SMARTNET / SITENET
RBU FIRMWARE DESIGN DOCUMENT

2001-DES-0002 – 03

July 2024

Document No:	2001-DES-0002	Client Ref:	

Document revision history

Rev	Description	Date Approved	Author	Checked	Approved
2000-SPC-0009					
01	Updated AT commands	03-Feb-20	Alastair Knight		
01A	Underway revise throughout to reflect TDM structure change and other modification	30-Nov-20	David King		
02	Completion of document revision. Changes made throughout.	19-Aug-21	Peter Conolly		
03	Added section 3.4.2 Internal Message Structure Added module File mapping, section 8	23-May-22	Peter Connolly		
04	Added Data Interfaces, section 4.6	10-Nov-22	Peter Connolly		
2001-DES-0002					
01	Added battery management section 3.11	16-Jul-23	Peter Connolly		
02	Added to Program Data (bootloader structure) and Volatile Data Added PPU & PPU bootloader. Updated AT commands.	09-Jan-24	Alastair Knight		
03	Changed battery management section 2.11 and +AT Commands section	25-Jul-24	Thomas Liu		

CONTENTS

TABLES

FIGURES

1 FUNCTIONAL DESIGN

1.1 Test Modes

The NCU and RBU support six test modes that are invoked using AT commands over the DEBUG serial interface.

The commands take the form:

ATMODE=<mode number>

Where the mode number identifies the required test mode according to the table below.

Test Mode	Mode Number
OFF	0
RECEIVE	1
TRANSPARENT	2
TRANSMIT	3
MONITORING	4
SLEEP	5
NETWORK MONITOR	6

1.1.1 Test mode OFF

On start-up test mode is OFF by default. The NCU or RBU will enter its normal mode of operation.

1.1.2 Test mode RECEIVE

Test mode RECEIVE is invoked with the AT command ATMODE=1. On receipt of this command the unit will drop out of the radio mesh, if it was connected, and switch to a ‘listening’ mode. Any messages that it receives are decoded and sent to the DEBUG serial interface as ASCII characters.

The messages are displayed in the following format:

:[index] Rx [msg type] [source] [status] [chan]:[rssi]:[snr]::[dev] : [raw msg bytes]

index	Slot index in super frame.
msg type	The name of the message type e.g. “HEARTBEAT”, “FIRESIG” etc.
source	The node ID for the message sender. For heartbeat messages the slot number is shown.
status	Either “OK” if the message was received intact or “ERR” if there was a problem decoding it.
chan	Frequency channel.

rssi	The signal strength indicator for the received message.
snr	The signal to noise ratio for the received message.
dev	Frequency deviation.
Raw msg bytes	A hexadecimal display of the message contents parsed into bytes.

e.g. :[41680] Rx HEARTBEAT 72 OK 9:-83:22::-973 : [002440FF007D888CFE]

1.1.3 Test mode TRANSPARENT

Test mode TRANSPARENT is invoked with the AT command ATMODE=2. On receipt of this command the unit will drop out of the radio mesh, if it was connected, and reconfigure as a radio modem. Messages that are received from the AT serial interface are packaged for broadcast and transmitted by the LoRa radio. Messages that are received from the LoRa radio are decoded and sent to the AT serial interface as a sequence of raw bytes.

In this mode the output to the serial interface is not converted to ASCII characters. The raw message bytes are output. For this reason this mode is not suitable for terminal output and should be read using a purpose-written test application.

The test messages that are broadcast by the LoRa radio are limited to a payload of 13 bytes. The test application must deliver messages as a sequence of 13 bytes for successful broadcast. The NCU/RBU wait until 13 bytes have been received before packing the message for transmission. There is no restriction on the content of the 13 bytes.

This test mode is designed for testing the transfer of messages over the air. Two radio units should be connected to the test application, allowing messages to be sent and received over the air.

While in TRANSPARENT mode, the NCU/RBU will only transmit and receive the special test messages from the test application. All other received messages will be discarded.

1.1.4 Test mode TRANSMIT

Test mode TRANSMIT is invoked with the AT command ATMODE=3. On receipt of this command the NCU/RBU will transmit one test message per second over the LoRa radio until the unit is powered down. No messages will be received over the LoRa radio.

The transmitted messages have the following format:

Frame Type 4 bits	Source Address 12 bits	Payload 104 bits (13 bytes)	Reserved bits 4 bits set to '0'	System ID 32 bits

The frame type is always set to a value of 5 (Test message type).

The source address is the configured node ID of the NCU/RBU.

The payload is 13 bytes of test data (see below).

The reserved bits are always set to zeros.

The system ID is the configured system ID of the NCU/RBU.

The payload of the test message is generated internally by the NCU/RBU and is a sequence of ASCII characters with the following format.

Node <ID> <count>

Where <ID> is the NCU/RBU network address and <count> is an incrementing count, starting at 1 and incrementing for each message sent.

e.g. "Node 1 1234"

Any unused bytes are set to 0x00.

1.1.5 Test mode MONITORING

Test mode MONITORING is invoked with the AT command ATMODE=4. This mode is very similar to TRANSPARENT mode, but has two important differences.

Firstly, this mode will accept all known message types and report them over the AT serial interface.

Secondly, the serial report contains extra information about the message and includes the entire message frame, not just the payload.

The data is output to the serial interface as raw binary bytes, so this mode is not suitable terminal applications.

As with TRANSPARENT mode, the NCU/RBU will accept a 13 byte sequence over the AT serial interface and pack it for transmission over the LoRa radio.

The output format to the serial interface is as follows.

MT	SRC	STATUS	RSSI	SNR	MSG BYTES

MT is the message type, see the table below.

SRC is the source of the message. It is the node ID of the unit that transmitted the message.

STATUS will be 0 if the message was decoded correctly or an error code otherwise.

RSSI is the received signal strength for the message, supplied by the LoRa radio.

SNR is the signal to noise ratio for the received message, supplied by the LoRa radio.

MSG BYTES is a binary sequence containing the received message. For the dedicated test message, this will be the 13 byte payload only. For other message types the whole message is shown.

The message type (MT) can be one of the following.

MT Value (hex)	Description
0	Fire signal
1	Alarm signal
2	Fault signal
3	Output signal
4	Command message
5	Response message
6	Logon request
7	Status indication message
8	Firmware update message
9	Route Add request
A	Route Add response
B	Route Drop message
C	Test Mode command
D	Test message
E	State signal
F	Load Balance request
10	Load Balance response
11	Acknowledgement
12	Heartbeat
13	RBU Disable
14	Status signal
15	Output state request
16	Output state

17	Unknown
18	Alarm output state signal
19	Ping
1A	Battery status signal
1B	Add Node link
1C	Drop Node link
1D	Day/Night status

1.1.6 Test mode SLEEP

Test mode SLEEP is invoked with the AT command ATMODE=5. In this mode the NCU/RBU is placed into sleep mode. In this mode the unit uses minimal power, waking only to hold off the watchdog, to prevent the unit from being restarted.

The NCU/RBU remains in sleep mode until it is powered down.

1.1.7 Test mode NETWORK MONITORING

Test mode NETWORK MONITORING is invoked with the AT command ATMODE=6. This mode is similar to RECEIVE mode, but the node remains synchronised to the mesh.

1.2 Peer to Peer Mode

Peer to Peer (PP) Mode is a special mode for communicating with a Portable Programmer Unit (PPU) within earshot on a dedicated radio channel without the need of an established mesh network.

Although Wireless Firmware Update was abandoned during the development of the project, the PPU is a useful mean to communicate with a Radio Board Unit to modify its NVM-stored configuration and possibly perform diagnostics or testing.

PP Mode is enabled by default, but it can be disabled using the NV_PP_MODE_ENABLE_E NVM parameter through the ATPPEN command

After each start-up, except when explicitly requested, PP mode is activated, so that RBU devices periodically send a heartbeat-like radio packet that contains their serial number and state. The radio board embedded within the PPU would receive these packets and update a packet list internally. Upon an ATPPLST command from the PPU main microcontroller, the PPU radio board would return the list of available devices.

When in PP Mode, the RBUs are in LoRa RX continuous mode, except when transmitting. If they receive a radio packet from the PPU intended for them (based on the serial number in the destination field of the packet), then the indicated command is executed, and a response sent back to the PPU. For simplification, the layout of the command and output signal payload packet used by the mesh protocol was re-used for PP protocol, except that the serial number instead of the Mesh unit address is used to specify destination.

When RBU_PP_MODE_MAX_TIME_MS milliseconds since the last received PPU command expires, the unit resets and boots in normal mode to join the mesh network. Presently set to 180000 ms, thus 3 minutes.

PP Mode functionalities are implemented by file MM_RBUAppPPMode.c.

1.3 PPU-RBU Software

The PPU-RBU Software is a derivative build for a radio board used as part of a PPU device.

The PPU-RBU places the Semtech SX1273 radio modem IC in sleep mode and enters stop mode after start-up until it receives a request from the PPU to go active with an “ATSTATE=1\r\n” command.

When the PPU-RBU is in active state, it remains in LoRa RX continuous mode, to observe heartbeats, until it receives a command from the PPU main microcontroller to transmit a message to a remote unit. The ATCMD and ATOUTP commands are used to communicate with the PPU-RBU to send wireless messages to the unit specified by the destination field of these commands (as mentioned previously the serial number is used instead of the unit address).

On a unit exiting PP Mode, and in the case it cannot find a tracking node, the unit can be returned to PP Mode by mimicking an invalid rank heartbeat from the NCU in test mode. When the PPU-RBU receives the ‘ATPPBST+’ AT command it broadcasts this NCU heartbeat, and if any units within earshot receive this packet, they will reset and re-enter PP Mode.

1.4 Built-In Test

Built in test (BIT) is carried out on the major interfaces of the radio board on start-up.

The device combination configuration (DC) setting is used to identify which devices should be tested.

If the DC indicates that a plug-in device should be fitted (sensor or beacon) the radio board requests the device type and class. This tests the communication link and that the plug-in type matches the programmed configuration.

Where a sound and visual indicator (SVI) is expected, the radio board requests the SVI serial number. This checks for the presence of the device and that the communication link is working.

All radio board configurations carry out a check of the Semtech ID (silicon revision). This checks the communications with the radio module and that the expected radio device is fitted.

If any of the BIT tests fail, the radio board indicates this by flashing the amber status LED once every ten seconds. On joining the mesh, fault messages are issued to the control panel.

The BIT tests can also be initiated (IBIT) using the ATBIT+ command over the command interface.

1.5 Stop Mode

The STM32L4 MCU is placed into Stop 2 mode to reduce power consumption to a minimum as often as possible.

The MCU is woken periodically through the Low Power Timer (LPTIM) timer comparator interrupt to schedule the Time Division Multiplex (TDM) structure. The action of the TDM slot is performed and we then reload the LPTIM comparator for the next TDM action and then return to stop mode.

The MCU can leave stop mode also when an external interrupt is triggered such as for call points or tamper switches.

Stop mode is triggered from the RTX RTOS idle daemon, which runs only when no interrupt service routines are active, and the RTOS thread has entered sleep mode awaiting an event or delay. We check also that there is no ongoing activity from the Head interface or the serial port such as an in-progress transmission or an open AT session.

Every time we enter stop mode we:

1. Suspend the RTOS
2. Disable interrupts
3. Disable all supported UART ports and configure their RX pins in GPIO interrupt mode to capture a start-bit (falling edge) meaning that an external device has initiated a new communication.
4. Capture the value of the LPTIM timer as a timestamp
5. Set unneeded pins to analogue mode to reduce quiescent current
6. Disable the clock PLL and set the core to run from the High Speed Internal (HSI) oscillator

When we wake we:

1. Restore the MCU clocks
2. Restore the pin configuration to that prior to stop mode
3. Re-adjust the system timing according the time spent in stop mode
4. Restore interrupts
5. Restore the RTOS to full operational mode

When the UART is the wake source we wait 500ms (AT_MODE_SEQ_TIMEOUT_US) for data to arrive. If during this period we receive a valid AT session token "+++<CR><LF>" then we keep the system awake for 10s (AT_MODE_EXIT_TIMEOUT_US) for the AT command to be fully received. The 10s delay is timed from the last UART activity.

1.6 Comms

1.6.1 General Command Syntax

All AT Commands must commence with the characters "AT" and end with a carriage return character (ASCII decimal value 13 – hexadecimal 0x0D) and line feed character (ASCII decimal value 10 – hexadecimal 0x0A). All character strings used to express commands or form replies are in ASCII format. An entire command sequence will have the following format:

<Prefix><Command><Type/Action><Data><CR><LF>

Where,

<Prefix> is the characters "AT" (without the quotation remarks)

<Command> is the command string given in this document

<Type/Action> "=" for Write, "?" for Read, or "+" for special command

<Data> values associated with the command

<CR> is the carriage return character (ASCII character decimal value 13 – hexadecimal 0x0D)

<LF> is the line feed character (ASCII character decimal value 10 – hexadecimal 0x0A)

Character “,” (Comma) can be used, if required, to separate parameters within the Command or Data strings, however, no comma should be found preceding or following the Type/Action field.

Once a command is received by the device it will processed. Once a command is processed, a response will be generated. A response will have the following format:

<Command>:<Space><Response><CR><LF>

Where,

<Command> is the command string

<Space> is the “space” character (ASCII character decimal value 32 – hexadecimal 0x20)

<Response> is the command specific response (depending on command) or one of the following strings (without the quotation marks):

“**OK**”, if the command is recognised and executed successfully

“**ERROR<,*Code*>**”, if it was an unrecognised command or after an unsuccessful execution. The “,*Code*” section of the response is optional. By default, this is not present unless explicitly declared in this document.

<**CR**> is the carriage return character (ASCII character decimal value 13 – hexadecimal 0x0D)

<**LF**> is the line feed character (ASCII character decimal value 10 – hexadecimal 0x0A)

Example 1 – Write Command: Success

[TX] – ATFREQ=1<CR><LF>

[RX] – FREQ: OK<CR><LF>

Example 2 – Write Command: Error

[TX] – ATFREQ=81<CR><LF>

[RX] – FREQ: ERROR<CR><LF>

Example 3 – Read Command

[TX] – ATFREQ?<CR><LF>

[RX] – FREQ: 1<CR><LF>

1.6.2 DEBUG Serial Interface

The NCU and RBU support the output of debug information from the USART4 port of the STM32 processor. The output from this port can be viewed using a terminal application connected to header J5 on the radio board.

This port also supports AT commands that are detailed in **Section 6**, of this document, allowing some properties of the unit to be queried or set.

The port is configured as follows:

Baud Rate = 2000000

Data Bits = 8

Parity = None

Stop Bits = 1

Flow Control = None

1.6.2.1 DEBUG Message Considerations

All messages output on this interface are preceded with an identification code. Each message is terminated with a carriage return character (ASCII decimal value 13 – hexadecimal 0x0D) and a line feed character (ASCII decimal value 10 – hexadecimal 0x0A). An entire debug message sequence has the following format and does not exceed 384 ASCII format characters.

<IDCode><Space><Message><CR><LF>

Where,

<IDCode> is the message specific code. These can be found in the table below

<Space> is the "space" character (ASCII character decimal value 32 – hexadecimal 0x20)

<Message> is the message string in ASCII format

<CR> is the carriage return character (ASCII character decimal value 13 – hexadecimal 0x0D)

<LF> is the line feed character (ASCII character decimal value 10 – hexadecimal 0x0A)

Identification Codes

Code	Description
+SYS:	System or CPU related messages
+INF:	Informative messages
+BIT:	Built in Test specific messages
+ERR:	Error message
+DAT:	Terse data interpreted by development software

Example – Informative message printed on debug interface

[RX] – +INF: Interface task created successfully on unit power-up<CR><LF>

1.6.3 PPU Serial Interface

The RBU supports a serial connection to a Portable Programmer Unit (PPU) via the USART1 port of the STM32 processor. The output from this port can be viewed using a terminal application connected to header J4 on the radio board.

This port supports the AT commands that are detailed in section 6.4, **Table 23 AT Commands**, of this document, allowing some properties of the unit to be queried or set.

The port is configured as follows:

Baud Rate = 115200

Data Bits = 8

Parity = None

Stop Bits = 1

Flow Control = Hardware (RTS/CTS)

1.6.4 I2C Interface

The RBU communicates with the Sound and Visual Indicator (SVI) via I2C, with the RBU as master.

The SVI has settings that configure its behaviour. The settings and their register addresses are defined in document HKD-17-0153-D.

The I2C driver provides functions to read/write SVI registers via the I2C interface.

2 SOFTWARE ARCHITECTURE

2.1 Radio Board Software Architecture

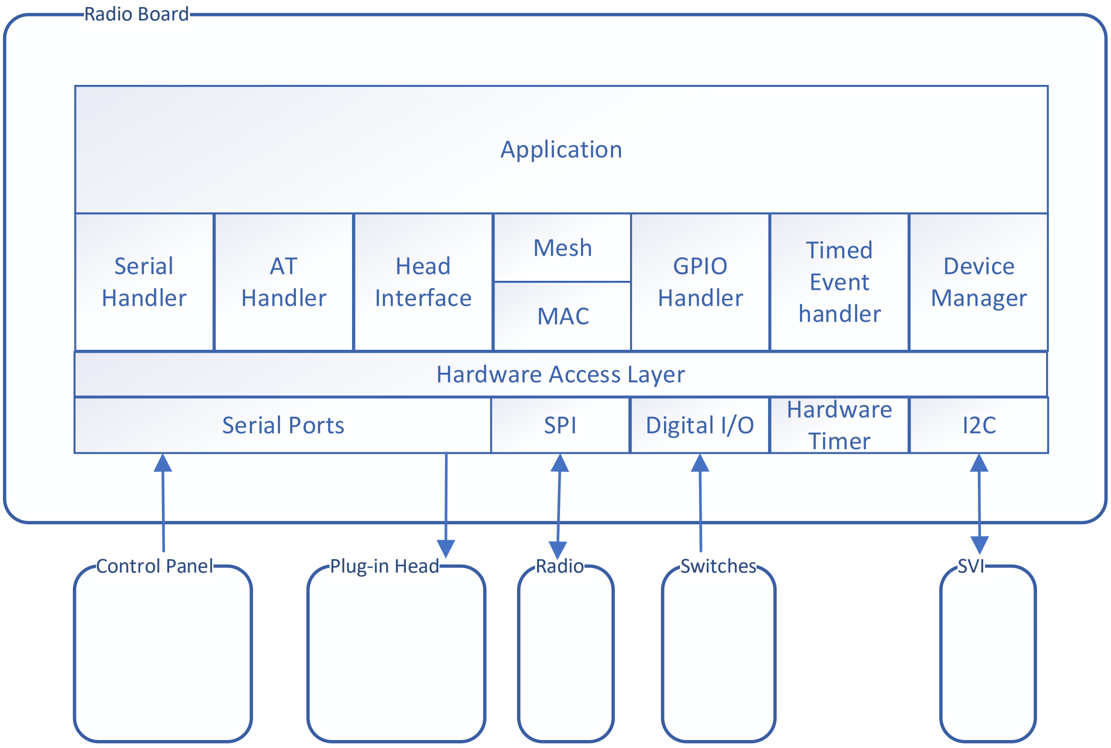


Figure 1:Radio Board Software Architecture

Name	**Description**
Application	The Application code carries out all of the system set-up when powered on. Its primary function is to implement the RBUs required behaviour. It processes commands and sensory inputs and takes the appropriate action.
Serial Handler	This module monitors all serial inputs and forwards received data to the Application
AT HAndler	This module interprets the commands from the serial inputs And forwards them to the Application module.
Head interface	This module manages the interface to the plug-in head via a serial link. It interprets commands from the Application and formats the response from the Head. It manages the wake-up process when the Head sends a wake-up signal.
Mesh	This is the top half of the radio communications stack. It performs mesh forming and synchronisation, session management, message packing and routing. It receives incoming messages and sends outgoing messages via the MAC module
MAC	This is the bottom half of the radio communications stack. It is responsible for the transfer of messages to and from the radio within the constraints of the time division multiplexed (TDM) protocol. The TDM slot times are calculated relative to the

	synchronisation point calculated by the MESH module. This module manages the frequency hopping for all logical channels. This module has hard real-time constraints and runs at a higher priority than all other modules.
GPIO Handler	This module performs regular checks on digital switched inputs and generates change notifications for the Application module.
Timed Event handler	This module manages events that require timing services at a higher resolution than the Application event cycle.
Device Manager	This is a sub-component of the Application module. It is responsible for the configuration of the radio board in accordance with the programmed device combination. It offers an interface between the Application and the configured device interfaces.

Table 1 Software Modules

2.2 Software Hierarchy

The high-level functionality of the radio board software is performed by the Application module. It is supported by the other modules which have more specific responsibilities (Table 4).

The Application runs in the main thread. On start-up it creates the threads for all of the other modules before entering its own cycle of operation. The application can gather data, make decisions and implement the required behaviour through a few chains of dependency with the other modules as described below.

2.2.1 Serial Communication Hierarchy

The Application module supports a command interface via the serial ports. This enables the radio board software to be configured our updated via serial link. Once operational, the NCU communicates with the control panel via serial link. The command interface is not used by the remote radio units, but they do use the low power serial interface for communication with plug-in sensors. This interface has complex behaviour which the Application delegates to the Head interface module.

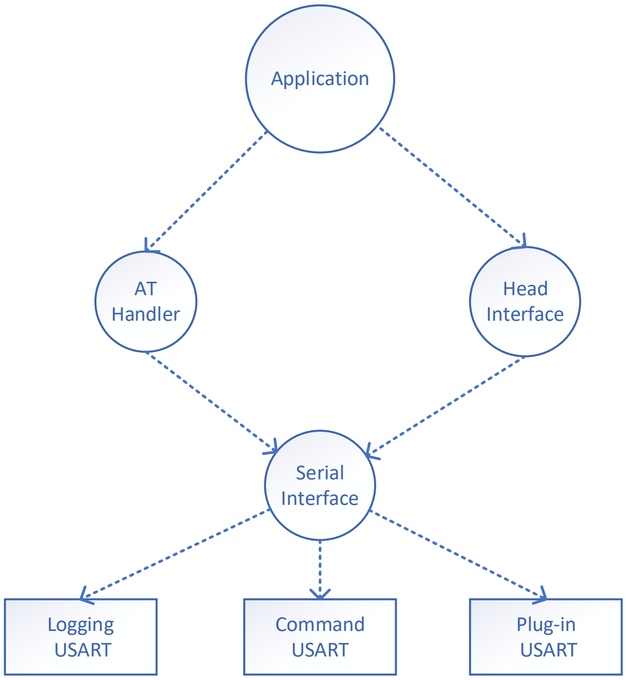


Figure 2: Serial Interface Dependency Chain

2.2.1.1 Command USART

This hardware interface is utilised by the radio board software for transfer of commands from the control panel or configuration software. Commands are interpreted by the AT Handler module (named so because all commands start with the sequence "AT"). Decoded commands are forwarded to the Application module for implementation.

2.2.1.2 Logging USART

This hardware interface is primarily used for outputting log messages in plain text which can be recorded with a standard terminal application.

2.2.1.3 Plug-in USART

This hardware interface id dedicated to communications with plug-in sensors, beacons or sounders. Plug-ins are exclusively managed by the Head Interface Module via the Serial interface module.

2.2.1.4 Serial Interface Module

This software module runs concurrently and manages the transfer of data between internal data buffers and the USART hardware. It communicates with the hardware via the Hardware Application Layer library, a software library supplied by the manufacturer of the microcontroller.

2.2.1.5 AT Handler

This software module reads data from the receive buffers, populated by the Serial Interface module, and interprets it into internal commands which are sent to the Application Module. Responses received from the Application are placed into the output buffers for transfer by the Serial Interface module.

2.2.1.6 Head Interface

This software module handles the interface with plug-in sensors, sounders and beacons. It communicates commands from the Application to the plug-in using the plug-in protocol. It also accepts messages from the plugin and raises the appropriate messages for the Application module. It communicates with the plug-in via the serial interface module.

2.2.1.7 Application

This software module is the main controller where the operational decisions are made.

2.2.2 GPIO Hierarchy

The GPIO module runs concurrently and is responsible for monitoring the digital inputs and informing the Application module when an input changes. It calls upon the Timed Event module for switch debouncing protection.

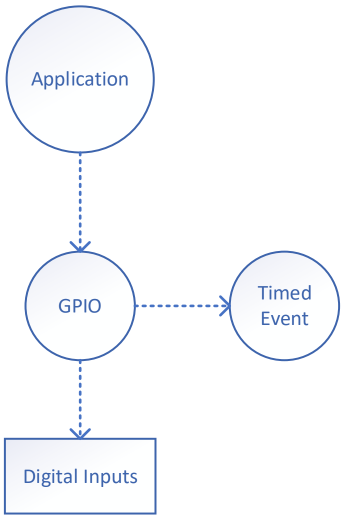


Figure 3: GPIO Dependency Chain

2.2.2.1 Digital Inputs

These are the hardware pins on the microcontroller that are connected to monitored digital circuits. There are several digital inputs that are managed by the GPIO module including tamper switches and fire call points. Pin states are determined using the Hardware Application Layer library, a software library supplied by the manufacturer of the microcontroller.

2.2.2.2 GPIO

This software module regularly checks the state of several digital inputs. Emergency call points (Fire and first aid buttons) are configured to raise an interrupt in the micro controller. There are insufficient resources to give all inputs an interrupt channel, so the GPIO module uses regular polling to determine the state of tamper switches and the battery monitor circuit.

The interrupt driven events trigger the Timed Event module.

2.2.2.3 Timed Event

This software module provides timed services where high precision is required. It is pre-programmed with timed events which are triggered via a mutex protected interface. The GPIO module depends upon this module to provide a debouncing service for the emergency switches.

2.2.2.4 Application

This software module is the main controller where operational decisions are made in response to the GPIO events.

2.2.3 Radio Mesh Hierarchy

Two modules work together to perform the radio mesh interface. Time-critical functions are performed by the Media Access Control module (MAC). Less time-critical functions are performed by the Mesh Module, which interfaces with the Application module.

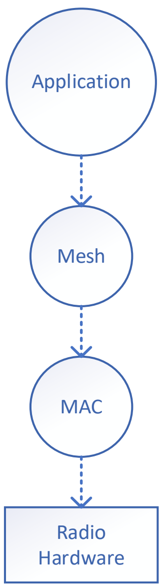


Figure 4: Radio Mesh Interface Dependency Chain

2.2.3.1 Radio Hardware

The radio hardware is contained in a separate integrated circuit, connected to the microcontroller via the SPI interface. SPI communications are performed using the Hardware Application Layer library, a software library supplied by the manufacturer of the microcontroller.

2.2.3.2 MAC

The Media Access Control module runs concurrently at the highest priority in order to manage the hard real-time constraints of implementing the mesh protocol. Its primary role is to transfer messages between the radio hardware and the radio board software. To do this it must ensure that it transmits and receives at the appointed times to satisfy the protocol.

2.2.3.3 Mesh

The mesh module unpacks received messages and sends their content to the Application module for processing. It receives data packets from the Application module and packs them for transmission. The Mesh module runs concurrently in order to perform automated mesh synchronisation and session management, without intervention from the Application module.

2.2.3.4 Application

The Application module receives messages from the radio mesh and actions them on the radio board. Events that require a radio message to be sent are passed to the Mesh module for packaging and transmission.

2.3 Concurrency Model

The radio board software is comprised of the modules listed in table 4. Each of these modules runs concurrently, with the exception of the Device Manager, which is a subcomponent of the Application Module.

The modules are designed to be event driven and remain dormant until called upon. Events are passed to modules via an OS Queue. The operating system suspends a module when its queue is empty and restores it when a message is placed into the queue. The module processes the message then is re-suspended.

If no modules are ready to run the operating system invokes the idle thread which holds-off the hardware watchdog and puts the microcontroller into stop mode, to reduce power consumption. The microcontroller remains in stop mode until woken by an event.

If a module becomes ‘locked’ the idle thread will never be invoked. This results in a failure to hold-off the hardware watchdog and the microcontroller is reset.

The Media Access Control (MAC) module is not queue driven. It is responsible for meeting the hard real-time requirements of the mesh protocol (time division multiplexed) and is therefore timer driven. After completing each event, the MAC module determines its own wake up time for its next event, making use of the low power hardware timer which continues to operate when the microprocessor is in stop mode.

The MAC module reads messages that arrive over the radio link and drops them into Mesh Module’s queue. The Mesh queue decodes the message after the MAC suspends. It handles Mesh synchronisation messages locally; all other messages are put in the Application Module’s queue for processing.

The Application Module contains all of the decision-making and is the focal point for data derived in the other modules.

