Developers Guide

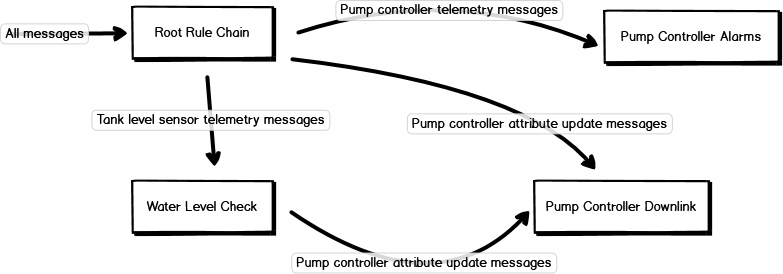
# Rules Chains

Most of the dashboard logic is expressed in ThingsBoard rules chains. All telemetry from devices and other events such as device attribute changes cause messages to flow through rules chains where they can be examined and acted upon.

See this page for an introduction to working with rules chains: <https://thingsboard.io/docs/user-guide/rule-engine-2-0/re-getting-started/>

The pump controller dashboard uses four rules chains. The Root Rule Chain is a rule chain defined initially by ThingsBoard to perform basic operations such as recording telemetry data to the database. Every message or other event that can be processed by a rule chain goes through the Root Rule Chain. The other three rule chains have been developed to avoid the root rule chain becoming too complicated and handle specific areas of functionality – automatic control of the pump based upon the tank water level, sending the pump control messages, and processing pump controller events and alarms.

The diagram below shows the connections between the rules chains, and the types of messages that are passed into them.



## Complications

There are a number of difficulties in working with Thingsboard rules chains when the logic is anything other than trivial. This section describes some of the odd techniques required.

### Only script nodes can change messages

An example of when this is required is setting an attribute value on a device based upon telemetry values in the current message – as when setting the estimate fill time attribute based upon the current tank water level.

The message being processed is a POST\_TELEMETRY\_REQUEST message with the current tank level in it. The message required by the save attributes node is a POST\_ATTRIBUTES\_REQUEST message with the fillTimer attribute value in it.

The message type and content can only be changed by a script node so you generally need a script node before some type of action node.

This is also done when switching the pump on and off. The trigger for that command is either tank level messages or an attribute update, but the message needs to be an RPC request in the end.

### The message originator is important

You cannot just transform a message with a script node when working with RPC requests. The “originator” of the message – the device it was supposedly sent from – must be the device the request is destined for. There is a change originator node to make this change.

An example of when this is required is when a telemetry message from the tank level sensor triggers a pump command, so the message originator must be changed to the pump controller before it is processed by the send RPC command node.

This is also required if a message from one device causes telemetry or attribute changes to another. The telemetry or attribute update will be recorded against the message originator.

### RPC requests only work for devices subscribing to the ThingsBoard MQTT broker

RPC requests work for the WiFi version of the pump controller Feather because it connects to the ThingsBoard MQTT broker and subscribes to the device RPC topic. They do not work for the LoRaWAN version.

To send a downlink to a LoRaWAN Feather a message must be sent to an MQTT topic on the *ThingsNetwork* broker, because ThingsNetwork cannot be instructed to subscribe to the ThingsBoard broker. The professional edition of ThingsBoard has a ThingsNetwork gateway but we are not using the PE edition of ThingsBoard.

The result of this is that you cannot simply use an RPC widget on the dashboard – if you do, the LoRaWAN device will always be marked as offline, even though you did get a message to the device via rules chain trickery.

So, the process of sending a command went from using an RPC switch widget to a very complicated set of message transformations ending with either an MQTT message downlink for LoRaWAN devices, or an RPC call for WiFi devices.

### Other tricks

* After editing a rule node in a rule chain you *must* click on the tick button at the lower right hand corner of the screen for the change to take effect.
* To help debug a rule node you can enable “Debug mode” in the node’s detail tab while editing the rule node. After this is done the messages flowing through the node can be viewed in the node’s event tab. You must also follow the advice of the note above for this change to take effect.
* ThingsNetwork uses base64 encoding for sending binary message payloads and ThingsBoard do not provide base64 conversion functions so we had to add these in where necessary.

### Device Attributes

The dashboard makes extensive use of device attributes to store the state of several variables. The table below describes the function of each variable. See the User Manual for set up of attributes.

