commands that will place the device into a desired stated.

We are using the term device configuration block to refer to the device configuration as well as the configurations for the sensor devices and bio-medical applications.

The configuration block will contain – where each is series of M2M2 commands:



Figure 22 Device Configuration Block

The configuration block has two fixed at the start of the table:

* Checksum, which is a CRC-32 for the full block, excluding the checksum
* Index, which is a listing of the index offset from byte 0 to the start of each block; where there are 4 user defined bock indices.
  + From the table above, we can expect
    - index[ANALOG\_GENERAL\_BLOCK\_IDX] = 44;
    - index[ANALOG\_AD5940\_BLOCK\_IDX] = 1068;
    - index[ANALOG\_ADPD1081\_BLOCK\_IDX] = 1580;
    - …
  + If the index is equal to 0 then the block is empty.
    - Ex, if there is no configuration data in the AD5940 block then
      * index[ANALOG\_AD5940\_BLOCK\_IDX] = 0;

|  |
| --- |
| #define ANALOG\_CONFIG\_BLOCKS (16)  #define ANALOG\_GENERAL\_BLOCK\_IDX (0)  #define ANALOG\_AD5940\_BLOCK\_IDX (1)  #define ANALOG\_ADPD1081\_BLOCK\_IDX (2)  #define ANALOG\_ADXL362\_BLOCK\_IDX (3)  #define ANALOG\_AD7146\_BLOCK\_IDX (7)  #define ANALOG\_PPG\_BLOCK\_IDX (4) #define ANALOG\_ECG\_BLOCK\_IDX (5)  #define ANALOG\_EDA\_BLOCK\_IDX (6)  #define ANALOG\_PEDOMETER\_BLOCK\_IDX (8)  #define ANALOG\_TEMPERATURE\_BLOCK\_IDX (9)  #define ANALOG\_WRIST\_DETECT\_BLOCK\_IDX (10)  #define ANALOG\_UI\_CONFIG\_BLOCK\_IDX (11)  #define ANALOG\_USER0\_BLOCK\_IDX (12)  #define ANALOG\_USER1\_BLOCK\_IDX (13)  #define ANALOG\_USER2\_BLOCK\_IDX (14)  #define ANALOG\_USER3\_BLOCK\_IDX (15)  typedef struct {  uint32\_t checksum;  uint16\_t index[ANALOG\_CONFIG\_BLOCKS];  } analog\_device\_configuration\_block\_t; |

On start-up two things happen:

1. Block is verified via checksum.
2. The generic device block is processed.

The sensor blocks (AD5940, ADPD1081, ADXL362, Pedometer and Temperature), bio-medical blocks (PPG, ECG, EDA and BCM) and the remaining two blocks (AD7156 and the user defined area) are process on demand by applications in the system.

This block of data should be placed into the App Data section on the flash. It is expected that this block of data can be securely updated via BLE and USB.

### Programmatic Access to the Device Configuration Block

The Device Configuration Block (DCB) will need to be:

1. Updated via external software such that an external agent can erase config areas and update config area: read and write access. Reduce complexity by allow updates to only be done for the full DCB. In other words, do not allow for individual sub-blocks (ex. ADPD area) to be updated.  
   This will require:
   1. Suitable M2M2 API commands
   2. External codes (SDK) to enable access via M2M2 API
   3. Firmware to enable block updates. The System App is best suitable for this task
2. Updated via OTA firmware update. The full DCB will be sent as a part of the firmware update package.
3. Access from firmware to read a block. All apps other than the System App will have read-only access.

#### Case 1 Update via M2M2 API

In this case a complete DCB is pushed via M2M2 to the flash. The DCB when pushed will over write the existing DCB. To repeat, a DCB consists of sub-blocks representing the various DCFGs and LCFGs that make up the DCB.

Since the DCB is larger than the max payload on an M2M2 command, several M2M2 commands will be needed.

The size of the DCB block should be hardcoded in firmware. It must be impossible to send a DCB larger than the designated area. Currently only 8KB is assigned, the full 8KB must be sent.

The following M2M2 commands will be required:

* Erase DCB area
* Start DCB update
* Send block with start offset of i bytes
  + Stores block in RAM from offset i
  + Repeated until the full 8KB DCB is sent
* End DCB update
  + DCB checksum is calculated then compared with the sent checksum; an error is indicated is the checksums do not match
  + DCB is written to the designated non-volatile area;
    - If successful, respond with such
    - If failed, respond with such

#### Case 2 Update via OTA Firmware Update

In this case, after the firmware update, the updater should update the DCB in the App area via the M2M2 API. However, the DCB area should be erased prior to doing an OTA. This will prevent the device from restarting in a possibly bad state due to firmware and DCB incompatibility.

#### Case 3 Read-Only Programmatic Firmware Access of DCB

Firmware Apps must be able to read the DCB to retrieve specific sub-block. In this mode the DCB will be read and never written. However, if the DCB is being updated, Apps should be blocked from reading.

