IoT - Software Design Document

Andrew Bonneville

Table of Contents

1 Introduction

1.1 Purpose

Define software design document (SDD) for the IoT Sensor Project. The SDD captures internal software design details and derived requirements to implement a software product.

1.2 References

STMicroelectronics, https://www.st.com:

- UM2153 User manual, Discovery kit for IoT node (B-L475E-IOT01A).
- UM1734 User manual, STM32Cube USB device library.
- PM0214 Cortex®-M4 programming manual.
- RM0351 Reference manual, STM32L4x5
- DS10969 Datasheet, STM32L475xx
- AN4760 Application note, Quad-SPI (QSPI) interface
- UM1718 User Manual, STM32CubeMX C Code Generation

Other:

- FreeRTOS, code and online documentation for kernel, https://www.freertos.org
- mbed TLS, code and online documentation, https://tls.mbed.org/
- Amazon FreeRTOS, device registration and security requirements, https://aws.amazon.com/freertos/

2 Design Considerations

2.1 Design Approach

This project handles various events that can happen both in an asynchronous and synchronous manner. There is no high-rate computation or handling of large data sets, but the system must remain responsive to events with minimal latency to prevent loss of data.

Overall design approach:

- Foreground tasks - these will consist of interrupt handlers servicing time critical events.
- Background tasks - these will handle any extended processing outside of an interrupt handler, and will be implemented as thread(s).

2.2 RTOS

2.2.1 Overview

Thread management will be handled by a commercially available RTOS that features:

- Mechanism for task to task communication
- Semaphore/mutex for accessing shared resources
- Preemptive scheduler for elevating higher priority tasks

2.2.2 FreeRTOS

FreeRTOS has been selected for this project, which is entirely written in C. Therefore, a C++ wrapper will be used by the application layer to provide an adaptive layer should we need to switch out the RTOS on this or future projects.

2.3 Software Directories

The following directory structure shall be used for source code and related libraries:

- <root> = IDE workspace
- <root>\Application\ = main application folder.
- <root>\CppUTest-Target\ = unit and system testing on target hardware. Creates an independent application (CppUTest-Target.elf) for testing.
- <root>\Startup\ = contains the linker command files and is the final build folder for the overall application (Startup.elf).
- <root>\<library>\ = various static libraries that get linked into the final Startup.elf. Source files provided by third- party will be located in their own independent library for ease of maintenance and update. Refer to the respective library folders for vendor information

2.4 C++ Exception Handling

C++ exception handling system (EHS) will be used for all C++ development to improve readability and maintenance of the code. In particular, the application will be able to monitor and handle exceptions thrown by the C++ STL rather than manually adding code to monitor and check for various error conditions.

When the EHS overhead affects performance, the use of noexcept and nothrow can be evaluated on a case by case to eliminate the overhead in critical paths.

2.5 Hardware Interrupt Handling

2.5.1 Guidelines

Nested interrupt handling will not be used to simplify code development and debugging. Therefore, any and all interrupt service handlers will adhere to the following guidelines to reduce interrupt latency:

- All interrupt processing should be as brief as possible; any extended processing should be deferred to a background task and not performed in the interrupt handler.
- Interrupt priorities will be assigned in the interrupt controller.

2.5.2 Interrupt Priorities

The STM32 Nested Vector Interrupt Controller (NVIC) will be configured to support an interrupt priority range of 0 to 15, with 0 being the highest priority interrupt. The following table identifies all enabled interrupts and their priority of execution (high to low):

Priority	Description
0	HAL driver code. This pertains to software timers in the HAL driven by a periodic hardware interrupt. Source: Internal, Timer 2
13	SPI3 global interrupt, handles WiFi data exchange over SPI. Source: Internal, SPI3 controller
13	Wi-Fi module, data ready interrupt. Source: External, ISM43362 module
14	USB global interrupt, handles all USB related activity; data receive, data transmit, enumeration, etc... Source: Internal, USB OTG controller
15	All RTOS specific interrupts (SVC, Systick, and PendSV) will use the same priority, the lowest priority. Source: Internal, SVC instruction, SysTick timer, PendSV control bit

Internal = generated on-chip STM32
External = generated outside the STM32, at the board-level

2.6 Hardware Timers

Core/system clock shall be 80Mhz, all peripherals are clocked at the same frequency to maximize performance.

All timers are internal to the STM32 processor, allocation and periodic intervals are as follows:

Timer	Period	Description
Systick	1 mS	Periodic timer that runs the RTOS scheduler at the specified period.
1	--	Note used
2	1 mS	Periodic timer used by the STM32CubeMX HAL code to implement software timers/delay loops.
3 - 8	--	Not used

2.7 Memory Allocation

2.7.1 Overview

The STM32L475VG internal memory segments shall be allocated as follows:

Address Range	Size (Kbyte)	Description
0x080F_FFFF 0x0808_0000	512	Bank 2 - field definable configuration information (e.g. cloud key received over USB interface)
0x0807_FFFF 0x0800_0000	512	Bank 1 - main executable program and initialization values for RAM memory
0x2001_7FFF 0x2000_0000	96	SRAM1 - see below
0x1000_7FFF 0x1000_0000	32	SRAM2 - see below

Internal memory mapped peripherals are not included in this table, refer to the manufacturers documentation for more information.