### Pump Controller

|  |  |  |
| --- | --- | --- |
| **Type** | **Key** | **Value/Description** |
| Shared Attributes | | |
| String | Protocol | wifi/lora depending on device |
| Server Attributes | | |
| Bool | autoControl | True: auto control on, false: auto control off |
| Double | lowLevel | Value of low threshold of tank for auto control to turn pump on |
| Double | highLevel | Value of high threshold of tank for auto control to turn pump off |
| Int | timerType | Determines timer type: 0: No Timer, 1: Est. Fill Timer, 2 Custom Timer |
| Int | customPumpTimer | Custom pump run time in minutes |
| Bool | requestedPumpState | The on/off value of the state we want the pump to be in: True: pump running, false: pump off |
| Int | transmitStatus | Status of last request sent: 0: Failure, 1: In-Transit, 2: Success |
| Int | requestTime | Timestamp of the last message sent to the pump. |

### Tank Sensor

|  |  |  |
| --- | --- | --- |
| **Type** | **Key** | **Value/Description** |
| Server Attributes | | |
| Int | fillTimer | The estimated time to fill tank in minutes |
| Double | tankfillRate | Fill rate of the tank in Meters per Minute: 0.0042 for 1m per 4 hrs |

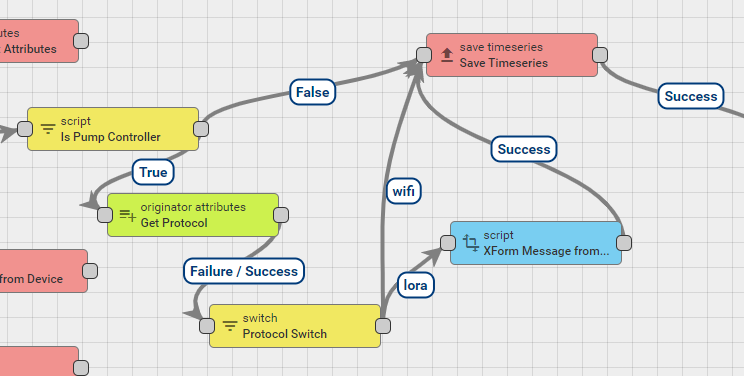
### Root Rule Chain

In an attempt to cap the complexity of the Root Rule Chain and organise functional logic groups, the Root Rule Chain has been customised for the bore pump controller dashboard so messages and events of interest are sent into more specialised rules chains.

* Pump controller device activity/inactivity and telemetry messages are forwarded to the Pump Controller Alarms rule chain to check for alarm conditions.
* If the requestedPumpState attribute has been modified in response to the pump on/off switch being changed manually, then the message is forwarded to the Pump Controller Downlink rule chain.
* Telemetry messages from the tank level sensor are forwarded to the Water Level Check rule chain to see if automatic control of the pump is required.

The Root Rule Chain is also responsible for updating the pump controller transmit status server attribute.

### Telemetry Source Determination



The source of telemetry messages is sorted into three categories within the root rule chain; Lora, Wifi and everything else. Telemetry messages are first checked in the Is Pump Controller node to determine if the metadata.devicetype is Pump Controller. Messages determined to be from a pump controller are then checked for their protocol Shared Attribute and split into either Lora or Wifi messages.

Lora messages receive additional treatment to decode the byte flags into Thingsboard readable JSON using a custom implementation of the [atob](https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/atob) function which is unavailable in this version of Thingsboard.

Messages then have their timeseries data saved.

### Transmit Status and Requested Pump State

transmitStatus and requestedPumpState are server Attributes of pump controller devices and control the Transmit status indicator and the manual control switch on the dashboard respectively.

