# Interfacing with LoRaWAN

RM191 LoRa + BLE Module

*Application Note* *v1.2*

## Introduction

The Laird RM1xx is a series of wireless communications modules that combines a Nordic nRF51822_QFAC (256/16) BLE device and a Semtech SX1272 860 to 1020 MHz low power, long range transceiver. The RM186 and RM186_PE are designed to function in the EU863-870MHz ISM band as opposed to the RM191 and RM191_PE, which is a very similar device designed for the US 902-928 MHz ISM band.

Both versions can collect data from external sources by interfacing directly with a sensor over a UART, SPI or I$^2$C serial communication channel, or over a wireless BLE connection with a sensor that is physically connected to a BLE peripheral device. The RM1xx can also gather data internally using the analog or digital IO pins. Once the data has been collected and arranged in packets, it may be transmitted to a remote LoRaWAN Gateway up to 16 kilometers away. From there, the data can be transmitted to a server or database over a TCP/IP link.

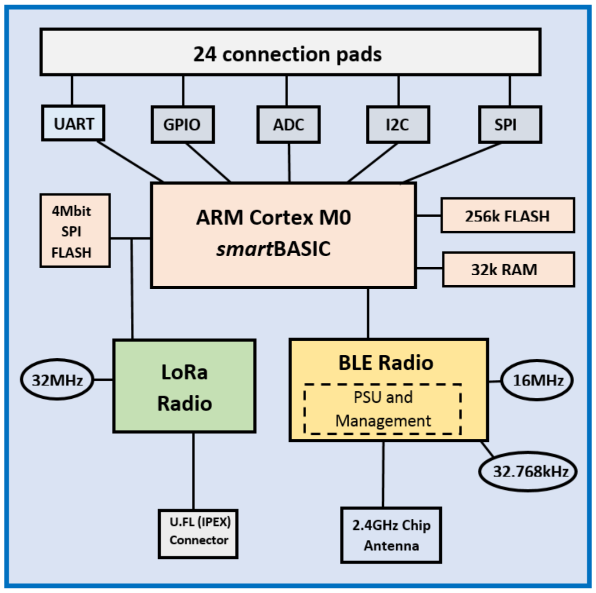
As well as supporting two distinct frequency ranges, the LoRaWAN specification also supports two different protocols. Although there are slight differences relating to power levels and data rates, the main differences between these protocols are how a module Joins a network, specifically which frequency it must use, and how the network then configures the module once it has joined.

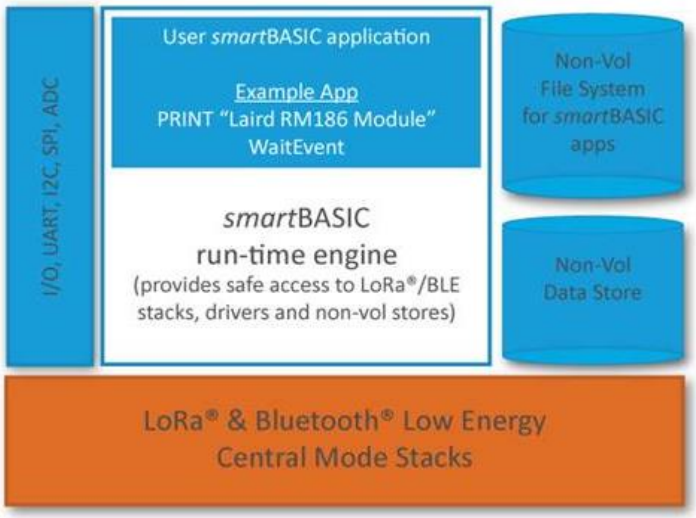
The first protocol, used in the EU and AS (Asia) regions, requires that the modules must support a specified number of mandatory channels on which a JoinRequest must be transmitted. Then, as part of a JoinAccept message, the network configures up to five additional channels that the module can then transmit data packets on.

The second protocol, used in the US and AU regions, initially enables 64 125kHz and 8 500kHz channels, and will transmit JoinRequests on random alternate 500kHz and 125kHz channels until it transmits on a frequency supported by a receiving gateway. Once the module has joined the network, the network can then update the enabled channels by means of a specific MAC command which modifies an internal ChannelsMask parameter which tracks the state of the channels.

The purpose of this document is to describe how the RM191 and RM191_PE, which support the US region protocol, interface with the LoRaWAN specification and explains what happens inside the module to support this protocol.

All smartBASIC APIs and LoRa Events that are referenced below are described more fully in the RM1xx Series *smart*BASIC LoRa Extensions user guide.




I/O, UART, I2C, SPI, ADC
User *smart*BASIC application
Example App
PRINT "Laird RM186 Module"
WaitEvent
*smart*BASIC
run-time engine
(provides safe access to LoRa®/BLE stacks, drivers and non-vol stores)
Non-Vol File System for *smart*BASIC apps
Non-Vol Data Store
LoRa® & Bluetooth® Low Energy
Central Mode Stacks

## Gateways and Servers

LoRaWAN is a wireless standard developed by the LoRa Alliance. It enables low-power wide-area networking using low data rates and minimal power usage. The RM1xx modules are designed to communicate with any LoRaWAN gateway which is within RF range, can receive a signal within the correct frequency range, and meets the requirements defined in the appropriate section of the LoRaWAN specification.

The gateway does nothing with the data, except to timestamp it and then send it to a network server via a TCP/IP link. The network server must be running LoRaWAN-compatible software to handle the decryption of messages and authentication of devices. There are several vendors that provide such services.

A module must be set up as a member of this network. This involves a network supporting the application that the module provides, so the network server knows which Application server to send the data to, and the Application server having a list of all the modules that support that application.

It is possible that a packet could be received by a LoRa gateway that is not connected to the desired network. However, in this instance the data should be discarded by the network server as it cannot be decrypted correctly. The only network server that would be able to decrypt the data successfully is the one to which the RM191 is a member. This is explained in greater detail in the Connecting to the LoRaWAN Network section.