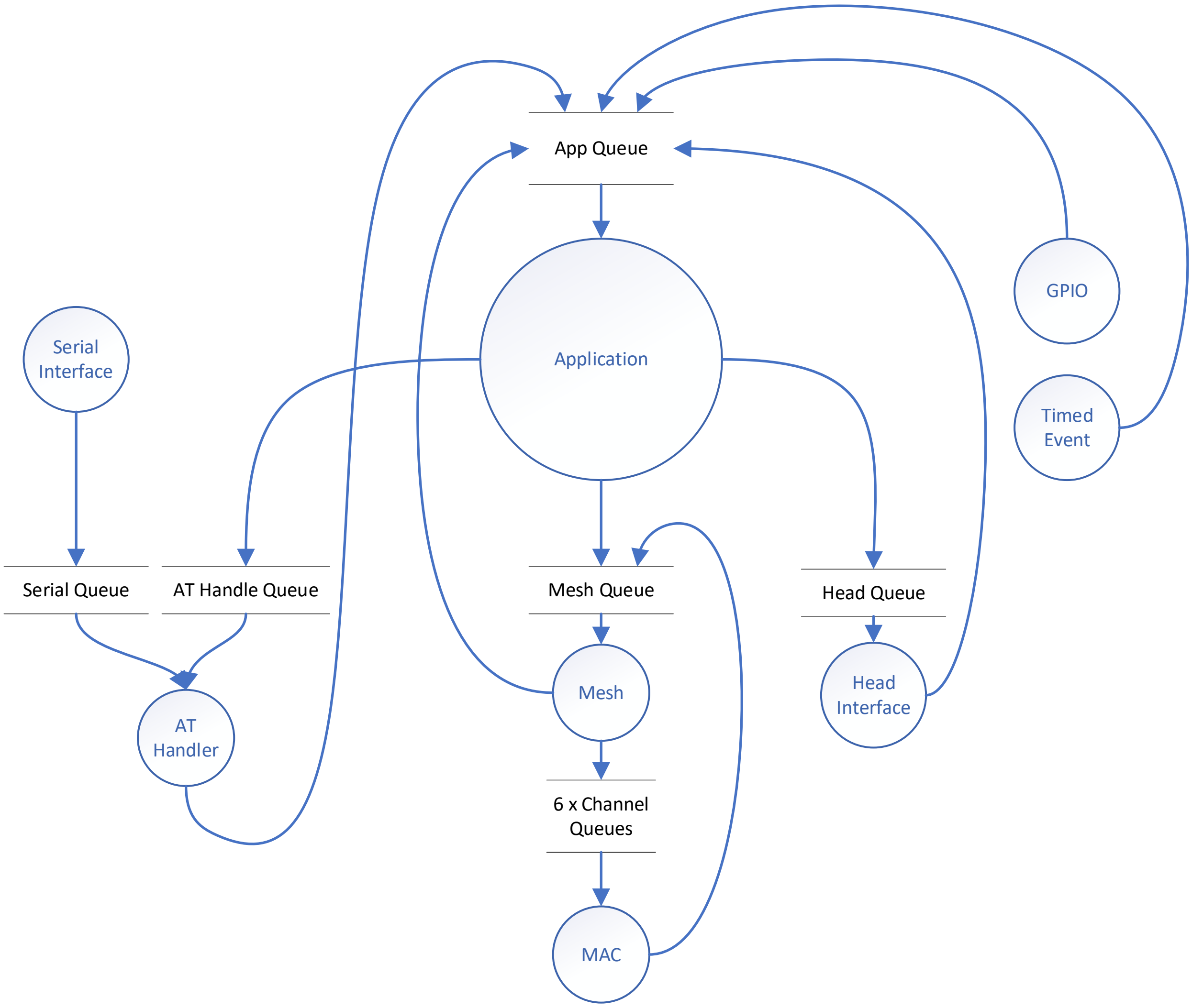


Figure 5: Radio Board Concurrency Model

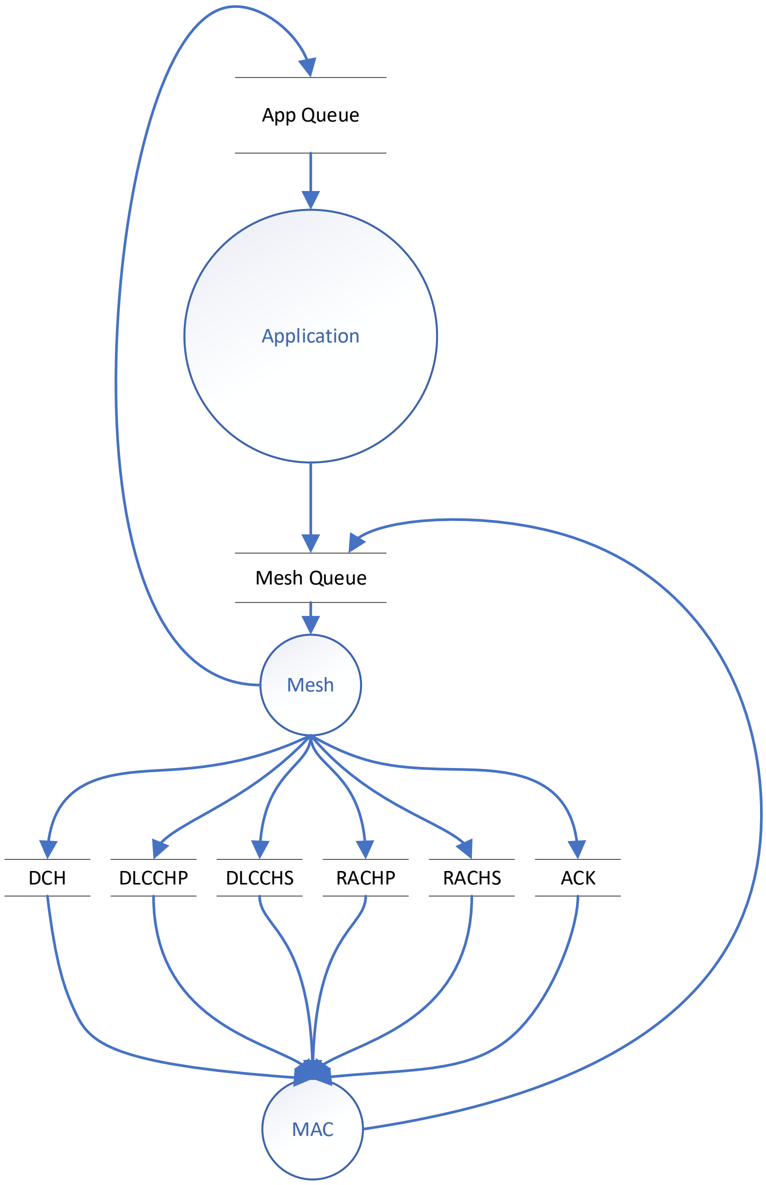


Figure 6: Mesh Protocol Data Flow Diagram

2.4 Communications and Messaging System

The concurrent modules that comprise the radio board software communicate using message queues to pass packets of data between modules.

External interfaces include USART interfaces for the control panel, programming, logging and plug-in interfacing. Standard SPI and I2C interfaces connect the microcontroller to the radio chip and internal sounder respectively.

2.4.1 Software Interfaces

Communication between the software modules is achieved by message structures, deposited in the appropriate queue for the destination module. The internal inter-process message queues are described in the table below and represented in figures 6 and 7 above.

Interface	Description
App Queue	The message queue for the main application. Receives radio messages and status messages from the MESH module, sensor events from the Head interface and GPIO module.
Mesh Queue	The message queue for the Mesh Module. Receives incoming radio messages from the MAC and messages for transmission from the Application
Head Queue	The Head Queue receives messages from the Application when some interaction with a plug-in head is required. The Head interface module reads from the queue and implements the necessary action.
DCH	Queue for outgoing synchronisation messages
DLCCHP	Queue for outgoing Priority downlink messages
DLCCHS	Queue for outgoing Standard downlink messages
RACHP	Queue for outgoing Priority data messages
RACHS	Queue for outgoing Standard data messages
ACK	Queue for outgoing message acknowledgements

Table 2 Key Software Interfaces

2.4.2 Internal Message Structure

An internal message is defined as one that is used to transfer data from one module to another. A standard message structure is used throughout the design.

```
typedef struct
{
  CO_MessageType_t Type;
  CO_MessagePayload_t Payload;
} CO_Message_t;
```

Figure 7: Standard Internal Message Structure

The message ‘Type’ identifies the ‘Payload’ contents so that it can be correctly interpreted by the receiving module.

The payload contains the message data and, if required, some supporting metadata. For example, a message received over the radio will be sent to the application module. The payload will include the received message and metadata such as the frequency, signal strength, timestamp etc.

The message data can be one of many structures, defined in CO_Messages.h. One of the most common is the command structure, used for distributing commands around the software modules.

```
typedef struct
{
  uint8_t TransactionID;
  uint16_t Source;
  union{
    struct
    {
      uint16_t Destination;
      uint16_t SourceUSART;
    };
    uint32_t UnitSerno;
  };
  uint8_t CommandType;
  uint8_t Parameter1;
  uint8_t Parameter2;
  uint8_t ReadWrite;
  uint32_t Value;
  uint8_t NumberToSend;
} CO_CommandData_t;
```

Figure 8: Command Message Structure

It is common practice to use enumerated fields where a parameter has a predefined range of values. For example, the '*CommandType'* field in the above structure has a limited set of values that the RBU will understand. This set of values is captured in a defined enumeration structure. In this case it is the *ParameterType_t* enumeration, shown here, reduced for brevity.

```
typedef enum
{
  PARAM_TYPE_ANALOGUE_VALUE_E,        //0
```

```
    PARAM_TYPE_NEIGHBOUR_INFO_E,         //1
    PARAM_TYPE_STATUS_FLAGS_E,           //2
    PARAM_TYPE_DEVICE_COMBINATION_E,     //3
  .
  .
  .
    PARAM_TYPE_BATTERY_TEST_E,           //63
    PARAM_TYPE_PRODUCT_CODE_E,           //64
    PARAM_TYPE_PPU_MODE_ENABLE_E,        //65
    PARAM_TYPE_DEPASSIVATION_SETTINGS_E, //66
    PARAM_TYPE_ENTER_PPU_MODE_E,         //67
    PARAM_TYPE_MAX_E
  } ParameterType_t;
```

Note the final entry is a ‘MAX’ marker. This is used to conveniently range-check a received command. The *CommandType* field must hold a lower value than PARAM_TYPE_MAX_E to be valid. This is a common practice throughout the software for message fields with similar constraints.

2.4.3 Command USART

The command USART carries two-way messaging between the radio board and the control panel or configuration programmer. The Application sends messages to the AT Handler, which put them into a send buffer. The Serial Interface module utilises the microcontroller direct memory access (DMA) hardware to transfer the data to the USART hardware.

Incoming messages are transferred using DMA into a receive buffer. The Serial Interface module reads from the receive buffer and constructs message packets which are sent to the AT Handler for decoding.

During stop mode, all USART hardware is shut down to save power. The receive pin is reconfigured to generate a wake-up interrupt, bringing the microcontroller out of stop mode and reconfiguring the USART to receive the message.

2.4.4 Logging USART

The Logging USART operates in an identical fashion to the Command USART, above, but its primary role is to output logging messages posted by the radio boards software.

2.4.5 Plug-in USART

The Plug-in USART is used for communications between a plug-in head, which could be a sensor or an output device. The Serial Interface module is utilised to transfer data to and from the USART hardware into software buffers. The Head Interface module reads and writes to these buffers.

2.4.6 SPI Interface

The microcontroller communicates with the radio hardware via the SPI interface. Under control of the MAC module, data packets are transferred to and from the SPI hardware using the HAL library functions, supplied by the manufacturer of the microcontroller.

2.4.7 I2C Interface

The microcontroller communicates with the internal sounder via the I2C interface. Under control of the Application module, data is transferred to and from the I2C hardware using the HAL library functions, supplied by the manufacturer of the microcontroller.

2.5 System Interrupts

The following sections describe the interrupt handling in the radio board software.

2.5.1 Fire MCP Interrupt

The Fire MCP interrupt is triggered on both rising and falling edges of the digital signal from the call point button. The interrupt service routine (ISR) is implemented by function MM_FireMCPInputIrq, defined in file MM_Interrupts.c.

The ISR triggers the debounce event in the Timed Event module by calling the Mutex protected function TE_FireMCPStateChange(new_state).

2.5.2 First Aid MCP Interrupt

The First MCP interrupt is triggered on both rising and falling edges of the digital signal from the call point button. The interrupt service routine (ISR) is implemented by function MM_FirstAidMCPInputIrq, defined in file MM_Interrupts.c.

The ISR triggers the debounce event in the Timed Event module by calling the Mutex protected function TE_FirstAidMCPStateChange(new_state).

2.5.3 Head Wake-up Interrupt

When the microprocessor is in stop mode, the USART hardware for plug-in communication is disabled and the receive pin is reconfigured to generate and interrupt on both rising and falling edges. If the plug-in has an event to report it signifies this to the radio board by pulsing the receive line low for a fixed time period. Each logic transition results in a call to the interrupt service routine (ISR) MM_HeadWakeUpIntIrq, defined in file MM_PluginInterfacetask.c

The ISR records the time of each interrupt and, on recognising the wake-up pulse, it puts a message into the message queue for the Head Interface module for processing under normal priority. The Head Interface module reconfigures the serial port and queries the head.

2.5.4 Radio DIO0 Interrupt

When the radio hardware has received a message it interrupts the microcontroller by setting its DIO0 output pin to a logic low. This generates an interrupt in the microcontroller, waking it from stop mode. The interrupt service routine(ISR), SX1272OnDio0Irq, is defined in file sx1272.c, which is part of the library code supplied by the radio chip's manufacturer. The ISR handles the transfer of the message into the microcontroller memory then calls the registered callback function OnRxDone (if in receive mode) or OnTxDone (if in transmit mode), defined in MM_MACTask.c. OnRxDone releases a semaphore to activate the MAC module, which processes the message after the ISR has completed.

2.5.5 Radio DIO1 Interrupt

The radio hardware signifies that the receiver has timed out without receiving an expected message it signifies this by setting its DIO1 pin to a logic low. This generates an interrupt in the microcontroller, waking it from stop mode. The interrupt service routine(ISR), SX1272OnDio1Irq, is defined in file sx1272.c, which is part of the library code supplied by the radio chip's manufacturer. The ISR calls the registered callback function OnRxTimeout, defined in MM_MACTask.c. OnRxTimeout releases a semaphore to activate the MAC module, which processes the message timeout.

2.5.6 Radio DIO3 interrupt

The radio hardware signifies the completion of a channel activity detection (CAD) action by setting its DIO3 pin to a logic low. This generates an interrupt in the microcontroller, waking it from stop mode. The interrupt service routine(ISR), SX1272OnDio3Irq, is defined in file sx1272.c, which is part of the library code supplied by the radio chip's manufacturer. The ISR calls the registered callback function OnCadDone, defined in MM_MACTask.c. OnCadDone releases a semaphore to activate the MAC module, which processes the CAD signal.

2.5.7 LPTIM Timer Interrupt

The low power timer (LPTIM) continues to function while the microprocessor is in stop mode.

The timer is programmed by the MAC module to wake up the microprocessor at a precise time to implement the next mesh protocol event. On completion of the event the MAC module calculates the next wake-up time and reprograms the LPTIM timer.

On reaching the programmed time, the LPTIM generates an interrupt that is handled by interrupt service routine MM_MAC_TimerISR, defined in file MM_MACTask.c. The ISR releases a semaphore that schedules the MAC module to run when the ISR completes.

2.5.8 Application Module Timed Interrupt

The Application module performs periodic tasks that must be scheduled at regular intervals.

This is achieved by means of a software timer that the Application module programs to interrupt at the required interval. On expiry of the software timer and interrupt is generated that calls interrupt service routine (ISR) PeriodicTimerCallback, defined in MM_ApplicationCommon.c, which puts a message into the input message queue of the Application module.

2.5.9 Head Interface Timed Interrupt

The Head Interface module performs periodic tasks that must be scheduled at regular intervals.

This is achieved by means of a software timer that the Head Interface module programs to interrupt at the required interval. On expiry of the software timer and interrupt is generated that calls interrupt service routine (ISR) HeadPeriodicTimerCallback, defined in MM_PluginInterfaceTask.c, which puts a message into the input message queue of the Head Interface module.

2.6 Data Storage

All of the data used on the radio board is stored in the internal memory of the microcontroller and is distributed as follows.

2.6.1 Program Data

The firmware for the radio board is stored in FLASH memory, located at address range 0x08000000 to 0x0807DFFB (516 KB).

- Serial bootloader at 0x08000000 to 0x08003FFF (16 KB available)
- PPU bootloader at 0x08004000 to 0x0801FFFF (112 KB available)
- Main application at 0x08020000 to 0x0807DFFB (375 KB available)
    - A 32-bit checksum is generated for the installed firmware and stored in FLASH memory at location 0x0807DFFC.

2.6.2 Configuration Data

Configuration data is stored in emulated EEPROM at address range 0x0807E000 to 0x0807FFFF (8 KB). This is FLASH memory managed by library code, supplied by the microcontroller manufacturer.

2.6.3 Volatile Data

Volatile data is stored in RAM. There are two RAM areas on the microcontroller:

1. System data is stored in the main RAM area of 96 KB, located at address 0x20000000 to 0x20017FFF. This is used for system stacks and global variable data. The last 144 bytes of this space is not initialised on startup and contains:
    - rbu_pp_mode_request (32 bits)
    - rbu_pp_master_address (32 bits)
    - reprogram_request_status (32 bits)
    - sw_reset_msg_indication_flag (32 bits)
    - sw_reset_debug_message (126 bytes)
2. A second RAM area of 512 x 44 = 26624 B, located at address 0x10000000 is utilised by the Mesh component for data related to the current session on the Mesh.

2.7 Logging

MCU logging will use the debug serial port. The following reporting levels will be supported:

- Level B, Debug build only
- Level A, Debug and Release builds

2.8 Folder Structure

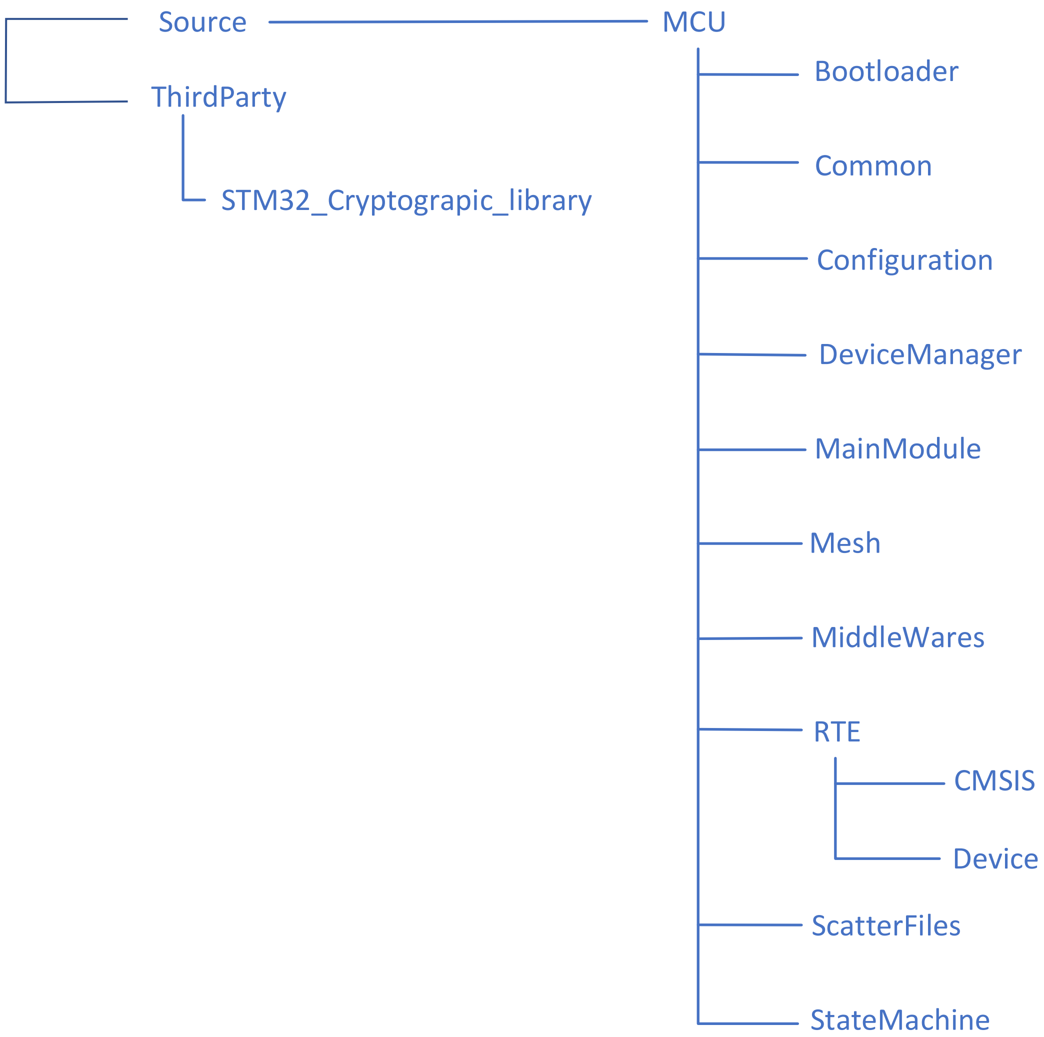


Figure 9: Code Repository Structure

Directory Name	Description of Contents
MCU / MainModule	Main routine for MCU processor, initialises the board having the MCU
MCU / StateMachine	The main state machine for the MESH module message handling
MCU / Mesh	Radio mesh protocol

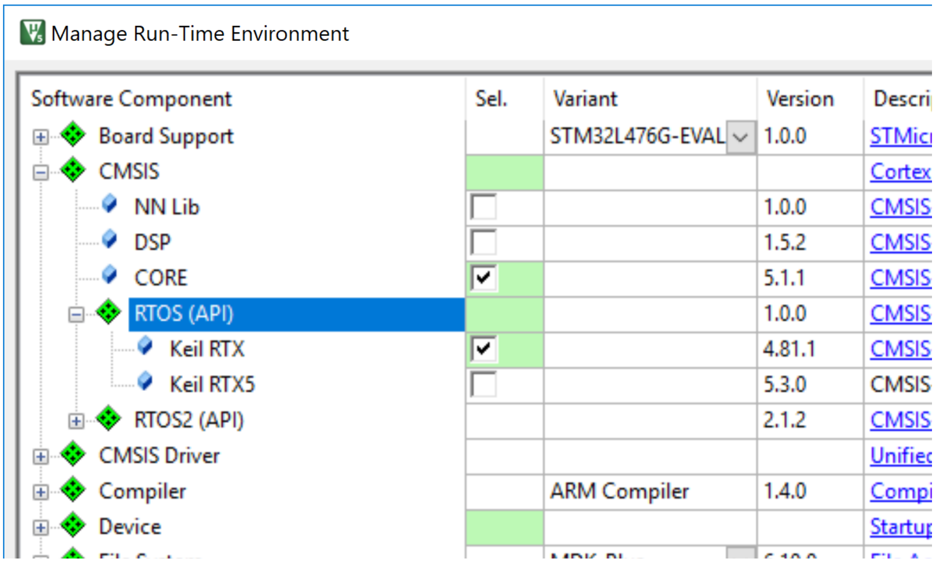
MCU / DeviceManager	Device drivers and interrupt service routines
MCU / Configuration	Device and functional element configuration
MCU / Common	Common defines and functions
MCU / Bootloader	Bootloader files
MCU/RTE/Device	STM library files for STM32L4 device

Table 3 Description of Folders

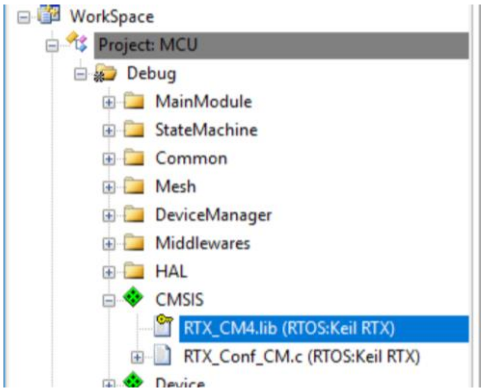
2.9 Third Party Software

2.9.1 RTX Operating System

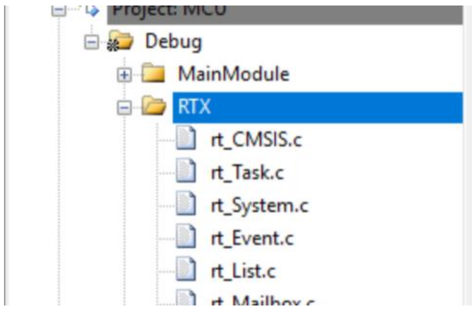
The software project used for the STM32L4 MCU based hardware uses the Keil RTX Real-Time Operating System. RTX is fully integrated in the uVision5 IDE, and is activated through the Run-Time Environment built-in utility as shown below:



The utility allows the RTX to be included in the project as a library without the need to recompile the source files and to avoid any unwanted modification of the RTOS. File RTX_Conf_CM.c allows configuration of RTX aspects by manual edit or using the Configuration Wizard.



Note that the RBU software project used for the STML0 MCU based hardware included the RTX source files and compiled them with rest of the SW.



2.9.2 STM32 device drivers

These were obtained from the ST STM32Cube suite.

2.9.3 Semtech driver

This is a software driver for the SX1273 IC. It was obtained from Semtech.

2.9.4 STM32 Cryptographic Library

Manufacturer’s library files for data encryption on STM32 devices.

2.10 Tasks

The table below shows the tasks used in the software.

Task Name	Priority	Description
MAC	Above Normal	Mac protocol
RBU Application	Normal	The application code for the RBU

Task Name	Priority	Description
NCU Application	Normal	The application code for the NCU
Mesh	Normal	The main state machine for the mesh protocol network and Radio Resource Control (RRC) layers
Head interface	Normal	Comms stack for head interface
GPIO	Normal	Handler for GPIO inputs
Config Serial	Normal	Serial interface stack
AT Handle	Normal	AT command handler
Timed Event	Normal	Manages high speed timed events e.g. Switch debouncing.
Idle	Low	OS idle task

Table 4 Task List

2.11 Battery Management

The Cygnus2 radio board software has two battery management procedures, SiteNet and SmartNet. The device configuration setting determines which procedure is invoked.

2.11.1 SiteNet Battery Management

SiteNet devices are powered by a 9V battery pack and two 3V backup cells connected in series to provide a 6V backup voltage.

The SiteNet battery management algorithm performs voltage tests on the primary and backup batteries in isolation, with a test load applied.

Best efforts are made to manage the case where the primary or backup batteries are missing, in which case a limited test is performed for the battery that is fitted.

The test is run over an extended period and a state machine, driven once per second by the main application, is used to manage the required steps for the test. The state machine for SiteNet devices is shown below.

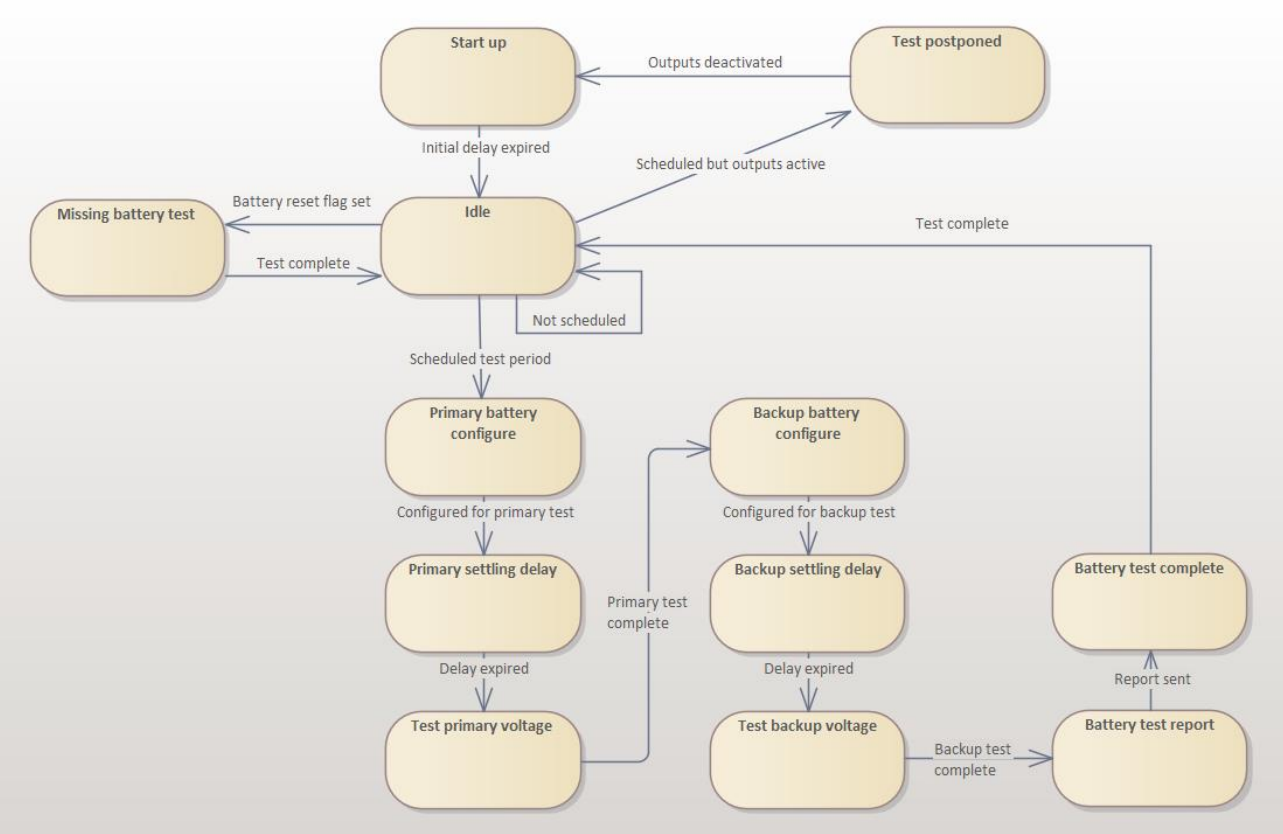


Figure 10 : SiteNet Battery Management State Diagram

2.11.2 SiteNet Battery Test Procedure

The SiteNet battery test procedure is detailed below.

