UM2726

User manual

BlueNRG-LP, BlueNRG-LPS, and STM32WB0 series 2.4 GHz radio proprietary drivers

Introduction

This document describes the BlueNRG-LP, BlueNRG-LPS, and STM32WB series 2.4 GHz radio proprietary low-level driver, which provides access to BlueNRG-LP, BlueNRG-LPS, and STM32WB0 series devices in order to send and receive packets without using the Bluetooth link layer. An application using a central data structure and APIs can control different features of packets such as: interval, channel frequency, data length and so on.

Note: The document content is valid for the BlueNRG-LP, BlueNRG-LPS, and STM32WB0 series devices. Any reference to the BlueNRG-LP device and platform is also valid for BlueNRG-LPS.

1 BlueNRG-LP, BlueNRG-LPS, and STM32WB0 series radio operation

The BlueNRG-LP, BlueNRG-LPS, and STM32WB0 series 2.4 GHz radio low level driver interface controls 2.4 GHz radio. Furthermore, it interacts with the wake-up timer, which runs on the slow 32 kHz clock, the RAM memory and the processor.

RAM is used to store radio settings, the current radio status, the data received and data to be transmitted. The radio low level driver can manage up to 8 different radio configurations also called state machines.

Several features are autonomously managed by the radio, without intervention of the processor:

- Packet encryption
- Communication timing
- Sleep management

A number of additional features are present and they are specifically related to the Bluetooth low energy standard like the Bluetooth channel usage.

The STM32WB0 devices are Arm® Cortex® core-based microcontrollers.

For more information on Bluetooth®, refer to http://www.bluetooth.com.

Note: Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

2 Data packet format

Only one packet format is used in the BlueNRG-LP, BlueNRG-LPS, and STM32WB0 series. It is shown below.

Figure 1. Packet format

	Preamble	NetworkID	Header	Length	Data	CRC
BlueNRG-LP, BlueNRG-LPS, STM32WB0	1 byte	4 bytes	1 byte	1 byte	0 - 255 bytes	3 bytes

A packet consists of six fields which, only four are user-accessible:

- By default, the preamble is 1-byte long. But, the user can define how many times to repeat the preamble through `RADIO_SetPreambleRep()`.
- NetworkID is the address of the device, expressed in 4 bytes. The receiver device accepts only those packets whose NetworkID field is the same as the one in its own address. The NetworkID should satisfy the following rules:
  - It has no more than 6 consecutive zeros or ones
  - It has not all 4 octets equal
  - It has no more than 24 transitions
  - It has a minimum of 2 transitions in the most significant 6 bits

  The NetworkID field is user-accessible through API `RADIO_SetTxAttributes()` or API `HAL_RADIO_SetNetworkID()`.
- Header can accept any values and its length is 1 byte. It can be used as a byte of data, but no encryption is applied to this field.
- Length represents the length of the data field. The user sets this value for a packet to transmit or reads this value from a received packet.
  The maximum number of bytes of the payload (with or without encryption) that the BlueNRG-LP, BlueNRG-LPS, or STM32WB0 series devices link layer can accept in reception is 255. The user can set the value of this threshold (from 0 to 255) at hardware level through API `RADIO_SetMaxReceivedLength()`.

Note: Refer to Section 3.2: API architecture for the APIs list and the naming convention used in the context of STSW-BNRGLP-DK SW package and STM32CubeWB0 SW package.

Then, the maximum value of the length field is 255 for the BlueNRG-LP, BlueNRG-LPS, or STM32WB0 series devices with some exceptions. If the encryption is enabled, at the maximum length of the data field, it must subtract 4 bytes. These 4 bytes are reserved for the MIC field added to the packet as shown in Figure 2. Packet format with encryption enabled. Packet format with encryption enabled. The table below contains a summary about the length field.