## Transmit/Receive Sequence

Before looking at the various commands and events it is very important to understand that once you send a packet into the LoRa stack you are starting a predefined sequence that must be allowed to complete before you can send another packet.

In loRa an uplink packet refers to a packet transmitted by the LoRa module and a downlink packet refers to a packet transmitted by the network server/Lora Gateway.

In Class A devices, once an uplink packet is transmitted, the module waits a predefined time for a downlink packet. This packet may or may not be transmitted by the gateway, however, the module must still open up receive windows to receive it if necessary.

For unconfirmed packets this is the end of the cycle. There are no resends. The module assumes that the transmitted packet has been received at the gateway and onto the server. The server may have data to send back to the module and these should be picked up in whichever Receive window they coincide with and dealt with accordingly by the stack.

However, in the case of confirmed uplink packets the cycle is more complicated as the stack actively waits or an acknowledgement. If that acknowledgment isn't received the stack automatically resends the packet a set number of times. The sequence only finishes after an acknowledgement is received or all the retries are attempted or an error has occurred causing the sequence to abort.

Similarly, for the JoinRequest, if the stack fails to receive a JoinAccept, it resends the JoinRequest a set number of times. Again, the sequence only finishes after a JoinAccept is received or all retries are attempted or an error has occurred.

This can be very offputting in the RM1xx because there is no indication of all that was happening in the background; it could seem that the module had frozen. However, this is not the case so it is very important not to interfere with this sequence by sending another packet to the stack.

The events that mark the end of the sequence are indicated in section 5.3. It is very important to monitor all these sequence complete events and act on them accordingly.

For testing purposes, it is possible to use the LoramacSetDebug command or monitor the EVLORAMACTXDONE or LORAMACNOSYNC events. These should reassure that there is actually something happening in the background. In the case of the US/AU modules this should be quite quick as there are no duty cycle restrictions. In the case of the EU modules, however, it is likely to be minutes between resends. However here you can use the LoramacGetOption(LORAMAC_OPT_NEXT_TX) command to check if there is another packet due for transmission.

## RM191 Frequencies and Duty Cycles

The RM191 module functions in the US 902-928 MHz ISM band. This band is divided into three different sets of channels. Uplink data can be sent on any one of 72 upstream channels. The first 64 of these channels have a bandwidth of 125 kHz starting at 902.3 MHz and spaced evenly at 200 KHz increments up to 914.9 MHz. The final eight uplink channels have a bandwidth of 500 kHz starting at 903.0 MHz and spaced evenly at 1.6 MHz increments up to 914.2 MHz. There are also eight downlink channels which also have a bandwidth of 500 KHz starting at 923.3 MHz and spaced evenly at 600 KHz increments up to 927.5 MHz. Unlike the EU 863-870 band, there are no duty cycle restrictions for the US 902-928 band.

While the LoRaWAN specification clearly states that 72 uplink channels are available for US 902-928 mode, this is often overlooked with networks that have smaller number of gateways. For this reason, in most of the available gateways the full spectrum of frequencies has been split up into a series of sub-bands of eight 125kHz channels and one 500kHz channel.

The RM191 module defaults to having all 72 uplink channels enabled. This is done by setting all the bits in the ChannelsMap, which is an internal value that controls which channel are enabled. Each bit of the channelsmap represents an individual channel. This is because the protocol is designed so that a module will transmit a JoinRequest on alternate random 500kHz and 125kHz channels until it eventually transmits on a frequency supported by a gateway. The gateway can then send the request to the network server, and if successful it transmits the JoinAccept back to the module.

When the module starts transmitting data to the network the network should then respond with a series of MAC Commands that will configure the module's channelsmap to match that of the available gateways. Once this configuration is completed the module should only then transmit on the correct frequencies for that network. This means that it should only be the initial JoinRequest that will have to be transmitted a number of times. Data packets should only be transmitted on supported frequencies.

## LoRaWAN Events

A series of events are raised by the LoRaWAN stack to indicate the success or failure of a specific task. These events are then routed through the RM191 firmware and finally output as *smart*BASIC events. Some of these events are raised by the stack. Some have been created by Laird to make the system more user friendly in terms of feedback.

The events listed in Table 1 are referenced later in this document.

***Table 1: LoRaWAN events***

| Event | Description |
|---|---|
| EVLORAMACJOINING | A JoinRequest has been sent to the gateway/server. |
| EVLORAMACJOINED | The JoinRequest has been successfully received by the gateway/server and the subsequent JoinAccept has been received by the RM191. The module is now connected to the network. |
| EVLORAMACJOINFAIL | Indicates that a JoinAccept message contains a MIC error. However, no direct action should be taken on receipt of this event. |
| EVLORAMACTXDONE | A signal from the module indicating that a packet was transmitted from the SX1272. This can be a JoinRequest, LinkCheck or data packet. This is an important event as all subsequent receive timings are taken from this event. However, no direct action should be taken on receipt of this event. |
| EVLORAMACTXCOMPLETE | Created when an uplink packet is loaded into the radio. However this event is not thrown until the successful end of the uplink/downlink sequence. |
| EVLORAMACRXDATA | A downlink packet has been received that contains data from the server to the module. |
| EVLORAMACRXCOMPLETE | A downlink packet has been received by the module. This event can indicate a combination of an acknowledgement, a MAC command or actual data from the server. Downlink packets are not limited to confirmed uplink packets. MAC commands and/or downlink data packets can be transmitted in response to an unconfirmed uplink packet. |
| EVLORAMACTXTIMEOUT | Very rarely thrown. Usually indicates that a packet has failed to be transmitted due to an internal SPI error. |
| EVLORAMACRXTIMEOUT | Only valid with JoinRequests, confirmed uplink or LinkCheck packets. For JoinRequests and confirmed uplink packets, it is sent after the module fails to receive a downlink after the configured number of transmission attempts. A LinkCheck is not retransmitted so it is sent if the LinkCheck doesn't receive a response. |
| EVLORAMACRXERROR | An RxError has been received from the SX1272. |
| EVLORAMACSEQUENCECOMPLETE | This event is an combination of all the events that can indicate the end of an uplink/downlink sequence. Two variables are returned with this event, the first indicates the terminating event via the |