1. Switch OFF the primary battery and switch ON the backup battery load. Wait for 5 seconds.
2. Take three voltage readings of the backup battery at 50ms intervals.
3. Switch OFF the backup battery load and switch ON the primary battery.
4. Switch OFF the backup battery and switch ON the primary battery load. Wait for 5 seconds.
5. Take three voltage readings of the primary battery at 50ms intervals.
6. Switch OFF the primary battery load and switch ON the backup battery.
7. If all three primary battery readings are below the voltage threshold, send a battery error message to the control panel and start the double amber LED pattern.
8. If all three backup battery readings are below the voltage threshold, send a battery error message to the control panel and start the double amber LED pattern.
9. Schedule the next battery test.

The process is represented by the flow chart below.

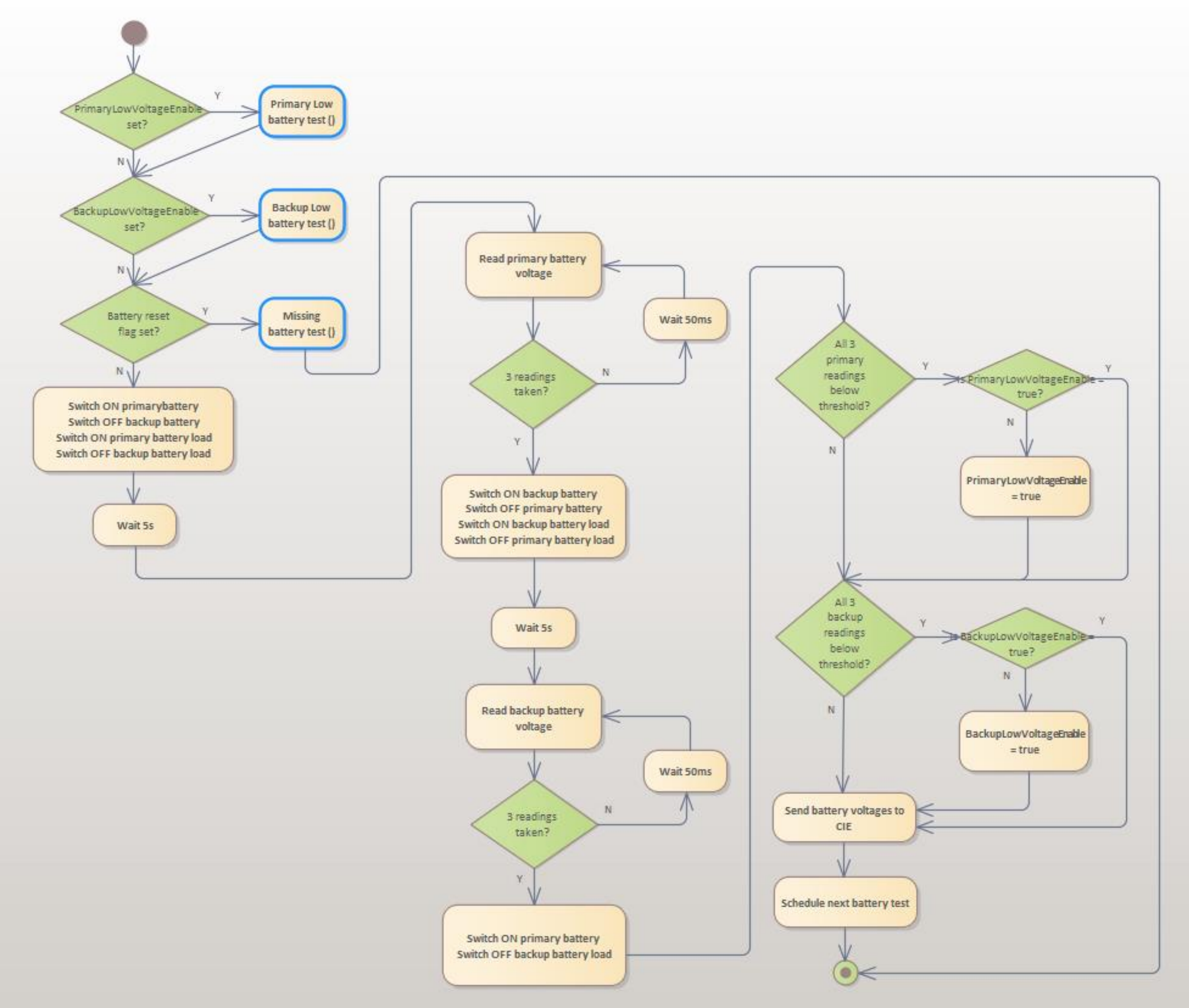


Figure 11 : SiteNet Battery Management Flow Chart

2.11.3 SmartNet Battery Management

SmartNet devices are powered by three 3.6V primary batteries and one 3V backup cell.

The SmartNet battery management algorithm performs voltage tests on the primary and backup batteries in isolation, with a test load applied.

Best efforts are made to manage the case where the primary or backup batteries are missing, in which case a limited test is performed for the battery that is fitted.

The test is run over an extended period and a state machine, driven once per second by the main application, is used to manage the required steps for the test. The state machine for SmartNet devices is shown below.

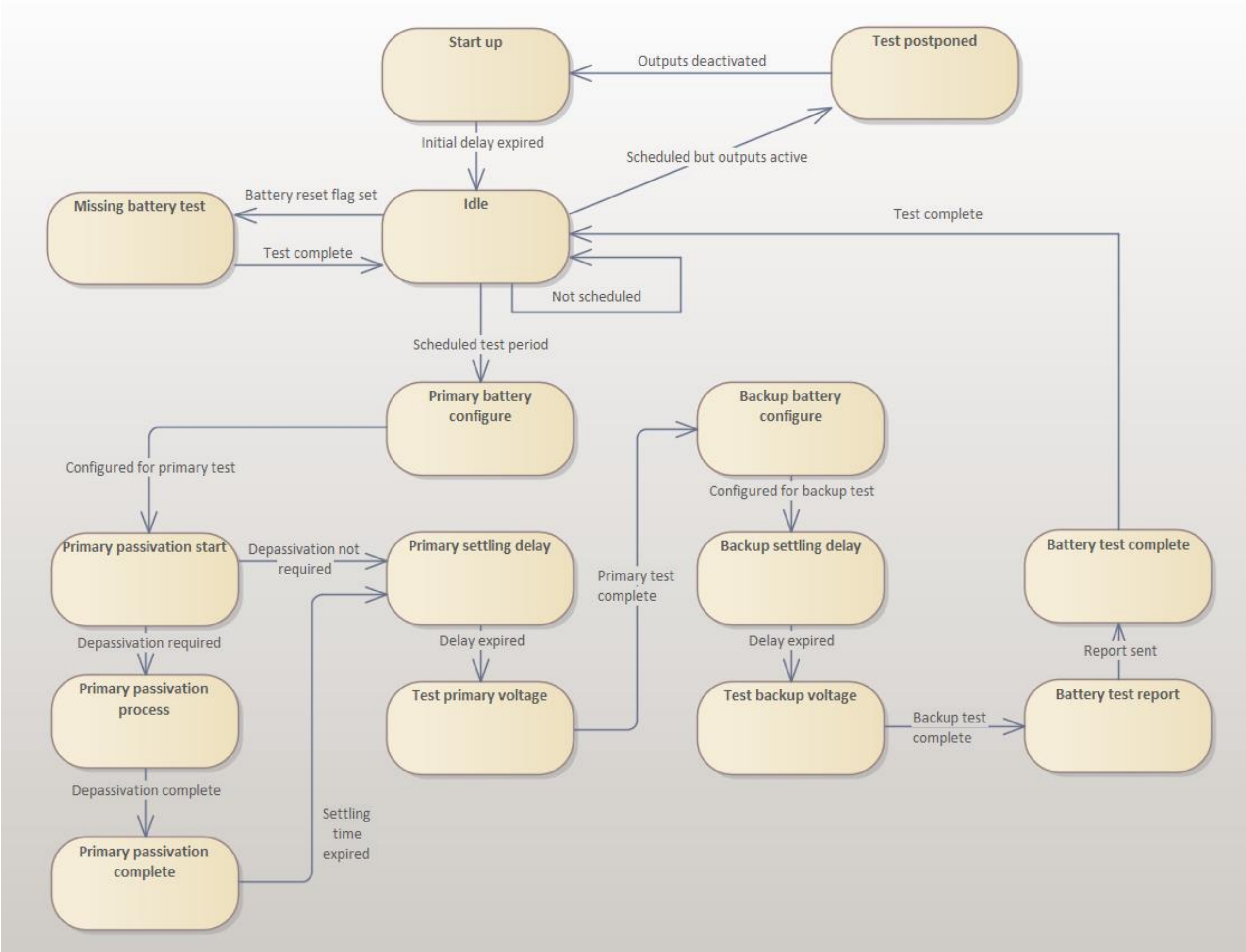


Figure 12 : SmartNet Battery Management State Diagram

2.11.4 SmartNet Battery Test Procedure

The SmartNet battery test procedure is detailed below.

1. Switch OFF the backup battery and switch on the battery load. Wait for 10 seconds.
2. Take two voltage readings of the primary battery, one second apart.
3. If the second reading is more than 199mV greater than the first reading, run the de-passivation process on the primary batteries.
4. Wait for 5 seconds.
5. Take three voltage readings of the primary batteries at 50ms intervals.
6. Switch ON the backup battery and switch OFF the primary battery. Wait for 5 seconds.
7. Take three voltage readings of the backup battery at 50ms intervals.
8. Switch ON the primary battery and switch OFF the battery load.
9. If all three primary battery readings are below the voltage threshold, send a battery error message to the control panel and start the double amber LED pattern.
10. If all three backup battery readings are below the voltage threshold, send a battery error message to the control panel and start the double amber LED pattern.
11. Schedule the next battery test.

The process is represented by the flow chart below.

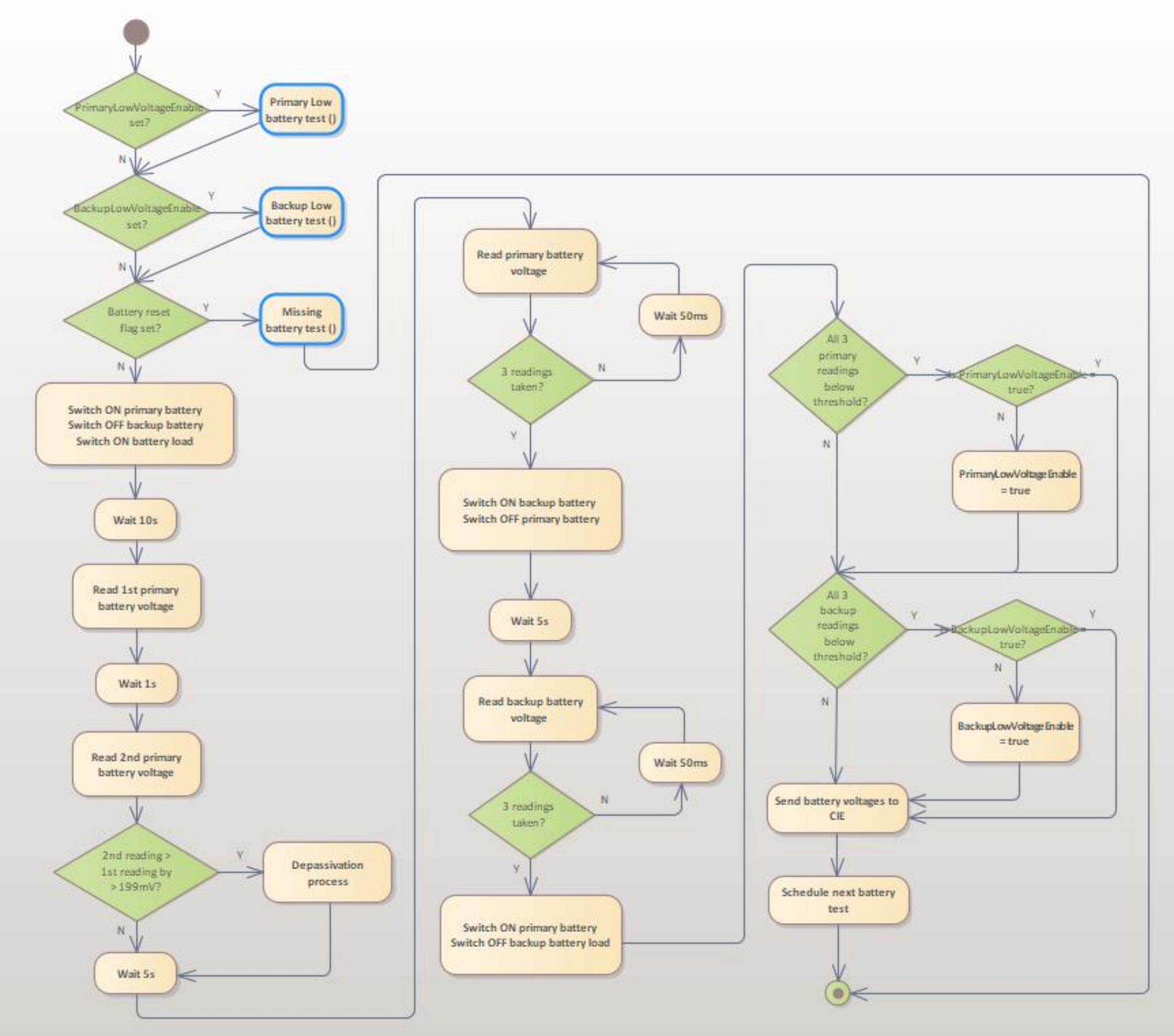


Figure 13 : SmartNet Battery Management Flow Chart

2.11.5 SmartNet De-passivation Procedure

The primary battery cells used by the SmartNet devices experience a loss of voltage during long periods of storage, this process is called passivation. The voltage is recovered when a load is applied and current flows.

The SmartNet battery monitor tests for passivation before running the battery test. A load is applied for 10 seconds, then two battery voltage readings are taken, with a one second interval between them. If the second reading is greater than the first by 200mV or more, the de-passivation procedure I invoked.

The SmartNet de-passivation software has the capability of pulsing the load that is applied to the battery during de-passivation. The 'ON' and 'OFF' times are configurable and are stored in non-volatile memory. Voltage readings are only taken while the battery load is on.

The de-passivation process is described by the flow chart below.

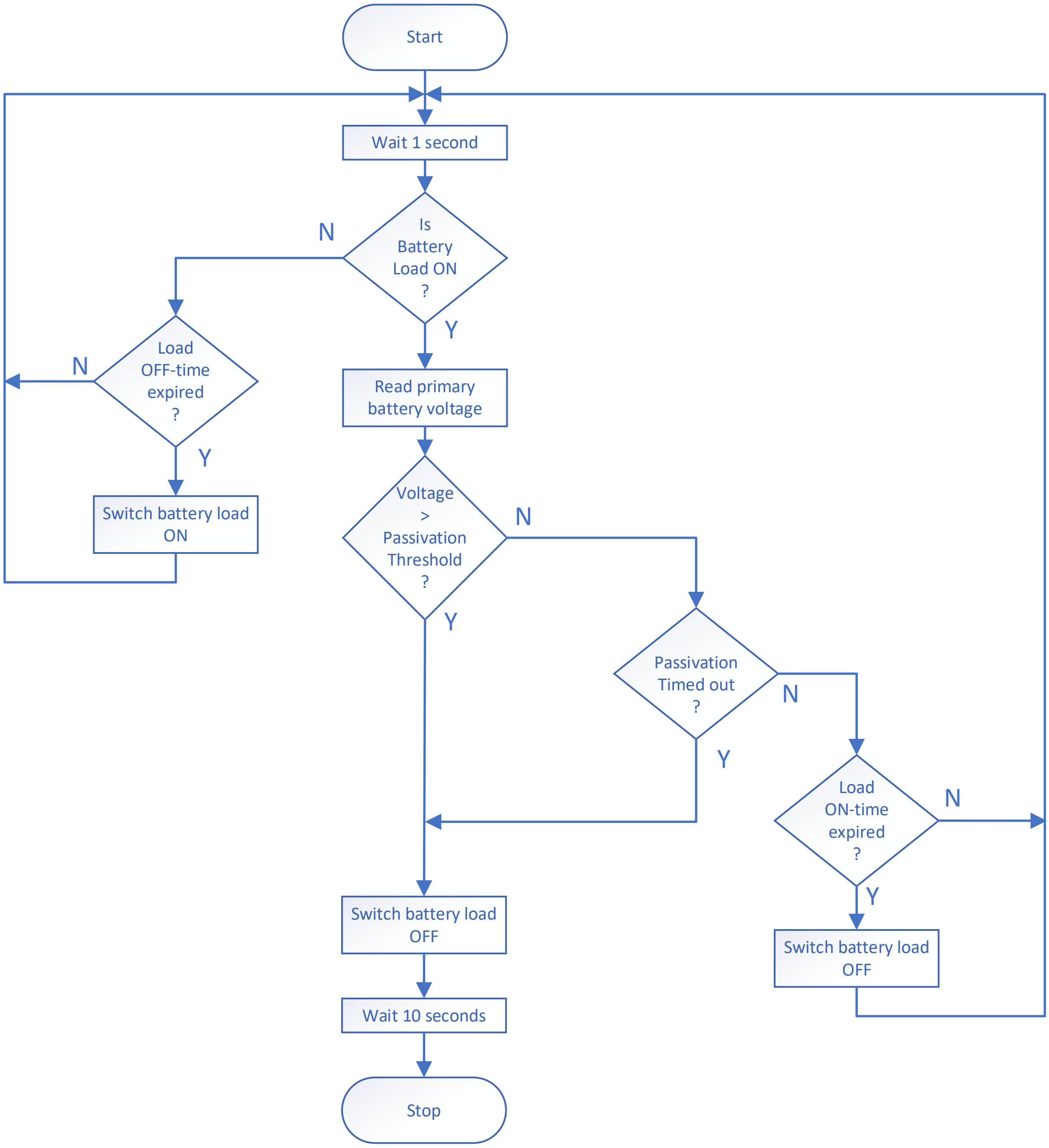


Figure 14 : SmartNet De-passivation Process Flow Chart

2.11.6 Missing Battery Test

The Cygnus2 software has two methods of determining whether batteries are missing. Firstly, the presence of a battery can be confirmed if its voltage can be read in isolation, without turning off the other batteries. Not all batteries can be isolated, however, so the second method must be applied.

During the battery test, two non-volatile 'flags' are used to indicate which battery is under test. The first flag is set when the primary battery test procedure begins. The second flag is set when the backup battery test procedure begins.

On a completion of the battery test, both flags are reset.

If a battery is not fitted, the device will experience a loss of power when the other battery is switched off during the test. This will result in the radio board restarting. On starting the battery test again, the state of the flags indicates that power was lost during an earlier test, and the state of the two flags indicates which battery is missing. The device can then do a limited test on the fitted battery, referred to as the Missing battery test.

On detection of a missing battery, the device sends a battery error message to the control panel where the "Device battery error" message will be displayed.

The device will double flash its amber LED every 12 seconds if a battery error is detected (this applies to both primary and backup batteries).

The missing battery test procedure is described by the flow chart below.

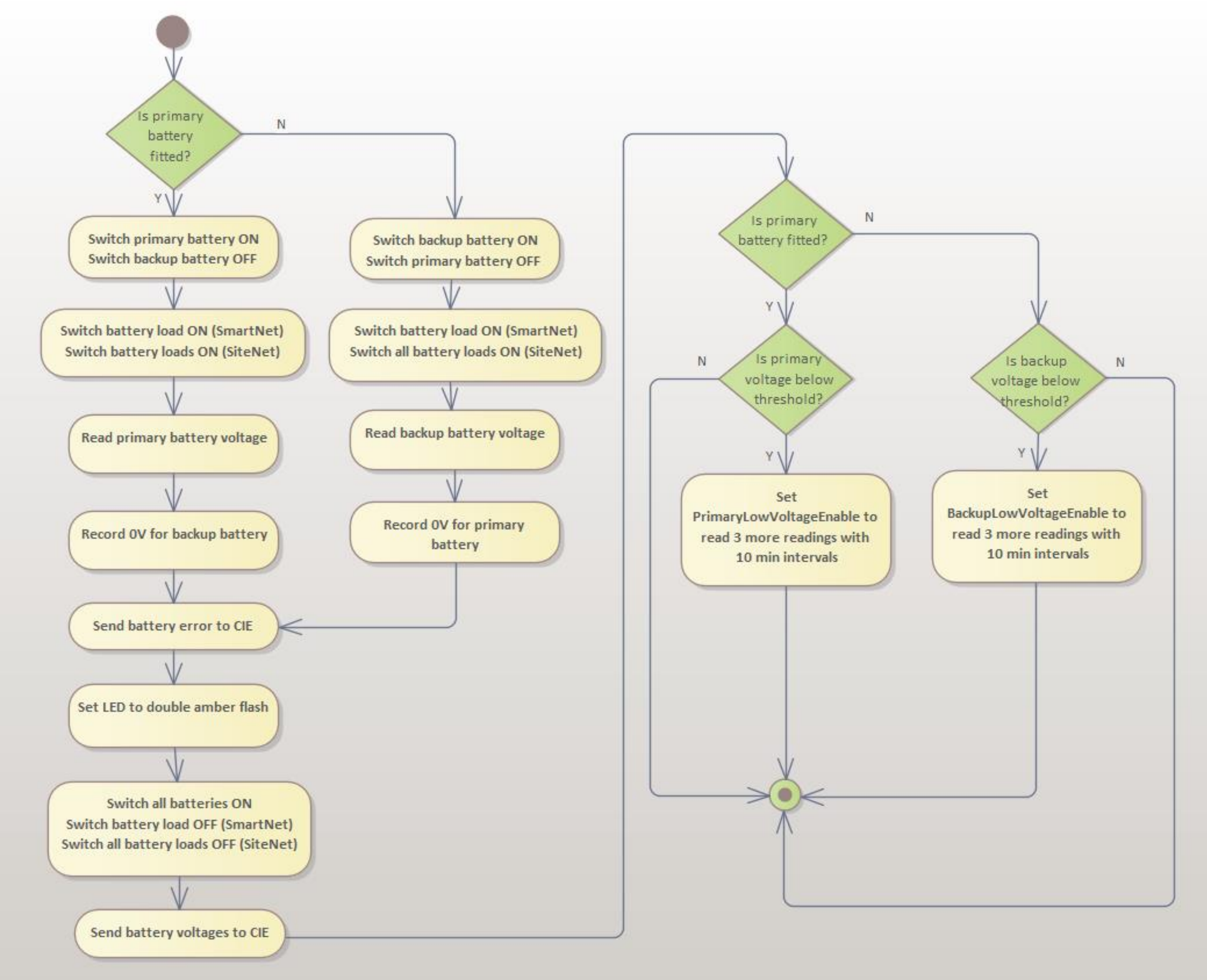


Figure 15 : Missing Battery Test Flow Chart

2.11.7 Low Battery Test

A low battery fault will be declared when either the primary or backup battery has reached a level where they can maintain only 30 days of operation or 30 minutes in an alarm state.

There are separate low battery threshold values for the primary and backup batteries.

The low battery thresholds are stored as configurable items that can be provisioned during production. Note this feature is required to allow update of thresholds without updating the firmware.

The low battery fault is latching. It will reset once a valid battery measurement has been received.

Upon the initial detection of a low battery, the RBU will take 3 further measurements to validate the status before indicating a low battery fault. This periodicity of these further measurements is 10 minutes.

On detection of a low battery, the device sends a low battery fault message to the control panel where the “Device low battery” message will be displayed.

The device operation will be changed such that the device is powered from the primary battery until the low battery condition is reached and that at this point the device will switch to the backup battery (unless the backup battery has already been determined as end of life).

The device will double flash its amber LED every 12 seconds if a low battery fault is detected (this applies to both primary and backup batteries).

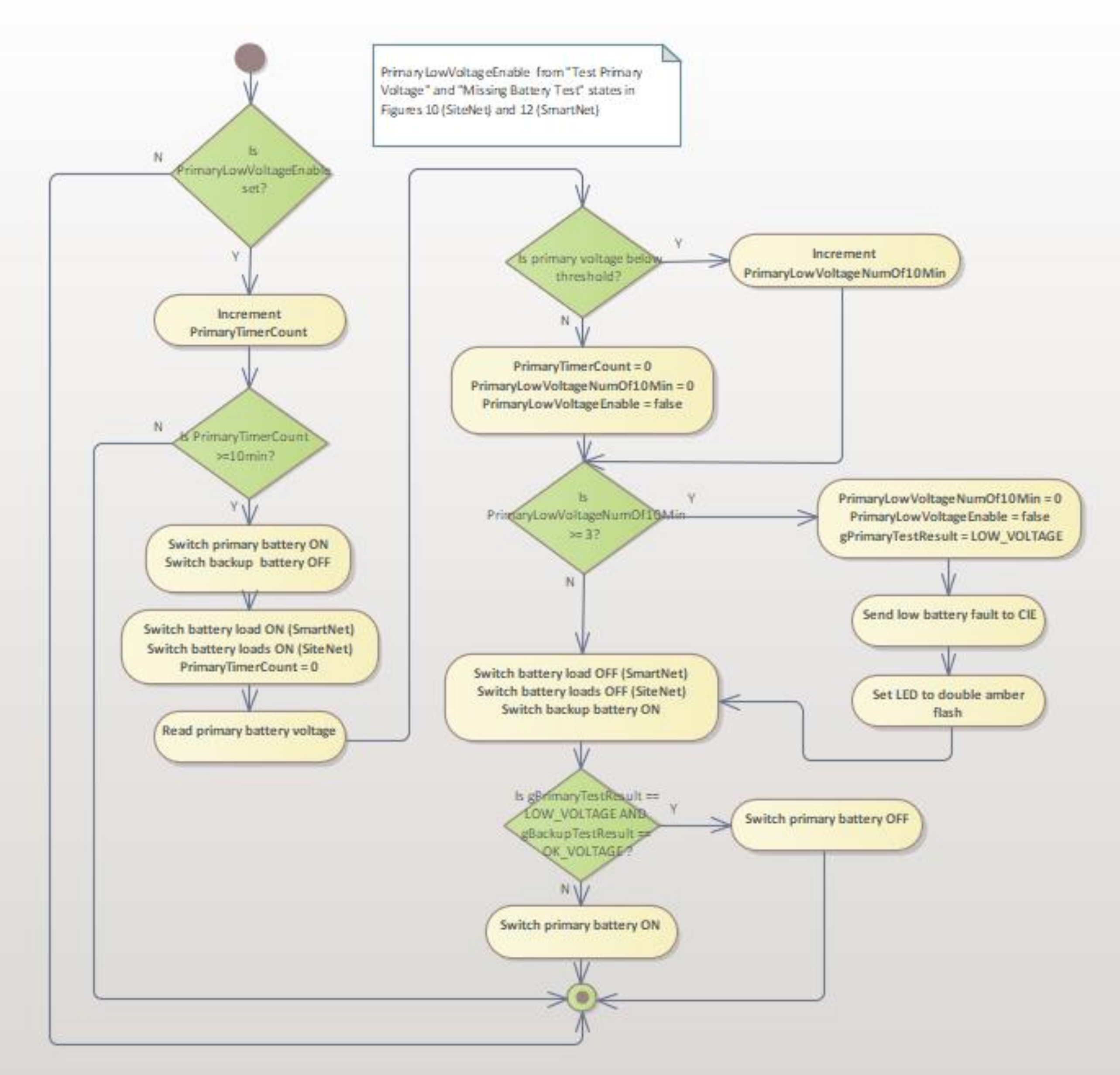


Figure 16 : Primary Low Battery Test Flow Chart

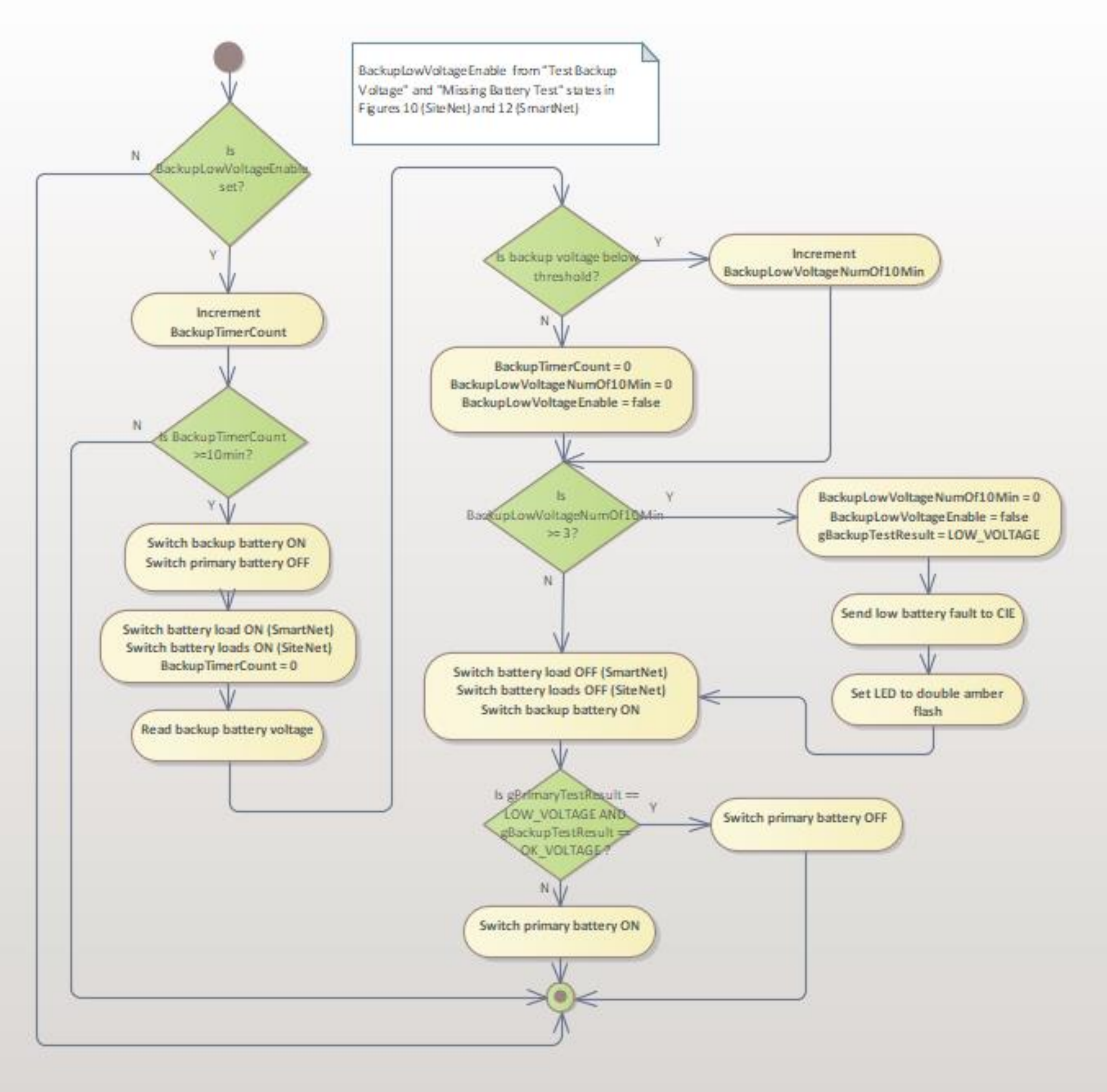


Figure 17 : Backup Low Battery Test Flow Chart

3 DETAILED DESIGN

3.1 Modules

3.1.1 Main module

MM_Main.c
MM_Main.h

Table 5 Main module files

This module contains the app_main() function for the application software. This is called after bootloader execute, which follows RTOS initialisation.

