AN1119: Using RAIL for Wireless M-Bus Applications with EFR32

This document describes the usage of Flex SDK for Wireless M-Bus development on EFR32™ Wireless Geckos. It includes the features and limitations, the usage of the radio configurator and supplied software components, and the description of the examples

KEY POINTS

- Features and limitations
- Usage of the configurator in M-Bus mode
- Usage of the plugin
- Description of the examples

Table of Contents

1. Overview

Wireless Gecko supports all Wireless M-Bus PHY configurations, according to EN13757-4 (both directions in all cases):

- Mode S
- Mode T
- Mode R2
- Mode C
- Mode N (submodes a, b, c, d, e, f, and g, or channels 1a/b, 2a/b, 3a/b, and 0)
- Mode F

In all modes, we provide support to basic data link layer features for both frame format A and frame format B

- Segmenting frames into blocks, and reassembling them to frame
- CRC calculation and checking
- Frame length decoding
- Data encoding and decoding (Manchester and 3 out of 6)
- Postamble transmission

The software is built on top of RAIL, therefore it inherits RAIL features that can be useful for implementing Wireless M-Bus, such as timestamping or scheduled reception and transmission.

The supplied example includes an easily reusable module, which provides some useful features, including mode 5 (AES-128 with dynamic initialization vector) security.

2. Limitations

2.1 T Mode Meter to Other Device

While the hardware supports 3 out of 6 coding, it can't generate the postamble required by this mode (this doesn't matter in receive mode). For T Mode Meter to Other transmission, we recommend using the supplied encoder function (`WMBUS_phy_software() functi on`), which calculates the CRC using the GPCRC peripheral, encodes the packet using a software based 3 out of 6 encoder, and adds the required postamble.

2.2 Modes Requiring Multiple PHY Configurations

The following modes require multiple PHYs:

- Mode T2 (bidirectional only)
- Mode C2 (bidirectional only)
- Mode N has a separate PHY for each bitrate (channel 1a-1b-3a-3b; 2a-2b and 0)
- Modes C, N, and F have separate PHYs for frame formats A and B

The current configurator cannot generate multiple PHYs into the same rail config.c. However, RAIL does support multiple PHYs, and it is possible to manually merge multiple PHYs.

2.3 Select Frame Type Based on Sync Word

Modes C, N, and F support frame formats A and B, where the sync word selects the frame format. Receiving frame A and frame B with the same configuration is currently not supported. You can either:

- Set up the frameA configuration and receive only frame format A.
- Set up the frameB configuration and receive only frame format B.
- Manually create a multi-PHY configuration and switch between the configurations runtime. However, you must decide which frame you expect before you switch to RX mode.
- Set up a custom C/N/F configuration with no frame coding option. In that case, you must decode the length and check CRC in software, but you can decide the frame type during reception as well.

Modes that support frame format B always have both sync words configured:

- FrameA and noFrame configurations have sync word 0 configured for frame type A and sync word 1 for type B.
- FrameB configurations have sync word 0 configured for frame type B and sync word 1 for frame type A.

You cannot receive frame type B with a frame A configuration or vice versa, but you can enable the second sync word in RAIL, and use the sync detect event to recognize that you are receiving something with the other frame type.

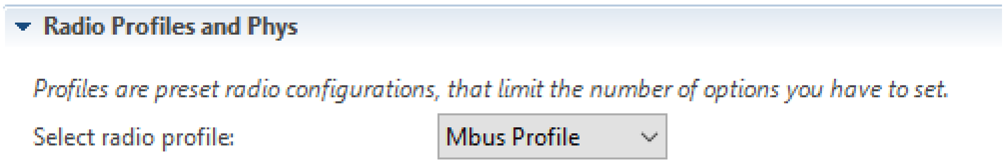
2.4 Decoder for Both C and T Mode Meter to Other

It is theoretically possible to receive mode C frames with a mode T configuration, because the first 10 bits of the sync word is the same: If the next 6 bits is a valid 3 out of 6 code, it is a T frame, and if the next 16 bits are the second part of mode C's sync word, it is a C frame. Since the second part of mode C's sync word is not a valid 3 out of 6 code, it is not possible that both conditions are true for a correctly formatted frame.

We don't provide a decoder that supports both C and T mode, but it's possible to write a software decoder using the `PHY wMbus ModeT C M2O 100k noFrame` configuration, which is a mode T configuration without a frame decoder. You must set the length during reception when using this configuration in RX mode

3. Using the Configurator

For Wireless M-Bus configurations, you should always use the Mbus Profile:



Radio Profiles and Phys

Profiles are preset radio configurations, that limit the number of options you have to set.