| Event | Description |
|---|---|
| | following values:<br>1: **EVLORAMACTXCOMPLETE**<br>2: **EVLORAMACRXCOMPLETE**<br>3: **EVLORAMACRXTIMEOUT**<br>4: **EVLORAMACRXCRCERROR**<br>5: **EVLORAMACTXTIMEOUT**<br>6: **EVLORAMACRXERROR**<br>7: **EVLORAMACTXDRPAYLOADSIZEERROR**<br>The second parameter is the time until there is available duty cycle to transmit another uplink packet. This is also the time until the **EVLORAMACNEXTTX** Event will be thrown. For US/AU modules this will always be 0. |
| EVLORAMACNEXTTX | This event indicates that there is now duty cycle available to transmit the next uplink packet. For modules that do not have duty cycle constraints this event will always be thrown at the same time as the **EVLORAMACSEQUENCECOMPLETE** event. |
| EVLORAMACNOSYNC | Notification that a receive window has closed without receiving a sync pulse. This event is for information only and no action should be taken on receipt of this event. |
| EVLORAMACADR | Notification that an ADR command has been received from the gateway and that some transmit parameters may have changed.<br>This event also returns 2 parameters -<br>Packet Type:<br>3: unconfirmed downlink packet<br>5: confirmed downlink packet<br>Frame Pending :<br>If 1 it indicates that the server has any packets yet to be transmitted. |
| EVLORAMACLINKCHECKRESPMSG | Notification that the module has received the response to a LinkCheck packet. |
| EVLORAMACMICFAILED | There has been a MIC error in a received downlink packet. |
| EVLORAMACDOWNLINKREPEATED | The received downlink packet is a repeat of the previous downlink packet. |
| EVLORAMACTXDRPAYLOADSIZEERROR | Notification that a packet is too large to be transmitted with the current datarate setting.  This event can only occur during the resending of a packet where the datarate has been reduced by the stack. |
| EVLORAMACFRAMESLOSS | Indicates that the received downlink sequence number is larger than the maximum expected value.  This event should not be acted upon directly, the stack should correct this automatically. |

## Data Rate

The LoRaWAN specification states that the following transmit data rates must be supported by the RM191.

***Table 2: LoRaWAN transmit data rates for RM191***

| Data Rate | Configuration | Physical Bit Rate (bit/s) |
|---|---|---|
| 0 | SF10 @ 125kHz | 980 |
| 1 | SF9 @ 125kHz | 1760 |
| 2 | SF8 @ 125kHz | 3125 |
| 3 | SF7 @ 125kHz | 5470 |
| 4 | SF8 @ 500kHz | 12500 |
| 5-7 | Reserved for future use | - |
| 8 | SF12 @ 500kHz | 980 |
| 9 | SF11 @ 500 kHz | 1760 |
| 10 | SF10 @ 500 kHz | 3900 |
| 11 | SF9 @ 500 kHz | 7000 |
| 12 | SF8 @ 500 kHz | 12500 |
| 13 | SF7 @ 500 kHz | 21900 |
| 14:15 | Reserved for future use | - |

SF7 to SF12 refers to the spreading factor of the system. The spreading factor refers to how many chips of information represent each bit (or symbol) of payload data. The actual chip value is calculated by $2^x$. So for SF7 there are $2^7$ (128) chips per symbol and for SF12 there are $2^{12}$ (4096) chips per symbol.

This explains why the physical bit rate is heavily dependent on the spreading factor. The higher the spreading factor, the more chips that must be sent for each bit of information and, consequently the longer the time it takes to transmit the data. This is very important when you look at the potential data throughput of a module.

The default configured bit rate for the RM191 is data rate 3, SF7@125kHz. At any time, you can reconfigure this value using the **LORAMACSetOption(LORAMAC_OPT_DATA_RATE,var$)** command and read it back using the **LORAMACGetOption(LORAMAC_OPT_DATA_RATE,var$)** command. These are important commands because the server can modify the data rate at any time using the **LinkAdrReq** command.

As with any changes to the configuration of the frequency channels by the gateway/server, notification of changes to the data rate are not automatically passed back to the user. You must use the *smart*BASIC **LORAMACGetOption** command to obtain the latest configuration.

As with any changes to the configuration of the frequency channels by the gateway/server, notification of any changes to the data rate are not automatically passed back to the user. However the **EVLORAMACADR** is thrown on receipt of an ADR command from the server. Then call the appropriate *smart*BASIC **LORAMACGetOption** commands to obtain the latest configuration.

**Note:** Increasing the data rate reduces the maximum range of the module by making the signal more susceptible to noise. At high spreading factors, the receiver can receive data at much lower signal strengths than it can with low spreading factors.

## UPLINK/DOWNLINK CYCLE

Before looking at the various commands and events it is very important to understand that once you send a packet into the LoRa stack you are starting a predefined sequence that must be allowed to complete before you can send another packet.

In loRa an uplink packet refers to a packet transmitted by the LoRa module and a downlink packet refers to a packet transmitted by the network server/Lora Gateway.

In Class A devices, once an uplink packet is transmitted, the module waits a predefined time for a downlink packet. This packet may or may not be transmitted by the gateway, however, the module must still open up receive windows to receive it if necessary.

For unconfirmed packets this is the end of the cycle. There are no resends. The module assumes that the transmitted packet has been received at the gateway and onto the server. The server may have data to send back to the module and these should be picked up in whichever Receive window they coincide with and dealt with accordingly by the stack.

However, in the case of confirmed uplink packets the cycle is more complicated as the stack actively waits or an acknowledgement. If that acknowledgment isn't received the stack automatically resends the packet a set number of times. The sequence only finishes after an acknowledgement is received or all the retries are attempted or an error has occurred causing the sequence to abort.