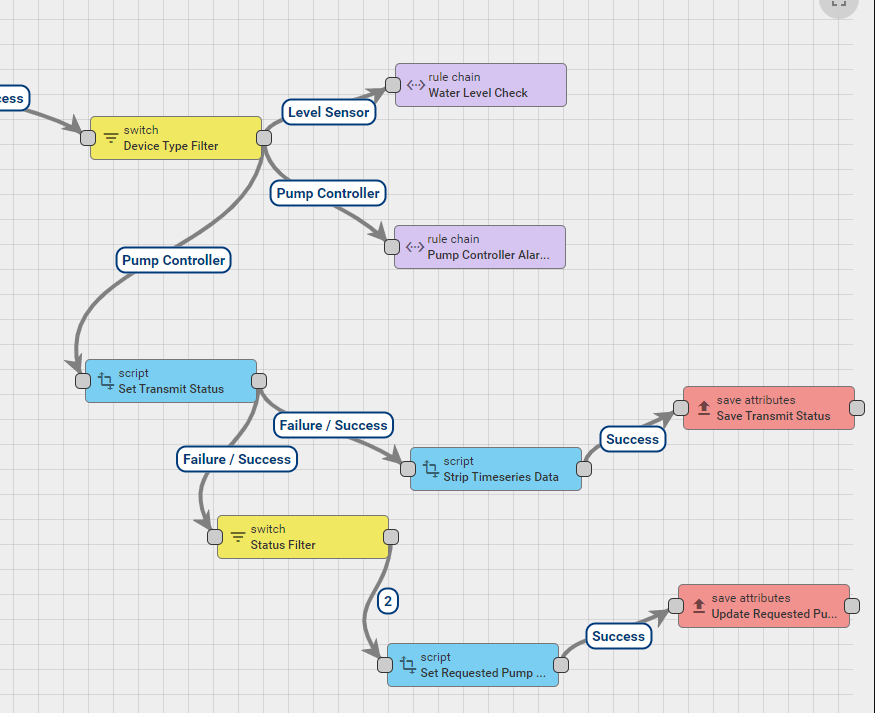
transmitStatus is an integer variable and is used to store 3 states:

0 : RED/Transmit Failed status – The pump controller has failed to update to the last status we requested by the dashboard, we infer that the messages are not being received or not being actioned. An example of the latter case is trying to switch the pump on when there is an active alarm. The pump controller will not start the pump so the status messages will continue to have pumpRunning: 0 in them.

1 : YELLOW/In-Transit status – The pump controller has not yet updated to the last status we requested, but not enough time has passed to know if it has received the message or not.

2 : GREEN/Successful status- The pump controller successfully reflects the status of our last request to it.

requestedPumpState is a Boolean variable and stores the On/Off value of our last request to the pump.

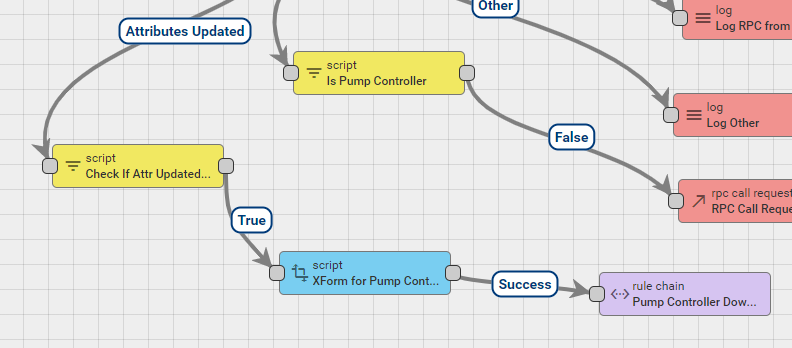


Level Sensor messages are directed to the Water Level Check rule chain, Pump Controller messages are sent to the alarms rule chain to check for alarms. Here we also set the transmitStatus and requestedPumpState Server Attributes.

The transmistStatus is determined by whether the pumpRunning flag on the incoming message matches the requestedPumpState. If they match; the transmitStatus is set to Successful. If they don’t match, and the pump controller has completed at least one status message cycle the status is Transmit Failed. If they don’t match, but it’s within the time it takes for a status cycle to process we set the transmitStatus to In-Transit.

The transmit status decision is time-based, not message-count-based. The Set Transmit Status node in Root Rule Chain contains the code that decides if enough time has passed since the requested state change that the transmit status should be changed to 0/red. Counting messages would have required even more rule nodes.

While the last transmission was successful, and the transmitStatus is in a Successful state: If we receive a status from the pump where the pumpRunning flag no longer matches transmitStatus the requestedPumpStatus is updated to match the last pumpRunning flag. This allows the dashboard to update the manual on/off switch to reflect the state of a pump that has been turned off by alarm or timer expiry.