Table 1. Values in bytes for the length field

Product	Data channels	Data channels with encryption	Advertising channels	Advertising channels with encryption
BlueNRG-LP, BlueNRG-LPS, STM32WB0 series	255	251	255	251

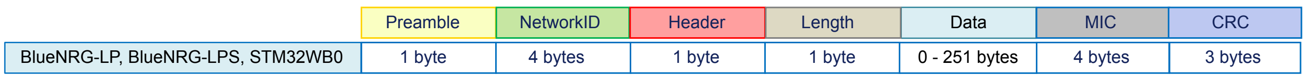
To avoid memory corruption due to bad length field received (in packet where the CRC check fails), the user must reserve the maximum memory for packet received that includes 2 bytes of header field as well as the data field

- Data can accept any value and its length is decided by the length field. The user defines a memory buffer in order to set the header field, the length field and data field as follows:

```
PacketBuffer[0] = 0x01; // Header field
PacketBuffer[1] = 5; // Length field
PacketBuffer[2] = 0x02; // Data byte 1
PacketBuffer[3] = 0x03; // Data byte 2
PacketBuffer[4] = 0x04; // Data byte 3
PacketBuffer[5] = 0x05; // Data byte 4
PacketBuffer[6] = 0x06; // Data byte 5
```

  If the encryption is enabled, only the data field is encrypted. The other fields including the header field and the length field are not encrypted.
- The CRC is used to identify corrupted packets. Its length is 3 bytes and the radio generates and checks it during transmission and reception respectively. The user can configure the initial value for the CRC calculation, except in the advertising channels where the initial value is set to 0x555555. The CRC hardware feature can be disabled. It means that the hardware neither appends the CRC in transmission nor checks it during reception.

Figure 2. **Packet format with encryption enabled**



	Preamble	NetworkID	Header	Length	Data	MIC	CRC
BlueNRG-LP, BlueNRG-LPS, STM32WB0	1 byte	4 bytes	1 byte	1 byte	0 - 251 bytes	4 bytes	3 bytes

3 Radio low level driver framework

3.1 Description

The radio low level driver consists of the following files:

- BlueNRG-LP, BlueNRG-LPS STSW-BNRGLP-DK software package:
  - *rf_driver_ll_radio_2g4.h*
  - *rf_driver_ll_radio_2g4.c*
  - *rf_driver_hal_radio_2g4.h*
  - *rf_driver_hal_radio_2g4.c*
- STM32CubeWB0 software package:
  - *stm32wb0x_ll_radio.h*
  - *stm32wb0x_hal_radio.h*
  - *stm32wb0x_hal_radio.c*

3.2 API architecture

The radio low level driver interface provides a set of APIs which allows the following functions to be addressed :

- Radio initialization
- Encryption
- Set receiver and transmitter Phy (1 Mbit/s, 2 Mbit/s, S = 2, S = 8)
- Communication channel management
- Set the network ID, CRC initial value, power level
- Set the maximum received packet length and the receive timeout
- Test commands (tone)

In the context of BlueNRG-LP, BlueNRG-LPS SW package (STSW-BNRGLP-DK), the list of APIs managing these settings are:

- `RADIO_Init()`
- `RADIO_SetEncryptionCount()`
- `RADIO_SetEncryptionAttributes()`
- `RADIO_SetEncryptFlags()`
- `RADIO_EncryptPlainData()`
- `RADIO_Set_ChannelMap()`
- `RADIO_SetChannel()`
- `RADIO_SetTxAttributes()`
- `RADIO_SetBackToBackTime()`
- `RADIO_SetTxPower()`
- `RADIO_SetReservedArea()`
- `RADIO_MakeActionPacketPending()`
- `RADIO_SetPhy()`
- `RADIO_SetMaxReceivedLength()`
- `RADIO_SetPreambleRep()`
- `RADIO_SetDefaultPreambleLen()`
- `RADIO_DisableCRC()`
- `RADIO_StopActivity()`
- `RADIO_StartTone()`
- `RADIO_StopTone()`

Note: *In the context of STM32CubeWB0 SW package, an equivalent list of APIs with prefix HAL_ is available.*

Most of the APIs modify the parameters of the state machine passed as parameter. On the other hand, some parameters are global, that is they are valid for all the state machines. One of which is the receive timeout that is set calling `RADIO_SetGlobalReceiveTimeout()` in the context of the STSW-BNRGLP-DK SW package, and `HAL_RADIO_SetGlobalReceiveTimeout()` in the context of the STM32CubeWB0 SW package.

This value sets the duration of the receive window in microseconds.

The radio low level driver uses a central data structure that consists of a linked list of ActionPackets. An ActionPacket is a structure (C language) that, in conjunction with the APIs above, defines a complete operation of transmission or reception. It also includes a number of callbacks, which allow the user to define a chain of actions.

The ActionPacket is composed of input fields used to configure the action and output fields holding information on the action once it has been executed. The table below contains the information on these fields.