3.1.2 Mesh Task

MM_MeshTask.c
MM_MeshTask.h

Table 6 Mesh Task files

This module contains a task that implements the mesh protocol state machine.

3.1.3 MAC Task

MM_MACTask.c
MM_MACTask.h

Table 7 MAC Task files

This module contains a task that implements the MAC protocol state machine.

Interrupts are received from the LoRa modem or from the MCU LPTIM timer comparator. These interrupts set flags and release the MACSemId semaphore to trigger the MAC task.

The MAC task runs in a continuous loop. Mostly the loop waits for activity on the MACSemId semaphore. When the semaphore is triggered the loop calls handler functions based on flag states.

3.1.4 GPIO Task

MM_GpioTask.c
MM_GpioTask.h

Table 8 GPIO Task files

This module contains a task that polls the GPIO input pins. This uses the DM_InputMonitor module.

The task waits for a semaphore to be triggered by interrupts from the GPIO pins.

When the semaphore is triggered the task polls the GPIO pins at regular intervals until the new state is confirmed by DM_InputMonitor.

If a state change occurs this is handled by callback functions.

3.1.5 NCU Application Task

MM_NCUApplicationTask.c
MM_NCUApplicationTask.h
MM_ApplicationCommon.c
MM_ApplicationCommon.h
MM_CommandProcessor.c
MM_CommandProcessor.h
MM_CIEQueueManager.c
MM_CIEQueueManager.h

Table 9 NCU Application Task files

This module performs the application layer functionality of the NCU. Its primary role is to interface the control panel (CIE) to the radio mesh. The CIE and NCU Application form a master-slave relationship, with the CIE as the master. Commands from the CIE are processed by the NCU Application, enabling the CIE to request information about the state of the radio mesh and its constituent RBUs.

The main components of the NCU Application are shown in Figure 18: NCU Application Components.

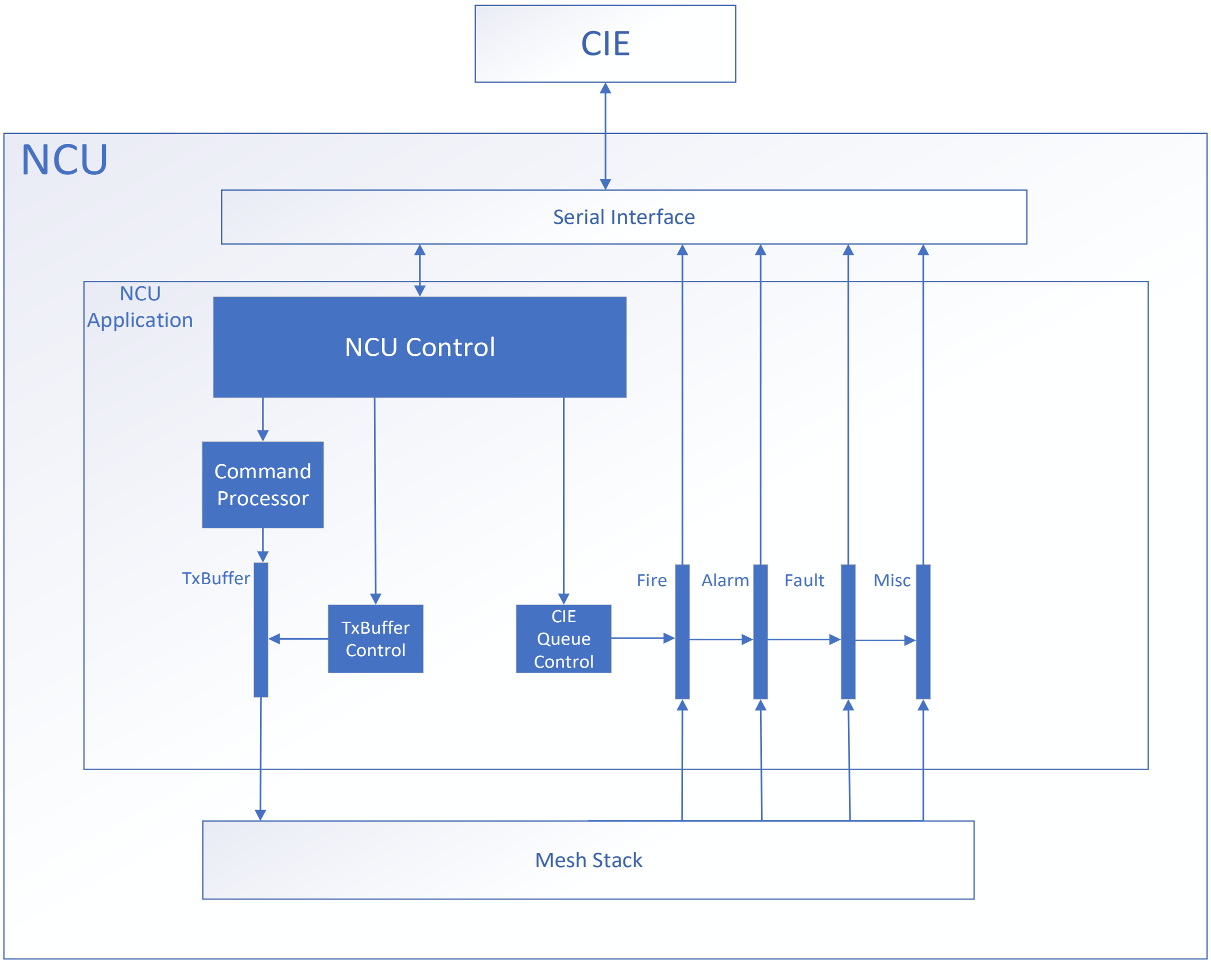


Figure 18: NCU Application Components

3.1.5.1 CIE Queue Control

The CIE queue control procedures are detailed in the CIE-NCU Protocol Specification 2008-SPC-0008. A summary follows.

All communications between the CIE and NCU are initiated by the CIE (master) and responded to by the NCU (slave). The CIE periodically queries the NCU to see if any events have been reported on the Mesh Network. Reported events are held in a set of message queues until they are read by the CIE. Four queues are implemented for different message types. These queues are labelled as ‘Fire,’ ‘Alarm,’ ‘Fault,’ and ‘Misc.’ Fire, Alarm and Fault queues are dedicated to that type of message, enabling the CIE to prioritise the order that messages are read. The Misc queue (miscellaneous) is for general messages such as status reports or responses to specific queries initiated by the CIE. The queues are managed by the CIE Queue Control component which queues messages that are received from the Mesh Network, and forwards them to the CIE on demand.

Messages going out to RBUs on the Mesh Network are queued in another queue called the TxBuffer. This is managed by the TxBuffer Control component which periodically checks for messages to be sent over the Mesh. After a message has been sent it remains at the front of the queue until a response is received from the Mesh. The message is then removed from the front of the queue and the next one is sent. A timeout mechanism discards unanswered messages to ensure that the link to the Mesh is not blocked indefinitely.

3.1.5.2 Command Processor

All commands received from the NCU are passed to the Command Processor component which interprets the command and translates it into Mesh Protocol messages. These are pushed onto the TxBuffer queue for transmission over the Mesh.

3.1.5.3 NCU Control

All messages entering the NCU Application are received by the NCU Control component which provides the appropriate routing and co-ordination activity. It also maintains records of the Mesh modules that have logged on, including their network address and zone number. The NCU Control component forms the main process of the NCU Application, utilising the other components as required.

3.2 State Machine

3.2.1 Overview of the State Machine

The State Machine is responsible for routing raw messages from the MAC to the appropriate handling functions. There are three modes of behaviour that must be implemented. One occurs during the network discovery period while the node establishes its connection to the network, another is the normal operation of the node after its connection is established, and the third is a test mode that enables specific functionality.

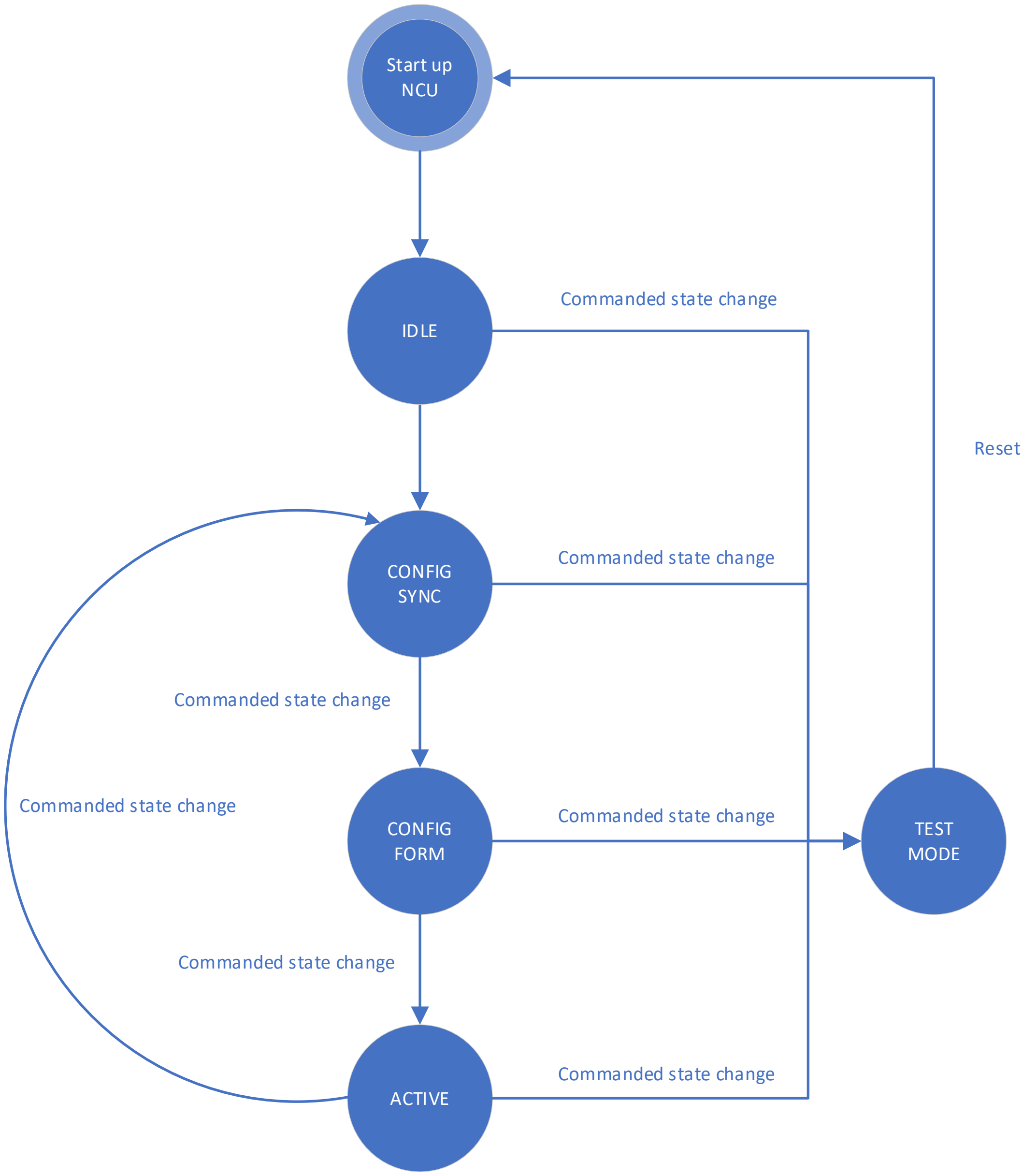


Figure 19: State Diagram for the NCU

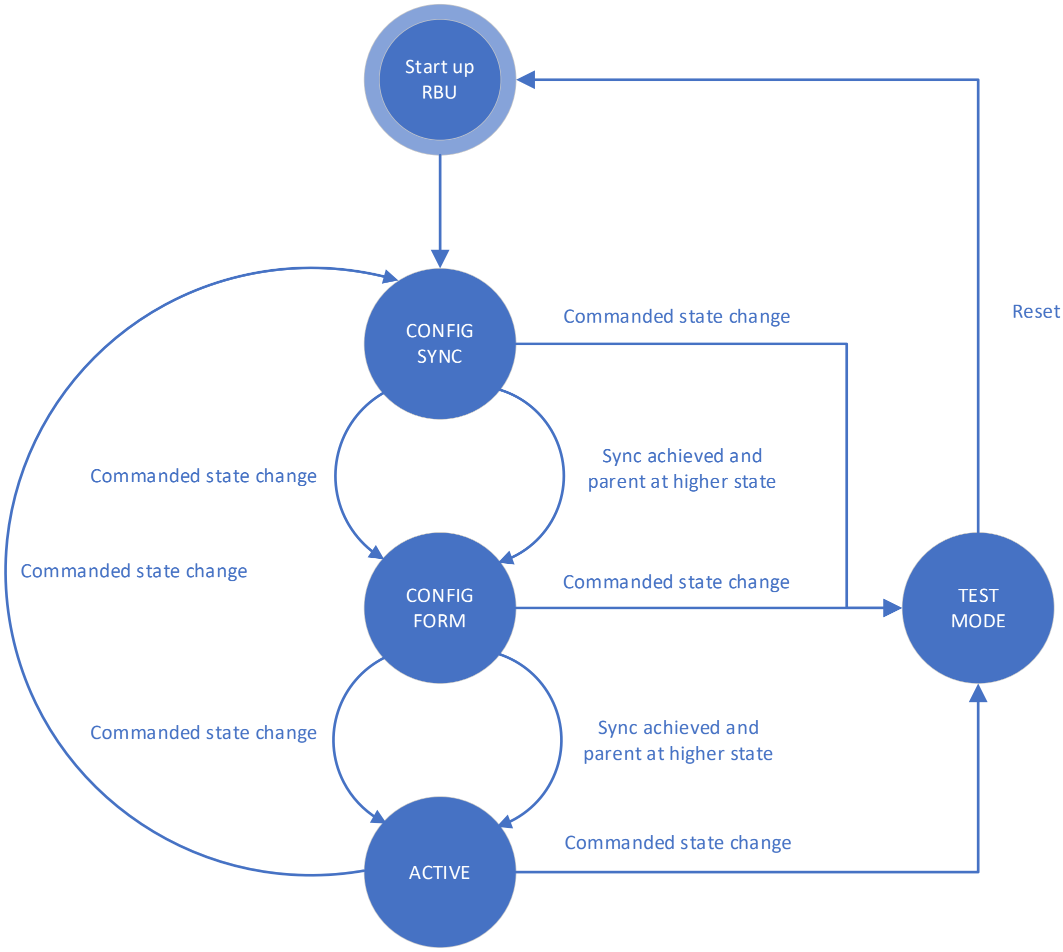


Figure 20: State Diagrams for RBU Devices

The state machine has five states:

3.2.1.1 State: IDLE

This state that the NCU enters on start-up. The NCU remains in this state until commanded by the CIE to advance to CONFIG SYNC state.

3.2.1.2 State: CONFIG SYNC

This is the state that an RBU enters on start-up. During this state the device performs the synchronisation phase. The device is set to continuous receive while it listens for a heartbeat from a neighbouring node. On receipt of the heartbeat it designates the sending node as its initial tracking node. The contents of the received heartbeat are read to establish the current position of the mesh TDM, then the device starts its own TDM in synchronisation with the mesh. A logon message is sent to the NCU via the initial tracking node. If the initial tracking node is in a more advanced state than CONFIG SYNC, the device automatically advances its own state. Otherwise the device remains in CONFIG SYNC until it receives a command from the NCU to advance its state to CONFIG FORM.

3.2.1.3 State: CONFIG FORM

On advancing to CONFIG FORM the device attempts to establish its place in the mesh. It begins by 'listening' for heartbeats from neighbouring nodes for several long frames. Once it has established which neighbours are available and the quality of their signals, the device selects the two best candidates as parents. If enough neighbours are available it also selects two tracking nodes, which are held in reserve as possible replacements for parents, should they stop communicating.

If the neighbouring nodes are in the ACTIVE state, the device automatically advances to the same state. Otherwise it remains in CONFIG FORM until it receives a command from the NCU to advance its state to ACTIVE.

3.2.1.4 State: ACTIVE

This is the normal state of the device when it is in active service. In this state the device communicates all fire, alarm and status messages to the NCU via its parent nodes and responds to commands from the NCU to activate output devices such as sounders and beacons.

An RBU device remains in the ACTIVE state while it retains contact with at least one of the parent nodes. Should contact be lost with a parent, the device promotes a tracking node to replace it. If no tracking nodes are available the device performs a parent scan, which involves listening for all heartbeats to identify a suitable replacement. Should no replacement be found, the device will continue to operate in the ACTIVE state with a single parent. If the single parent is lost, the device is no longer able to communicate with the mesh and performs a reset. This puts the node back into the CONFIG SYNC state and it attempts to re-join the mesh.

3.2.1.5 State: TEST MODE

This mode is entered in response to an external command received via the AT serial interface. It can be entered from any of the other states and the unit remains in this mode until it is restarted.

While in test mode, state machine events are mapped to special test functions that enable the RBU to be exercised and report results over the AT serial interface.

3.2.2 State Machine Design

The state machine contains a number of service functions that provide the behaviour associated with specific events. It holds an array of function pointers that associates each event with its corresponding service function.

The appropriate function is selected by using an event enumeration to index into the array.

The array is two-dimensional so that a different function association can be made for each state.

The NCU and RBU use a different array because of the differences in their function mapping for each state.

Events enter the state machine via a message queue, defined as MeshQ in the software. The queue is serviced by the Mesh Task, which reads messages from the queue and identifies the type of message using utility functions that examine the raw binary message.

This information is used to determine the event type, and an event enumeration is passed to the state machine, along with the raw message.

In the state machine, function SMHandleEvent accepts the event enumeration and uses it to index into the function pointer array, selecting the appropriate service function to process the raw message. The array is two-dimensional, with the second index being provided by the current State.

The service functions unpack the raw messages and act upon them.

3.2.2.1 State Machine Public Functions

SMInitialise

Initialises the data structures used by the state machine and sets the device type (NCU or RBU) and the Network Address (Node ID) and System ID of the Mesh to be used.

INPUTS:

bool isNCU	Set to true if the device is an NCU, or false if it is an RBU.
uint32_t address	The node ID to be used when joining the Network Mesh.
uint32_t systemId	The system ID of the Mesh.

RETURN:

None.

SMHandleEvent

Selects the appropriate service function for a received event.

INPUTS:

SM_Event_t event	Enumerated event type.
uint8_t* pData	A pointer to the event data structure.

RETURN:

None.

SMCheckDestinationAddress

A validation function that checks whether the destination address in a received message is valid for the device.

INPUTS:

uint32_t destAddress	The address from the received message.
uint32_t targetZone	The zone that the message was sent to.

RETURN:

bool TRUE if destAddress is valid for the local device.

MC_STATE_ScheduleStateChange A function to schedule a state transition.

INPUTS:

CO_State_t state	The new state that the state machine should be scheduled to transition to.

Bool synchronised	True if change should occur on next long frame. False for immediate.

RETURN:

None.

MC_STATE_SetDeviceState

A function to invoke a state transition.

INPUTS:

CO_State_t state	The new state that the state machine should transition to.

RETURN:

None.

MC_STATE_GetDeviceState

A function to query the current state.

INPUTS:

None.

RETURN:

CO_State_t	The current state of the state machine.

MC_STATE_SetMeshState

A function to set the current Mesh state.

INPUTS:

CO_State_t state	The new Mesh state.

RETURN:

None.

MC_STATE_GetMeshState

A function to query the current Mesh state.

INPUTS:

None.

RETURN:

CO_State_t The current Mesh state.

SM_SendConfirmationToApplication

A function to send network status information from the Mesh to the Application.

INPUTS:

uint32_t Handle	Transaction ID generated by the Application.
uint32_t ReplacementHandle	Replacement ID if a duplicate message was overwritten.
AppConfirmationType_t ConfirmationType	Message type being confirmed.
int32_t Error	Message success indicator for the Application.

RETURN:

None.

SM_SendEventToApplication

A function to send network status information from the Mesh to the Application.

INPUTS:

CO_MessageType_t MeshEventType	Event type.
ApplicationMessage_t* pEventMessage	Pointer to the message structure.

RETURN:

None.

SM_ActivateStateChange

A function to to allow the MAC to notify the State Machine that its scheduled state change can now take place.

INPUTS:

uint8_t LongFrameIndex	Used to set the start and stop long frames for SNR averaging that occurs before selecting the rank of the device in CONFIG_FORM state.

RETURN:

None.

MC_STATE_ReadyForStateChange

Check whether the device is ready to advance to the requested state.

INPUTS:

CO_State_t requested_state	The desired new state.

RETURN:

Bool TRUE if the state change can be actioned, FALSE otherwise.

MC_STATE_GetScheduledState

Gets the current scheduled state.

INPUTS:

None.

RETURN:

CO_State_t The scheduled device state.

MC_STATE_ActivateStateChange

Callback function, called by the TDM when a new longframe begins.

INPUTS:

uint8_t LongFrameIndex The long frame number.

RETURN:

None.

MC_STATE_Initialise

Initialise the Mesh and Device states.

INPUTS:

None.

RETURN:

None.

MC_STATE_AdvanceState

Advance the device state to the next level.

INPUTS:

None.

RETURN:

None.

3.3 Mesh

3.3.1 Synchronisation

3.3.1.1 Overview of the Sync Algorithm

The main functions of the synchronisation module are to align the timing of the mesh node with that of its tracking nodes (which, in order of priority, are the Primary Parent, the Secondary Parent, the Primary Tracking Node and the Secondary Tracking Node), and to calculate the expected MCU RTC value in subsequent slots so that interrupts can be scheduled to wake the MCU in time to perform the necessary actions in those slots.

Alignment with the tracking nodes is achieved by noting the LoRa modem's RxDone arrival time (the "timestamp") of the received sync message on the DCH channel. This is done independently for each node. The number of tracking nodes is set by the macro MAX_NO_TRACKING_NODES (4) in CO_Defines.h.

TDM module public functions MC_TDM_StartTDMMasterMode() and MC_TDM_StartTDMSlaveMode(), and private function MC_TDM_Advance(), generate a timestamp corresponding to the Sync RxDone event for the requested slot. Since the Sync RxDone event only really occurs in DCH slots, this event is not a real event in most slots, but simply represents a common location within each slot. The functions subtract a counter offset from the timestamp in order to wake the MCU sufficiently early to process whatever needs to be done in each slot. Within each slot, private functions MC_TDM_Process<type>Slot() configure the LoRa modem, where <type> is DCH/PRACH/SRACH/ACK/DLCCH/DULCH. Each function then sets the MCU to rewake after a leading guard space, to send or receive the message payload.

The timestamp for the next active slot is based on the slot timing of the present sync node. Individual parent nodes are unlikely to be exactly coincident in a real mesh, but any variation as encountered on switching sync node, is accommodated by means of the guard space, typically 3.3ms. For DCH receive, guard space is instead 1.5ms, until frequency lock with the mesh is achieved, and thereafter 3.5ms.

Two sync messages received from the sync node are required for a node to achieve frequency lock. The node will then know how many Low Power Timer (LPTIM) clock cycles correspond to a mesh long frame. Once long frame duration has been measured, function MC_SYNC_UpdateSync() signals this to the upper layers by setting Boolean flag *gFreqLockAchieved* TRUE. The first tracking node discovered is the initial tracking node, at which point long frame duration is initialised to a default value that would be correct if LPTIM clocks were exactly matched between devices. In practice, clocks will be different, so the recorded value of long frame duration is measured, and then applied to the stored value through a low pass filter $y(n) = 3/4\ y(n-1) + 1/4\ x(n)$. This filter is necessary to prevent timing jitter from being amplified by each node, and is appropriate since the long frame duration is determined entirely by the interval between heartbeats from the NCU, which should be stable.

(Note, this is not yet implemented – see CYG2-786.) Prior to achieving frequency synchronisation, the device must allow at least +/-0.713ms guard space for detecting the second sync message from each tracking node. (This figure comes from 118.75 second long frame duration, with 3ppm XTAL maximum error in the receiving and sending node. 0.713ms = 2 * 118.75 * 3/1000000 = 0.000713 seconds).

Timing Offsets for Other Events

The synchronisation algorithm deals primarily with timing of RxDone events generated when receiving synchronisation messages on the DCH channel. The timing of other events for the TDM structure of the DCH, DLCCH, RACH and ACK packets are depicted in figures in Sections 6.2, 6.3 and 6.4 of 2001-SPC-0012 Mesh Protocol Design.docx.

Overview of Software Interface

The synchronisation module is controlled using the three external functions listed below:

- Updating the timestamp of the sync node and performing sync checks are performed by

MC_SYNC_UpdateSync

- Assigning a node to be the current sync node is performed by *MC_SYNC_SetSyncNode*
- Returning the node ID of the current sync node is performed by *MC_SYNC_GetSyncNode*

3.3.1.2 Detailed Description of the Sync Algorithm

The sync algorithm operates by maintaining for each other tracked node, within static structure array ShortListElement_t gNodes[], LastTimestamp and AveragePeriod member variables. LastTimestamp is that of the received sync message on the DCH channel. AveragePeriod is the low pass filtered delta between the last two timestamps. The variables for the present sync node are used. The concept is embodied in Figure 21.

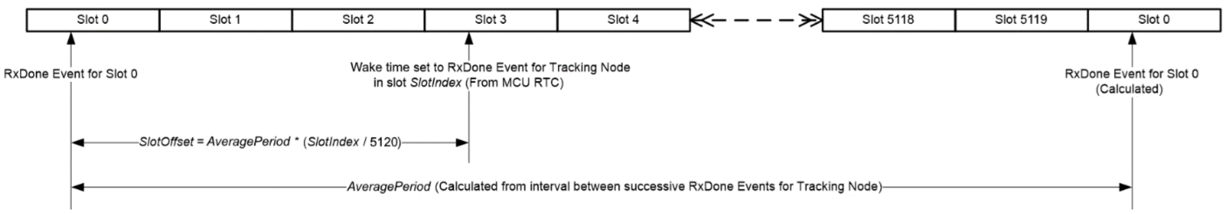


Figure 21: Depiction of LastTimestamp and AveragePeriod

Sync timing is maintained by accurately noting the MCU RTC value whenever a sync message is received, and passing that value to the algorithm. Whenever such a value is received the algorithm:

1. Calculates AveragePeriod (in MCU RTC cycles) for that node by calculating the interval between successive RxDone events and low pass filtering,
2. Sets LastTimestamp to that for the RxDone event.

The timing of subsequent active slots is then predicted based on the known Slot Index of those future active slots and the value of AveragePeriod, from which a Slot Offset is calculated, and added to the value of LastTimestamp to give the timestamp of the subsequent active slot.

To support these calculations it is necessary to extend the 4 second wrapping interval of the MCU RTC to a duration that must exceed the long frame duration of 118.75 seconds. For this reason a Long Counter is maintained within the algorithm, which extends the counter to 22 bits. The 16 LSBs of the Long Counter at RxDone events are made to exactly align with the 16-bit MCU RTC value at RxDone events, so conversion from Long Counter to Timestamp is simply to read the 16 LSBs. The value of the Slot Index has no direct correlation with the actual long counter value, but is maintained as the difference between the RxDone event for the slot in question and the value of LastTimestamp. With a perfect accuracy RTC, the interval between RxDone events in subsequent slots will be exactly 380 counter values. Though with a finite (3ppm) accuracy RTC, some intervals round to 379 or 381 clock counts.

The final concept is that all values related to the RTC wrap with a 16-bit period, and all values related to the Long Counter wrap with a 22-bit period (where the cycle frequency of these clocks is 16384 counts per second, +/- 3ppm). Thus arithmetic performed using the MCU RTC count or the long counter must be performed taking account of the wrapping.