Select radio profile: Mbus Profile

Figure 3.1. Mbus Profile

3.1 Recommended Configurations

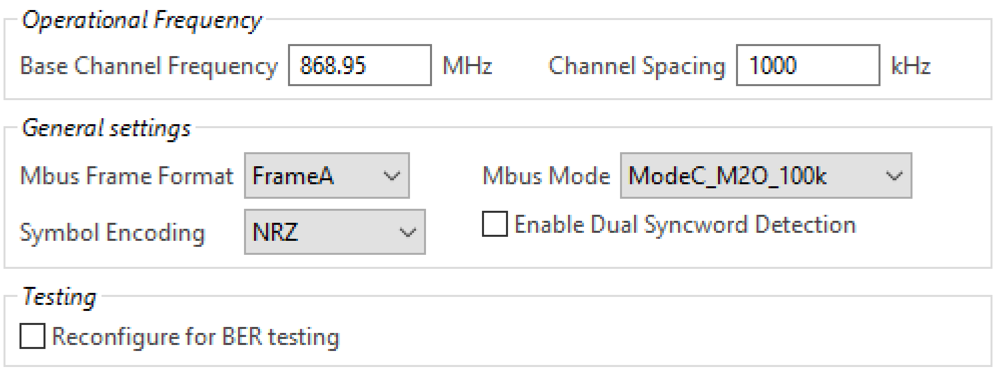
Mode	Submode	Configuration	Channel
Mode S	N/A	PHY wMbus ModeS 32p768k frameA	0
Mode T	Meter to Other, Rx	PHY wMbus ModeT M2O 100k no postamble frameA[1]	0
	Meter to Other, Tx	PHY wMbus ModeTC M2O 100k noFrame[2]	0
	Other to Meter	PHY wMbus ModeT O2M 32p768k frameA	0
Mode R2	N/A	PHY wMbus ModeR 4p8k frameA	0
Mode C	Meter to Other frame format A	PHY wMbus ModeC M2O 100k frameA[3]	0
	Other to Meter frame format A	PHY wMbus ModeC O2M 50k frameA[3]	0
	Meter to Other frame format B	PHY wMbus ModeC M2O 100k frameB[4]	0
	Other to Meter frame format B	PHY wMbus ModeC O2M 50k frameB[4]	0
Mode N	a (channel 1a), frame format A	PHY wMbus ModeNabef 4p8K frameA[3]	0
	b (channel 1b), frame format A	PHY wMbus ModeNabef 4p8K frameA[3]	1
	c (channel 2a), frame format A	PHY wMbus ModeNcd 2p4K frameA[3]	0
	d (channel 2b), frame format A	PHY wMbus ModeNcd 2p4K frameA[3]	1
	e (channel 3a), frame format A	PHY wMbus ModeNabef 4p8K frameA[3]	2
	f (channel 3b), frame format A	PHY wMbus ModeNabef 4p8K frameA[3]	3
	g (channel 3b), frame format A	PHY wMbus ModeN2g 19p2k frameA	0
	a (channel 1a), frame format B	PHY wMbus ModeNabef 4p8K frameB[4]	0
	b (channel 1b), frame format B	PHY wMbus ModeNabef 4p8K frameB[4]	1
	c (channel 2a), frame type B	PHY wMbus ModeNcd 2p4K frameB[4]	0
	d (channel 2b), frame type B	PHY wMbus ModeNcd 2p4K frameB[4]	1
	e (channel 3a), frame type B	PHY wMbus ModeNabef 4p8K frameB[4]	2
	f (channel 3b), frame type B	PHY wMbus ModeNabef 4p8K frameB[4]	3
	g (channel 3b), frame type B	PHY wMbus ModeN2g 19p2k frameB[4]	0
Mode F	Frame type A	PHY wMbus ModeF 2p4k frameA[3]	0
	Frame type B	PHY wMbus ModeF 2p4k frameB[4]	0

Note:

1. Could also work in Tx mode, but it does not generate the postamble required by the standard.
2. Should be used with the supplied software encoder (`WMBUS_phy_software() function`)
3. Sync word for frame format A is set up for sync word 0, sync word for frame format B set up as sync word 1, but not enabled by default. Receiving frame format B is not possible, but the sync detect event could be used.
4. Sync word for frame format B is set up for sync word 0, sync word for frame format A set up as sync word 1, but not enabled by default. Receiving frame format A is not possible, but the sync detect event could be used.