Table 2. **ActionPacket structure**

Parameter name	Input/output	Summary
`StateMachineNo`	In	This parameter indicates the state machine number for this action. From 0 to 7.
`ActionTag`	In	The configuration of the current action. Details of the flags in the `ActionTag` table
`MaxReceiveLength`	In	Set the maximum number of bytes that the link controller accepts in reception. It is between 0 and 255 bytes.
`WakeupTime`	In	Contains the wake-up time in microseconds if it is relative. If it is absolute, the time is expressed in system time units (STU). More about STU can be found in the BlueNRG-LP, BlueNRG-LPS, and STM32WB0 series devices timer module application note
`*next_true`	In	Pointer to next `ActionPacket` structure if `condRoutine()` returns `TRUE`
`*next_false`	In	Pointer to next `ActionPacket` if `condRoutine()` returns `FALSE`

`(*condRoutine)`
`(ActionPacket*)`
	In	User callback necessary to decide the next action in a linked list of ActionPackets. The routine is time critical and it must end within 45 µs.

`(*dataRoutine)`
`(ActionPacket*, ActionPacket*)`
	In	User callback to manage data.

`*data`	In/out	Pointer to the array with the data to send (header, length and data field), for TX. Pointer to the array where the data received are copied, for RX. In case of RX, the array must have the max. size as explained in Section 2: Data packet format
`timestamp_receive`	Out	This field contains the timestamp when a packet is received. It is intended to be used in the `dataRoutine()` callback routine. RX only. It is expressed in STU. One STU is 625/256 microseconds.
`status`	Out	The status register with the information on the action.
`rssi`	Out	The RSSI of the packet was received with. RX only.

The ActionTag is a bitmask used to enable different features of the radio, used by the ActionPacket. The table below explains these parameters.

Table 3. **ActionTag field description**

Bit	Name	Description
7	`INC_CHAN`	This bit activates automatic channel increment. The API `RADIO_SetChannel()` [1][2] sets the value of the increment. 0: no increment 1: automatic increment
6	`TIMESTAMP_POSITION`	This bit sets where the position of the time stamp is taken, at the beginning of the packet or at the end of it. 0: end of the packet, when the demodulator receives the last bit of the packet received or when the last transmitted bit has been shifted out from the transmit block. 1: beginning of the packet, when the demodulator detects the preamble + access address. Rx only.
5	`RELATIVE`	It determines if the `WakeupTime` field of the `ActionPacket` is considered as absolute time or relative time to the current. 0: absolute 1: relative
4	`WHITENING_DISABLE`	This bit determines whether whitening is disabled or not 0: whitening enabled 1: whitening disabled
3	`RESERVED`	Reserved
2	`TIMER_WAKEUP`	In the Radio handler, this bit determines if the action (RX or TX) is going to be executed based on the back-to-back time or based on the value of `WakeupTime`. If it is the first action, this bit is ignored since it is going to be executed always based on the value of `WakeupTime`. 0: based on the back-to-back time (default 150 µs). 1: based on the value of `WakeupTime`
1	`TXRX`	This bit determines if the action is an RX action or a TX action. 1: TX action 0: RX action
0	`PLL_TRIG`	This bit activates the radio frequency PLL calibration. 0: radio frequency calibration disabled. 1: radio frequency calibration enabled. The user should set this in the first action.

1. *In the advertising channels, the frequency hopping is limited to 1 hop.*
2. `RADIO_SetChannel()` *in the context of STSW-BNRGLP-DK SW package, and* `HAL_RADIO_SetChannel()` *in the context of STM32CubeWB0 SW package.*

The bits of the status field of the ActionPacket represent the map of the interrupts triggered by the last radio action. A description of the status field of the ActionPacket is reported below. For full details, refer to the:

- BlueNRG-LP radio controller reference manual (RM0480)
- BlueNRG-LPS radio IP reference manual (RM0498)
- STM32WB09xE reference manual (RM0505)
- STM32WB07xC, STM32WB06xC reference manual (RM0530)
- STM32WB05xZ reference manual (RM0529)

Table 4. **Status_table**

Bit name	Bit position	Description
RCVOK	31	Receive data without errors.
RCVCRCERR	30	Receive data fail (CRC error). This error is raised only if at least preamble and access address have been detected.
TIMECAPTURETRIG	29	Time captured in the Time Capture register.
RCVCMD	28	Received command.
RCVNOMD	27	Received MD bit embedded in the PDU data packet header was zero.
RCVTIMEOUT	26	Receive timeout (no preamble found).
DONE	25	Receive/transmit done.
TXOK	24	Previous transmitted packet received OK by the peer device.
RCVLENGTHERROR	18	The received payload length exceed the maximum.
PREVTRANSMIT	6	Previous event was a transmission (1) or reception (0).