Volatile memory (SRAM) shall be subdivided as follows:

Address Range	Size (Kbyte)	Description
0x2001_7FFF 0x2001_2000	24	Secondary location for variables
0x2001_1FFF 0x2000_0000	72	Heap memory, which includes dynamic objects and thread / process stack pointers (PSP)
0x2001_7FFF 0x2000_0000	96	SRAM1 - total available memory for this hardware segment
0x1000_7FFF 0x1000_0400	31	Primary location for global and static variables
0x1000_03FF 0x1000_0000	1	Interrupt handlers, main stack pointer (MSP)
0x1000_7FFF 0x1000_0000	32	SRAM2 - total available memory for this hardware segment

2.7.2 Non-volatile Memory Allocation

STM32 on-chip flash implementation restricts how the memory can be allocated. Code executing from one bank, cannot be erased or written too regardless if the configuration area is page aligned and outside the code image. Therefore, to support setting field configurable parameters, code will reside in one bank, and configuration parameters will be maintained in a separate bank.

2.7.3 Stack Allocation

The STM32 will be configured to use both of its available stack pointers as follows:

- Main stack pointer (MSP) - statup code, and all exception/interrupt handlers use the main stack pointer. The initial memory location for the MSP is set in the linker command file.
- Process stack pointer (PSP) - once the RTOS scheduler is started, all background code uses the PSP. The PSP is managed by the RTOS for each thread created, and is allocated at runtime from heap memory. Exception handlers continue to use the MSP.

Separate stack pointers has an advantage, in that sizing the stack for individual threads does not include stack depth due to interrupt handlers. This means that the stack usage for interrupt handlers impacts only the main stack pointer, and does not get replicated for each thread stack when compared to using a common stack pointer.

2.7.4 Heap Allocation

One memory region will be setup to handle all dynamic memory allocation, heap memory. The heap will contain objects, as well as the individual thread/process stack pointers managed by the RTOS.

3 Application Modules

3.1 High Level

There are four basic levels to the design:

- Application - contains the proprietary code that uniquely defines the product functionality and features. The application consists of three threads:
    - Command Interface - processes commands received via the USB interface, if a response is required notifies the Response Interface to handle request.
    - Response Interface - waits for a request from the Command Interface to generate a response message and sends it out the USB interface
    - Cloud Interface - periodic thread that collects and submits new sensor information to a cloud server via the Wi-Fi interface.
- Middleware - predominately contains third-party code that implements a specific functionality (e.g. RTOS, protocol stack).
- Drivers - low level code that provides I/O access to hardware resources.
- Hardware - physical components such as the processor, peripherals, and on board modules and sensors.

Application	Command Interface (USB Receive)	Response Interface (USB Transmit)	Cloud Interface (Wi-Fi Tx/Rx)
Middleware	C/C++ Runtime Library and C++ Wrappers		
	FreeRTOS	TLS	MQTT
Drivers	USB	Sensors	Wi-Fi
	Hardware Abstraction Layer		
Hardware			

In anticipation of new features not yet addressed, the design mandates an abstraction layer between each design level. This mitigates coupling between non-adjacent layers and allows the implementation to adapt to changing needs. For example, code in the Application layer cannot directly call a WiFi driver, instead must use the C/C++ Runtime Library or a C++ Wrapper (abstraction layer).

Unless otherwise stated in this document, code in the Middleware and Driver must be thread-safe:

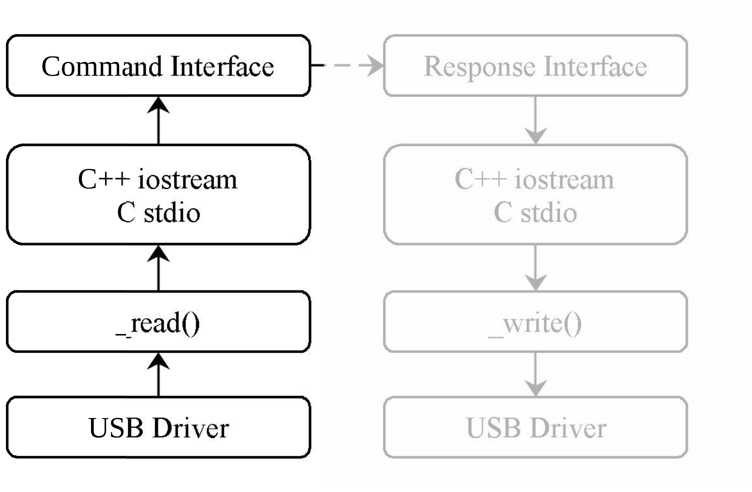
- "A thread-safe function can be called simultaneously from multiple threads, even when the invocations use shared data, because all references to the shared data are serialized."