3.2 Using Custom Settings

All the above configurations (and more) can be set up using Custom settings.


Operational Frequency

Base Channel Frequency: 868.95 MHz

Channel Spacing: 1000 kHz

General settings

Mbus Frame Format: FrameA

Mbus Mode: ModeC_M2O_100k

Symbol Encoding: NRZ

☐ Enable Dual Syncword Detection

Testing

☐ Reconfigure for BER testing

Figure 3.2. Custom Settings

Base Channel Frequency

Sets the base channel frequency/operational frequency.

Channel Spacing

Sets the channel spacing frequency. This setting is only used in Na, Nb, Nc, Nd, Ne, and Nf modes, since the other modes always use channel 0.

Mbus Frame Format

Sets the frame format to use:

- *FrameA*: Frame format A (10B first block, 16B further blocks, CRC after each block, length does not include CRCs)
- *FrameB*: Frame format B (10B first block, max 115B second block, optional block. CRC after second and optional block, length includes CRCs)
- *NoFormat*: No frame decoder; fixed length mode. Length must be set with `RAIL_SetFixedLength()` before packet Tx and during packet Rx (e.g. using `RAIL_PeekRxPacket` to check the length field when it's received)

Mbus Mode

Sets the mode (modulation, deviation, bit rate, etc). Possibilities are:

- ModeS_32p768k
- ModeT_M2O_100k
- ModeT_O2M_32p768k
- ModeR_4p8k
- ModeC_M2O_100k
- ModeC_O2M_50k
- ModeN1a_4p8K
- ModeN1c_2p4K
- ModeNg

Symbol Encoding

Sets the symbol coding:

- *NRZ* – No symbol coding
- *Manchester* – Manchester (zero is “01”). Also adds 2 bits of “01” postamble for Tx packets
- *3 of 6* – 3 out of 6 coding. Not recommended for Tx as it does not send postamble.

Enable Dual Syncword Detection

Enables second sync word for ModeC, ModeN, and ModeF

Reconfigure for BER Testing

Test mode for BER testing. See application note *AN971: EFR32 Radio Configurator Guide* for details.

4. The WMBus Plugin

The plugin provides the following features:

- Processes the packet before sending it, if needed. Currently needed for mode T meter to other only.
- Helper functions to get the timing in limited access mode
- Helper functions to fill the payload with EN13757-3 compatible payload
- Helper functions to encode/decode manufacturer field
- Helper functions for cryptography using mbedTLS. Currently only supports mode5 (AES128-CBC with dynamic init vector)
- Helper types for data link and transport layer header

For detailed documentation, see `wmbus.h`, which is documented via doxygen style.

5. Example Application

The provided example application pair implements a very basic meter-collector interaction.

Meter

The meter periodically sends synchronous SND-NR messages with some hardcoded value with mode 5 security. While waiting, it sleeps in EM2 in idle or EM1 if the main oscillator is required for scheduling or Rx mode. By default, the meter is configured for limited access mode (short receive window after transmission), but it doesn't handle any received packets, it is just implemented to demonstrate the scheduling required for an application.

Please make sure to update the following line for the mode you configured in the radio configurator:

```
static const WMBUS_Mode_t mode = WMBUS_MODE_S;
```

Collector

The collector prints the received packet on serial terminal, with some information (like the first block) detailed. If the packet is EN13757-3 compatible with the short header, it also decodes mode5 security if required.

Once the meter and collector is running, the collector should print messages like this to the serial terminal:

```
RX:[Time:163580960]
Block-1:[L:30,C:0x44,M:SIL,ID:00000001,Version:0x01,devType:0x07]
AppHeader:[CI:0x7A,AccessNr:60,Status:0x00,encMode:5,Accessibility:02,encBlocks:1,sync:1]
[0x2F 0x2F 0x04 0x13 0x39 0x30 0x00 0x00 | 0x02 0x3B 0x7B 0x00 0x2F 0x2F 0x2F 0x2F]
```

A Wireless M-Bus sniffer can also be used. In that case, the default crypto key used by the application is: `00112233445566778899AABBCCDDEEFF`.

For unencrypted packets, RAILTest also can be used as a sniffer with the right meter to another device to receive configuration.

6. Manually Merging Configurations

To merge multiple configurations (e.g., for bidirectional T mode), create applications with the configurations you want to use. In this description, we lead you through the example of bidirectional T mode meter, which needs `PHY wMbus ModeTC M2O 100k noFrame` for transmit, and `PHY wMbus ModeT O2M 32p768k frameA` for receive.