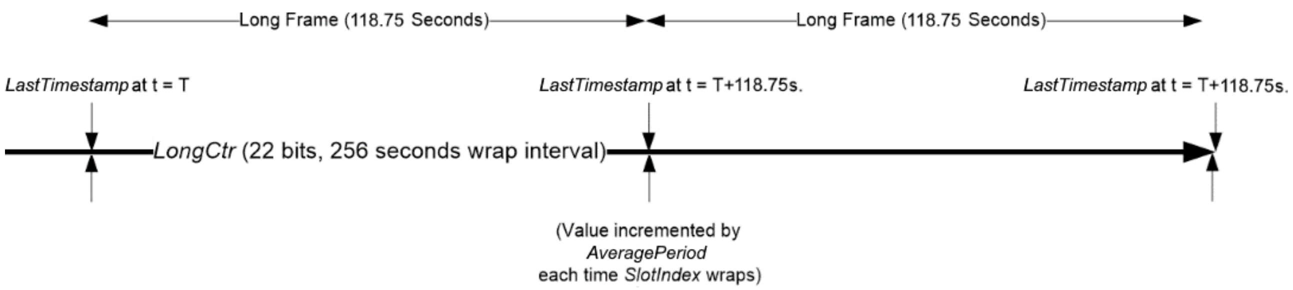


Figure 22: Depiction of LastTimestamp

The principle sequence of events in the sync algorithm is thus as follows. The description omits for simplicity, actions on invalid heartbeat receipt, and other off normal conditions.

- In response to a prospective received heartbeat message, the NCU or RBU state machine calls corrresponding service function ncuHeartbeat() or rbuHeartbeat().
- If the function unpacks the message successfully, and payload parameters are per a heartbeat message, and the message was received in a heartbeat slot, the function calls Mesh Forming and Healing module public function MC_MFH_Update().
- Within that module, MC_MFH_Update() calls private functions as follows: MC_MFH_UpdateShortList(), to update ShortListElement_t gNodes[] with the received heartbeat information. If for a previously seen node, MC_MFH_UpdateShortList() calls MC_MFH_UpdateNode(), else calls MC_MFH_AddNode().
- The information added by MC_MFH_AddNode() includes message node ID, Signal to Noise Ratio (SNR), and Received Signal Strength Indicator (RSSI). Also the function sets LastTimestamp to the MCU RTC count for the received heartbeat. And defaults AveragePeriod to the number of cycles per long frame.
- MC_MFH_UpdateNode() likewise, except the function checks the count against the expected value, per the stored AveragePeriod. And if within MFH_AVE_LONG_FRAME_THRESHOLD, updates AveragePeriod with the low pass filtered delta between the count and the stored LastTimestamp. And applies the message SNR and RSSI to the stored value through low pass filter $y(n) = 7/8\ y(n-1) + 1/8\ x(n)$.
- MC_MFH_UpdateNode() calls Sync Algorithm module function MC_SYNC_UpdateSync(), then updates LastTimestamp. If the received heartbeat is for the present tracking node (can be Primary Parent, Secondary Parent, Primary Tracking Node or Secondary Tracking Node), MC_SYNC_UpdateSync() again

determines the expected timestamp value, per the stored AveragePeriod. And if within SYNCH_LOCK_TOLERANCE, sets the gFreqLockAchieved true.

- MC_SYNC_UpdateSync() calls TDM module function MC_TDM_SetSynchReferenceTime() to set the sync reference counter to the computed start of the next long frame.

Tracking node management is handled by the Session Management module and Mesh Forming and Healing module. Refer respective sections 3.3.6 and 3.3.7. The maximum number of tracking nodes is 4, though this can be increased by modifying the value of #define MAX_NO_TRACKING_NODES.

3.3.1.3 Source and Header Files

The code is split into six files. These are described in detail later in this section.

- MC_SyncAlgorithm.h — Public header file for the Mesh Comms Synchronisation Algorithm
- MC_SyncAlgorithm.c — Source code for the Mesh Comms Synchronisation Algorithm
- STSyncAlgoTest.h — Test harness header file
- STSyncAlgoTest.c — Test harness source code
- MC_SyncPublic.h — Stub public header file included only by test harness
- MC_SyncPrivate.h — Stub private header file included only by test harness

The public functions are as follows. These are described in the following subsections.

- MC_SYNC_Initialise — Initialise the sync module
- MC_SYNC_SetSyncNode — Assign a node to be the current sync node
- MC_SYNC_GetSyncNode — Return the node ID of the current sync node
- MC_SYNC_UpdateSync — Update the time stamp of the sync node and perform sync checks

MC_SYNC_Initialise

Initialise the sync module (function has no content)

INPUTS:

- const uint16_t nodeID : Node ID of the sync node

RETURN:

- None

MC_SYNC_SetSyncNode

Assign a node to be the current sync node.

INPUTS:

- const uint16_t nodeID : Node ID of the sync node

RETURN:

- Type is ErrorCode_t. Returns SUCCESS_E (0) if the function succeeds, else returns one of the other values of ErrorCode_t to convey the reason for the error or warning. (See **Error! Reference source not found.**.)

MC_SYNC_GetSyncNode

Return the node ID of the current sync node.

INPUTS:

- None

RETURN:

- Type is uint16_t. Node ID.

MC_SYNC_UpdateSync

Update the timestamp of the sync node and perform sync checks. If last timestamp valid, check for sync lock, comprising modulo average period obtain, and expected timestamp determine, latter used if present timestamp invalid. If valid, set sync reference time, and if present timestamp within tolerance of expected timestamp, deem frequency lock.

INPUTS:

- ShortListElement_t* pSyncNode : Sync node pointer
- const uint32_t timeStamp : The timestamp for the received heartbeat

RETURN:

- None

3.3.1.4 Function Call Hierarchy

The function call hierarchy is depicted in Figure 23 overleaf. External functions are shaded, and functions called from more than one place have italicised text.

Input parameter range checking is only performed in the external functions, to avoid duplication of code.

3.3.1.5 Return Codes

The following return codes can be returned by the extern functions in the sync module.

Table 10 Function Return Codes

ErrorCodes - Enumeration and Description
SUCCESS_E Successful call to function.
ERR_NOT_FOUND_E Node not found when attempting to add as sync node.
ERR_NO_SYNC_LOCK_E Node has no valid timestamp when attempting to add as sync node.

3.3.2 Time Division Multiplex

3.3.2.1 Overview of the Time Division Multiplex Algorithm

TBA

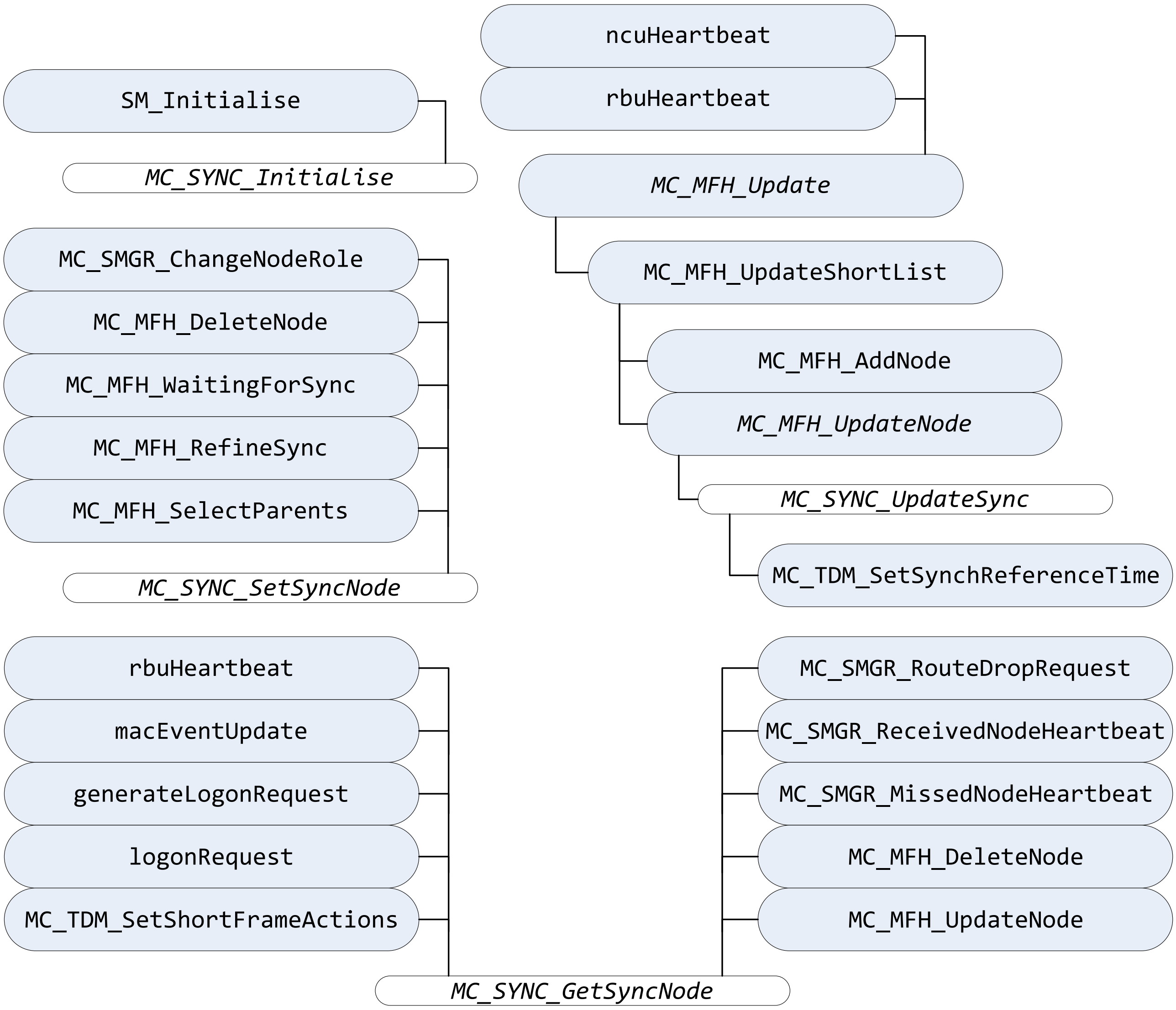


Figure 23: Synchronisation Module Function Call Hierarchy

Overview of Software Interface

The time division multiplex module is controlled using the external functions listed below:

- TBA

3.3.2.2 Detailed Description of the Time Division Multiplex Algorithm

TBA

3.3.2.3 Source and Header Files

The code is split into two files. These are described in detail later in this section.

- MC_TDM.h Public header file for the Time Division Multiplexing of the mesh protocol
- MC_TDM.c Source code for the Time Division Multiplexing of the mesh protocol

The public functions are as follows. These are described in the following subsections. The three in italics are uncalled.

- MC_TDM_Init Initialise the TDM properties
- MC_TDM_SetDCHSlotBehaviour Set the behaviour of a DCH slot
- MC_TDM_StartTDMSlaveMode Start the TDM for the RBU
- MC_TDM_StartTDMMasterMode Start the TDM for the NCU
- MC_TDM_TimerOperationTdm Run the TDM
- MC_TDM_GetSlotReferenceTime Return the expected RxDone time for the current slot
- MC_TDM_IsReceivingDCHSlot Return true if the current slot is a receiving DCH slot
- MC_TDM_IsTrackingDCHSlot Return true if the current slot is a tracking node DCH slot
- MC_TDM_GetCurrentSlot Return the details of the current TDM slot
- MC_TDM_GetCurrentSlotType Return the current slot type
- MC_TDM_GetSlotType Return the slot type for a specific slot number
- MC_TDM_GetSlotTypeDefault Return the default slot type (unchanged by dynamic disablement) for a specific slot number
- MC_TDM_ConstructSlotInSuperframeIdx Calculate the super frame slot from the frame indexes
- *MC_TDM_GetCurrentSlotInShortframeIdx* *Return the current short frame index*
-
- *MC_TDM_GetCurrentSlotInLongframeIdx* *Return the slot index in the long frame*
- *MC_TDM_GetCurrentSlotInSuperframeIdx* *Return the slot index in the super frame*
- MC_TDM_SetSynchReferenceTime Set the sync reference counter to the start of a long frame
- MC_TDM_StateChangeAtNextLongFrame Schedule the change of state
- MC_TDM_OpenAckSlot Open ACK slot to send or receive ACK
- MC_TDM_DchConfigurationValid Check have at least one DCH slot configured to receive

- MC_TDM_EnableChannelHopping Start channel hopping

MC_TDM_Init

Initialise the TDM properties. Default the heartbeat behaviour to receive, and set the local device behaviour to transmit. Calculate the long frame heartbeat slot (0 to 5119). And set the type of each short frame slot (0 to 39).

INPUTS:

- None

RETURN:

- Type is ErrorCode_t. Always returns SUCCESS_E (0).

MC_TDM_SetDCHSlotBehaviour

Set behaviour for the specified node, if within the MAX_DEVICES_PER_SYSTEM range.

INPUTS:

- const uint16_t address : The node ID to map to the DCH slot
- const MC_TDM_DCHBehaviour_t behaviour : The behaviour to set

RETURN:

- Type is bool. Returns true if behaviour set, else returns false.

MC_TDM_StartTDMSlaveMode

Start the TDM for the RBU. Calculate the sync reference time (start of long frame slot 0). Set the next long frame sync, and next slot properties. Generate heartbeat if due for this RBU. Identify the current heartbeat slot, and configure for the next wake. Next slot can heartbeat or primary RACH. Set the LPTIM comparator, and enable the LPTIM interrupt.

INPUTS:

- const uint32_t slotInSuperframeIdx : Slot in super frame (0 to 327679)
- const uint16_t slotRefTime : Last received heartbeat timestamp
- const bool enableFrequencyHopping : Frequency hopping configure (false to disable, true to enable)

RETURN:

- None

MC_TDM_StartTDMMasterMode

Start the TDM for the NCU. Get the current slot time. Set the timer to wake after 1 slot duration to give the initial heartbeat time to be generated. Set the slot record for the next wake. Generate heartbeat. Set the LPTIM comparator, and enable the LPTIM interrupt.

INPUTS:

- None

RETURN:

- None

MC_TDM_TimerOperationTdm

Run the TDM. Read the LPTIM counter. Call private handler function for the present slot type. Comprises DLCHH, primary RACH, secondary RACH, DULCH, ACK (primary RACH, secondary RACH or DULCH), and DCH. Select LED pattern for present mesh state. Comprises Sync, Form, and Active.

INPUTS:

- None

RETURN:

- None

MC_TDM_GetSlotReferenceTime

Return the expected RxDone time for the current slot. Depends on whether the slot is DLCHH, primary or secondary RACH, DCH, or ACK (primary RACH, secondary RACH or DULCH).

INPUTS:

- None

RETURN:

- Type is uint16_t. The predicted RxDone time.

MC_TDM_IsReceivingDCHSlot

Return true if the current slot is a receiving DCH slot.

INPUTS:

- None

RETURN:

- Type is bool. Returns true if current slot is a receiving DCH slot, else returns false.

MC_TDM_IsTrackingDCHSlot

Return true if the current slot is a tracking node DCH slot.

INPUTS:

- None

RETURN:

- Type is bool. Returns true if current slot is a tracking node DCH slot, else returns false.

MC_TDM_GetCurrentSlot

Return the details of the current TDM slot.

INPUTS:

- MC_TDM_Index_t* const slot : Pointer to the slot properties structure to be populated

RETURN:

- None

MC_TDM_GetCurrentSlotType

Return the details of the current TDM slot.

INPUTS:

- None

RETURN:

- Type is MC_TDM_SlotType_t. The slot type.

MC_TDM_GetSlotType

Return the slot type for a specific slot number.

INPUTS:

- const uint16_t shortFrameSlot : The short frame slot index

RETURN:

- Type is MC_TDM_SlotType_t. The slot type.

MC_TDM_GetSlotTypeDefault

Return the default slot type (unchanged by dynamic disablement) for a specific slot number.

INPUTS:

- const uint16_t shortFrameSlot : The short frame slot index

RETURN:

- Type is MC_TDM_SlotType_t. The slot type.

MC_TDM_ConstructSlotInSuperframeIdx

Calculate the super frame slot from the frame indexes.

INPUTS:

- const uint32_t LongFrameIndex : The long frame index (0 to 63)
- const uint32_t ShortFrameIndex : The short frame index (0 to 127)
- const uint32_t SlotIndex : The slot index (0 to 39)

RETURN:

- Type is uint32_t. The super frame slot (0 to 327679).

MC_TDM_GetCurrentSlotInShortframeIdx

Return the current short frame slot index.

INPUTS:

- None

RETURN:

- Type is uint32_t. The short frame slot index (0 to 39).

MC_TDM_GetCurrentSlotInLongframeIdx

Return the slot index in the long frame.

INPUTS:

- None

RETURN:

- Type is uint32_t. The long frame slot index (0 to 5119).

MC_TDM_GetCurrentSlotInSuperframeIdx

Return the slot index in the super frame.

INPUTS:

- None

RETURN:

- Type is uint32_t. The super frame slot index (0 to 327679).

MC_TDM_SetSynchReferenceTime

Set the sync reference counter to the start of a long frame.

INPUTS:

- const uint16_t nextFrameStart : The calculated LPTIM count at the start of the next long frame
- const uint32_t aveFrameLength : The long frame duration in clock ticks

RETURN:

- None

MC_TDM_StateChangeAtNextLongFrame

Schedule the change of state at next long frame start, or the following one if insufficient remaining time to schedule.

INPUTS:

- const bool ChannelHopping : True if channel hopping is to be started
- void (* callbackFunc)(const uint8_t) : Pointer to function to be called when the state changes

RETURN:

- None

MC_TDM_OpenAckSlot

Called when we need to open an ACK slot to send or receive an ACK. If sending an ACK, the TDM will have already decided to sleep through the slot, so we need to revise the wake time. Received ACKs were known about in advance, so the wake time will be correct.

INPUTS:

- const bool sendingAck : True if we are opening a slot to send an ACK, false to receive

RETURN:

- None

MC_TDM_DchConfigurationValid

Called each long frame to ensure that we have at least one DCH slot configured to receive.

INPUTS:

- None

RETURN:

- Type is bool. Returns true if at least one Heartbeat slot is configured to receive, else returns false.

MC_TDM_EnableChannelHopping

Start channel hopping.

INPUTS:

- None

RETURN:

- None

3.3.2.4 Function Call Hierarchy

TBA

3.3.2.5 Return Codes

The following return codes can be returned by the extern functions in the time division multiplex module.

Table 11 Function Return Codes

ErrorCodes - Enumeration and Description
SUCCESS_E Successful call to function.

WARNING_NODE_ALREADY_BEING_TRACKED_E An attempt was made to add a tracking node that is already being tracked. (No change has been made to the tracking algorithm.)
ERR_ALL_TRACKING_NODES_ALLOCATED_E An attempt was made to add more tracking nodes than are supported.
ERR_TRACKING_NODE_NOT_FOUND_E Specified tracking node not found.
ERR_SLOT_INDEX_OUT_OF_RANGE_E Call to function unsuccessful as the slot index was an invalid value
ERR_UNEXPECTED_ERROR_E There should be a free element in SyncData.aTrackingNode, but code failed to find it.
ERR_NO_TRACKING_NODES_ALLOCATED_E Function called expects there to be active tracking nodes, but there are none active.
ERR_NEXT_SLOT_INDEX_OUT_OF_RANGE_E The value of the Next Slot Index is too large.
ERR_INVALID_POINTER_E The pointer in the function call is set to NULL.

3.3.3 Channel Hopping Sequence Generation

3.3.3.1 Overview of the Channel Hopping Sequence Generator Algorithm

This module populates an array with a sequence of specified length of hopping channel indicies that satisfy two criteria:

- First Adjacent Hop Rule: the minimum hop interval (specified as an integer number of channels) can be specified, so that the EN54 requirement that all frequency hops are a minimum of 1MHz. It is essential to note that the 1.0 MHz rule must be applicable for all cyclic offsets of the hopping sequence. For example, consider channels 0 to 7 where a hop of 3 channels is required to meet the 1MHz separation. If hop n uses channel 0, then channels {3, 4, 5, 6} are all suitable for the next hop, as these are always at least 1.0 MHz .

- Second Adjacent Hop Rule: the PA/Cygnus requirement that any three consecutive hopping channels will all be on different channels. This rule is automatically applied for adjacent hops by the rule above, and so is implemented specifically for the second adjacent channel. This rule exists to prevent a hopping sequence from merely toggling between two different channels.

The rules are simple to implement progressively whilst the array is being populated from the first to the last entry, but special measures are taken in the code to ensure that the rules are also followed when the sequence wraps back from the last sequence channel to first sequence channel, and continuing thereon.

The channels are selected using a 16-bit maximum length PRBS, which generates each number from 1 to 65535 once in a sequence of 65535 values[1]. Provision is also made to reset and then seed the random number generator using a seed stored in a byte array: this allows all devices in a given network to generate a random hopping sequence that can be set using any 16-bit sequence that is preferably unique to that one mesh. (The seed value can be longer than 16 bits, but must always a multiple of 8 bits packed into a byte array.)

3.3.3.2 Detailed Description of the Channel Hopping Sequence Generator Algorithm

There are two pre-requisites to meet before generating a hopping sequence.

- An array of type Channel_t [2] must be declared, having sufficient elements to hold the sequence (for Cygnus II, RACH_ DLCCH_SEQ_LEN should be 68, and DCH_SEQ_LEN should be 16). Initialise all elements of the array to zero.
- The function MC_GetRandNumber() must be called with the second parameter being a pointer to a structure containing the seed. The seed is of type PRBSGeneratorSeed_t, which is a structure of two elements:
  - uint8_t NumBytes; // An integer specifying the number of bytes in the seed value.
  - uint8_t * pSeedByteArray; // Pointer to array of NumBytes elements containing the seed value.

Note that the order and endianness of the bytes containing the seed are not important, so long as they are used consistently by all units in the mesh. We could use either the encryption key (168 bits) or the hash key (128 bits) for this purpose.

Once the pre-requisites are met, the hopping sequence is generated by calling the public function thus:

ErrorCode = MC_GenChanHopSequence(aRachDlcchChanHopSeq, RACH_DLCCH_SEQ_LEN,

NUM_HOPPING_CHANS, MIN_CHAN_INTERVAL);

The returned value, ErrorCode will be SUCCESS_E if the hopping sequence is successfully created, or an error value if the sequence generation fails, indicating the reason for the failure.

In the event of an error, the process is repeated up to 100 times before the whole procedure is abandoned. It has never failed to produce a suitable sequence within the 100 iteration limit.

The function MC_GenChanHopSequence () selects channels for each hop in the sequence. To do this it calls MC_GetRandNumber() to generate random numbers, and GetPermittedChans() to provide a candidate list of channels that meet the hopping rules from which to make a random selection.

There is also a requirement to meet the hopping sequence rules when the sequence repeats. This is validated by the function TestChanHopSeq() on completion of the sequence generation.

---

[1] See https://uk.mathworks.com/help/comm/ref/pnsequencegenerator.html, section "Sequences of Maximum Length" table row r=16, Generator Polynomial [16 15 13 4 0].

[2] typedef uint8_t Channel_t

3.3.3.3 Source and Header Files

The code is split into four files. These are described in detail later in this section.

- MC_ChanHopSeqGenPublic.h Public header file
- MC_ChanHopSeqGenPrivate.h Private header file
- MC_ChanHopSeqGen.c Source code
- STChanHopSeqGenTest.h Test harness header file
- STChanHopSeqGenTest.c Test harness source code

3.3.3.4 Public Functions

The public functions are as follows. These are described in the following subsections.

- MC_GenChanHopSequence
- MC_GetRandNumber
- MC_SelectInitialChannel

MC_GenChanHopSequence

Called during the MAC initialisation to generate a table of hopping channels. The address of the array to store the channels is passed to the function, along with the desired length of the hopping sequence, the number of radio channels, and the minimum number of channels that any hop must make in order to meet EN54 channel hopping requirements (1.0 MHz). In our application, there are 7 channels and we require a hop of 3 channels to achieve a 1MHz separation, so we should set NumChans to 7 and MinChanInterval to 3. The DCH sequence length should be set to 16, and the RACH/DLCCH sequence length should be set to 68.

INPUTS:

- Channel_t * const pChanHopSeqArray : Pointer to the array to contain the hopping sequence.
- const uint16_t SeqLength : Number of hops in the whole sequence (after which, sequence should be repeated).
- const Channel_t NumChans : Number of hopping channels. Should not exceed the s/w constant NUM_HOPPING_CHANS.
- const uint16_t MinChanInterval : Minimum number of channels to hop (e.g. EN54 specifies minimum of 1.0 MHz). If channel spacing is not constant, specify the worst case interval.

RETURN:

- Type is ErrorCode_t. Returns SUCCESS_E (0) if the function succeeds, else returns one of the other values of ErrorCode_t to convey the reason for the error or warning. (See Table 12.)

MC_GetRandNumber

Produces a pseudo-random sequence, returning a different value for each time it is called. It must be called once with a seed value before it is used. All nodes on a network should seed with the same value, unique to that network. We use the System ID for this purpose.

INPUTS:

- uint16_t * const pRandomNumber : Pointer to the variable to contain the random number.
- const PRBSGeneratorSeed_t * const pPRBSGeneratorSeed : Pointer to the seed structure. If this is set to NULL the function will return the next random number in the series. If it points to a valid byte array, and the length of the array is specified as non-zero, the function will re-seed the random number generator and will generate the first number in the series.

RETURN:

- Type is ErrorCode_t. Returns SUCCESS_E (0) if the function succeeds, else returns one of the other values of ErrorCode_t to convey the reason for the error or warning. (See Table 12.)

MC_SelectInitialChannel

Called to select the best channel on which to search for the DCH if the mesh is already hopping when the RBU attempts to synchronise to it. The preferred channel is any one that has the smallest max-interval between instances of using that channel. The concept of “the smallest max_interval” is cumbersome to word, but means the following: if one channel “a” is used twice within the 16-step hopping sequence, at intervals of 7 and 9, whereas another channel “b” is used twice at intervals of 6 and 10, then the max_interval for these two are 9 and 10 for channels a and b respectively, and the minimum max_interval is 9 for channel a. This means that the longest the RBU would have to wait to find the first DCH would be 9 long frames, whereas for channel b the longest the RBU might have to wait would be 10 long frames.

INPUTS:

- Channel_t * const pChanHopSeqArray : Pointer to the array to contain the hopping sequence.
- const uint16_t SeqLength : Number of hops in the whole sequence (after which, sequence should be repeated).
- const Channel_t NumChans : Number of hopping channels. Should not exceed the s/w constant NUM_HOPPING_CHANS.
- uint16_t * const pInitialSearchChannel. Pointer to variable to return the selected hopping channel index.

RETURN:

- Type is ErrorCode_t. Returns SUCCESS_E (0) if the function succeeds, else returns one of the other values of ErrorCode_t to convey the reason for the error or warning. (See Table 12.)

3.3.3.5 Return Codes

The following return codes can be returned by the extern functions in the channel hopping module.

Table 12 Function Return Codes

ErrorCodes - Enumeration and Description
SUCCESS_E Successful call to function.
ERR_PRBS_SEED_NOT_SET_CORRECTLY_E The structure containing the PRBS seed value does not contain valid values.
ERR_RAND_NUM_GENERATOR_NOT_SEEDED_E The PRBS random number generator was called without having first seeded it.
ERR_TOO_FEW_CHANNEL_OPTIONS_E A sequence could not be resolved because there were insufficient channel options to satisfy all constraints.
ERR_INVALID_POINTER_E The pointer in the function call is set to NULL.

3.3.4 RACH Protocol – Lost Message Management

The RACH Protocol requires that all messages that are addressed to a specific recipient (i.e. non broadcast messages) must be acknowledged. If no acknowledgement is received, the message shall be resent.

If, having sent a message, no acknowledgement has been received, it is assumed that the message did not reach its destination, perhaps due to a network collision with another transmission. The sender makes several retries, selecting random slots for retransmission in accordance with the back-off algorithm defined in the Mesh Protocol Design document (2001-SPC-0012). If the final destination is not one of the parent nodes, the repeated messages are sent via alternate parents in an attempt to find a route that works. Once the maximum number of resends have been performed without receiving an acknowledgement, the message is discarded.

The resend mechanism is implemented by the Acknowledgement Manager in the application layer of the software.

3.3.4.1 Acknowledgement Manager

The Acknowledgement Manager (implemented in source file MC_AckManager.c) is responsible for processing the ACK responses from neighbouring devices on the network, and for resending messages that are not acknowledged.

The ACK response does not contain information that could map it to the original message. For this reason, only one message can be active at any time. Further messages must be held until the active message has been acknowledged.

Each channel (P-RACH & S-RACH) is managed separately so that, for example, P-RACH can still operate normally while S-RACH is blocked by an unacknowledged message.

The Acknowledgement Manager does not distinguish between neighbouring nodes. All P-RACH messages share a single P-RACH queue, all S-RACH messages share a single S-RACH queue.

When the application wants to send a message, it asks the Acknowledgement Manager if there is an outstanding message on the RACH channel. If the channel is not active, it queues the message on the appropriate RACH queue for the MAC to transmit in the next slot. It also adds the message to the Acknowledgement Manager (function MC_ACK_AddMessage) which makes a copy of the message and stores it in an internal queue. ACK responses from the neighbouring node are routed to the Acknowledgement Manager (function MC_ACK_AcknowledgeMessage). It is assumed that the ACK is a response to the first message in the internal queue, because only one message may be outstanding at any time. The message will then be discarded from the queue, cancelling any resend operation.