3.2 Detailed Design

3.2.1 Command Interface

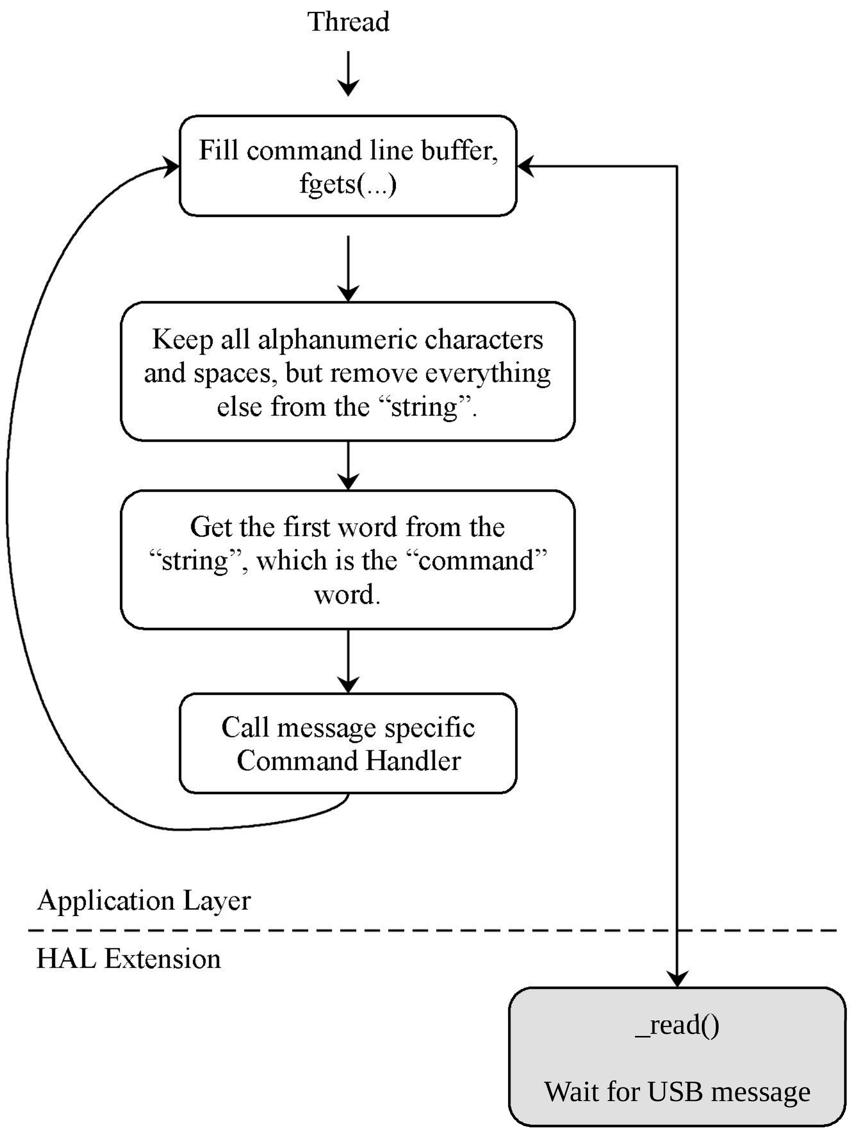
3.2.1.1 Data Flow

The Command Interface is responsible for processing all data that is received over the USB port. Data flows from the USB Driver to the system call function _read(), which delivers the data to the runtime library for double buffering. Upon receipt of a newline character, the runtime library delivers a single command string to the Command Interface.