RAIL 2.0 can use different configurations for different channels that can be used here.

By default, the configurator prefixes everything with “generated_”, so it is a good idea to rename these to something more understandable, such as transmit and receive; however, do not rename `generated_channels` and `generatedChannelConfig`, because we only need one of them.

First copy the variables for both configurations into one `rail_config.c` file. You should end up with the following:

- `transmit_irCalConfig`
- `receive_irCalConfig`
- `transmit_phyInfo`
- `receive_phyInfo`
- `transmit`
- `receive`
- `transmit_entryAttr`
- `receive_entryAttr`

Also, make sure that you include `rail_config.h` and `em_common.h`

Next, we should create a `RAIL_ChannelConfig_t` and a `channelConfigs` array, which should look like this:

```
const RAIL_ChannelConfig_t generated_channelConfig = {
  NULL,
  NULL,
  generated_channels,  2
};
const RAIL_ChannelConfig_t *channelConfigs[] = {
  &generated_channelConfig,
  NULL
};
```

The only difference we made is that `generated_channels` should now have two entries. So let us do that:

```
const RAIL_ChannelConfigEntry_t generated_channels[] = {
  {
    .phyConfigDeltaAdd = transmit,
    .baseFrequency = 868950000,
    .channelSpacing = 0,
    .physicalChannelOffset = 0,
    .channelNumberStart = 0,
    .channelNumberEnd = 0,
    .maxPower = RAIL_TX_POWER_MAX,
    .attr = &transmit_entryAttr
  },
  {
    .phyConfigDeltaAdd = receive,
    .baseFrequency = 868300000,
    .channelSpacing = 0,
    .physicalChannelOffset = 0,
    .channelNumberStart = 1,
    .channelNumberEnd = 1,
    .maxPower = RAIL_TX_POWER_MAX,
    .attr = &receive_entryAttr
  },
};
```

The difference from the original channel arrays is that the channel spacing is set to 0, and the `channelNumberStart/End` fields are set to 0 for transmit and 1 for receive.

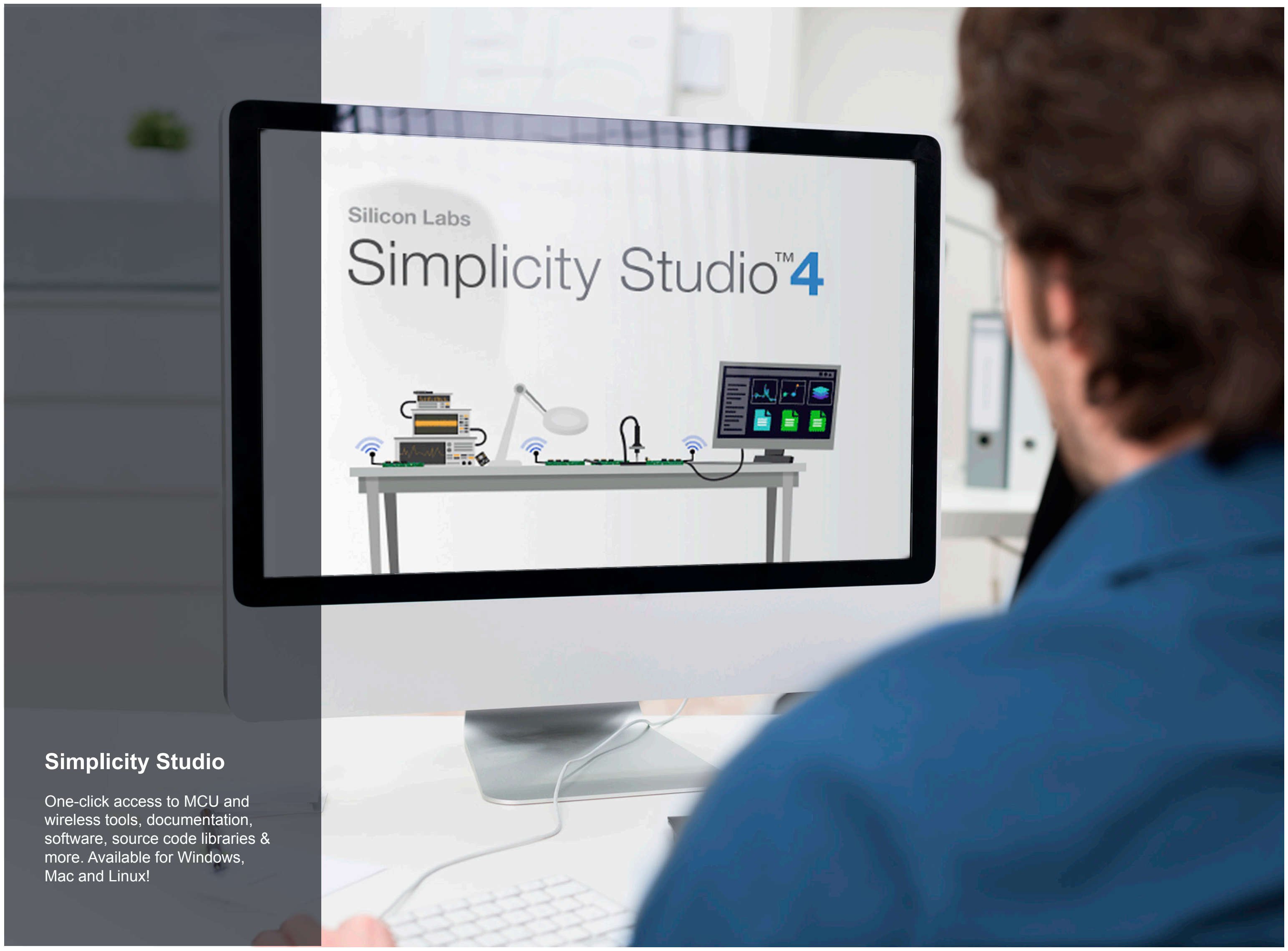
The `ChannelConfigEntry` above would set up channel 0 as the transmit channel and channel 1 as the receive channel. In mode N, it makes sense to use multiple channels with the same configuration, but otherwise, the channel spacing does not matter, and one channel per configuration is enough.