Similarly, for the JoinRequest, if the stack fails to receive a JoinAccept, it resends the JoinRequest a set number of times. Again, the sequence only finishes after a JoinAccept is received or all retries are attempted or an error has occurred.

This can be very offputting in the RM1xx because there is no indication of all that was happening in the background; it could seem that the module had frozen. However, this is not the case so it is very important not to interfere with this sequence by sending another packet to the stack.

The events that mark the end of the sequence are indicated in section 5.3. It is very important to monitor all these sequence complete events and act on them accordingly.

For testing purposes, it is possible to use the LoramacSetDebug command or monitor the **EVLORAMACTXDONE** or **LORAMACNOSYNC** events. These should reassure that there is actually something happening in the background. In the case of the US/AU modules this should be quite quick as there are no duty cycle restrictions. In the case of the EU modules, however, it is likely to be minutes between resends. However here you can use the **LoramacGetOption(LORAMAC_OPT_NEXT_TX)** command to check if there is another packet due for transmission.

## CONNECTING TO THE LORAWAN NETWORK

Before the RM191 can transmit any data to a LoRaWAN network it must first be connected to that network. As the RF link between the RM191 and a gateway is over a secure channel, this connection process involves the exchange of certain keys between the module and the network server.

This can either be achieved by calculating the keys afresh every connection or by configuring the RM191 and network server with identical key information in advance.

In both cases below, the module will throw the **EVLORAMACJOINING**, **EVLORAMACJOINED** and **EVLORAMACNEXTTX** during the Join process.

## Over-the-Air Authentication

The recommended method of connecting an RM191 to a network is by using the Over-the-Air Authentication (OTAA) method. With this method the module must be configured with certain IDs so that the Network Session Key (NwkSKey) and the Application Session Key (AppSKey) can be calculated. These IDs are also transmitted to the server as part of the JoinRequest command so that the NwkSKey and AppSKey can be calculated and stored there as well. These keys are unique between a module and a server.

The following IDs must be configured:

- **Application Identifier (AppEUI) –** This is a global application ID in IEEE EUI64 which uniquely identifies the application provider of the module.
- **End-device Identifier (DevEui) –** This is a global end-device ID in IEEE EUI64 which uniquely identifies the module/end-device.
- **Application Key (AppKey) –** This is an AES-128 application key specific to the module which is used to derive the session keys NwkSKey and AppSKey.

These values can be stored and read back from the RM191 using the following commands listed in Table 3. The values set using the **LORAMACSETOPTION** are not persisted over a power cycle.

*Table 3: OTAA IDs*

| ID | Data Length (Bytes) | Write Command | Read Command |
|---|---|---|---|
| AppEui | 8 | at+cfgex 1010 "xxxx" (For firmware versions prior to 17.4.1.0 the AppEui Id is 1000) | at+cfgex 1010? |
| | | LORAMACSetOption(7, xxxx) | LoramacGetOption(5) |
| DevEui | 8 | N/A. Configured during production | ati 25 |
| | | at+cfgex 1011 "xxxx" (For firmware versions prior to 17.4.1.0 the DevEui Id is 1001) | at+cfgex 1011? |
| | | LORAMACSetOption(5, xxxx) | LoramacGetOption(7) |
| AppKey | 16 | at+cfgex 1012 "yyyy" (For firmware versions prior to 17.4.1.0 the AppKey Id is 1002) | Write only |
| | | LORAMACSetOption(6,yyyy) | |

**Note:** The **xxxx** above represents an 8 byte hexadecimal value and **yyyy** a 16 byte hexadecimal value.

The IDs configured using the **at+cfgex** command only take effect after a module reset. This is not performed automatically as part of the command; it must be manually initiated.

IDs configured using the **LORAMACSetOption** API are only valid while the module is powered. The value is not persisted so is discarded when the module is powered down or reset.

**Note:** There are two options for configuring a permanent DevEui. The device is initially configured with a global DevEui during production. However, it is possible to override this value with a local value using the **at+cfgex** option. When it comes to selecting which one to use, the code will select the local value first if it is available. If not, it reverts to the global value. The **ati 25** command only returns the global value and **at+cfgex 1011?** only returns the local value.

The **LORAMACSetOption** only temporarily replaces **the at+cfgex** option.

On receipt of a **LORAMACJoin(LORAMAC_JOIN_BY_REQUEST)** command, the module first checks that all the required parameters have been configured. If not, the join request is cancelled and an error message is returned.

If everything is working, then the join request is transmitted to the gateway/server on a random 500kHz channel. If successful, an **EVLORAMACJOINED** event is passed to the user. This event is not received for at least five seconds after the transmission of the join request command due to the preprogrammed delays in the system. This is explained in the Data Reception section and the expected delays are defined in Table 5 below.

If the join request fails, the firmware will resend the JoinRequest on a random 125kHz channel. If this fails the default action of the RM191 is to stop transmitting JoinRequests and throw an **EVLORAMACRXTIMEOUT** event to the smartBASIC app**.** So both attempts have a 1 in 8 chance of success of selecting a supported frequency. The module remembers which frequencies have been attempted and should not transmit again on those frequencies until all others have been attempted. So the chances of success should improve.

However, on receiving the **EVLORAMACRXTIMEOUT** event the app needs to send another JoinRequest. This causes the record of previous attempts to be cleared so the chances of success return to 1 out of 8. Therefore it could take many attempts to Join before you are eventually successful. This should only happen with the JoinRequests because once joined the network server should configure the module with the correct channlesmap for that network.

There are very good reasons for limiting the number of Join attempts. The first has to do with saving power. In the previous version of the stack the module would just keep on sending retries. If there were no gateways in the area this would only waste power. Secondly, when the module moves into this retry sequence it doesn't change the packet in any way. So in the case of JoinRequests the DevNonce value doesn't change. This brought about the slim possibility that if a gateway/server receives and responds to the JoinRequest with a JoinAccept but that JoinAccept is not received by the module, any further JoinRequests with the same DevNonce will be rejected. So the module would never be able to connect with any amount of retries.