3.2.1.2 Thread

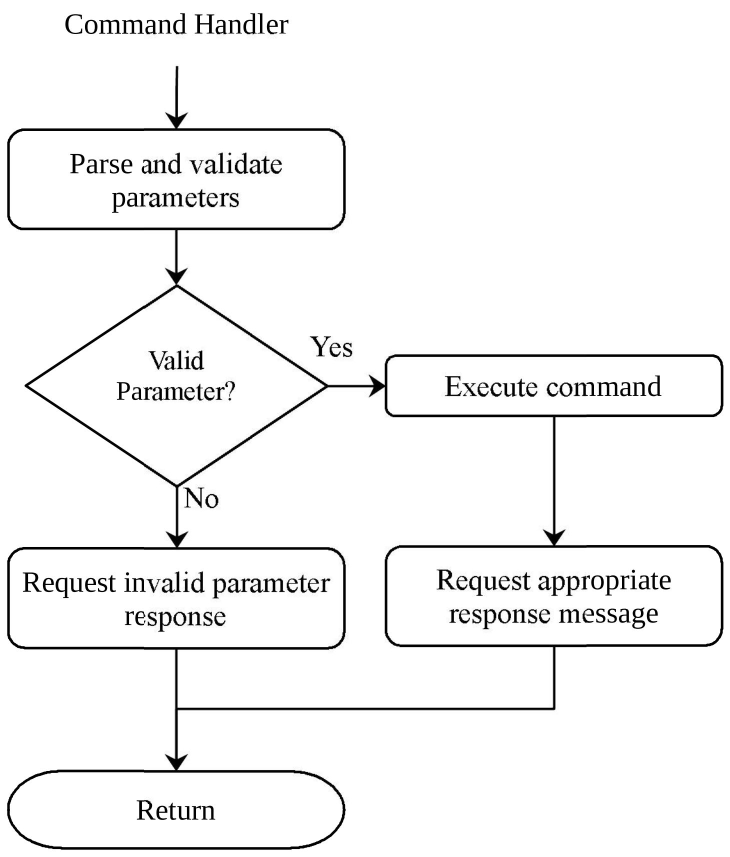
Command Interface will be implemented as a persistent RTOS thread that will block indefinitely when waiting for user to enter a newline. When a newline character is received, the Command Interface will parse and execute the command string. The runtime library will implement the command line buffer, buffer each USB message(s) until a newline character is received.


Thread
Fill command line buffer, fgets(...)
Keep all alphanumeric characters and spaces, but remove everything else from the "string".
Get the first word from the "string", which is the "command" word.
Call message specific Command Handler
Application Layer
HAL Extension
_read()
Wait for USB message

3.2.1.3 Command Handler

A unique command handler is implemented for each command word, and is responsible for parsing and validating any and all parameters associated with the command. Once validated, the command handler will then execute the command and if appropriate send a request to the Response Interface to generate and send a message back to the PC host.

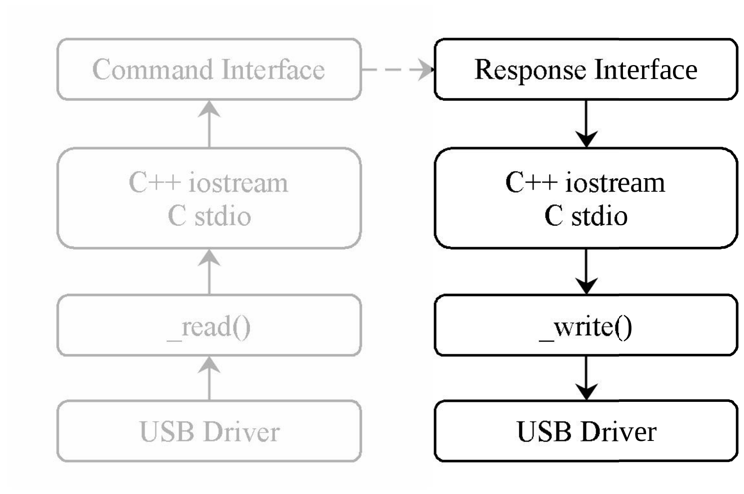
The following is a generic command handler:


Command Handler
Parse and validate parameters
Valid Parameter?
Yes
Execute command
No
Request invalid parameter response
Request appropriate response message
Return