The result should be something like this:

```
#include "em_common.h"
#include "rail_config.h"

const uint8_t transmit_irCalConfig[] = {…};
const uint8_t receive_irCalConfig[] = {…};

const uint32_t transmit_phyInfo[] = {
  1UL,
  0x00200000, // 32.0
  (uint32_t) NULL,
  (uint32_t) transmit_irCalConfig,
#ifdef RADIO_CONFIG_ENABLE_TIMING
  (uint32_t) &generated_timing,
#else
  (uint32_t) NULL,
#endif
};
const uint32_t receive_phyInfo[] = {
  1UL,
  0x001B6DB6, // 27.4285714286
  (uint32_t) NULL,
  (uint32_t) receive_irCalConfig,
#ifdef RADIO_CONFIG_ENABLE_TIMING
  (uint32_t) &generated_timing,
#else
  (uint32_t) NULL,
#endif
};


const uint32_t transmit[] = {
  0x01040FF0UL, (uint32_t) transmit_phyInfo,
  …
};
const uint32_t receive[] = {
  0x01040FF0UL, (uint32_t) receive_phyInfo,
  …
};


RAIL_ChannelConfigEntryAttr_t transmit_entryAttr = {…};
RAIL_ChannelConfigEntryAttr_t receive_entryAttr = {…};

const RAIL_ChannelConfigEntry_t generated_channels [] = {
  {
    .phyConfigDeltaAdd = transmit,
    .baseFrequency = 868950000,
    .channelSpacing = 0,
    .physicalChannelOffset = 0,
    .channelNumberStart = 0,
    .channelNumberEnd = 0,
    .maxPower = RAIL_TX_POWER_MAX,
    .attr = &transmit_entryAttr
  },
  {
    .phyConfigDeltaAdd = receive,
    .baseFrequency = 868300000,
    .channelSpacing = 0,
    .physicalChannelOffset = 0,
    .channelNumberStart = 1,
    .channelNumberEnd = 1,
    .maxPower = RAIL_TX_POWER_MAX,
    .attr = &receive_entryAttr
  },
};
const RAIL_ChannelConfig_t generated_channelConfig = {
  NULL,
  NULL,
  generated_channels,
  2
};
const RAIL_ChannelConfig_t *channelConfigs[] = {
  &generated_channelConfig,
  NULL
};
```

With this configuration loaded, channel 0 can be used for transmission, and channel 1 for reception. The configuration change will be automatically applied by RAIL.

RAIL also enables loading only the configuration difference at channel change. In that case, transmit and receive should be the difference for each mode, while the common part could be loaded with the first parameter of `RAIL_ChannelConfig_t`. However, this configuration is out of scope of this document.



Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701