If the application wants to send another message, and the Acknowledgement Manager reports that there is an outstanding message that is waiting for an ACK response, the application does not queue the message on the MAC for transmission, but just adds the message to the Acknowledgement Manager. This will place the new message in the internal queue, behind any previously scheduled messages. When a message is acknowledged and removed from the internal queue, the next message comes to the front of the queue. The Acknowledgement Manager does not automatically send the message, but marks it as 'ready to send.' The application checks for 'ready' messages for each RACH slot by calling the function MC_ACK_MessageReadyToSend(<RACH channel>). If this reports that a message is ready, the application reclaims the message (function MC_ACK_GetMessage) and pushes it into the appropriate RACH queue for the MAC to transmit it. The Acknowledgement Manager retains a copy of the message at the front of the internal queue and activates the back-off algorithm to schedule the retransmission of the message, should it be unacknowledged.

The back-off algorithm uses a random number generator to select a wait period, counted in RACH slots, before setting it back to 'ready to send.' The random number generator is seeded on initialisation with the Mesh Network Address, which will be different for every node on the network.

The back-off algorithm uses an incremental exponent to increase the scope of the delay that is applied before each retransmission. After several retransmissions have received no response, the message is discarded.

The delay is implemented as a countdown of slots on the appropriate RACH channel. The countdown is initialised with the number of slots to wait, initialised using the random number generator. The Mesh Application receives a message from the MAC for each RACH slot. On receiving this message the Application updates the Acknowledgement Manager, which decrements the countdown for that RACH channel. When the countdown reaches zero, the outstanding message is marked for retransmission.

A virtual channel, the Dedicated Uplink Channel (DULCH), is also defined, which is mapped to the secondary RACH (every 8th slot). The DULCH slot ensures that each device has its own RACH slot: It is used when many devices are expected to send a RACH message at a similar time. Using the DULCH means that each device takes a turn in sending a RACH message, thus allowing each one to dispatch a message before another device attempts to do so, and thus avoiding the packet collisions that might otherwise occur. The DULCH is described in detail in 2000-SPC-0007 Mesh Forming and Healing.docx.

3.3.4.2 Acknowledgement Manager public functions

MC_ACK_Initialise

This function is called to fully initialise the Acknowledgement Manager. It resets the internal message queues and back-off values. It also seeds the random number generator with the network address of the device.

Parameters

None.

Return void.

MC_ACK_AddMessage

The application should call this message if it wants the Acknowledgement Manager to process the acknowledgement of a message and manage the delay before resending.

If another message is already being managed, i.e. waiting for an acknowledgement, the new message is queued for sending after the outstanding message has been resolved.

If there are no outstanding messages, the new message is added to the queue and comes straight to the front, and an acknowledgement will be expected for the message.

Parameters

const uint32_t neighbourID An identifying number for the destination device. This should be the Mesh Network address of the destination device.

const AcknowledgedQueueID_t channel. The RACH channel for the message. The Acknowledgement Manager defines two RACH channels, ACK_RACHP_E and ACK_RACHS_E.

const ApplicationLayerMessageType_t MsgType. The message type that is being sent. This is used to report the type of message that has been acknowledged, without having to unpack the stored message.

Const bool MultiHop. This parameter is set to true if the destination is more than one hop away. If the initial hop is to a parent node, the retransmissions will be alternated between both parents.

Const uint32_t MsgHandle. This is the transaction ID that is supplied by the application layer when sending messages. It is reported back to the application when the message is sent.

Const CO_Message_t * pNewMessage. A pointer to the message to be sent. A copy of the message

is made and queued so that it can be resent if required.

Return bool True if the message was successfully queued, false otherwise.

MC_ACK_Acknowledge_Message

When the application receives an acknowledgement over the Mesh Network, it should call this function to notify the Acknowledgement Manager. The message at the front of the queue is discarded and the back-off delay is cancelled. If another message was queued, it is brought to the front of the queue and marked as ‘ready to send.’

Parameters

const AcknowledgedQueueID_t_t channelID The RACH channel for the acknowledgement. The Acknowledgement Manager defines two RACH channels, ACK_RACHP_E and ACK_RACHS_E.

ApplicationLayerMessageType_t* pMsgType. This pointer is an 'out' parameter which receives the message type that was acknowledged.

uint32_t* pMsgHandle. This pointer is an 'out' parameter. It is the transaction ID that is supplied by the application layer when sending messages. It is reported back to the application when the message is acknowledged.

Return bool True if the acknowledgement was matched to a message, false otherwise.

MC_ACK_WaitingForACK

The application can call this function to query whether there is an outstanding acknowledgement on a specified RACH channel. It returns true if a message is waiting to be acknowledged.

This enables the application to determine whether a message can be sent immediately, or if it should be queued (by calling MC_ACK_AddMessage).

Parameters

const AcknowledgedQueueID_t channel. The RACH channel for the acknowledgement. The Acknowledgement Manager defines two RACH channels, ACK_RACHP_E and ACK_RACHS_E.

Return bool True if there is an outstanding ACK on the channel, false otherwise.

MC_ACK_UpdateAckExpectedFlags

This function sets flags to indicate that acknowledgements are outstanding. These flags are set for quick access by the high priority MAC, which uses them to determine whether the unit can sleep through the next ACK slot.

Parameters

None.

Return None..

MC_ACK_UpdateSlot

The MAC informs the application layer that a new RACH slot has begun. The application calls this function to inform the Acknowledgement Manager. The Acknowledgement Manager uses this update to progress the backup algorithm and so determine when an unacknowledged message should be resent. The return value indicates whether a message became 'ready to send' after the update.

Parameters

const AcknowledgedQueueID_t channelID The RACH channel for the slot. The Acknowledgement Manager defines two RACH channels, ACK_RACHP_E and ACK_RACHS_E.

Return bool True if a message became 'ready to send', false otherwise.

MC_ACK_MessageReadyToSend

After the application informs the Acknowledgement Manager that a new RACH slot has occurred (i.e. called MC_ACK_UpdateSlot), it should call this function to discover whether the update resulted in a message becoming ready to send. If this function returns true, the application should call MC_ACK_GetMessage and put the returned message onto the appropriate RACH queue for transmission.

Parameters

const AcknowledgedQueueID_t channelID. The RACH channel to be queried.

Return bool True if a message is ready for retransmission, false otherwise.

MC_ACK_GetMessage

The application calls this function if the Acknowledgement Manager reports that it has a message that is ready to send (i.e. MC_ACK_MessageReadyToSend returns true). The returned message can then be placed on the appropriate RACH queue for transmission.

The Acknowledgement Manager retains its copy of the message for acknowledgement processing.

Parameters

const AcknowledgedQueueID_t channel. The RACH channel to be queried.

Const CO_Message_t * pMessage. A pointer to a message object that the 'ready' message should be copied into.

Return bool True if a message was copied into pMessage, false otherwise.

MC_ACK_GetRandomDelay (private function)

The Acknowledgement Manager implements a back-off delay to determine when an unacknowledged message should be resent. Part of this process requires the selection of a random slot within a range of values, determined by the back-off exponent.

This function implements the selection of the random slot in which the message should be resent.

Parameters

const uint32_t backOffExp The exponent to be used when selecting the random slot. This sets the range of slots that the random slot should be chosen from.

Return uint32_t The delay, in slots, that was selected.

MC_ACK_RescheduleMessage (private function)

The Acknowledgement Manager calls this function internally to determine when a message should be resent. It manages the back-off exponent value, which determines the range of slots that may be selected for the retransmission, then calls upon function MC_ACK_GetRandomDelay to select a slot within the range.

If the back-off exponent has been incremented to its maximum value, the decision is taken to stop resending the message and discard it. This is done by making an internal call to MC_ACK_AcknowledgeMessage, which removes the message from the front of the queue and moves on to the next one.

Parameters

const uint32_t neighbourID The identifier for the network device that the message was destined for. This should be unique to the parent node, typically the Mesh Network address of the unit.

const uint32_t channel The RACH channel for the message. The Acknowledgement manager defines two RACH channels, ACK_RACH1 and ACK_RACH2.

Return bool True if the resend was scheduled, false if the message was discarded.

3.3.5 Message Packing and Unpacking

Message packing is performed to minimise the length of transmitted messages.

The application software utilises structures to represent each message type. The structures uniformly use 32 bit values for each message property. When the application needs to send a message over the radio, it must first reduce the size of the message properties (i.e. the number of bits used to represent the property) and pack them into a continuous binary packet.

The messages and their properties are defined in the Cygnus 2 Mesh Protocol Design document 2001-SPC-0012.

Message packing is performed in the source file MC_PUP.c. The corresponding header file defines the bit lengths to be used for each message property.

Message packing is implemented using a shift buffer, managed by functions defined in MC_PUP.c. Each message type has a corresponding MC_PUP_PackXxx() function, where Xxx is the name of the message to be packed e.g. MC_PUP_PackAlarmSignal(). The MC_PUP_PackXxx() functions pass each message property to the shift function to be packed into the binary packet. When all of the properties are packed, the contents of the shift buffer are transfered into a message structure for transmission.

Special consideration is given to the Application Payload section of the message. This must be encrypted before being included in the transmission packet (except for heartbeats). Encryption requires a minimum message size of 64 bits. Since almost all of the Application Payloads are shorter than this, a padding function has been implemented to pack extra zeros to the payload.

Finally, a message integrity check (MIC) field is added to the end of the packet. This is calculated by applying an encryption algorithm to the entire packed message and appending the first four bytes of the result to the packet. Note that we do not encrypt the message with this process, we only supply it to the algorithm to calculate the MIC.

Unpacking is achieved using a reverse of the packing procedure. MC_PUP.c contains unpacking functions, declared with the format MC_PUP_UnpackXxx(), where Xxx is the message name e.g. MC_PUP_UnpackAlarmSignal().

A message is received over the radio in the form of a binary packet, which is copied into the shift buffer. MC_PUP.c has functions defined to enable the message type to be identified from the packet contents. The message is then shifted out of the shift buffer into an application structure under the control of the appropriate MC_PUP_UnpackXxx() function.

3.3.6 Session Management

Each node in the Mesh Network must maintain its place in the network by monitoring its parent and child nodes. This is the responsibility of the Session Management, implemented in MC_SessionManagement.c.

When an RBU joins the network, it selects two nodes as candidate parents, based on SNR, the candidate's rank within the mesh and the number of children it has. The Session Management sends a Route Add message to both candidates. On receiving an acknowledgement from a parent node, the Session Management marks it as an active parent. Uplink messages will be routed via the active parents.

In the parent node, the Route Add message is received by the Session Management and the calling node is added to its list of child nodes. The heartbeat signal for the child is monitored from that point onwards. If several consecutive heartbeats are missed, the Session Manager sends a fault message to the NCU, notifying it that the child node has stopped communicating and the node is removed from the child list.

The Session Management updates the MAC DCH behaviour for each addition or removal of a parent/child node. This enables the MAC to activate the LoRa receiver only when a heartbeat is expected.

The heartbeats of the parent nodes are monitored in a similar way, but missing heartbeats are not reported to the NCU. The Session Management keeps a count of the number of consecutive missed heartbeats for each parent, child and tracking nodes.

The Session Management is also responsible for deciding the network destination address for uplink messages. It alternates the address between the primary and secondary parents to provide route diversity for uplink messages.

If the decision is taken to drop a parent node, due to missing heartbeats or low signal quality, the Session Manager will promote a tracking node to replace the parent. If the RBU wishes to change a parent, it sends a Route Drop message to the parent. In the parent, the message is directed to the Session Manager which sends an acknowledgement and removes the child from its list.

3.3.6.1 Session Management Public Functions

MC_SMGR_Initialise

Initialises the data structures used by the Session Management for monitoring parent and child nodes.

INPUTS:

None.

RETURN:

None.

MC_SMGR_SetParentNodes

Called to update the Session Management with candidate parent nodes. The parent selection routine identifies the candidates by the DCH slot in which they transmit their heartbeat. For convenience, this function accepts the slot number and converts it to a node ID. If the slot number does not correspond to a DCH slot, the parent is rejected.

INPUTS:

Uint16_t primary_dch_slot. The DCH slot in which the primary parent transmits its heartbeat.

Uint16_t secondary_dch_slot. The DCH slot in which the secondary parent transmits its heartbeat.

RETURN:

Bool. True if the parents were accepted

MC_SMGR_IsParentNode

Called to find out if a node ID corresponds to a parent of the RBU.

INPUTS:

Uint16_t node_id. The node ID you want to query.

RETURN:

Bool. True if the node ID matches one of the parent nodes.

MC_SMGR_GetNextParentForUplink

Called before a RACH uplink message is sent in order to get the network address (node ID) of the parent to which the message should be sent. The returned value will alternate between parent node IDs for each call so that some route diversity is utilised by the system.

INPUTS:

None.

RETURN:

Uint16_t. The node ID of the selected parent.

MC_SMGR_RouteAddRequest

This function is called when another node selects the local node as a parent and sends a Route Add message. This function adds the calling node to the list of child nodes

INPUTS:

Uint16_t node_id. The node ID of the new child.

RETURN:

Bool. True if the node ID was added to the child list.

MC_SMGR_RouteAddResponse

Called when a parent node acknowledges a Route Add message. The parent node is marked as 'active' and will be used for routing future uplink messages.

If no acknowledgement was received, the 'response' parameter will be false and the parent will not be used to route messages.

INPUTS:

Uint16_t node_id. The node ID of the parent.

Bool response True if the parent node accepted the Route Add request. False otherwise.

RETURN:

Bool. True if the node ID matched a parent ID, and the active status was updated.

MC_SMGR_RouteDropRequest

Called when a child node sends a Route Drop message. This function removes the calling node from the list of child nodes

INPUTS:

Uint16_t node_id. The node ID of the child.

RETURN:

Bool. True if the node ID was found and removed from the child list.

MC_SMGR_IsAChild

Called to find out if a node ID corresponds to a child of the RBU.

INPUTS:

Uint16_t node_id. The node ID you want to query.

RETURN:

Bool. True if the node ID was found in the child list.

MC_SMGR_MissedNodeHeartbeat

Called each time a heartbeat is not received from another node. If the node ID matches a parent or a child, a count is incremented for the node. After a predetermined number of consecutive missed heartbeats, the node is removed from the child list.

INPUTS:

Uint16_t node_id. The node ID for the missed heartbeat.

RETURN:

Bool. True if the node ID was found in the parent/child list and the count was updated.

MC_SMGR_GetNumberOfChildren

Returns the number of children in the child list. Called when populating heartbeat messages.

INPUTS:

None.

RETURN:

Uint16_t.   The number of children.

MC_SMGR_SetHeartbeatMonitor

Used to activate or deactivate the heartbeat monitor for a specified DCH slot. If the second parameter is set to true, the Session Manager will configure the MAC to wake up the LoRa receiver for the DCH slot to listen for a heartbeat. If it is false, the MAC is configured to sleep through the DCH slot.

INPUTS:

Uint16_t super_slot_index.   The super frame slot index of the DCH slot.

Bool.   True to start listening for heartbeats, false to stop.

RETURN:

Bool.   True if the MAC behaviour was updated.

MC_SMGR_ParentSessionActive

Returns true if communication with either parent node is open. Used to determine when the node should revert its state from OPERATIONAL to CONNECTING.

INPUTS:

None.

RETURN:

Bool.   True if either parent is communicating.

3.3.7 Mesh Forming and Healing

Mesh forming and healing is described in the document 2000-SPC-0007 Mesh Forming and Healing.docx.

3.3.8 MAC

MC_MAC.c
MC_MAC.h

Table 13 MAC files

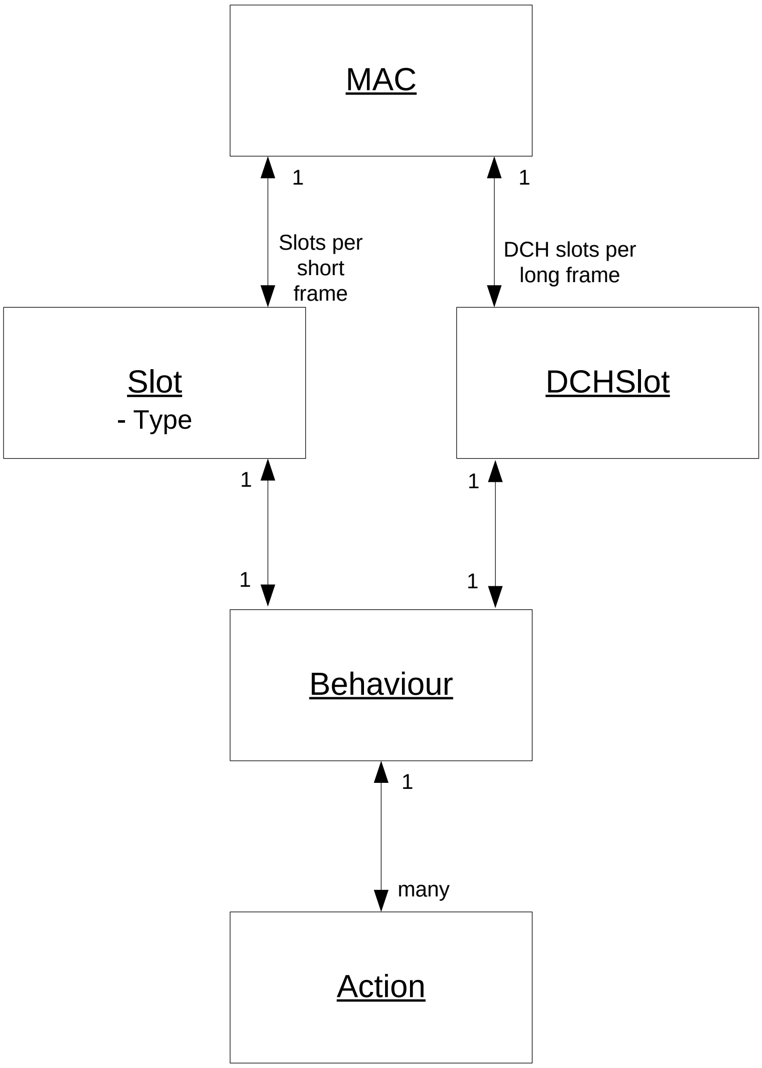


Figure 24: MAC Data Model

The MAC module maintains the TDM slot structure. The data model for the MAC is shown in Figure 24.

The MAC tracks all the slots in a short frame and all the DCH slots in a long frame. These types for each of these slots are configured at start time and do not change.

Each slot and DCH slot is allocated a behaviour. This describes how the MAC behaves in any given slot. The behaviour of each slot changes dynamically as the RBU joins the mesh and when children join or leave.

Each behaviour has a number of actions associated with it. Each action is a timed event triggered by the LPTIM comparator. All of the actions associated with a single behaviour take place in the same slot. Most behaviours have two actions – the first to configure the radio modem, and the second to start the transmit / receive operation.

3.3.9 DLCCH – Lost Message Management

The down-link control channel issues broadcast commands which originate from the NCU and are rebroadcast by each node down the ranks to the bottom of the mesh network.

A problem is seen where two nodes broadcast in the same slot and the resulting collision cannot be decoded by the nodes at the next rank. This means that some nodes may not receive the command.

In the case of State change commands, there is an automatic recovery as each node monitors its parent's rank in the heartbeat signal. If the parents advance to a higher state, the child node automatically raises its state to the same level.

Output Command messages must always be delivered, so there is a procedure that each primary parent follows to verify that its child nodes received the command, as described below.

3.3.9.1 Output Command Delivery verification

The NCU and RBU application layers implement a state machine that is invoked when an Output State command is issued. The State Machine manages the process of verifying that all child nodes have received the Output Command. The sequence that is implemented by the state machine is shown below.

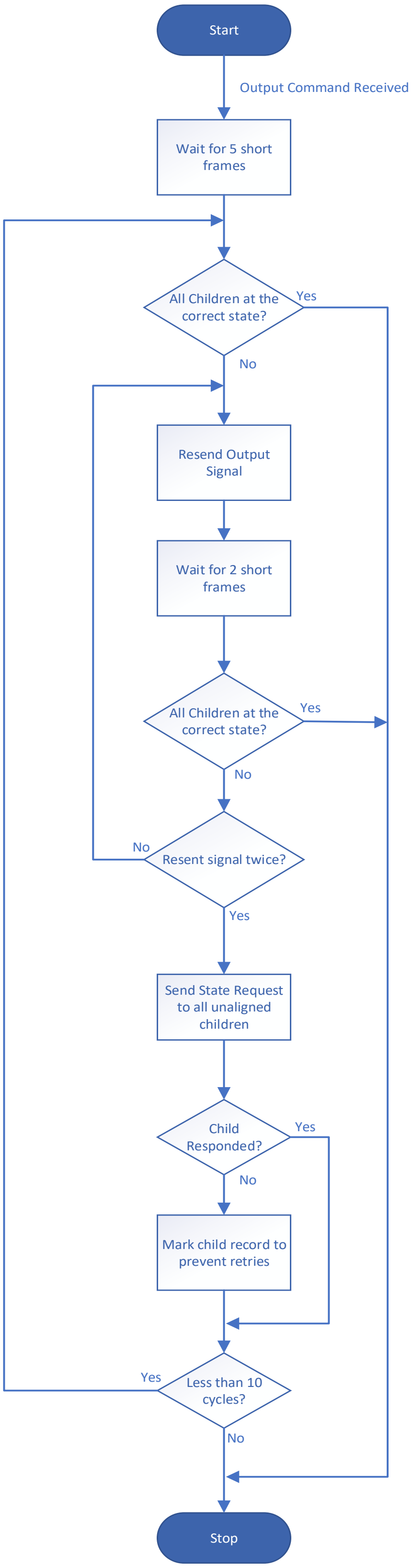


Figure 25: Child Output State Verification Flow Diagram

The functions that implement the above process are as follows. Note that the NCU functions are prefixed MM_NCU.

MM_ApplicationProcessNewShortFrame

Implements the state machine that is shown in the figure above. The mesh stack sends a message to the application layer for each new short frame. This function is called on receipt of that message and executes the procedure for the current state.

INPUTS:

Uint32_t ShortFrameIndex. The index of the short frame in the TDM.

RETURN:

None.

MM_ApplicationRequestChildOutputState

Iterates through the child records for children that do not report the correct output state and sends an Output State Request message for that node to report its output state. The child is only queried if the current node is the child's primary parent and no request was sent previously. It will also ignore any child that has been marked unresponsive (the 'isMuted' flag is set to true in the child record).

INPUTS:

None.

RETURN:

Bool. True if an Output State Request was sent, false otherwise

MM_RBUOutputStateRequestCnfCb

A call-back function that the Mesh stack calls when an Output State Request message has been acknowledged by the child node, or to report an error in sending the message.

INPUTS:

Uint32_t Handle. A unique tag that the application generated for the request.

Uint32_t New Handle. If the message is replaced in the transmission queue by a newer duplicate, this parameter identifies the new handle to associate with the child request.

Uint32_t Status Indicates the success of the message or the error code if it failed to send.

RETURN:

None.

MM_RBUOutputStateRequestIndCb

The Mesh stack calls this function in the child node on receipt of an Output State Request from the parent. It places a message in the application message queue to generate the response.

INPUTS:

None.

RETURN:

None.

MM_RBUOutputStateMessageIndCb

The Mesh stack calls this function in the parent node on receipt of an Output State message from a child. This is the response to the parent's Output State Request. It places a message in the application message queue to update the child record.

INPUTS:

Uint16_t nodeID. The network ID of the responding child.

Uint8_t outputs. The last commanded output that the child actioned.

RETURN:

None.

3.3.10 Programmed Zones

Devices on the Mesh may be grouped into zones, enabling the control panel to set the outputs of one part of the system without affecting the others.

Each RBU in the system can be assigned to a zone by programming the zone number into its non-volatile memory. Unprogrammed devices default to zone 1.

Programming the zone number can be done locally using the ATZONE command over the serial interface, or remotely from the NCU using the ATCMD command. Both commands also support the read option.

The down-link OUTPUT message contains a zone field which the Control Panel populates when sending the command. On arriving at the RBU the zone number in the message is compared to the programmed zone in non-volatile memory. If the two match the RBU sets its outputs accordingly.

The range of zones is limited to 1→4094 by the 12-bit field in the message. Zone number 4095 (0xFFF) is reserved for activating all zones.

The primary parent is responsible for verifying that a child has set its outputs correctly. The child uses the Output State message to relate its output state, and its programmed zone number, to the parent. The Primary Parent keeps a record of the last OUTPUT command that applies to each child, enabling it to verify that the child's reported outputs are correct for the latest command.

3.3.11 PPU

The radio board has two PPU components.

1) PPU management in the main Radio Board Software.
2) The PPU Bootloader.

Together, these enable the radio devices to be configured or reprogrammed over the radio.

To perform configuration or firmware updates the radio board needs to be dropped from the mesh and put into PPU DISCONNECTED mode. In this state, the radio board changes its radio frequency to channel 10, outside the normal operating band of the Cygnus2 system. It generates randomly spaced announcement messages and transmits them over the radio. The announcement message contains identifying fields for the radio device.

A PPU Master device will pick up the announcement signal and inform the Cygnus Config tool that the device has been found. The Cygnus Config tool may then put the device int PPU CONNECTED mode. In this state the RBU stops sending announcements and will accept configuration commands from the PPU Master.

The RBU software has several states that it could be in during PPU mode. They are identified by this enum, defined in CO_Messages.h.

```
typedef enum
{
   PPU_STATE_NOT_ACTIVE_E,      //Not in ppu mode (normal operation)
   PPU_STATE_REQUESTED_E,       //Starting PPU mode but waiting for outstanding
event (e.g. ROUTE DROP confirmation)
   PPU_STATE_DISCONNECTED_E,    //PPU Disonnected. Waiting for PPU Master to
connect. Announcement messages broadcast.
   PPU_STATE_CONNECTED_E,       //PPU Connected.  PPU Master connected. No
announcement messages.
   PPU_STATE_AUTO_DISCONNECT_E,//Automaticcally go to the PPU Disconnected state
on startup
   PPU_STATE_AUTO_CONNECT_E,    //Automaticcally go to the PPU Connected state
(used after firmware update reboot)
   PPU_STATE_MAX_E
} PpuState_t;
```

The RBU starts in PPU_STATE_NOT_ACTIVE_E. To get it into PPU Disconnected mode, there are two mechanisms, one for a meshed RBU and one for an RBU that is not meshed.

1) Meshed. The panel sends a command when the user clicks the “Drop to PPU” button.
2) Unmeshed. The PPU Master sends a message over the radio.

The first one comes as a downlink command. Just search for PARAM_TYPE_ENTER_PPU_MODE_E.

The second comes over the radio from the PPU Master when the RBU is in idle mode i.e. waiting to see its first heartbeat after reset. The message uses a special frame type. All RBU radio messages begin with a 4-bit frame type, define in CO_Message.h.

```
/* Frame Types */
typedef enum
{
   FRAME_TYPE_HEARTBEAT_E,
   FRAME_TYPE_DATA_E,
```

```
    FRAME_TYPE_ACKNOWLEDGEMENT_E,
    FRAME_TYPE_AT_E,
    FRAME_TYPE_PPU_MODE_E,
    FRAME_TYPE_TEST_MESSAGE_E,
    FRAME_TYPE_ERROR_E,
    FRAME_TYPE_MAX_E
} FrameType_t;
```

When the MAC receives the message it drops it in the queue for the Mesh Task. The mesh task, in MM_MeshTaskMain, reads the queue and picks the frame type out of the message by calling MC_PUP_GetMessageType(). It maps this to an event for the state machine (SM_StateMachine.c) by calling MM_MapMessageToEvent (), to route it to the correct state machine hander function when SM_HandleEvent() is called. It ends up in function ppuModeMesssage() which checks that the desination address matches, then sends it to the Application via the queue AppQ. It ends up in function MM_PPU_ApplicationProcessPpuCommand() in MM_PpuManagement.c, where the RBU handles the PPU commands.

3.4 Device Manager

3.4.1 Non-Volatile Memory

DM_NVM.c
DM_NVM.h

Table 14 Non-Volatile Memory driver files

The NVM driver provides an interface for accessing non-volatile parameters located in the EEPROM on the STM32 processor. Functions are provided to read or write values up to 32 bits wide.

The table below lists the parameters supported and the width of each parameter.

Parameter	Offset in EEPROM	Parameter range	Default	Comments
Bootloader Use PP UART	0	0-1	0x1	Use PPU UART = 1, Use Debug UART = 0
Address	4	0-0xFFFF	0x0020	Address = 0, Zone = 1
Is Sync Master	8	0-1	0x0	RBU = 0, NCU = 1
Frequency Channel	12	0-15	0x0	Channel 0
Device Combination	16	0-0xFF	0x00	No devices fitted, repeater
System Id	20	0-0xFFFF	0x0001	Id default value is 1

Unit Serial Number	24	0-0xFFFF	0x0000	N/A
Tx Power Low	28	0-15	0x7	Set to 7 dB
Tx Power High	32	0-15	0xA	Set to 10 dB
Zone	36	0-512	1	
PP Mode Enable	40	0-1	1	

Table 15 NV Parameters

3.4.2 UART Driver

DM_SerialPort.c
DM_SerialPort.h

Table 16 UART driver files

The Serial Port driver files provides functions to configure the USART module as well as for accessing the interface for sending and receiving data. In addition, this driver provides functionality for:

- Handling USART RX Interrupt
- Handling USART TX Interrupt
- Writing TX buffer
- Reading RX buffer
- Determining the number of bytes in RX buffer
- Searching <CR><LF> (0x0D, 0x0A) sequence in input buffer
- Custom Debug print function to transmit data over the debug port

Two circular buffers per USART module are implemented. The RX buffer has a size of 128 bytes and the TX buffer has a size of 64 bytes, these can be modified if necessary by changing #define found in DM_SerialPort.h.