3.2.2 Response Interface

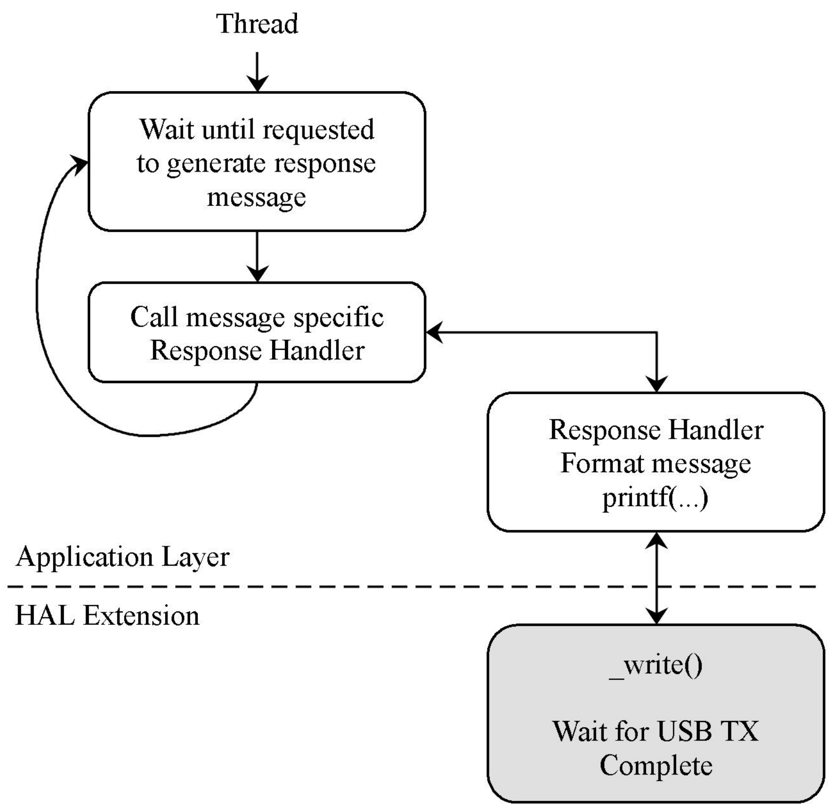
3.2.2.1 Data Flow

The Response Interface is responsible for generating all response messages to be transmitted over the USB port. Data flows from the Response Interface to the runtime library. The runtime library then calls the system call function _write() which deliveries the buffered data to the USB Driver.



3.2.2.2 Thread

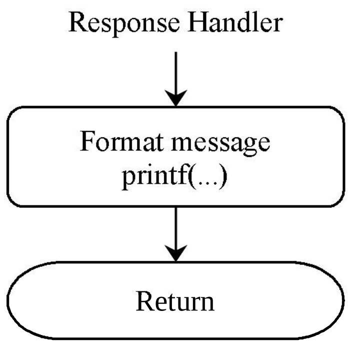
Response Interface will be implemented as a persistent RTOS thread that will block indefinitely when waiting for a request to send a response message. When a request is received, the Response Interface will format a message and call the standard IO library to buffer and transmit message.


Thread
Wait until requested to generate response message
Call message specific Response Handler
Response Handler Format message printf(...)
Application Layer
HAL Extension
_write()
Wait for USB TX Complete

3.2.2.3 Response Handler

A unique response handler is implemented for each message, and is responsible for formatting the message content and initiating the transfer back to the PC host.

The following is a generic response handler:



3.2.3 Cloud Interface

Cloud Interface will be implemented as a persistent RTOS thread, that will provide a basis for running IoT demos and sample code. The demos will primarily interact with the network stack to demonstrate functionality. At this time there are no hard design requirements for a specific demo.

4 Operating System

4.1 High Level

The operating system provides the low-level functionality required by the application, and consists of:

- Wrappers - abstraction layer between the application and the lower layers
- Middleware - predominately contains third-party code that implements a specific functionality (e.g. RTOS, protocol stack), and is not hardware dependent
- Drivers - low level code that provides I/O access to hardware resources.
  - HAL Extension - binds or redirects various modules together with a goal of providing a porting layer that simplifies maintenance and future adaptations.
- Hardware - physical components such as the processor, peripherals, and on board modules and sensors.

The remainder of this section will document the implementation, or in the case of third party libraries where the documentation can be obtained.

Application			
Middleware	C/C++ Runtime Library and C++ Wrappers		
	FreeRTOS	TLS	MQTT
Drivers	HAL Extension		
	USB	Sensors	Wi-Fi
	Hardware Abstraction Layer (HAL)		
Hardware			

4.2 Detailed Design

4.2.1 STM32CubeMX

“STM32CubeMX is a graphical tool that allows a very easy configuration of STM32 microcontrollers and microprocessors, as well as the generation of the corresponding initialization C code for the Arm® Cortex®-M core or a partial Linux® Device Tree for Arm® Cortex®-A core), through a step-by-step process.” - STMicroelectronics

The code is maintained and documented by STMicroelectronics, for additional information refer to the References section at the beginning of this document.

For this project, STM32CubeMX generates the following code:

- USB Driver
- Hardware Abstraction Layer (HAL)
- Sensors

4.2.1.1 USB Driver Configuration

Generation of the USB driver shall have the following settings:

- Device only - this project will connect to an external host, such as a PC host.
- Endpoint size 64-bytes - this is largest endpoint size supported for full-speed devices, and will maximize throughput on the data link without adversely affecting CPU load.
- Communication Device Class (CDC) - class will be set to CDC mode to support bulk transport of data between host and this device.
- Enable self power and link power management features. All other features will remain disabled.
- USBD TX and RX buffer size settings are 1024-bytes and 64-bytes respectively. Contents are directly accessible by the runtime libraries when exchanging data.

4.2.1.2 Sensor Configuration

Under review, not yet designed.

4.2.1.3 Hardware Abstraction Layer (HAL) Configuration

The HAL is implementation specific and not discussed at the design level. For additional information refer to the References section at the beginning of this document for the STM32CubeMX.

4.2.2 HAL Extension

4.2.2.1 Overview

The HAL Extension is not generated by the STM32CubeMX tool, and is a design choice to provide a dedicated module that binds and redirects various modules together. The overall goal of the HAL Extension is to provide a porting layer that simplifies maintenance and future adaptations.

As a result:

- Consists mostly of system calls that bind or redirect C/C++ libraries to various drivers.
- The collection of functions and methods may not be directly related to one another in functionality.