This can be very frustrating during testing. So Laird has developed several workarounds. The simplest is to increase the number of JoinRequests that will be attempted before throwing **EVLORAMACRXCTIMEOUT** event. This can be carried out using:

**LORAMACSetOption(LORAMAC_OPT_MAX_JOIN_ATTEMPTS, “x”)**

where x is the number of Join attempts, up to a maximum of 8. The value is passed in as a string. This value must be set before sending a JoinRequest as this is when the number of retries is set up. Increasing this value means you increase the probability of the module selecting a supported frequency. It is also in keeping with the expected operation, in that the module will “search” for a gateway.

It is also possible to modify the channel configuration of the RM191, the channelsmap parameter, by using one of the 2 available methods. These methods can be set either through a configuration command or through a smartBASIC API. As is usual with all RM191 modules, the configuration commands are persisted, and the smartBASIC commands are lost after a reset.

It is important to realise that for these options to work you must first know how a gateway is configured. If you set up the channelsmap incorrectly it is possible that the module will never connect to a network. The default method of just trying random channels may be slow, but it will eventually succeed.

Because there are multiple options to configure the channelsmask, the firmware needs to know which to use. So there is another configuration variable required using **at+cfg 1002 Y** which defines the method that must be used. The valid values for Y in this command are as follows:

| at+cfg 1002 Y | Methods |
|---|---|
| 0 | Default : all channels enabled |
| 1 | Sub-band setting configured via **at+cfg 1001** or **LORAMACSetOption(LORAMAC_OPT_SUBBAND, …)** |
| 2 | ChannelsMask configured directly via **at+cfgex 1009** or **LORAMACSetOption(LORAMAC_OPT_CHANNELMASK,…)** |

In most instances it is recommended that option 1 is selected, which sets a specific sub-band. This is how the majority of gateways work. The only time that option 2 would be useful is if you want to split the channels map over multiple sub-bands. This is not an option that is generally supported in most gateways, but it is an option on Laird's Sentrius gateway.

More information on these commands are given in the RM1xx Series LoRa *smart*BASIC Extensions User Guide.

## Activation by Personalization

When using Activating by Personalization, the NwkSKey and the AppSkey must be configured on both the RM191 and any server with which it might communicate. By default, it is likely that the DevEui and AppEUi will also be set in the module as the values must be set on the network server, however these IDs are not used in the Join process.

- **Network Session Key (NwkSKey):** Specific to the end device. Used by the module and gateway/server to calculate the checksum of all data messages. Also used to encrypt and decrypt the payload field of MAC only data messages.
- **Application Session Key (AppSKey):** Specific to the end device. Used by the module and gateway/server to encrypt and decrypt the payload data. It may also be used to calculate the optional payload checksum.
- **End Device Address (DevAddr):** 32 bits (4 bytes) that identifies the module within the current network.

*Table 4: Personalization IDs*

| ID | Data Length (Bytes) | Write Command | Read Command |
|---|---|---|---|
| NwkSKey | 16 | at+cfgex 1013 "xxx.xxxx" (For firmware versions prior to 17.4.1.0 the NwkSKey Id is 1003) | Write only |
| AppSKey | 16 | at+cfgex 1014 "xxx.xxx" (For firmware versions prior to 17.4.1.0 the AppSKey Id is 1004) | Write only. |
| DevAddr | 4 | at+cfgex 1015 "xxxx" (For firmware versions prior to 17.4.1.0 the DevAddr Id is 1005) | at+cfgex 1015? |

**Note:** The **xxxx** above represents a hexadecimal value. For example:

**at+cfgex 1013 "2b7e151628aed2a6abf7158809cf4f3c"**

As with the OTAA, once a **LORAMACJoin(LORAMAC_JOIN_BY_PERSONALIZATION)** is received by the module it checks that the above IDs (Table 4) are configured. If not, an error message is returned and the request is cancelled.

When you join a network using the personalization method, there is no handshaking required between the RM191 and the server. Both sides of the link should be configured with the same key values and the RM191 assumes to have joined the network as soon as the command is sent. The module is ready to start transmitting. See the section on personalization and the handling of the sequence numbers below.

If using the Multitech gateway, refer to the Conduit mLinux: LoRa Use with Third-Party Devices webpage for details on how to configure the gateway.

## DATA TRANSMISSION

Transmitting data to the server is a simple task. Call the *smart*BASIC **LoramacTxData** command containing the data you wish to send to the server along with the port number and the confirm flag. Everything else is handled by the firmware.

A recent change to the RM1xx firmware has now made it possible to query the stack as to the maximum possible packet size using the **LORAMACQueryTxPossible** API, which returns the theoretical and actual maximum packet sizes. With this command it is now possible to control the data packets more efficiently. It is worth pointing out here that in the case of confirmed packets it is possible that although the packet is valid when you first send it, if it enters the resend phase, as the datarate reduces the packet size may become too large and the **EVLORAMACTXDRPAYLOADSIZEERROR** will be thrown.

When a packet is loaded into the stack it randomly selects one of the enabled channels and transmits the packet on that channel. Once a packet has been loaded it cannot be deleted or overwritten. As there are no duty cycle limitations with the RM191, any packet is transmitted immediately.

There are a number of different ways the successful completion of an uplink/downlink sequence can be determined using the LoRa events.

You should always receive an **EVLORAMACTXCOMPLETE** event, provided nothing has gone wrong with the transmission or reception of data, i.e. you received an **EVLORAMACTXDONE** event, and nothing has gone wrong with any received packet. If there is an error in the received packet the **EVLORAMACTXCOMPLETE** will not be thrown.

The only exception is when a confirmed transmit packet is rejected by the server, but the server has a downlink packet to transmit. The downlink packet would still be transmitted so in this case you would receive the **EVLORAMACRXCOMPLETE** event but not the **EVLORAMACTXCOMPLETE** event as the uplink would not have been acknowledged.