The receive interrupt handler routine will copy all the received bytes into the RX buffer and if the buffer is full the bytes are discarded. The main task or program shall check regularly the RX buffer to process the incoming bytes. The function “SerialPortWriteTxBuffer” copy bytes into the TX buffer and enables the USART interrupt which issue the bytes out.

To configure serial port settings of the USART module use function “SerialPortInit”, this function accepts parameters for setting up the baud rate speed and the parity.

The function “SerialPortDebugPrint” is provided as a custom debug printout tool for sending messages over the debug port. This function accepts format parameters to help the creating of the message.

The function “SerialPortWriteTxBuffer” works by appending the data parameter to the circular TX buffer which is later on accessed during the TX interrupt process when a message is being printed out.

3.4.2.1 USART Code Example

Initialization code example

```
// (1) Initialization of USART data structure and pointers

// (2) Serial port settings configuration: 115200bps, no parity

SerialPortDataInit(DEBUG_UART);      // (1)

SerialPortInit(DEBUG_UART, 115200, UART_PARITY_NONE);  // (2)
```

Transmit message example

```
// (1) Data buffer declaration. Populate buffer with test message.

// (2) Write TX buffer and send message via the serial port

uint8_t test_string[] = "Hello World";  // (1)

SerialPortWriteTxBuffer(DEBUG_UART, test_string, strlen((char*) test_string));  // (2)
```

Debug print example

```
// (1) Declare and load register

// (2) Print debug message

uint8_t aux_reg = 2; // (1)

SerialPortDebugPrint("Cygnus%d Ready\r\n", aux_reg);          // (2)
```

3.4.3 Input Monitor

DM_InputMonitor.c
DM_InputMonitor.h

Table 17 Input Monitor driver files

The Input Monitor driver provides the following functions:

- Configuration of the Input to be monitored
- Initialise the monitoring data
- Maintains the current state of the monitored Inputs
- De-bounces values read from pins and decides when to change state
- Provides a flag for whether polling is enabled / disabled

The following inputs are monitored:

- Fire
- First Aid
- Tamper1
- Tamper2
- Low Main Battery

Each Input can be in any of the following states:

- Idle
- Active

The Input Monitor driver is called by the GPIO task module. This initialises the input monitoring in use at startup and contains a task for polling the inputs.

The Input interrupts are configured to trigger on either edge. When it detects activity the ISR uses a binary semaphore to start the Polling task. This reads the value from each monitor enabled input pin at a fixed interval.

Each time a value is read from the input pin the DM_InputMonitorPoll () is called. The Input Monitor uses the current state, the read value and a timer vallue to decide when the Input should change state. If a state change is confirmed then this is communicated using a callback function.

When the state of all the monitored inputs are confirmed the polling is considered disabled and the GPIO task enters a sleep state until a new event occurs.

3.4.4 The Independent Watchdog - IWDG

DM_IWDG.c
DM_IWDG.h

Table 18 Independent Watchdog driver files

The independent Watchdog – IWDG generates a system reset when it is not serviced for a certain delay, this allows the system to recover from a Software failure. This can be very helpful in the case of a SW design issue or a HW failure. For instance, the micro will be reset if the SW will wait for too long through a while loop for an event that does not occur as expected.

If the system restarts due to a watchdog reset a flag is set in the hardware (RCC_FLAG_IWDGRST). The software checks for the presence of this flag and reports a watchdog reset over the debug logging output.

The Watchdog is serviced in the Idle Task in *os_idle_demon*() function, this way we ensure that all of the other threads have run to completion.

The Macros defined in CO_Define.h and listed below help to Enable and Configure the IWDG:

- IWDG_ENABLE*:* when defined, the IWDG is enabled through the call of *DM_IndependentWdgInit()* function is called from the main module
- IWDG_TIMEOUT_PERIOD_MS: defines the timeout in Ms for the Watchdog to trip if not serviced and generates a System reset
- IWDG_DEBUG_FREEZE: allows Debug operations while the IWDG is enabled. For instance, no Reset would be generated when the CPU hits a breakpoint

3.4.5 LED driver

DM_LED.c
DM_LED.h

Table 19 LED driver files

The LED driver controls the tri-colour LED on the Radio Base. The possible states supported by the LED are:

- Off
- Red
- Green
- Blue
- White

The driver can be configured to execute a sequence of states of the LED. The sequence is defined as a list of four states. The driver can either run through the sequence once and then stop or repeat continuously. A repeatable sequence can be configured to have a limited execution time.

The generation of a sequence can be either controlled by the driver itself through the DM_LedPeriodicTick() which is called by the idle daemon or by an external event through the DM_LedExternalControl() API that specifies what step of the requested pattern to be applied

When the driver reaches the end of the sequence it stops or goes back to the start depending on the current configuration.

Each step of a pattern can have a determined limited duration or unlimited if set to zero

Table 20 contains the flash sequence configurations. These are defined in the state machine and passed to the LED driver when the RBU enters a state.

State	Step 0	Step 1	Step 3	Step 4	Is it Repeated
CONFIG	Red	Off	Off	Off	Yes
ACTIVE	Green	Green	Green	Off	No
TEST MODE	Blue	Off	Blue	Off	No
BIT Failure	RED	OFF	N/A	N/A	Yes

Table 20 LED Flash Sequences

3.4.6 I2C Driver

DM_i2c.c
DM_i2c.h

Table 21 I2C driver files

The I2C driver provides an initialisation function, to set up the I2C interface, a read function and a write function, which manage the transfer of data over the I2C link during a transaction.

The RBU is not permitted to sleep during an I2C transaction. For this reason, the read and write functions are sequential operations. It is acceptable for a higher priority task to interrupt the transaction because the I2C interface employs clock stretching to maintain synchronisation.

3.4.6.1 I2C Driver Functions

DM_I2C_Initialise

Initialises the I2C hardware for data transfer on the RBU.

INPUTS:

Uint32_t i2c_address. The bus address to be configured on the RBU.

RETURN:

ErrorCode_t An error code reporting success or failure of the operation.

DM_I2C_Read

Manages a 'read' transaction between the RBU and the SVI board. Parameters indicate which SVI register is to be read and the returned value is available when the function ends.

The function allows multiple bytes to be returned (consecutive memory locations in the SVI starting at the supplied register address). In practice, we will only read single bytes, as defined in the SVI protocol document HKD-17-0153-D.

INPUTS:

Uint8_t device_addr. The I2C bus address of the SVI board.

Uint8_t register_addr. The address of the I2C register to be read.

Uint16_t data_length. The number of bytes to be transferred.

Uint8_t* rx_buffer. [OUT] parameter which receives the returned data.

Uint16_t rx_buffer_size. The size, in bytes, of rx_buffer to ensure that no buffer overrun occurs.

RETURN:

ErrorCode_t An error code reporting success or failure of the operation.

DM_I2C_Write

Manages a 'write transaction between the RBU and the SVI board. Parameters indicate which SVI register is to be updated and the value to be written.

The function allows multiple bytes to be written (consecutive memory locations in the SVI starting at the supplied register address). In practice, we will only write single bytes, as defined in the SVI protocol document HKD-17-0153-D.

INPUTS:

Uint8_t device_addr. The I2C bus address of the SVI board.

Uint8_t register_addr. The address of the I2C register to be updated.

Uint16_t data_length. The number of bytes to be transferred.

Uint8_t* data_buffer. The data to be written to the SVI.

RETURN:

ErrorCode_t An error code reporting success or failure of the operation.

3.4.7 SVI Driver

DM_svi.c
DM_svi.h

Table 22 SVI driver files

The SVI driver implements the protocol for communication between an RBU and a Sounder and Visual Indicator (SVI).

An initialisation function accepts the I2C address of the RBU and initialises the I2C driver.

Read and Write functions provide an interface to the SVI registers. The write function filters out any attempts to write to read-only registers.

3.4.7.1 SVI Driver Functions

DM_SVI_Initialise

Configures the I2C address of the RBU.

INPUTS:

Uint32_t i2c_address. The bus address to be configured on the RBU.

RETURN:

ErrorCode_t An error code reporting success or failure of the operation.

DM_SVI_GetRegister

The RBU application uses enumerations to manage the SVI read and write commands. This function returns the SVI register address for a given enumeration.

INPUTS:

Uint8_t command. The command enumeration used by the RBU.

Uint8_t* registerAddress. An 'OUT' parameter used to return the value of the requested register.

RETURN:

ErrorCode_t An error code reporting success or failure of the operation.

DM_SVI_ReadRegister

This function reads the value of the specified SVI register and returns it.

INPUTS:

Uint8_t registerAddress. The register address to be read.

Uint8_t* pResponse. An 'OUT' parameter used to return the value that was read.

RETURN:

ErrorCode_t An error code reporting success or failure of the operation.

DM_SVI_WriteRegister

This function writes a value to the specified SVI register. If the supplied register address is for a read-only register, the command is rejected and an error code returned.

INPUTS:

Uint8_t registerAddress. The register address to be written to.

Uint8_t value. The value to be written.

RETURN:

ErrorCode_t An error code reporting success or failure of the operation.

3.5 RBU Bootloader

The bootloader SW component is first to be executed after the RTX RTOS initialisation. The BLF_Main module is in charge of a minimal configuration comprising just that needed by the boot/reprogramming operations and calls the Boot manager module. On a Firmware update request the App Updater Manager is called to handle serial port Ymodem protocol communication and Flash firmware reprogramming. If no reprogramming request is received. the SW enters application mode. The bootloader is due to revert to a standalone application.

3.5.1 App Updater

BLF_AppUpdater.c
BLF_AppUpdater.h

This module implements the features related to the reprogramming operation such us initiating the Ymodem protocol and verifying the application integrity.

3.5.2 Boot

BLF_Boot.c
BLF_Boot.h

This module is responsible of the boot operation and decides whether we can proceed to the application or remain in the bootloader mode waiting for a new application to be downloaded

3.5.3 Main

BLF_Main.c
BLF_Main.h

This module contains the main() function for the software. This is called after OS initialisation.

3.5.4 Serial Interface

BLF_serial_if.c
BLF_serial_if.h

This is a wrapper of the serial Interface Device manager in order to build the API and blocks that are necessary to the bootloader especially the modules originally provided by ST such as the Ymodem

3.5.5 Common

common.c
common.h

Common functions and the features for the bootloader modules

3.5.6 Ymodem

ymodem.c
ymodem.h

Implements the Ymodem protocol, this was initially provided by ST with AN4767-STM32L_Dualbank_flash_memory_fieldupgrade demonstration project but was adapted to use the DMA through the serial Device manager.

3.6 PPU Bootloader

The PPU bootloader will do one of three things on startup, depending on the value of variable rbu_ppu_mode_request. It can be one of:

STARTUP_NORMAL. Transfers control to the main RBU application for a normal start-up.

STARTUP_CONNECTED. Transfers control to the main RBU application, which will go straight to PPU CONNECTED mode.

STARTUP_PPU_DISCONNECTED. Transfers control to the main RBU application, which will go straight to PPU DISCONNECTED mode.

STARUP_SERIAL_BOOTLOADER. Invokes the code that expects a firmware update over the serial link (managed in file PPUB_SerialUpdate.c). This is almost a clone of the first bootloader, installed for when the first bootloader is removed. It is not called in the current version.

STARTUP_RADIO_BOOTLOADER. Invokes the code that expects a firmware update over the radio (managed in file PPUB_RadioUpdate.c).

On a normal start-up, the main function calls PPUB_PMGR_ValidateApplicationFirmware(), which does a checksum on the main RBU code area and comperes it to the value held in FLASH. If they match, control is transferred to the main RBU code. If the checksum fails, the radio updater is invoked.

The main focus for this document is the radio updater code in PPUP_RadioUpdate.c.

3.6.1 Radio Firmware Updater

The main function starts a radio update by calling PPUB_RadioFirmwareUpdate(). This function initialises the radio and sets it to start receiving. It then enters a while loop which cycles until the update is complete, or abandoned.

The while loop does the post processing on any radio interrupts, checking flags to see whether an interrupt gas occurred, in much the same way as the MACTask does in the radio board software. In this case, however, the loop does not block on a semaphore, it free-runs continuously. AT the end of the loop there is a 1ms delay, setting the cycle period of the loop.

After processing the interrupts, if there were any, the loop calls PPUB_PerformRoutineTasks(). This function does the routine stupp like flashing the LED, checking for timeout, sending the PPU Announcement message and testing for completion of an update.

Incoming radio messages are read from the LoRa chip by the SX1272 library code in response to an interrupt. The ISR in sx1272 calls function PPUB_OnRxDone(), passing the payload and other radio metrics, all of which are stored in globals for later processing. The received payload is copied into global buffer gRxBuffer, and the gRxDoneFlag is set to true.

The while loop checks gRxDoneFlag on its next iteration. Function PPUB_Deserialise() is called to decode the received payload into a global structure called gRxMessage. It also performs a checksum to validate the message.

The while loop then checks that the message is addressed to this device (PPUB_AddressMatch()) then passes it to PPUB_ProcessRadioMessage() to be actioned. This is a simple switch statement on the received command, which calls the appropriate handler function. It should be fairly straight forward to read through those functions.

Two files worthy of mention are:

PPUB_FlashInterface.c This is a straight copy of library code from STMicroelectronics. It handles the low level tasks for writing to FLASH.

PPUB_RadioPageManagement.c This is our application code for managing a firmware update over the radio. Much of it is straight forward and easily picked up by following the code.

One noteworthy thing is the way that the code tracks which packets have been received.

The PPU Master sends a config message which says how many packets (messages) make up the firmware update, and how many bytes each one has in the payload. Typically the payload is 32 bytes, but we can configure that.

The software maintains an array called gRxPacketBitfield. This array is used as a bit-field, with one bit representing each expected message. The bit-field is zero'd at the start of the update and, as each message is successfully written to FLASH, the corresponding bit is set to '1.'

When all packets have been sent, the PPU Master asks each device which packets they missed. The bitfield is used to build a response, where a '0' in the bitfield corresponds to a lost packet, and the location of the bit in the array corresponds to the packet number.

3.7 Data Interfaces

3.7.1 Radio Interface

Signals from the radio are decoded into digital packages by the SX1273 radio chip. The SX1273 is connected to the MCU via a SPI serial interface, over which the data is transferred. Three GPIO lines are connected to the SX1273 to provide event interrupts. The connections are as follows.

PA6 : SX_MISO (data transmit)

PA7 : SX_MOSI (data receive)

PA5 : SX_SCK (serial clock)

PB7 : SX_DIO0 (transmit/receive event)

PB5 : SX_DIO1 (Receiver timeout)

PB3 : SX_DIO3 (channel activity detect)

3.7.2 PPU Serial Interface

The PPU serial interface utilises USART1 on the MCU. It is used for configuration settings, and communications with the control panel (NCU configuration) or for firmware updates (RBU configuration). The hardware serial lines connect to the MCU on pins:

PA9 : Data transmit

PA10 : Data receive

3.7.3 Plugin Serial Interface

Used for communications with a plug-in sensor or beacon head. It utilises LPUART1 connected to MCU pins:

PC1 : Data transmit

PC0 : Data receive

3.7.4 Debug Interface

The debug interface is used for logging purposes in a test environment, and can also be used to update configuration values. It utilises UART4 connected to MCU pins:

PC10 : Data transmit

PC11 : Data receive

3.7.5 SVI Interface

The SVI interface is used for communications with the Sounder and Visual Indicator board. It utilises the I2C2 serial interface connected to MCU pins:

PB10 : I2C2_SDA (bidirectional serial data)

PB11: I2C2_SCL (serial clock)

4 SOFTWARE DEVELOPMENT TOOLS

4.1 Programming Language

- C language is selected as the main programming language during coding and will be written following the PA C Coding Standard (QS505)

4.2 Software Development Environment

- The software development environment is Keil µVision 5
- The ARM compiler tool chain is used

4.3 Revision Control System

- GIT is used to manage changes in source code and programs.
- The repositories are hosted on Git Hub. They are configured to allow access from developers at PA and from Cygnus Group Ltd.

4.4 CUnit Automated Test Framework

- CUnit has been selected as the test framework.
- The automated tests are run on a PC and will automatically generate a test report for each test run.

4.5 Bug Tracking

- JIRA has been selected as the bug tracking tool
- New tasks and bugs which are found can be recorded such that all team members can access and updated them
- As bugs are fixed progress can be recorded and reports can be generated.

5 PC TOOLS

5.1 Mesh DLL

Mesh DLL is a Dynamic Link Library for use on 64 bit PCs. It provides a function to decode mesh messages that are expressed as hex byte strings.

The code consists of MC_PUP.c (shared with the MCU code) and a simple application in mesh.c. This is compiled using MS Visual studio.

The decode takes in hex bytes and attempts to decode the bytes. The output is returned to the caller of the function.

The mesh DLL is used by the following PC applications:

- GUI Decode.py
- Log Decode.py
- Log Parse.py

5.2 Python

A number of python scripts have been created to add software development and test.

The scripts all use python 3.6. Some scripts use the pyserial module.

Script Name	Options	Description
firmware_update.py	none	Prompts the user for a binary file then upgrades all the units in a test rig. This uses the configuration in terminal_configurations
guidecode.py	None	Graphical User Interface for decoding mesh messages. This requires mesh.dll to be present in the same directory as the python script.
logdecode.py	-h help -l list com ports -f INFILE select file input -c COMPORT select serial input	Command line program to take input from either a file or a COM port. This data is decoded with results printed on the standard output. This requires mesh.dll to be present in the same directory as the python script.
logparse.py	<input files> system_ID	Python script to analyse a number of log files collected from RBUs.

Script Name	Options	Description
		When this is invoked the input files and the system ID must be specified. Wildcards can be used for files names. The script collects statistics on the log files and produces a number of CSV files to help analysis. The DCH_analysis file contains numbers of heartbeats transmitted and received. This requires mesh.dll to be present in the same directory as the python script.
terminal_start.pyw	-l create logs	Python script to start terminal windows for all RBUs in a test rig. This uses the configuration in terminal_configurations. If the script is called with the –l option then a new directory is created with a name based on the date and time. Log files are created in this directory for each RBU.
terminal_stop.pyw	none	Python script to stop all terminal windows in use. If logging is running then that is also stopped.
unit_configuration.py	None	Python script to configure all the RBUs in a test rig. This uses the configuration in terminal_configurations.

5.3 Teraterm

Teraterm is a terminal emulator program with its own scripting language. A number of scripts were created during the development and testing of Cygnus2.

Script Name	Description
AT_commands.ttl	Graphical User interface that allows the user to select and run AT commands to communicate with RBUs
Serial_reprogramming.ttl	Script to program a single RBU.

5.4 Debug GUI

The Debug GUI is an internal tool that connects to an RBU over the debug serial port and monitors the output. It is capable of issuing AT commands to the RBU and reading the responses. When connected to the NCU it can be used to advance the network through the configuration states and into active mode.

Some debug messages are output buy the RBU especially for the Debug GUI, allowing it to gather network information and build an image of how the mesh has formed.

These special outputs take the form:

+INF:{<event>,[value],[value]……}

The event is a code number for a specific occurrence within the RBU. The table below describes the event for each number.

The event tag is followed by comma-separated values that give details relating to the event.

e.g. if the event is "Added child node" then this will be followed by the node ID of the child. So if an RBU adds node 23 as a child, the debug output will be:

+INF:{4,23}

The following table maps event numbers to their associated event.

Event	Description
0	Added Primary Parent
1	Dropped Primary Parent
2	Added Secondary Parent
3	Dropped Secondary Parent
4	Added Child
5	Dropped Child
6	Promoted Secondary Parent
7	Promoted Primary Tracking Node
8	Promoted Secondary Tracking Node
9	Moved to new state

+AT COMMANDS

This section contains the AT commands supported by the RU. These are provided to the Portable Programmer using the wired connection.

It is implied that all AT commands are terminated by <CR><LF>.

Further detail for NCU AT commands can be found in 2008-SPC-0008 CIE-NCU Protocol Specification.

Command	Description	Flow	Examples and Parameters
+++	Enter AT mode	-	+++<CR><LF>
AT200	Start/stop the 200 hour (1h on/1h off) EN54 Duration of Operation test	w	AT200=ZxUx,<v>[, channel][,channel][,channel][,channel] Where: ZxUx = any address (only works locally). v = 0 to stop test or 1 to start test Channel = channel index of output to be tested Up to four channels can be specified: • channel=5 sounder • channel=6 beacon • channel=9 visual indicator • channel=10 SiteNet S+V combined
ATACKE	Acknowledge Alarm	w	ATACKE+ NCU only
ATACKF	Acknowledge Fire	w	ATACKF+ NCU only

Command	Description	Flow	Examples and Parameters
ATACS	Channel profile configuration	r/w	ATACS?<c> ATACS=<c>,<b> Response: ACS: c,b c = channel b = profile bitfield If bit is set then channel responds to that profile: • Bit 0 = fire profile • Bit 1 = first aid • Bit 2 = evacuate • Bit 3 = security • Bit 4 = general • Bit 5 = fault • Bit 6 = routing • Bit 7 = test • Bit 8 = silent test
ATADC	Set/get output channel local delays	r/w	ATADC?[ZzUu,]<c> ATADC=[ZzUu,]<c>,<delay1>,<delay2> Response: ADC: c,delay1,delay2 (local response) ADC: ZzUu,c,delay1,delay2 z = zone u = unit address c = channel delay1 = initial delay delay2 = investigative delay
ATAGD	Output Delays Overide. Instructs all devices to bypass output delays	w	ATAGD+<x> x = 0 (Enable output delays) / 1 (Bypass output delays)

Command	Description	Flow	Examples and Parameters
ATANA	Read analogue value	r	ATANA?<ZzUu>,<c> Response in MISC queue: ANA:<ZzUu>,<c>,0,1,<v> z = zone u = unit address c = channel v = value
ATANL	Add parent/child link to session manager. Only on RBU test builds with ENABLE_LINK_ADD_OR_REMOVE declared. See also ATDNL.	w	ATANL=<u>[,<is_parent>] u = unit address is_parent = 0 (or omitted): the node is added as a child 1: the node is added as the next available parent/Trk node
ATANN	Send PPU announcement message (PPU Disconnected mode only)	w	ATANN+
ATAOF	Set/get alarm output flags	r/w	ATAOF?[ZzUu,]<c>,CR><LF> ATAOF=[ZzUu,]<c>,<b> Response: AOF: c,b local response AOF:ZzUu, c,1,z,b remote response z = zone u = unit address c = channel b = bit field Bit 0 = inverted Bit 1 = ignore night delays Bit 2 = silenceable

Command	Description	Flow	Examples and Parameters
ATAOS	Broadcast Alarm Output Status	w	ATAOS+<s>,<u>,<d> Response: AOS: OK s = profile bit field for silenceable outputs u = profile bit field for unsilenceable outputs d = profile bit field for delay mask (1 in bit field means skip delays) Each bit field has one bit for each output profile which is set if the profile is active and reset if it is not. Bit 0 = fire profile Bit 1 = first aid Bit 2 = evacuate Bit 3 = security Bit 4 = general Bit 5 = fault Bit 6 = routing Bit 7 = test Bit 8 = silent test
ATATTX	Send an AT command to a unit in test mode (see ATSAT). Intended for production testing.	w	ATATTX+<destination>,<AT command> destination = unit address or serial number (with '-' seperators) AT command = command to be sent to unit Example to switch on sounder: ATATTX+1234-01-5678,ATOUT+Z1U4095,5,0,1,0
ATBATBC	Get/Set the backup low battery threshold in mV	r/w	ATBATBC? ATBATBC=<value> value = backup low battery threshold in mV for SiteNet.
ATBATBS	Get/Set the backup low battery threshold in mV	r/w	ATBATBS? ATBATBS=<value> value = backup low battery threshold in mV for SmartNet.

Command	Description	Flow	Examples and Parameters
ATBATP	Get/Set the battery test period	r/w	ATBATP? ATBATP=<x> x = battery test interval in days.
ATBATPC	Get/Set the primary low battery threshold in mV	r/w	ATBATPC? ATBATPC=<value> value = primary low battery threshold in mV for SiteNet.
ATBATPS	Get/Set the primary low battery threshold in mV	r/w	ATBATPS? ATBATPS=<value> value = primary low battery threshold in mV for SmartNet.
ATBATS	Request Battery Status	r	ATBATS? Send Battery voltages to debug log. ATBATS?ZxUx Remote request for battery status (from NCU). Response goes into the control panel queue as an MSI.
ATBFR	Get/Set beacon flash rate	r/w	ATBFR?<ZxUx>,<c>[,p] Response in MISC queue: BFR:ZzUu,<c>,<p>,<z>,<v> ATBFR=<ZxUx>,<c>[,p][,v] Response: BFR: OK z = zone u = unit address c = channel p = profile v = value
ATBID	Get/Set the branding ID of an RBU	r/w	ATBID=n Response: BID: OK ATBID? Response: BID: n n = decimal number
ATBIT	On-demand Built-In Test		ATBIT+

Command	Description	Flow	Examples and Parameters
ATBMODE	Battery test mode. Used in conjunction with ATRADC	r/w	ATBMODE? ATBMODE=<x> x = 0 Test mode off x = 1 Primary battery voltage test x = 2 Back-up battery voltage test x = 3 Primary battery current test
ATBOOT	Restart the unit in normal mode	w	ATBOOT+
ATBOOT1	Restart the unit in serial bootloader mode	w	ATBOOT1+
ATBPPU	Get / Set the if the bootloader uses the PPU or the Debug Serial Port	r/w	ATBPPU? ATBPPU=x x = 0 for Debug – 1 for the PPU
ATBTST	Initiate an immediate battery test	w	ATBTST+[ZxUx]
ATBV	Send the battery voltage readings to the debug port. The battery load is not applied. Only on RBU test builds with ENABLE_BATTERY_VOLTAGE_COMMAND declared.	r	ATBV?

Command	Description	Flow	Examples and Parameters
ATCHF	Get/Set channel flags	r/w	ATCHF=[ZzUu,]<c>,<b> ATCHF?[ZzUu,]<c> z = zone u = unit address c = channel b = bit field Bit 0 = inverted Bit 1 = ignore night delays Bit 2 = silenceable. Bit 3 = day enable Bit 4 = night enable
ATCHFW	Check firmware	r	ATCHFW+ Entered at the NCU only. Sends a message to all nodes containing the NCU firmware version. Nodes with a different version respond with firmware fault. Nodes with the same version do not respond.
ATCMD	Send Command to NCU/RBU Application	r/w	ATCMD?t,d,c,p1,p2 ATCMD=t,d,c,p1,p2,value t - Transaction ID d - Destination node ID c - Command type p1 - First command parameter p2 - Second command parameter value - The value to be written (Only for write operation) See Application Layer Parameters section of 2001-SPC-0012 Mesh Protocol Specification for command details.
ATCONE	Confirm alarm	w	ATCONE+ NCU only
ATCONF	Confirm fire	w	ATCONF+ NCU only

Command	Description	Flow	Examples and Parameters
ATDBC	Disable battery checks	r/w	ATDBC? ATDBC=x x = 0 enable battery checks x = 1 disable battery checks
ATDEC	Request remote device to report its device combination	r	ATDEC?<ZzUu> Response placed in MISC queue DEC:<ZzUu>,0,0,<z>,<v> z = device zone u = device unit address v = value
ATDEVCF	Get / Set the device configuration property (device combination)	r/w	ATDEVCF? ATDEVCF=x 0 ≤ x ≤ 41
ATDISC	Enable/disable a channel	w	ATDISC+[ZuUu,]<channel>,<enable_day>,<enable_night> z = device zone u = device unit address <channel> = the channel to enable/disable enable = 0 (disable channel) / 1 (enable channel)
ATDISD	Enable/disable a device	w	ATDISD+[ZzUu,]<enable> z = device zone u = device unit address enable = 0 (disable device) / 1 (enable device)
ATDISF	Disable fault reports	r/w	ATDISF? ATDISF=<x> x = 0 (fault reports enabled) / 1 (fault reports disabled)
ATDISRB	Disable an individual RBU. Write-only, NCU only.	w	ATDISRB=x x = The destination node ID

Command	Description	Flow	Examples and Parameters
ATDISZ	Enable/disable a zone	w	ATDISZ+<zone>,<enable> zone = the zone number enable = 0 (disable zone) / 1 (enable zone)
ATDLOUT	Start delayed output test function	w	ATDLOUT+[ZzUu,]<id>,<od>,<c> Response: DLOUT:OK z = zone u = unit address id = initial delay before output is activated (seconds) od = output duration (seconds) c = channel
ATDNI (NCU only)	Read neighbour information	r	ATDNI?<ZzUu> Response: DNI:ZzUu,pp,psnr,sp,ssnr z = zone u = unit address pp = primary parent ID psnr = primary parent SNR sp = secondary parent ID ssnr = secondary parent SNR
ATDNIR (NCU only)	Report neighbour information from cache	r	ATDNIR+[f,]<n> f = first node in report (default = 1) n = last node in report (max 511)
ATDNL	Remove a parent/child link from the session manager. Only on RBU test builds with ENABLE_LINK_ADD_OR_REMOVE declared. See also ATANL.	w	ATDNL=<u> u = unit address

Command	Description	Flow	Examples and Parameters
ATDPASD	Get / set battery depassivation settings	r/w	ATDPAS=[<ZzUu>,]<voltage>,<timeout>,<on time>,<off time> ATDPAS?[<ZzUu>] z = device zone u = device unit address voltage = target voltage in millivolts timeout = maximum depassivation time in seconds on time = on duty cycle in seconds off time = off duty cycle in seconds
ATDPOLL	Switch on/off automatic polling of digital inputs	r/w	ATDPOLL? ATDPOLL=<x> x = 0 enable automatic polling x = 1 disable automatic polling
ATDSF	Request fault status flags from remote device	r	ATDSF?<ZzUu> Response placed in MISC queue DSF:<ZzUu>,0,0,<z>,<v> z = device zone u = device unit address v = fault flag bits: 0x00001 Installation Tamper Fault 0x00002 Dismantle Tamper Fault 0x00004 Low Battery 0x00008 Detector Fault 0x00010 Beacon Fault 0x00020 Sounder Fault 0x00040 IO Input 1 Fault 0x00080 IO Input 2 Fault 0x00100 Dismantle Head Fault 0x00200 Mismatch Head Fault 0x00400 Battery Error 0x00800 Device Id Mismatch 0x01000 Dirty Sensor 0x02000 Internal Fault 0x04000 Input Short Circuit Fault 0x08000 Input Open Circuit Fault 0x10000 Output Fault 0x20000 BIT Fault

Command	Description	Flow	Examples and Parameters
ATDVD	Dump visible devices. See also ATSCAN.	r	ATDVD+[f,][n] f = first node to list (default = 1) n = last node to list (default = 511)
ATDWRAP	Set the number of short frames for DULCH wrap around	r/w	ATDWRAP? ATDWRAP=x x = the number of short frames to wrap around. Note that this reduces the max permissible node ID to x/2 - 1
ATEDP	Enable a channel on a plug-in	r/w	ATEDP?<ZzUu>,<c> Response placed in MISC queue EDP:<ZzUu>,<c>,0,<z>,<v> ATEDP=ZzUu>,<c>,<v> Response: EDP: OK z = zone u = unit address c = channel v = value
ATEDR	Send disable message to remote device. Puts device into sleep mode until power is reset.	w	ATEDR=<ZzUu> Response: EDR: OK z = zone of target device u = unit address of target device
ATEEI	Read/write to one of the ten reserved NVM values	r/w	ATEEI?[ZxUx,]<index> ATEEI=[ZxUx,]<index>,<value> ZxUx is an optional address to remotely read/write from the NCU. If omitted the command applies to the local device. <index> : range 0-9 addresses the parameter to read/write. <value> : The value to write. Responses are directed to the debug output where the command is entered.