4.2.2.2 USB Messages

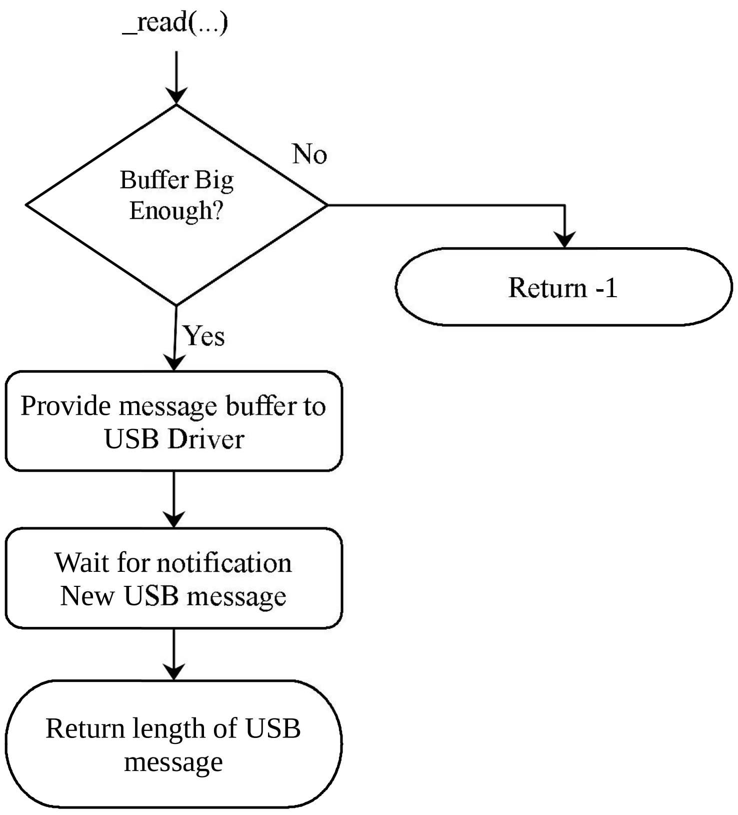
4.2.2.2.1 _read()

The _read() function is a standard system call that interfaces with the IO libraries, and coordinates the transfer of a USB message from a PC host to the IO library. This implementation will provide a zero copy interface between the IO library and USB driver, and will only verify that the IO library has provided a sufficiently large enough buffer for direct write access by the the USB driver.

When a valid buffer size has been provided, the _read() will then hand the buffer off to the USB driver and wait indefinitely until a new USB message arrives.

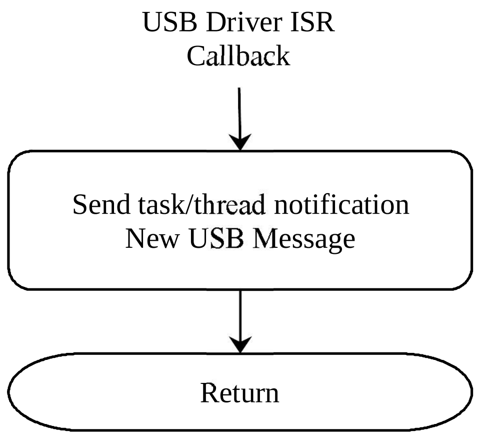
In addition, this design also blocks the PC host from sending another USB message until the function hands the USB Driver a new message buffer. This approach allows the application to process the contents of the message without risk of loosing data.

Based on the above design, the USBD RX buffer size will be set to 64-bytes (endpoint size), the maximum amount of data a PC host can send to the device in a single transaction.


_read(...)
Buffer Big Enough?
No
Return -1
Yes
Provide message buffer to USB Driver
Wait for notification New USB message
Return length of USB message

4.2.2.2.2 _read() ISR

The _read() will block indefinitely while waiting for a new USB message. To unblock the _read(), the USB driver has an ISR that will notify the RTOS when a new USB packet has arrived, and thereby wakup the _read() function to process the message.