The root rule chain also listens for changes to the requestedPumpState attribute. When this attribute is updated the message is transformed and sent to the Pump Controller Downlink rule chain to be sent to the pump.

### Pump Controller Alarms

This rule chain manages the pump controller offline alarm, the various alarms that can be raised by the pump controller Feather, and the pump controller restart event.

The pump controller restart event is handled here because it is similar to the pump controller alarms in that it is based upon a bit flag in the pump controller status message. Instead of raising a ThingsBoard alarm, a new telemetry message is created – similar to the pump on and off event messages – so it appears in the pump controller event list.

The pump controller offline alarm is driven by a ThingsBoard feature that monitors when a device was last heard from. The pump controller inactivityTimeout server attribute is used to set the length of time before the alarm is raised and ThingsBoard generates Activity and Inactivity messages based upon this value. These are the messages used to raise or clear the alarm.

The various pump controller alarms are raised or cleared depending on the value of their bit flag in the latest pump controller status telemetry message.

### Water Level Check

This rule chain is triggered by an incoming telemetry message from the tank level sensor. It checks if automatic control is enabled and if so, whether the current water level is below the low-level threshold or above the high-level threshold. If either of these are true and the pump is not already in the necessary state, then the pump is switched on or off via the Pump Controller Downlink rule chain.

This rule chain also updates the estimated fill time value based upon the current water level and the fill rate recorded in the tank level sensor tankFillRate server attribute. After the new estimated time is calculated the message content is changed to be an attribute update message with the new value of the tank level sensor fillTimer server attribute, and sent into a save attributes node.

NOTE! The Get Fill Time node uses a constant for the maximum desired tank level – see the expression that calculates msg.fillTimer. It might be better if the code used the highLevel server attribute of the pump controller.

The rule chain is complicated by the fact that it has to find the pump controller associated with the tank level sensor in order to check if automatic control is enabled and get the level thresholds, plus the communications protocol required if a command has to be sent to the pump controller. This is all done in the GetAutoControlAttr node. The “tank level sensor Manages pump controller” device relationship allows this to be done.

The LevelCheck node is where the current water level is checked against the threshold levels and pump state and a decision made whether to turn the pump on or off, or do nothing.

If the pump must be switched on or off, the incoming telemetry message from the tank level sensor must be transformed into a pump control message suitable for use by the Pump Controller Downlink rule chain. This involves changing the message originator from the tank level sensor to the pump controller and changing the message content to a pump on/off command.

### Pump Controller Downlink

The Pump Controller Downlink rule chain must be called with a POST\_ATTRIBUTE\_REQUEST message with values for requestedPumpState, mode, and command. The requestedPumpState attribute change is what would have caused the message to be send, after being changed from the manual switch on the dashboard or generated by the water level check rule chain.

If requestedPumpState is true then the pump should be switched on, if it is false the pump should be switched off. The command field is from earlier in development and should be removed. It is always 0 when requestedPumpState is false, and 1 when requestedPumpState is true.

The mode field is used to record whether the action was generated by the manual switch or automatic control. It becomes part of the event log telemetry entry.

One branch of this rules chain changes the message to a POST\_TELEMETRY\_REQUEST message with the mode value and a Pump on/off message.

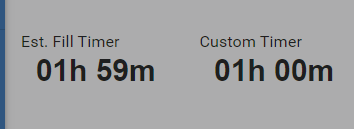
The other main branch transforms the input message into a message that can be sent to the pump controller, including information about whether a timer should be sent with a pump on message. The original and device attribute nodes gather this information and then the script nodes do the transformation into a protocol-specific format which are then sent using the MQTT downlink or RPC request nodes.

The transmitStatus server attribute for the pump controller is set to “1” as well, to indicate there is a message ‘In-Transit’.

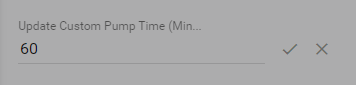
### Dashboard and Custom Widgets

The dashboard contains several custom widgets that work together to give a greater level of information to users controlling a bore pump.

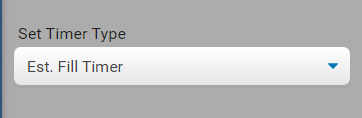
### Timer Display Widgets

These are probably the simplest of all our custom widgets. They take the fillTimer from the Tank Level Sensor and customPumpTimer from the Bore Pump Controller devices and convert the integer minute value and convert it to a more readable HH:MM value.