4 How to write an application

There are two ways to write an application: the former is based on the usage of four APIs without ActionPacket, and the latter based on the use of the ActionPacket data structure.

4.1 No ActionPacket approach

The simplest way is to use a set of APIs provided in HAL radio driver, that allows the radio to be configured to fulfill the actions below:

- send a packet
- send a packet and then wait for the reception of a packet (ACK)
- wait for a packet
- wait for a packet and if the packet is received, a packet is sent back (ACK)

In this contest, the user does not need to use the ActionPacket to configure the operations of the radio, but a pointer to a user callback is requested, which handles different information according to the executed action:

- TX action: IRQ status
- RX action: IRQ status, RSSI, timestamp and data received

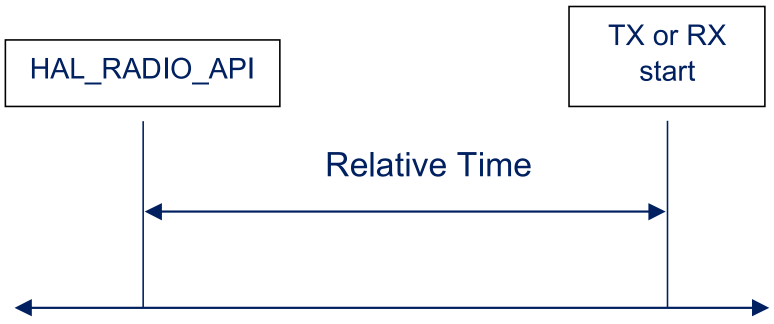
The user callback is called in interrupt mode, in particular in the `BLE_TX_RX_IRQHandler()` in the context of the STSW-BNRGLP-DK SW package, and `RADIO_TXRX_IRQHandler()` in the context of the STM32CubeWB0 SW package.

that has the maximum priority.

The second parameter of each API is a relative time in microseconds that represents when the next radio activity starts from the moment in which the API is called. This delay must be big enough as otherwise it is not possible to program the radio timer and an error code is returned.

The user can choose the desired time without taking into account the time that the radio uses for its setup. Then, the delay that is passed to the API, represents when the first bit is transmitted or the receive window starts.

Figure 3. **Relative time**



4.1.1 TX example with no ActionPacket (STSW-BNRGLP-DK SW package)

In the example below, the radio is programmed to send a packet periodically, with a time between two consecutive packets of 10 ms. Each packet contains 3 bytes of data.

Note: The radio initialization is followed by the timer module initialization.

The `ActionPacket` structure is not used directly in this example, but it is used through the APIs of the HAL driver.

```
uint8_t send_packet_flag = 1;
uint8_t packet[5];
int main(void)
{
   HAL_VTIMER_InitType VTIMER_InitStruct = {HS_STARTUP_TIME, INITIAL_CALIBRATION, CALIBRATION_INTERVAL}
;
   /* System initialization function */
   if (SystemInit(SYSCLK_64M, BLE_SYSCLK_32M) != SUCCESS) {
      /* Error during system clock configuration take appropriate action */
      while(1);
   }
   /* Radio configuration*/
   RADIO_Init();
   /* Timer Init */
   HAL_VTIMER_Init(&VTIMER_InitStruct);
   /* Set the Network ID */
   HAL_RADIO_SetNetworkID(0x88DF88DF);
   /* Set the RF output power at max level */
   RADIO_SetTxPower(MAX_OUTPUT_RF_POWER);
   packet[0] = 1; /* Header field */
   packet[1] = 3; /* Length field */
   packet[2] = 2; /* Data */
   packet[3] = 3;
   packet[4] = 4;
   while(1) {
      HAL_VTIMER_Tick();
      /* If ready, a new action is scheduled */
      if(send_packet_flag == 1) {
         send_packet_flag = 0;
         /* Schedule the action with the parameter - channel, wakeupTime, data,
         dataCallback */
         HAL_RADIO_SendPacket(22, 10000, packet, TxCallback );
      }
  }
  return 0;
}
void BLE_TX_RX_IRQHandler(void)
{
   RADIO_IRQHandler();
}
```