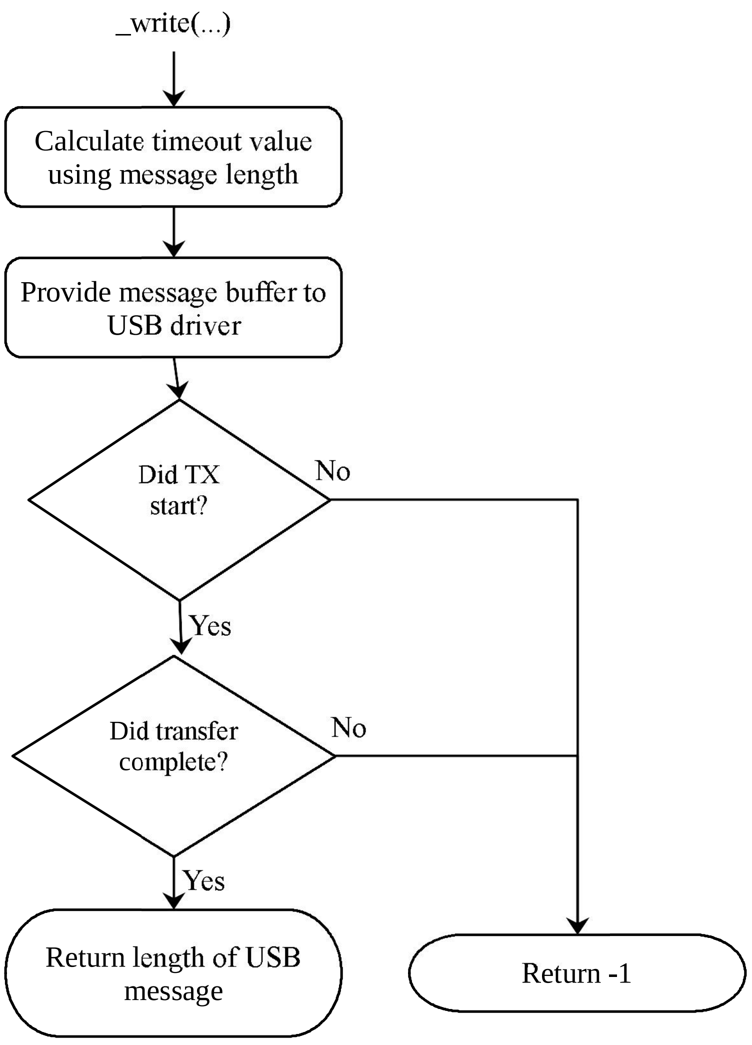


4.2.2.2.3 _write()

The _write() function is a standard system call that interfaces with the IO libraries, and initiates transfer of date from this device to the PC host. This implementation will provide a zero copy interface between the IO library and USB driver. There is no constraint on buffer/message size.

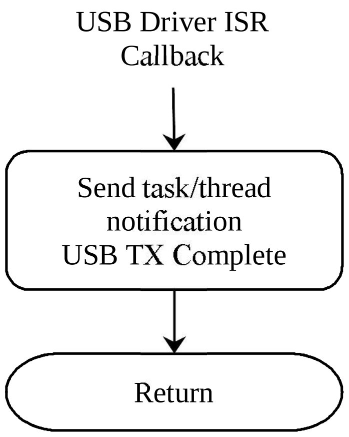
The _write() will hand the buffer off to the USB driver and then wait for the transfer to complete with the PC host, or timeout waiting. The timeout duration shall be based on the length of the message to be transferred.

To support system debug messages, the _write() method shall be thread safe and utilize a semaphore/mutex to coordinate access to the shared communication channel.


_write(...)
Calculate timeout value using message length
Provide message buffer to USB driver
Did TX start?
No
Yes
Did transfer complete?
No
Yes
Return length of USB message
Return -1

4.2.2.2.4 _write() ISR

The _write() function which will block waiting for the USB TX to complete. To unblock the _write(), the USB driver has an ISR that will notify the RTOS when a USB TX Complete has occurred, and thereby wakeup the _write() function to process further.



4.2.2.3 Data Storage

Under review, not yet designed.

4.2.3 FreeRTOS

4.2.3.1 Overview

FreeRTOS is a kernel that is maintained and documented by Amazon Web Services, for additional information refer to the References section at the beginning of this document.

Note, the STM32CubeMX tool can also generate a version of the FreeRTOS, however, that approach has the following limitations:

- Tool does not provide the latest version, and lags behind about 2 to 3 versions.
- Tool implements the CMSIS wrapper around the kernel, which is a C wrapper and not a C++ wrapper. The application is written in C++, and would require an additional wrapper layer that would hinder maintenance and debugging.
- Kernel support is from the FreeRTOS community, STMicroelectronics does not modify or provide any support for the kernel.

4.2.3.2 Configuration

FreeRTOS shall be configured as follows:

- Scheduler - kernel supports two options; preemptive or round robin (time slicing). This project shall use a preemptive scheduler to adhere to the overall design approach, remain responsive and minimize system latency.
- To aid maintaining the overall system, kernel statistics and stack management will be enabled for all builds. For consistency, all builds including release builds will have the feature enabled.

Task notification feature provides the ability for a task to be notified when an event happens. It is limited to providing notification to a single consumer, and cannot be used to broadcast an event. Therefore, to aid development of drivers, the FreeRTOS feature set for task notification shall be used exclusively for driver development. The application layer is not to use task notification.

All other RTOS configurations and features are considered implementation details, and not addressed at the design level.

4.2.4 Network

4.2.4.1 High Level

The following is a high-level overview for the network stack used on this project.