If there is a successful downlink packet you should also receive an **EVLORAMACRXCOMPLETE** event along with the **EVLORAMACTXCOMPLETE** . This confirms an acknowledgment to a confirmed uplink packet. If that packet also contains data there will also be an **EVLORAMACRXDATA** event.

To make the process simpler, a new event has been created: **EVLORAMACSEQUENCECOMPLETE**. Internally the firmware monitors what is supposed to happen and so which event should end the uplink/downlink sequence. This event is then thrown along with a variable indicating which event was the actual terminating event. So now all that is required is to monitor the **EVLORAMACSEQUENCECOMPLETE** and act on that.

Examples of when the events are thrown are shown below. The waveforms are obtained using the **LoramasSetDebug** API. The blue waveform represents when the module is transmitting and the red waveform represents when the module is in receive mode.

In figure 1 the module has been programmed to transmit a confirmed packet. Once the module stops transmitting, the **EVLORAMACTXDONE** event is thrown. One second later the first receive window opens. This window will open for a minimum of 8 preamble characters. If the module doesn't receive these preamble characters from the gateway it will close. In this case it does receive these characters and so stays open. At the end of the window, once all of the downlink packet has been received the window closes and the module throws the following events :

- **EVLORAMACTXCOMPLETE** event – because it has successfully transmitted a packet
- **EVLORAMACRXCOMPLETE** event – because the downlink packet was received correctly
- **EVLORAMACSEQUENCECOMPLETE** event – returned with a numeric value of 2
- **EVLORAMACNEXTTX** event – because the module can now transmit the next packet

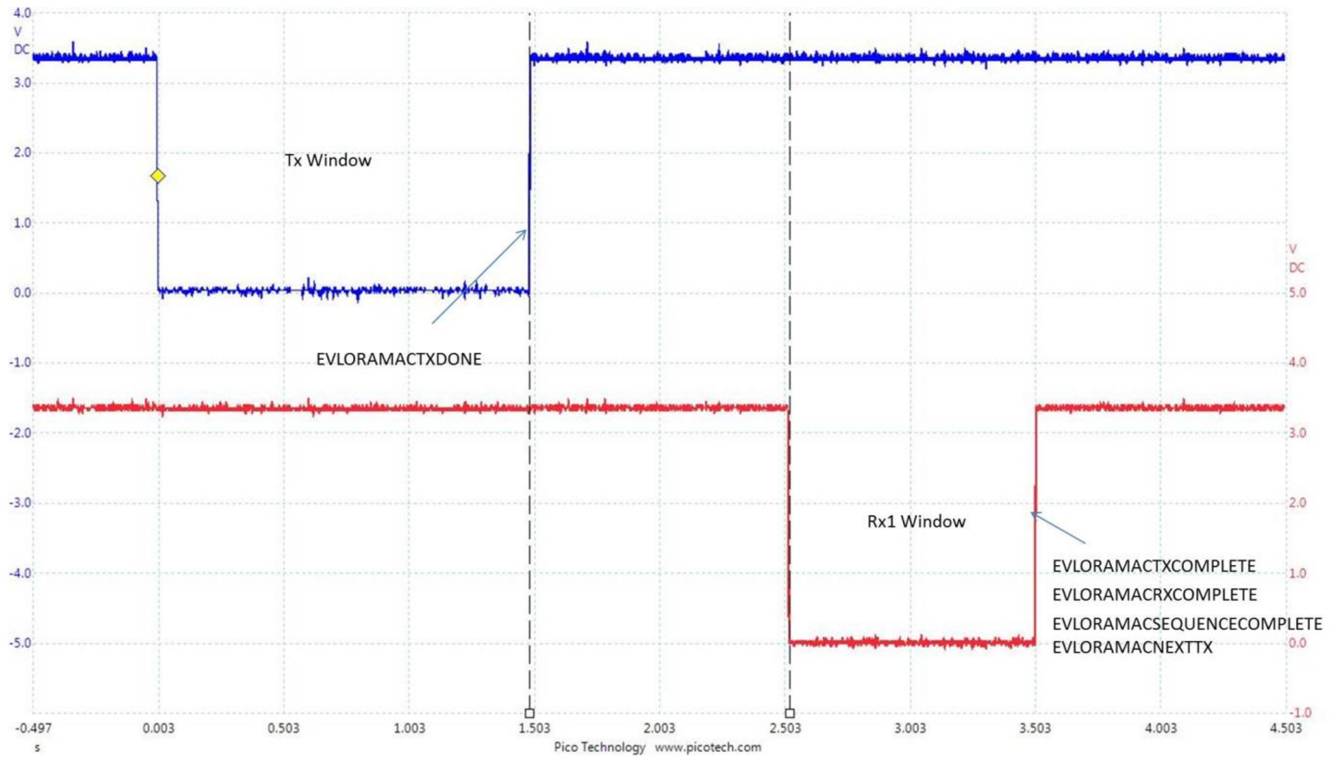


***Figure 1: Confirmed packet***

Figure 2 below shows an example of an unconfirmed packet transmitted to the gateway. The transmit trace is the same as before. However, because it was an unconfirmed uplink there is no downlink packet. So the receive windows will only open for the minimum time causing the **EVLORAMACNOSYNC** events to be thrown. In this example the numeric value returned with the **EVLORAMACSEQUENCECOMPLETE** event would be 1.

Again the **EVLORAMACNEXTTX** is thrown, indicating the module can now transmit the next packet.