### Update Custom Pump Timer

This widget is based on the default update Integer Server Attribute Widget set to control the updateCustomPumpTimer. The only custom change that has been made is the addition of a unique HTML id tag to all us to disable the widget from another widget on the dashboard.

### Set Timer Type Widget

The Set Timer Type Widget controls the timerType integer Server Attribute of a Pump Controller Device. The attribute only has 3 legal values:

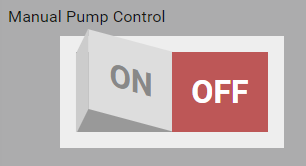
0: Not Timer

1: Estimated Fill Timer

2: Custom Timer

These values are mapped to the options of HTML select object and are updated when the select box value changes. Like the Update Custom Pump Timer widget, this widget has a unique HTML id tag so it can be disabled from another widget on the dashboard.

### Manual Pump Control Switch Widget

This widget controls the requestedPumpState Boolean Server Attribute. It is a CSS treatment of a standard HTML input checkbox. However, we have added an extra function to this switch. When the switch is turned on the function that updates the server attribute also adds disabled HTML tags to the Set Timer Type and Update Custom Timer widgets. When turned off the disabled tags are removed. This is a simple workaround to prevent users trying to update the timer value when the pump was unable to accept the updated value.

### Transmit Status Widget

This widget indicates to users the status of the last message they attempted to send to the Pump Controller. It is simply a display of the transmitStatus integer Server Attribute. When the attribute is updated, rather than displaying the integer value, we match the value as follows:

0: Transmit Failure

1: In-Transit

2: Succesful

# Firmware

The firmware is the software that runs on the AdaFruit Feather board and talks to the bore pump electronics. It can be compiled for either a WiFi or LoRaWAN Feather.

## Source Code Repository

The firmware source code can be found at: <https://bitbucket.org/borepumpcsu/firmware/src/master/>

## IDE Configuration

The firmware for the bore pump controller is an Arduino sketch and can be edited, compiled, and downloaded to the controller with the Ardunio IDE.

The Arduino IDE must be configured to work with Feathers following the instructions at: <https://learn.adafruit.com/adafruit-feather-m0-basic-proto/setup>

## The source files

|  |  |
| --- | --- |
| arduino\_secrets.h | WiFi SSID/password, ThingsBoard device access token, ThingsNetwork application and device identifiers. Any password/secret like information. |
| config.h | The main switch for whether to compile for LoRaWAN or WiFi is set here as USE\_LORAWAN or USE\_WIFI. The Thingsboard URL and I/O pin assignments are also set here. |
| PumpController.ino | The main sketch, has the overall pump control logic and timer and command callback functions. |
| PumpIO.h/.cpp | The interface for controlling the pump and reading the alarm signals. |
| NetworkAccess.h | The interface for abstracting network access. PumpController.ino only uses this interface for communications. |
| LoRaWAN.h/.cpp | The LoRaWAN implementation of the NetworkAccess interface. |
| WiFiNetwork.h/.cpp | The WiFi implementation of the NetworkAccess interface. |

## Compiling for LoRaWAN or WiFi

See §4 of the User Manual for detailed instructions on how to configure the firmware source code for LoRaWAN Feathers. In summary the USE\_LORAWAN macro must be defined in config.h, the USE\_WIFI macro must not be defined, and the various ThingsNetwork app and device IDs must be set in arduino\_secrets.h.

For WiFi Feathers, define the USE\_WIFI macro in config.h, ensure the USE\_LORAWAN macro is not defined, and set the WiFi SSID and password, and the ThingsBoard device access token in arduino\_secrets.h.

## Libraries

The following Arduino libraries are required.

**ZeroTC45 1.0.0**. This can be installed using the Arduino IDE library manager.

**ArduinoJson 6.16.1**. This can be installed using the Arduino IDE library manager.

**MCCI LoRaWAN LMIC library 3.2.0**. This can be installed using the Arduino IDE library manager. Only required for the LoRaWAN Feather. You must edit the MCCI\_LoRaWAN\_LMIC\_library/project\_config/lmic\_project\_config.h file in your Arduino libraries folder to ensure the correct LoRaWAN frequency range is defined so the Feather can talk to your gateway(s).