Application			**OSI Model**
Middleware	C++ STL / C++ Wrapper		
	FreeRTOS	MQTT	7
		TLS	6
		Sockets	5
Drivers		WiFi	1 - 4
Hardware			

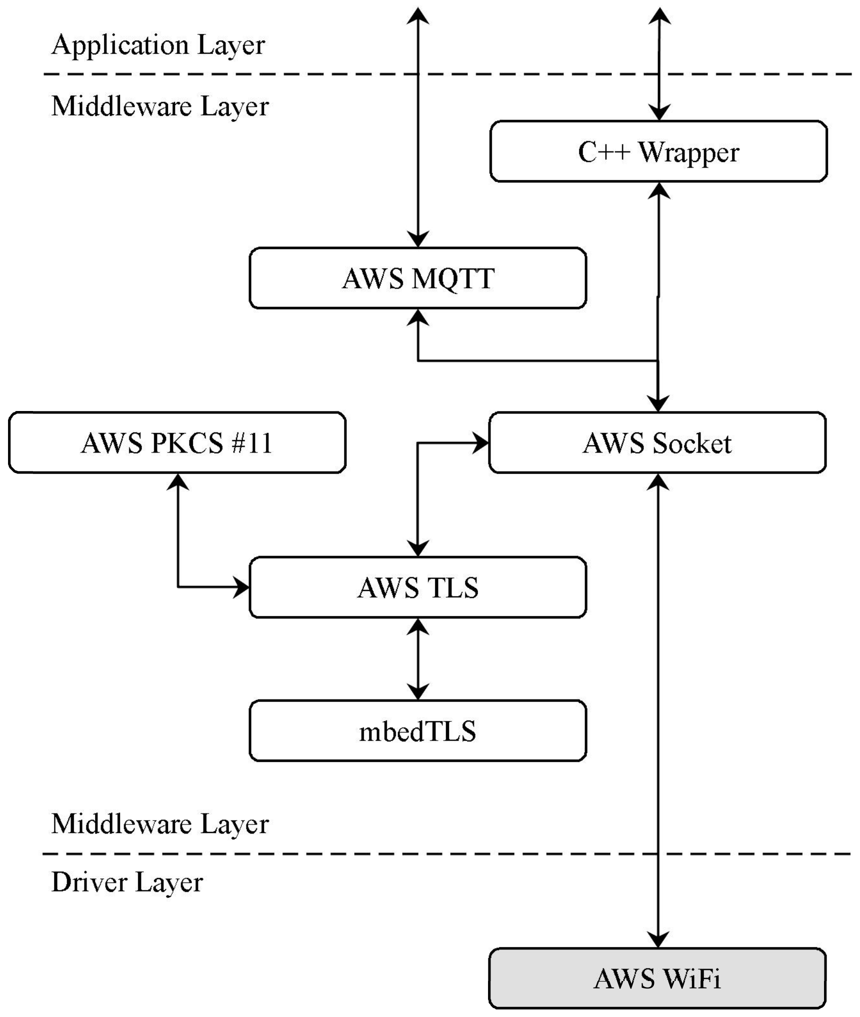
For each of the major modules; MQTT, TLS, Sockets, WiFi a third-party commercially available module will be used for implementation. That vendor will provide the necessary documentation and internal design details.

To anticipate changing needs, an abstraction layer shall be implemented between major modules. For example, the TLS module shall be abstracted such that the module can be swapped out with a different implementation that will be adapted to an abstraction layer between MQTT and TLS, as well as, TLS and Sockets.

All other features are considered implementation details, and not addressed at the design level.

4.2.4.2 Data Flow

Amazon Web Service (AWS) maintains a near production ready IoT operating system for microcontrollers, and will be the basis for providing network connectivity for this project. The following diagram depicts how data flows through the network stack:



Known limitations/exceptions:

- UDP is not supported, a local branch will be implemented to add the capability, and periodically merged with AWS updates. Implementing UDP impacts the AWS Socket and AWS WiFi modules.
- FreeRTOS kernel will be maintained separately from the AWS package, and will be obtained directly from the source.
- mbedTLS will be maintained separately from the AWS package, and will be obtained directly from the source.

4.2.4.3 Embedded Network Library

Under review, not yet designed.

5 Glossary

AWS - Amazon Web Services

CDC - Communication Class Device

FS - Full Speed

IoT - Internet of Things

MQTT - Message Queuing Telemetry Transport, data message protocol

OSI - Open Systems Interconnection model

OTG - On the Go

PKCS #11 - defines a platform-independent API to cryptographic tokens,

Sockets - in general, specify an IP address and read/write data from the location

SPI - Serial Peripheral Interface

SRS - Software Requirements Specification

TLS - Transport Layer Security (encrypt/decrypt data)

USB - Universal Serial Bus

USB OTG - USB On The Go

VCP - virtual communication port

Wi-Fi - technology for radio wireless local area networking of devices based on the IEEE 802.11 standards. Wi-Fi is a trademark of the Wi-Fi Alliance

YAT - Yet Another Terminal