Command	Description	Flow	Examples and Parameters
ATEVAC	Activate the evacuate outputs	w	ATEVAC+ Sets the outputs to activate in accordance with the ‘evacuate’ profile.
ATFA Discontinued in version 01.16.192	get / set the Firmware Active image and update the NVM data of the new Image (for write operation)	r/w	ATFA? ATFA=x x = 1 or 2
ATFAN Discontinued in version 01.16.192	get / set the Firmware Active image and do Not update the NVM data of the new Image (for write operation)	r/w	ATFAN? ATFAN=x x = 1 or 2
ATFBAND	Set the frequency band for the radio	r/w	ATFBAND? ATFBAND=<x> x = 0 (865MHz) / 1 (915MHz)
ATFCM	Enable or disable the 'faults clear' message that is sent on joining if no faults are present	r/w	ATFCM=[<ZzUu>,]<0|1> 0=disabled, 1=enabled ATFCM?[<ZzUu>]
ATFI	get the Firmware Information	r	ATFI? Response: FI nn.nn.nn,dd/mm/yy,0xhhhhhhhh nn = decimal number dd = day mm = month yy = year h = hexadecimal number (checksum)

Command	Description	Flow	Examples and Parameters
ATFIP	Get the firmware information from a plug-in head		ATFIP?<ZzUu> Response in MISC queue: FIP:ZzUu,0,0,2,nn.nn.nn,dd/mm/yy z = zone u = unit address n = decimal digit dd = day mm = month yy = year
ATFIR	Read the firmware version of a remote device		ATFIR?<ZzUu> Response in MISC queue: FIR:ZzUu,0,0,1,nn.nn.nn z = zone u = unit address n = decimal digit
ATFLEN	Get / Set number of short frames per long frame.		ATFLEN? ATFLEN=x x = short frames per long frame (valid range 16 to 128 in multiples of 16)
ATFREQ	Get / Set the initial frequency channel (determines hopping sequence).	r/w	ATFREQ? ATFREQ=x $0 \leq x \leq 9$
ATGADC	Sets the global delays	r/w	ATGADC? ATGADC=<output delay>,<investigate delay> Delay values in seconds.
ATGDLY	Set global delay override.	r/w	ATGDLY? ATGDLY=x X = 0 : Global override off. X = 1 : Global override on.

Command	Description	Flow	Examples and Parameters
ATGSET	Compound statement to set global delay and day/night settings	w	ATGSET+<delays_enable>,<global_override>,<global delay1>,<global_delay2>,<is_day>,<ignore_security_in_day>, <ignore security at night> Where delay1 and delay2 have a granularity of 30 seconds i.e. a value of 2 = 60s.
ATILP	Set initial IDLE mode listening period before Phase 2 sleep. See also ATISP2, ATISP3 & ATPH2.	r/w	ATILP? ATILP=<x> x = the initial listening period in hours (default 72).
ATIOUP	Get / Set polling interval on I/O unit in milliseconds		ATIOUP? ATIOUP=p
ATIOUTH	Get / set input thresholds on I/O unit (raw ADC value, 0 … 4095)		ATIOUTH? ATIOUTH=<th1>,<th2>,<th3> Response: IOUTH: th1,th2,th3 th1 = short cct threshold th2 = logic threshold th3 = open cct threshold
ATISP2	Set the sleep duration for start-up synch phase 2. See also ATILP, ATISP3 & ATPH2.	r/w	ATISP2? ATISP2=<x> x = the sleep period in hours (default 3).
ATISP3	Set the sleep duration for start-up synch phase 3. See also ATILP, ATISP2 & ATPH2.	r/w	ATISP3? ATISP3=<x> x = the sleep period in hours (default 6).

Command	Description	Flow	Examples and Parameters
ATJRSSI	Set the RSSI joining threshold for nodes selecting parents	r/w	ATJRSSI? ATJRSSI=<x> **x** **Threshold** 0 -107dBm / -112dBm (default) 1 -112dBm (non-EN54) 2 -117dBm (non-EN54) 3 -122dBm (non-EN54) 4 -102dBm 5 -97dBm 6 -92dBm 7 -87dBm 8 -82dBm 9 -77dBm 10 -72dBm
ATJSNR	Set the SNR joining threshold for nodes selecting parents	r/w	ATJSNR? ATJSNR=<x> x = decimal SNR value (default 5)
ATLDLY	Set RBU output to use local or global delay values	r/w	ATLDLY?[ZxUx,]<channel> ATLDLY=[ZxUx,]<channel>,<x> ZxUx : optional address for remote operation from the NCU. channel : The output channel to configure. x = 0 (use global delays) / 1 (use local delays)
ATLED (NCU only)	Set/read the state of the plug-in head status indicator LED	r/w	ATLED?<ZzUu> ATLED=<ZzUu>,<v> Response: LED:ZzUu,v z = zone u = unit address v = value. 0=OFF, 1=ON

Command	Description	Flow	Examples and Parameters
ATLEDP	Add/remove LED pattern	r/w	ATLEDP?<p> ATLEDP=<p>,<x> x = 0 (remove) / 1 (add) **p** **Pattern** 0 Constant Green 1 Constant Blue 2 Constant Red 3 Constant White 4 Constant Magenta 5 Constant Cyan 6 Constant Amber 7 Low Battery OR Missing Battery 8 Built In Test Fail 9 Mesh Sleep 1 10 Mesh Sleep 2 11 Mesh Idle 12 PPU Mode Active 13 Mesh Connected 14 Mesh State Config Sync 15 Mesh State Config Form 16 Mesh State Ready For Active 17 Mesh State Active 18 First Aid Indication 19 Fire Indication
ATLOZ	Set the enable bits for zones 1 to 48. See also ATUPZ.	r/w	ATLOZ? ATLOZ=<zone bits> Where zone bits is a single decimal number
ATMAXCH	Set the maximum number of children that a device will support	r/w	ATMAXCH? ATMAXCH=x Where x is the max number of children.
ATMAXDL	Set the number of repeats for downlink messages	r/w	ATMAXDL? ATMAXDL=<x> Where x is the max number of message repeats.
ATMAXHP	Set max number of hops for a message	r/w	ATMAXHP? ATMAXHP=x Where x is the max number of hops.

Command	Description	Flow	Examples and Parameters
ATMFS	Set/get the mesh state of a device	r/w	ATMFS=<s> Response: BID: OK ATMFS? Response: BID: s s = mesh state 0 = IDLE 1 = SYNCH 2 = FORM 3 = ACTIVE
ATMODE	Get / Set the test mode for the unit.	r/w	ATMODE? ATMODE=x x = 0 : Test Mode OFF x = 1 : Test Mode RECEIVE x = 2 : Test Mode TRANSPARENT x = 3 : Test Mode TRANSMIT x = 4 : Test Mode MONITORING x = 5 : Test mode NETWORK MONITOR x = 6 : Test Mode SLEEP

Command	Description	Flow	Examples and Parameters
ATMSI	Unsolicited status report for remote device	-	Reported in MISC queue MSI: ZzUu,<pv>,<bv>,9,<pp>,<sp>,<prssi>,<srssi>,<nc>,<e>,<ed>,<r>,<dc>,<f> z = zone u = unit address pv = primary battery voltage (mV) bv = backup battery voltage (mV) pp = primary parent unit address (-1 if none) sp = secondary parent unit address (-1 if none) prssi = RSSI of primary parent srssi = RSSI of secondary parent e = event number (reason for generating the status report) ed = event data (supporting information for event e.g. node ID) r = rank dc = device combination f = 'any faults' flag (0 if no faults, 1 if any fault, 2 if any warning, 3 if fault and warning)
ATMSN	Read message queue status	r	ATMSN?<CR>,<LF> Response is as follows. MSN: <fire queue msg count>,<fire queue lost msgs><LF> <alarm queue msg count>,<alarm queue lost msgs><LF> <fault queue msg count>,<fault queue lost msgs><LF> <misc queue msg count>,<miscqueue lost msgs><LF> <downlink queue msg count>

Command	Description	Flow	Examples and Parameters
ATMSR	Request mesh status report for remote device	r	ATMSR?<ZzUu> Read from NCU cache. ATMSR?<ZzUu>,1 Send request to remote device. Response in MISC queue MSR: ZzUu,<pv>,<bv>,10,<pp>,<sp>,<prssi>,<srssi>,<nc>,<e>,<ed>, <r>,<dc>,<f> z = zone u = unit address pv = primary battery voltage (mV) bv = backup battery voltage (mV) pp = primary parent unit address (-1 if none) sp = secondary parent unit address (-1 if none) prssi = RSSI of primary parent srssi = RSSI of secondary parent e = event number (reason for generating the status report) ed = event data (supporting information for event e.g. node ID) r = rank dc = device combination f = ‘any faults’ flag (0 if no faults, 1 if any fault, 2 if any warning, 3 if fault and warning)
ATNOD	Get the number of devices and zones that have been seen by the NCU.	r	ATNOD? Response: NOD: <zone count>,<device count>

Command	Description	Flow	Examples and Parameters
ATNODES	report the associated NODES on the Debug port		ATNODES+ Response: NODES: OK +DAT:E,R P,s S,s T1,s T2,s N1,s N2,s….. Where E=Event type R=Rank P=Primary Parent Node ID S=Secondary Parent Node ID T1=Primary Tracking Node ID T2=Secondary Tracking Node ID N=Child node ID s=SNR A value of -1 means ‘not present.’
ATNVM	Read/write directly to NVM config parameter	r/w	ATNVM?<index> ATNVM=<index>,<value> Index : range is 0 to 108, see DM_NVMParamId_t in DM_NVM_cfg.h. Value : the value to be written
ATOPC	Report output channel configuration over the debug output	r	ATOPC?

Command	Description	Flow	Examples and Parameters
ATOUT	Instruct the NCU to generate an output Signal (NCU only)	w	ATOUT+d,p,x,d u = the destination node unit address (4095 = all nodes) p=output profile 0=Fire 1=First Aid 2=Evacuate 3=security 4=General 5=Fault 6=Routing 7=Test 8=Silent Test x = outputs activated (0 or 1) d = duration (not implemented)
ATPF	Development aid. Instigate NCU power saving mode (Dynamic Tx power)	r/w	ATPF=0|1 0=normal, 1=power saving ATPF?
ATPFM	Override automatic parent selection in pre-formed mesh.		ATPFM? ATPFM=<primary parent>,<secondary parent> Set secondary parent to -1 if not wanted. ATPFM=0,0 will disable preformed mesh
ATPH2	Set the Phase 2 listening period for start-up synch. See also ATILP, ATISP2 & ATISP3.	r/w	ATPH2? ATPH2=<x> x = phase 2 listening period in hours (default 14).

Command	Description	Flow	Examples and Parameters
ATPIN (Debug tool)	Read/write the value of a GPIO pin on the MCU	r/w	ATPIN?p Response: PIN: v ATPIN=p,v Response: PIN: OK p = pin number, format is PB1, PC13 etc. v = value, always 0 or 1
ATPING	Send a ping to another device	w	ATPING=<u> u = unit address.
ATPPBST	Broadcast a message to request the PPU mode (PPU-RBU only)	w	ATPPBST+
ATPIRCN	Set/get the PIR strike count	r/w	ATPIRCN=<x> ATPIRCN? x = strike count (default 5)
ATPIROF	Set/get the PIR off debounce period	r/w	ATPIROF=<x> ATPIROF? x = off debounce period in µs (default 30000, max 1000000)
ATPIRON	Set/get the PIR on debounce period	r/w	ATPIRON=<x> ATPIRON? x = on debounce period in µs (default 30000, max 1000000)
ATPPEN	Enable/Disable the Peer to Peer mode (RBU only) No longer used.	r/w	ATPPEN? ATPPEN=n N=0 or 1
ATPPU	Send a message to put a device in PPU mode	w	ATPPU+<serial number> ATPPU+[<node_id>][,<system_id>] ATPPU+<destination>
ATPPUMD	Send a message to put a device into PPU Disconnected mode	w	ATPPUMD+[<node_id>]

Command	Description	Flow	Examples and Parameters
ATPRF	Get / Set profile configuration		ATPRF?<p > ATPRF=<p>,<t>,<f>,<x> Response: PRF: p,t,f,x p = profile t = tone f = flash rate x = SiteNet combined sounder pattern
ATPTYPE	Read the plug-in head type and class	r	ATPTYPE?[ZzUu] z = zone of target device u = unit address of target device
ATQAM	Read the next message from the alarm queue (NCU)	r	ATQAM? Response: QAM:< ZzUu>,<c>,<a>,<v> z = zone of source device u = unit address of source device c = channel number a = activated (1 for alarm condition, 0 for no alarm) v = senor reading
ATQFE	Read the next message from the fire queue (NCU)	r	ATQFE? Response: QFE:< ZzUu>,<c>,<a>,<v> z = zone of source device u = unit address of source device c = channel number a = activated (1 for alarm condition, 0 for no alarm) v = senor reading

Command	Description	Flow	Examples and Parameters
ATQFT	Read the next message from the fault queue (NCU)	r	ATQFT? Response: QFT:< ZzUu>,<c>,<t>,<v>,<f> z = zone of source device u = unit address of source device c = channel number t = fault type v = value (1 for fault condition, 0 for no fault) f = 'any fault' flag for the source device (1 or 0)
ATQMC	Read the next message from the misc queue (NCU)	r	ATQMC? Response varies with report type.
ATR	Reboot the unit	w	ATR+[ZzUu] z = zone u = unit address
ATRADC	Read battery values. Works in conjunction with ATBMODE which sets the battery test.	r	ATRADC?
ATRNK	Set the maximum rank at which a device should join the mesh	r/w	ATRNK?[ZzUu] ATRNK=[ZzUu,]<r> z = zone of device u = unit address of device r = maximum rank

Command	Description	Flow	Examples and Parameters
ATROFF	Enables and offset to be added to the RSSI of node IDs 0 to 9. Primarily, this is to make the target node seem farther away than it is for parent selection tests. Only on RBU test builds with APPLY_RSSI_OFFSET declared.	r/w	ATROFF=<node id>,<rssi offset> ATROFF?<node id> Valid range for rssi offset is -128 to +127. The offset is applied as: rx'd rssi + <rssi offset>
ATRP	Restart the unit in serial bootloader mode (same as ATBOOT1).	w	ATRP+
ATRRU	Reboot a remote device	w	ATRRU+<ZzUu> z = zone of device u = unit address of device
ATRST	System alarm reset	w	ATRST+ All devices reset their alarms and send new fire signals if inputs are still active. Existing faults are re-sent.
ATRXO	Shift the timing between RACH Tx and Rx. Only on RBU test builds with ENABLE_TDM_CALIBRATION declared. See also ATTIM.	w	ATRXO=<offset> Note: This sets `rach_tx_rx_offset` but this variable appears not to be used.

Command	Description	Flow	Examples and Parameters
ATSAT	Send a message to put a device in OTA AT mode. See also ATATTX.	w	ATSAT+<serial number> ATSAT+[<node_id>][,<system_id>]
ATSCAN	Scan for visible devices for one complete long frame.	r	ATSCAN+
ATSDN	Read/write day or night mode	r/w	ATSDN? Response: SDN: v ATSDN=v Response: SDN: OK v = 0 (day) / 1 (night)
ATSENV	Read Sensor Analogue Values from all devices	w	ATSENV+ Response: SENV: OK Sensor values are returned in the Battery Status message over the delayed channel. MSC:<ZzUu>,<pv>,<bv>,<dc>,<z>,<s>,<h>,<p> z = zone u = unit address pv = primary battery voltage (mV) bv = backup battery voltage (mV) dc = device combination s = smoke sensor value h = heat sensor value p = pir sensor value
ATSERNO	Get / Set the unit serial number. See also ATSERPX.	r/w	ATSERNO? ATSERNO=xxxx-xx-xxxx

Command	Description	Flow	Examples and Parameters
ATSERPX	Get / Set the unit serial product code. See also ATSERNO.	r/w	ATSERPX?[ZxUx] ATSERPX=[ZxUx,]<product code> product code = three-letter serial code, e.g. XAD
ATSHB	Test aid. Instructs the device to not transmit a number of heartbeats. Only on RBU test builds with ENABLE_HEARTBEAT_TX_SKIP declared.	r/w	ATSHB=<number to skip> ATSHB? Defaults to 1 if no value is given.
ATSL	Set the sound level of an output	r/w	ATSL=[ZzUu,]<channel>,<level> ATSL?[ZzUu,]<channel> z = zone u = unit address channel = the output channel level = the sound level to set (SmartNet 0/1/2, SiteNet 3)
ATSNDF	Development aid. Send a fault signal with the specified parameters	w	ATSNDF=<c>,<f>,<v>[,<d>] c = channel f = fault type v = value (‘1’ = fault active, ‘0’ = fault cleared) d = delay sending: 1 (delay on DULCH) / 0 (send immediately default)
ATSNP	Read the serial number of a plug-in	r	ATSNP?<ZzUu> Response in MISC queue: SNP:ZzUu,255,0,<z>,<xxx-xx-xxxxx> z = zone u = unit address x = decimal digit

Command	Description	Flow	Examples and Parameters
ATSNR	Read device serial number	r	ATSNR?<ZzUu> Response in MISC queue: SNR:ZzUu,255,0,<z>,<xxx-xx-xxxxx> z = zone u = unit address x = decimal digit
ATSNRAV	Set the SNR averaging strategy for visible devices	r/w	ATSNRAV? ATSNRAV=<x> x = 0 use normal averaging x = 1 reset average if older than two long frames x = 2 no averaging. Use latest received SNR.
ATSQC	Development aid. Report over debug usart how many uplink messages are queued for S-RACH and DULCH	r	ATSQC?
ATSTATE	Update the state of an RBU	w	ATSTATE=x x=0 : Set state to IDLE x=1 : Set state to CONFIG x=2 : Set state to ACTIVE x=3 : Set state to TEST MODE
ATSTM	Test aid. Enables dummy status messages to be passed to the NCU cache. Only on RBU test builds with ENABLE_NCU_CACHE_DEBUG declared.	w	ATSTM=<node id>,<prim parent>,<sec parent>,<rank>,<event>,<data>

Command	Description	Flow	Examples and Parameters
ATSTS	Read/write the sounder tone for a profile	r/w	ATSTS?<ZzUu>,<channel>,<profile> Response placed in MISC queue STS:<ZzUu>,<c>,<p>,<z>,<v> ATSTS=< ZzUu>,<channel>,<profile>,<value> Response: STS: OK z = zone u = unit address c = channel p = profile v = value
ATSVI	Read or write to the SVI board registers	r/w	ATSVI=p,x ATSVI?p p = The command parameter to be accessed. (0→22, zero-based index into register table in 2002-SPC-0003 Sounder VI Software Architecture and Design) x = The value to be written. Examples: ATSVI=4,n Sets tone pattern to n+1 (see page 2 of 2000-DBA-0011-04 Sounder Tones List.pdf) ATSVI=1,1 Turns sounder on ATSVI=1,0 Turns sounder off
ATSYNC	Get / Set the synchronisation flag.	r/w	ATSYNC? ATSYNC=x x = 0 (slave) / 1 (master = NCU)
ATSYSID	Get / Set the system ID for the unit	r/w	ATSYSID? ATSYSID=x x <= 65535

Command	Description	Flow	Examples and Parameters
ATTHL	Read/write lower threshold of detector	r/w	ATTHL?<ZzUu>,<c> Response placed in MISC queue THU:<ZzUu>,<c>,0,<z>,<v> ATTHL=<ZzUu>,<c>,<dv>,<nv> Response: THU: OK z = zone u = unit address c = channel dv = day value nv = night value
ATTHU	Read/write upper threshold of detector	r/w	ATTHU?<ZzUu>,<c> Response placed in MISC queue THU:<ZzUu>,<c>,0,<z>,<v> ATTHU=<ZzUu>,<c>,<dv>,<nv> Response: THU: OK z = zone u = unit address c = channel dv = day value nv = night value
ATTIM	Set or read timing parameters. Only on RBU test builds with ENABLE_TDM_CALIBRATION declared. See also ATRXO.	r/w	ATTIM?<index> ATTIM=<index>,<value> **index** **timing parameter** 0 gDchRxDoneLatency 1 gDchTxOffset 2 gDchRxOffset 3 gRachTxOffsetDownlink 4 gRachTxOffsetUplink 5 gRachCadOffset 6 gDlcchTxOffset 7 gDlcchCadOffset 8 gAckTxOffsetUplink 9 gAckCadOffset 10 gDchOffset 11 gSyncCorrection 12 gFrameSyncOffset

Command	Description	Flow	Examples and Parameters
ATTMP	Set/get the test mode of a plug-in (head)	r/w	ATTMP?<ZzUu>,<c> Response placed in MISC queue TMP:<ZzUu>,<c>,0,<z>,<v> ATTMP=<ZzUu>,<c>,<v> Response: TMP: OK z = zone u = unit address c = channel v = value
ATTMR	Put device into test mode	w	ATTMR=<ZzUu>,<m> Response: TMR: OK z = zone u = unit address m = test mode 0 = test mode OFF 1 = test mode RECEIVE 2 = test mode TRANSPARENT 3 = test mode TRANSMIT 4 = test mode MONITORING 5 = test mode SLEEP 6 = test mode NETWORK MONITOR
ATTOS	Initiate One-Shot Test	w	ATTOS+<s> s = 0 silent test s = 1 standard test
ATTST	Broadcast a Test Signal over the Lora Radio.	w	ATTST=<string> Where <string> is a max of 13 characters in length.
ATTXPHI	Write the high transmission power value to NVM.	r/w	ATTXPHI? ATTXPHI=<x> Where x=0→10

Command	Description	Flow	Examples and Parameters
ATTXPLO	Write the low transmission power value to NVM.	r/w	ATTXPLO? ATTXPLO=x Where x=0→10
ATTXRX (Debug tool)	Set/get the logic for the TxRx switch	r/w	ATTXRX? ATTXRX=<tx> Response: TXRX: tx tx=0 or 1, logic used for the Tx control line. Rx set to opposite logic. Default on start-up is tx=1
ATUA	Get / Set the unit address	r/w	ATUA? ATUA=u 0≤u≤511
ATUID	Read the MCU ID	r	ATUID?
ATUPZ	Set the enable bits for zones 49 to 96 See also ATLOZ.	r/w	ATUPZ? ATUPZ=<zone bits> Where zone bits is a single decimal number
ATXAM	Discard message from alarm queue	w	ATXAM+
ATXFE	Discard message from fire queue	w	ATXFE+
ATXFT	Discard message from fault queue	w	ATXFT+
ATXMC	Discard message from misc queue	w	ATXMC+
ATXMQ	Clear all message queues	w	ATXMQ+

Command	Description	Flow	Examples and Parameters
ATZONE	Read/write the programmed zone number of a device.	r/w	ATZONE? ATZONE=n
ATSET	Sets the number of LED boards, Zones and Devices for a CIE Panel	w	ATSET=96,96,511

Table 23 AT Commands

6 RADIO BOARD MODULE FILE MAPPING

6.1 Application Module

CFG_Device_cfg.c
CFG_Device_cfg.h
DM_ADC.c
DM_ADC.h
DM_BatteryMonitor.c
DM_BatteryMonitor.h
DM_Device.c
DM_Device.h
DM_FA_NonLatchingButton.c
DM_FA_NonLatchingButton.h
DM_i2c.c
DM_i2c.h
DM_IWDG.c
DM_IWDG.h
DM_LED.c
DM_LED.h
DM_LED_cfg.c
DM_LED_cfg.h
DM_MCUPower.c
DM_MCUPower.h
DM_NVM.c
DM_NVM.h
DM_NVM_cfg.c
DM_NVM_cfg.h
DM_OutputManagement.c
DM_OutputManagement.h
DM_RadioBaseBoard.h
DM_RelayControl.c
DM_RelayControl.h
DM_svi.c
DM_svi.h
DM_SystemClock.c
DM_SystemClock.h
lptim.c
lptim.h
MM_ApplicationCommon.c
MM_ApplicationCommon.h
MM_ATCommand.c
MM_ATCommand.h
MM_ATHandleTask.c
MM_ATHandleTask.h
MM_CIEQueueManager.c
MM_CIEQueueManager.h
MM_CommandProcessor.c
MM_CommandProcessor.h
MM_IdleDemon.c
MM_Main.c
MM_Main.h
MM_MainPPU.c

MM_MeshAPI.c
MM_MeshAPI.h
MM_NCUApplicationStub.c
MM_NCUApplicationStub.h
MM_NCUApplicationTask.c
MM_NCUApplicationTask.h
MM_NeighbourInfo.c
MM_NeighbourInfo.h
MM_PPUApplicationTask.c
MM_PPUApplicationTask.h
MM_RBUApplicationTask.c
MM_RBUApplicationTask.h
MM_RBUAppPPMode.c
MM_SviTask.h
spi.c
spi.h
system_stm32l4xx.c

6.2 AT Handler Module

MM_ATHandle_cfg.c
MM_ATHandle_cfg.h
MM_ATHANDLE_PPUcfg.c

6.3 GPIO Module

board.c
board.h
DM_InputMonitor.c
DM_InputMonitor.h
gpio-board.c
gpio-board.h
gpio.c
gpio.h
gpio_config.c
gpio_config.h
MM_GpioTask.c
MM_GpioTask.h
MM_Interrupts.c
MM_Interrupts.h
pinName-board.h
spi-board.c
spi-board.h
stm32l4xx_it.c
stm32l4xx_it.h

6.4 Head Interface Module

MM_PluginInterfaceTask.c
MM_PluginInterfaceTask.h

MM_PluginQueueManager.c
MM_PluginQueueManager.h

6.5 MAC Module

DM_CRC.c
DM_CRC.h
MC_ChanHopSeqGen.c
MC_ChanHopSeqGenPrivate.h
MC_ChanHopSeqGenPublic.h
MC_MacConfiguration.c
MC_MacConfiguration.h
MC_TDM.c
MC_TDM.h
MC_TDM_SlotManagement.c
MC_TDM_SlotManagement.h
MM_MACTask.c
MM_MACTask.h
radio.h
sx1272-board.c
sx1272-board.h

6.6 Mesh Module

MC_AckManagement.c
MC_AckManagement.h
MC_Encryption.c
MC_Encryption.h
MC_MAC.c
MC_MAC.h
MC_MacQueues.c
MC_MacQueues.h
MC_MeshFormAndHeal.c
MC_MeshFormAndHeal.h
MC_PingRecovery.c
MC_PingRecovery.h
MC_PUP.c
MC_PUP.h
MC_SACH_Management.c
MC_SACH_Management.h
MC_SessionManagement.c
MC_SessionManagement.h
MC_StateManagement.c
MC_StateManagement.h
MC_SyncAlgorithm.c
MC_SyncAlgorithm.h
MC_SyncPrivate.h
MC_SyncPublic.h
MC_TestMode.c
MC_TestMode.h
MM_MeshTask.c
MM_MeshTask.h

SM_StateMachine.c
SM_StateMachine.h

6.7 Serial Interface Module

DM_Log.c
DM_Log.h
DM_SerialPort.c
DM_SerialPort.h
DM_SerialPort_cfg.c
DM_SerialPort_cfg.h
MM_ConfigSerialTask.c
MM_ConfigSerialTask.h
MM_ConfigSerial_cfg.c
MM_ConfigSerial_cfg.h
serial.h
uart-board.h
uart.c
uart.h

6.8 Timed Event Module

MM_TimedEventTask.c
MM_TimedEventTask.h