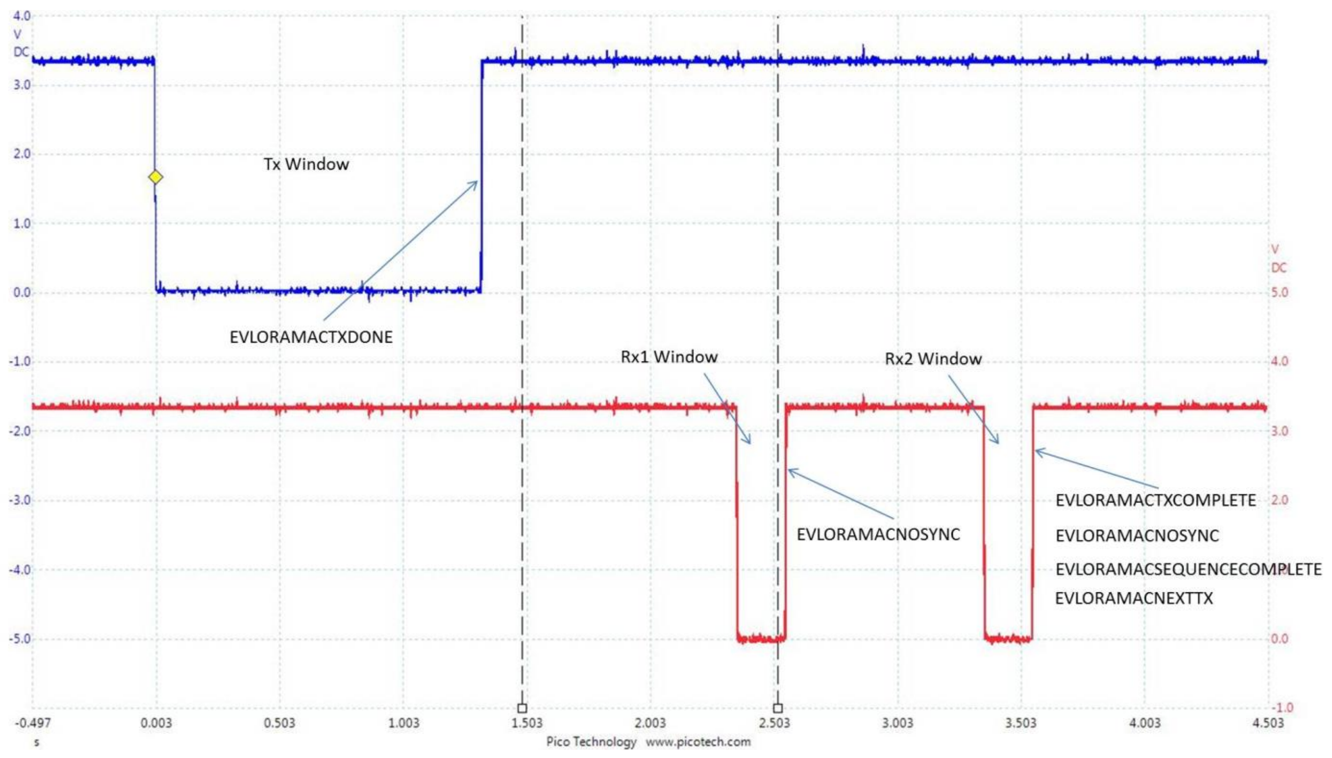


***Figure 2: Unconfirmed data packet***

As mentioned in an earlier section, if the transmission fails, the confirmed/not confirmed option determines what happens next. If the confirmed option is selected and the confirmation is not received, the module attempts to retransmit the packet. By default, the module attempts to send a specific packet eight times, i.e. the initial attempt and seven retries. The number of retries can be configured using the **LORAMACGetOption(LORAMAC_OPT_MAX_RETRIES,var$)** command. The number of retries cannot exceed eight. If the number of retries is reached without a receiving a response, then an EVLORAMACRXTIMEOUT event is thrown.

During this retransmit process the LoRaWAN stack automatically decrements the data rate after every two failed transmissions. So, depending on the starting data rate, the data rate at the end of this process could be much lower than at the beginning, potentially causing this process to take longer than might be expected. The module tends towards decreasing the data rate as this has the effect of increasing the range of the module. Reducing the datarate could have the effect of now making the packet too large for transmission causing a **EVLORAMACTXDRPAYLOADSIZEERROR** event to be thrown.

If an **EVLORAMACRXTIMEOUT** event is received, it is up to the user how to proceed. You can either try to join the network again or try sending another packet.

One the uplink/downlink sequence has completed it is possible to transmit the next uplink packet immediately, so the **EVLORAMACNEXTTX** event will fire immediately after the **EVLORAMACSEQUENCECOMPLETE** event.

## DATA RECEPTION

After each transmitted uplink packet on a Class A LoRaWAN device, the module must listen for a downlink packet which may or may not be sent from the gateway. There are two possible receive windows in which this downlink packet can be transmitted, illustrated in Figure .

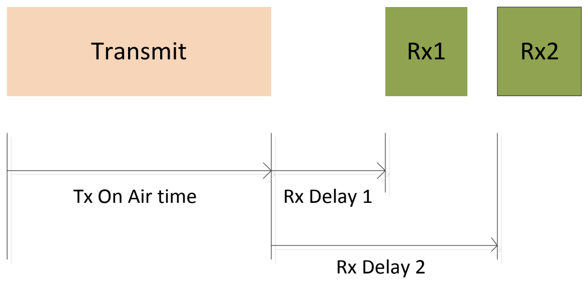


***Figure 3: Receive window timings***

The actual length of the delays (Table 5) is dependent on whether the RM191 is transmitting a join request or a normal data packet.

***Table 5: Receive delay times***

| Packet Type | Rx Delay 1 | Rx Delay 2 |
|---|---|---|
| Join Request | 5 seconds | 6 seconds |
| Data Packet | 1 second | 2 seconds |

In the first receive window, the downlink frequency channel is a function of the uplink channel, calculated by

Receive Channel = Transmit Channel modulo 8

With the second receive window, the gateway transmits on the same frequency as the uplink packet.

The downlink data rate depends on the uplink data rate.

If the downlink packet is transmitted in the first window and is successfully received by the module, then the second window is not required and does not open.