**PubSubClient 2.8.0**. This can be installed using the Arduino IDE library manager. Only required for the WiFi Feather.

**WiFi101 0.16.0**. This can be installed using the Arduino IDE library manager. Only required for the WiFi Feather.

## Logical Design

There are three logical components in the firmware – the *control logic*, the *pump* *interface*, and the *network stack*.

### Control Logic

The *control logic* is defined in PumpController.ino and runs during the standard Arduino sketch loop function and has callbacks to respond to timed events and commands from the dashboard. It checks for alarm conditions using the *electronics interface* and sends status messages triggered via a timer or a status change using the *network stack*.

The *control logic* is responsible for deciding when to switch the pump off (using the *pump interface*) if an alarm condition is raised while the pump is running.

The *control logic* has a few callback functions for working with timed events and commands from the dashboard.

The statusCallback function is called asynchronously in response to the status message timer going off. This is a continuous 10-minute timer running on the processor’s Timer/Counter 4. This function sets a flag to tell the main loop function it should send a status message at the next available opportunity.

The pumpTimerCallback function is called asynchronously when the optional timeout value sent with a pump on command expires and it unconditionally stops the pump running. The processor’s Timer/Counter 5 is used for this and runs in one-shot mode so once it goes off it does not run again until restarted by another pump on with timeout command.

The processCommand function is called from the network stack when a command is received from the dashboard. This callback is not asynchronous like the two timer related callbacks. The path to this function is loop -> network.loop -> (protocol dependent logic) -> processCommand. For either protocol it will be called before network.loop returns and should be kept relatively small and fast.

There is also a small serial port command interpreter that is used for development and testing purposes. It can be safely removed in production if desired.

### Pump Interface

The *pump interface* is defined in PumpIO.h/.cpp with the PumpIO class and StatusMsgBody structure and is responsible for controlling the pump and receiving alarm signals from it. The Feather does not control or communicate with the pump directly but uses an intermediary board that simplifies the electronic interface to the pump. A single output digital line is used to signal whether the pump should be running or not, and a set of input digital lines are used to signal alarm conditions. The *pump interface* is a fairly transparent mapping of these lines. The pump control line is controlled with the startPump and stopPump methods and the alarm lines are returned in a structure of bit flags via the getStatus method.

There is no input line that gives the actual pump running status. This means the controller does not *know* if the pump is running or not, it can only attempt to switch the pump on or off based upon the commands and alarm conditions it receives. The *pump interface* remembers whether the pump is supposed to be running and returns this in the pumpRunning flag of the StatusMsgBody structure.

The *pump interface* debounces the alarm input lines and only updates an alarm status 500ms after a change is detected on an input line. This may result in the line not actually getting a new value if there was a transient change that lasted less than the debounce period.

### Network Stack

The communications logic abstracted behind the NetworkAccess interface defined in NetworkAccess.h. There are WiFi and LoRaWAN implementations of the NetworkAccess interface, and only one of them is present in the compiled firmware.

There is a setup method that must be called from the main sketch setup function to initialize the network stack.

The network stacks are designed to be called every time the main loop function runs via the NetworkAccess.loop method.

Due to the relatively low-level interface provided by LMIC the NetworkAccess interface has a readyToSend method that should be called before attempting to send a status message. For LoRaWAN this will only return true when it seems the stack is in a state where it can send an uplink soon – ie the network has been joined and there is not current RX/TX operation in progress. For WiFi this will return true if a WiFi network has been joined and there is an open connection to the ThingsBoard MQTT broker.

### WiFi

The WiFi stack is defined in WiFiNetwork.h/.cpp and uses MQTT to communicate with ThingsBoard. The device RPC topic is subscribed to for receiving commands and status messages are sent to the broker.

Calling the WiFi loop method checks the MQTT topic for commands and will attempt to reconnect to the WiFi network and ThingsBoard MQTT broker if necessary.

The WiFi stack has been written to emulate the behaviour of LoRaWAN where commands from the dashboard are only delivered to the control logic after a status message is sent. This is not necessary in practice, it was only done to help ensure the control logic would work for LoRaWAN while developing with a WiFi Feather. If it is later decided a WiFi Feather should be used for some installations this delay in delivering commands can be removed.