A library should be provided for this function. The library will be accessed via C/C++ APIs and **not** M2M2 – since it is not an App.

API:

* GetBlock(blockindex)
  + Where the block indexes are formed via ENUM and **not** hardcoded numeric value.
  + On success, return the data in a memory block
  + On failure, return an error code

### Module Configuration Blocks

The Module configuration blocks contain register address-data pairs as would be required by the consuming application.

In the case of sensors, ADPD, ADXL and AD5940, these are sensor device register address-data pairs. Where for:

* ADPD – 16-bit register and address pairs
* ADXL – 16-bit register and address pairs
* AD5940 – 32-bit register and address pairs
* AD7156 – 16-bit register and address pairs

In the case of bio-medical apps, PPG, ECG and EDA,

See the sections Figure 5 Example of a Customer Sensor Configuration (pg 9) and Figure 8 Example of a Customer Bio-medical Configuration (pg 11) for examples of the data in the configuration blocks.

### User Blocks

The user blocks can contain any information as desired by the modules that will consume data in the user blocks.

# Other

## File System

TODO

## Secure Booting

The Nordic SDK secure bootloader will be used, unless advised otherwise. ADI should be provided the necessary documentation and tools for creating application images.

<https://infocenter.nordicsemi.com/topic/com.nordic.infocenter.sdk5.v15.2.0/ble_sdk_app_secure_bootloader.html>

## Secure Updating

The firmware must be updated via BLE and USB. The update method must be secure, where the image is signed.

Secure updating must be able to update all segments of the flash:

* Application
* Device configuration section of app data region
* SoftDevice
* DFU boot loader

OTA must not result in a dead device.

## RTOS

FreeRTOS will be used. The selected version must be under MIT License.

## Toolchain

Will use the Segger Embedded Studio based on GCC tool chain.

GCC also allows our customers to be independent of the host system since GCC is available across Windows, Linux and MacOS.

<http://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.sdk%2Fdita%2Fsdk%2Fnrf5_sdk.html>

## Unit Testing

The current firmware does not have any unit testing. Unit testing must be added to the full firmware source code base.

# BLE Dongle

Requirements for BLE dongle:

1. UART over USB interface
2. Ability to change and update firmware
3. Certified
4. nRF52
5. Housing

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Dongle | 1 | 2 | 3 | 4 | 5 | Cost per unit (Cdn) |
| [RayTac MDBT50Q-RX](http://www.raytac.com/product/ins.php?index_id=89) | Yes | Yes | Yes | Yes | Yes | ~$16 |

The RayTac MDBT50Q-RX appears to be the most suitable solution. In addition, the [MDBT50Q-DB](http://www.raytac.com/product/ins.php?index_id=81) development board is also available.

Nordic’s firmware for USBD BLE UART: <https://www.nordicsemi.com/DocLib/Content/SDK_Doc/nRF5_SDK/v15-2-0/usbd_ble_uart_example?2007#usbd_cdc_acm_example_setup>

Note, the dongle needs to support the Nordic UART/Serial Port Emulation over BLE and the ADI Custom profile. The Nordic UART/Serial Port Emulation over BLE profile has already been mentioned.

# Serial Number

Each device should have a serial number. The Serial number can be the BLE Mac Address which should be unique. The LAP of the MAC Address can be manually assigned or derived from the nRF52840 serial number.

<https://devzone.nordicsemi.com/f/nordic-q-a/529/how-to-get-nrf51822-serial-number-and-hw-id-through-segger/2749#2749>

Part IV: Software

# Log Download Tool

PC based log download tool using WinUSB.

TODO

# Device Configuration Block Generator and Push Tool

A host side tool is required to serve two functions:

1. Generate a device configuration block
2. Push the device configuration block to the device using M2M2 (via USB or BLE – the channel type should not matter).

We have this feature in the VSM WaveTool. It can be extracted for this purpose. However, the following must be changed:

1. The current VSM WaveTool inserts CFGs as M2M2 commands.
2. This new method required CFGs to be converted to binary, then inserted into the designation CFG block.

In addition, the ADPD400x DCFG has a change to the DCFG for the ADPD10xx series. This must be taken in account. Applications WaveTool 2.1.1 onwards supports this version of the ADPD400x DCFG.

# Firmware Update

Looking for Fenda to lead this, but we require:

1. Firmware Updates need to be supported via BLE and USB. Nordic has support for such: <https://www.nordicsemi.com/DocLib/Content/SDK_Doc/nRF5_SDK/v15-2-0/lib_bootloader_dfu_process>
2. Need to update:
   * Application
   * Soft Device
   * Bootloader
3. Also need to update:
   * Device Control Block (in User Application Space)
   * User applications
4. Dual bank or single bank – really determined by the size of our application
5. Secure
6. What tools are needed?