If the downlink packet is not received during the first Receive window, then the second Receive window is opened. The first receive window frequency is a function of the upstream channel and shares the same data rate as the upstream message. The second receive window has a set frequency and data rate. By default, the frequency and data rate of the second receive window are 923.3 MHz and DR 8.

As shown in Figure , all timings are taken from the end of the uplink transmission. This is the **EVLORAMACTXDONE** event. At the designated interval, the RM191 turns on the receiver for a short period and listens for a sync message. If the signal is not detected, the receiver is switched off and the RM191 then waits for the second window to open and repeat the process.

If the sync message is detected in any of the receive windows, then the RM191 receiver remains on until the full downlink packet is received. In this way the RM191 saves power by only having the receiver switched on for the minimum amount of time.

**Note:** If the sync message is transmitted outside this initial small receive window, the downlink packet is missed. The LoRa gateway has no knowledge of this fact. As far as the gateway is concerned, the packet was sent. It is not retransmitted in the second window.

Again, the uplink packet option (confirmed or not confirmed) determines the RM191's response to this. If confirmed is selected, then the same uplink packet is resent, following the procedure outlined above.

Receive timings are hardcoded. There are no configuration options available to modify these timings. The RM191 timings meet those of the LoRaWAN specification. It is the gateway's responsibility to send the downlink packet at the correct time.

## LINK CHECK

A module can transmit a Link Check request to the server using the LORAMACLinkCheck API.

The Link Check response from the server contains an indication of the signal strength of the last Link Check request received by the server and the number of gateways that have received that request.

## CONTROLLING THE PERSONALISATION SEQUENCE NUMBER

The LoraWan specification states that, when a module has joined a network using the personalisation option, it must keep track of the uplink and downlink counter values over a reset. The reason for this is that a module is deemed to have "Joined" a network when the keys are loaded into both the module and the network server. There are no handshaking messages. So when the module is powered down it is still, in effect, joined to that network.

For both uplink and downlink packets the main requirement is that the counter value of an incoming packet must be less than 16384 greater than that of the previous received packet. If the value is greater than 16384 more than the previous packet the incoming packet is rejected. No indication is sent back to the transmitting device on why the incoming packet was rejected.

Also worth noting is that if the incoming packet counter value is less than that of the previous packet, for example either the module or server has been reset, this is likely to result in a negative difference which will probably equate to a large positive number greater than 16384 and so will also be rejected.

It is important to note that the incoming counter value does not necessarily have to be just one more than the previous value. Provided the difference is less than 16384, the incoming packet will be accepted.

To support this requirement the RM1xx now stores uplink and downlink counter values in the internal serial eeprom that will be used to initialise the internal counters on boot up, before the first uplink packet is transmitted. However, to ensure we're not writing to the serial eeprom every time we transmit and receive packets the module uses an incremental step to determine when the values need to be updated.

This incremental value defaults to 256 but it can be modified in Flash using the **at+cfg 1003** command or temporarily using the **LoramacSetOption(LORAMAC_OPT_SEQUENCE_INCREMENT, ..)** smartBASIC command. In both cases the value must lie between 1 and 256.

Looking at the uplink counter first and starting with the scenario where everything has been reset, the first uplink counter value will be 0. If the incremental step is set to the default value of 256, this means that the uplink counter stored in the serial eeprom will be 256.

Every time an uplink packet is transmitted the counter value of the uplink packet will be increased by 1 byte the stack. When it reaches 256 then the value in the serial eeprom will be incremented by 256 up to 512. On subsequent uplink packets the counter value will continue to increase by one every packet.

If, for whatever reason, the module then resets it will lose this running counter value. However when it reboots it will read in the value stored in the serial eeprom and so start counting from, following from the previous paragraph, 512. This, in turn, will cause the value in the serial eeprom to update to 768.

In the scenario where a module will only send a single packet before going to deep sleep (which requires a reboot to wake up from), it the incremental step is 256 then the first 4 uplink packet sequence numbers will be 0, 256, 512 and 768.

The downlink counter is handled slightly differently. It uses the same incremental step value but the value stored in the eeprom is only updated when the actual counter value is the incremental step greater than the stored value. So, again using the default value of 256, for downlink counter values of 0-255 the value stored in the eeprom will be 0. When the downlink counter reaches 256 the value in the eeprom is updated to 256.

So if the module resets and the actual downlink counter lies between 0 and 255, it will initialise with 0. If the actual value is greater than 256 to 511 it will be 256. This ensures that the local value is close enough to the next incoming value to not cause problems.

Also supporting these counter operations are the **LoraMacSetOption/LoramacGetOption** commands, **LORAMAC_OPT_EEPROM_UPCOUNTER** and **LORAMAC_OPT_EEPROM_DOWNCOUNTER**. These can be used to set or read the stored sequence number value stored in the serial eeprom. This has the effect of setting the sequence number to a value of your choosing after a reset.

The actual real values being passed between the module and the server can be retrieved using the **LORAMAC_OPT_DOWNLINK_COUNTER** and **LORAMAC_OPT_UPLINK_COUNTER** ids. These are read only commands.

## References

- Lora Alliance - LoRaWAN Specification – The current version is available from the LoRa Alliance website.
- LoRaWAN Regional Parameters – The current version is available from the LoRa Alliance website.
- User Guide – RM1xx Series *smart*BASIC Extensions (documentation tab of RM1xx product page)
- User Guide - *smart*BASIC Core Functionality (documentation tab of RM1xx product page)
- Application Note – Connecting to Multitech Conduit Gateway (documentation tab of RM1xx product page)
- http://www.multitech.net/developer/software/lora/conduit-mlinux-lora-communication/conduit-mlinux-lora-use-third-party-devices/

## Revision History

| Version | Date | Notes | Approver |
|---|---|---|---|
| 1.0 | 24 May 2016 | Initial Release | Tim Carney |
| 1.1 | 10 Oct 2016 | Updates to several ID values | Colin Anderson |
| 1.2 | 11 Oct 2017 | Updated for versions 101.6.1.0 & 111.6.1 | Jonathan Kaye |