### LoRaWAN

The LoRaWAN stack is defined in LoRaWAN.h/.cpp and uses the MCCI LMIC library as it seems to be the most actively maintained LMIC variant. This has the downside of making the source incompatible with other variants due to the fact MCCI have extended the lmic\_pinmap structure and introduced new events. The extra events can be safely removed from the onEvent function and the lmic\_pinmap initialisation will need the extra fields commented out if a different LMIC variant is used.

Once there is confidence the LMIC stack is working well the onEvent function could be trimmed to only respond to the EV\_TXCOMPLETE event. Everything else is for debugging.

The LMIC os\_runloop\_once function must be called as often as possible because the library uses polling to check the radio I/O pins.

The standard logic from the LMIC demo programs has been kept which is to process downlinks as soon as they have arrived rather than copying the data to a buffer and setting a flag to process them next time through the main sketch loop function. In practice is seems to be much the same thing assuming LMIC does not produce the EV\_TXCOMPLETE event until all necessary LoRaWAN processing for the uplink/downlink window is complete, which is what the demos and comments imply.

The usual LMIC demo code behaviour of running a tight loop until the EV\_TXCOMPLETE event is received after attempting to send an uplink has *not* been implemented here. The uplink is scheduled via the usual LMIC\_setTxData2 function but the EV\_TXCOMPLETE event is not waited on. The “normal” pattern could be followed but it does not seem to be necessary and it would result in all normal processing being halted for at least 2 seconds every time an uplink was sent.

# Simulator

A simulator was developed to help with the development of the dashboard without the need of a Feather board or tank level sensor. It allows simple control over status messages and has been developed to mostly mirror the logic of the Feather firmware.

## Source Code Repository

The firmware source code can be found at: <https://bitbucket.org/borepumpcsu/pump-sensor-simulator/src/master/>

## Libraries

**Newtonsoft Json.NET 12.0.0**: Can be installed via Visual Studio. Or see <https://www.newtonsoft.com/json>

## Operation

Please see the provided User Manual for detailed instructions on the operation and usage of the Pump Simulator.

## Implementation

### Main Form

The main form consists of four main areas:

* Connection Details – Update the Thingsboard connection information
* Level Sensor – Update the level sensor telemetry data and send it to Thingsboard
* Pump Controller – Update the pump controller status flags and send to Thingsboard
* Event Log – Track incoming and outgoing messages

### Connection Details

Each field in the Connection Details section is bound to an attribute of the ThingsboardProperties class. When details are changed they update the bound attribute, which is then saved permanently in the programs settings.

### Level Sensor

Each field in the Level Sensor section is bound to an attribute of the TankMessage class. On initialization an instance of the TankMessage class is created and is used for the rest of the program’s life. When the Send button is clicked the PostToTB() function of this class is called, which in turn converts the values to a JSON message with the ToJSON() function. PostToTB() then posts the message to Thingsboard using the values in the Connections Details. If the response is not OK a message is logged.

### Pump Controller

Each field in the Pump Controller section is bound to an attribute of the StatusMessage class. On initialization an instance of the StatusMessage class is created and is used for the rest of the program’s life. When the Send button is clicked the PostToTB() function of this class is called, which in turn converts the values to a JSON message with the ToJSON() function. PostToTB() then posts the message to Thingsboard using the values in the Connections Details. If the response is not OK a message is logged.

The DownlinkCommandPoll instance is then checked to see if a command has been received from the dashboard.

### Event Log

The Main Form has an AddToLog() function which takes a string and adds it to the top of the event log object. The event log will check it’s item count, and if there is more than 50 items, delete the oldest one to keep it tidy.

## DownlinkCommandPoller

This class runs in a separate thread, started by the main form, and polls the ThingsBoard device RPC URL to receive commands from the dashboard.

When a command is received it is stored in the rpcCmmand field. Each received command overwrites any existing stored command to replicate the way LoRaWAN downlinks are scheduled – only the most recent downlink is sent.

The main form checks for a command every time a status message is sent. A stored command is nulled out after the main form asks for it, so the main form only receives the command once. This replicates the way downlink commands should only be delivered once to the pump controller.

Thread locking is used to ensure consistency of the stored command field.

The NewtonSoft Json.NET library is used to parse the JSON received from ThingsBoard.