The user callback, `TxCallback()`, is defined in order to reschedule another send packet action as follows:

```
uint8_t TxCallback(ActionPacket* p, ActionPacket* next)
{
  /* Check if the TX action is ended */
  if( p ->status & BLUE_STATUSREG_PREVTRANSMIT) != 0) {
   /* Triggers the next transmission */
   send_packet_flag = 1;
  }
  return TRUE;
}
```

4.1.2 TX example with no ActionPacket (STM32CubeWB0 SW package)

In the example below the radio is programmed to send a packet periodically, with a time between two consecutive packets of 10 ms. Each packet contains 3 bytes of data.

The HAL-driver APIs require neither an `ActionPacket` structure, nor a pointer to a user callback, because a set of default callbacks is defined.

```
HAL_RADIO_CallbackTxDone(void)
{
  send_packet_flag = 1;
}
```

`HAL_RADIO_CallbackTxDone()` is called when a transmission is successfully completed. To activate this callback, the user needs to pass `HAL_RADIO_Callback()` as an argument of `HAL_RADIO_SendPacket()`.

```
uint8_t packet[7];
uint8_t DataLen = 5;
uint8_t send_packet_flag = 1;
uint16_t data_val = 1;

int main(void)
{
  MX_RADIO_Init();
  MX_RADIO_TIMER_Init();

  /* Build packet */
  packet[0] = 0x02;      /* Header */
  packet[1] = DataLen;         /* Length */

  /* Channel map configuration */
  uint8_t map[5]= {0xFF,0xFF,0xFF,0xFF,0xFF};
  HAL_RADIO_SetChannelMap(0, &map[0]);

  /* Setting of channel and the channel increment*/
  HAL_RADIO_SetChannel(0, 22, 0);

  /* Sets of the NetworkID and the CRC.*/
  HAL_RADIO_SetTxAttributes(0, 0x88DF88DF, 0x555555);

  /* Configures the transmit power level */
  HAL_RADIO_SetTxPower(MAX_OUTPUT_RF_POWER);

  /* Infinite loop */
  while (1)
  {
    HAL_RADIO_TIMER_Tick();
    if(send_packet_flag == 1)
    {
      send_packet_flag = 0;
      for(uint8_t i = 0; i < (DataLen-2); i++)
     {
       packet[i+2] = i + data_val;
     }
     data_val++;
      HAL_RADIO_SendPacket(22, 10000, packet, HAL_RADIO_Callback );
    }
  }
}
```

4.1.3 RX example with no ActionPacket (STSW-BNRGLP-DK SW package)

In the example below, the radio is programmed to go to RX state periodically. The delay between each RX operation is 9 ms. This ensures that after the first good reception, the RX device wakes up always 1 ms before the TX device (configured in Section 4.1.1: TX example with no ActionPacket (STSW-BNRGLP-DK SW package)) starts to send the packet (guard time).

Note: *The maximum length of the payload is set to 255 bytes.*

The RX timeout is 20 ms. This value is big enough to ensure that at least one packet should be received.

The ActionPacket structure is not used directly in this example, but it is used through the APIs of the HAL driver.

```
uint8_t rx_flag = 1;
uint8_t packet[MAX_PACKET_LENGTH];

int main(void)
{
   HAL_VTIMER_InitType VTIMER_InitStruct = {HS_STARTUP_TIME, INITIAL_CALIBRATION, CALIBRATION
_INTERVAL};
   /* System initialization function */
   if (SystemInit(SYSCLK_64M, BLE_SYSCLK_32M) != SUCCESS) {
      /* Error during system clock configuration take appropriate action */
      while(1);
   }
   /* Radio configuration*/
   RADIO_Init();
   /* Timer Init */
   HAL_VTIMER_Init(&VTIMER_InitStruct);
   /* Set the Network ID */
   HAL_RADIO_SetNetworkID(0x88DF88DF);

   while(1) {
      HAL_VTIMER_Tick();
      /* If ready, a new action is scheduled */
      if(rx_flag == 1) {
         rx_flag = 0;

         /* Schedule the action with the parameter - channel, wakeupTime,
         receive buffer, receive timeout, max received length, dataCallback */
         HAL_RADIO_ReceivePacket ( 22, 9000, packet, 20000, 255, RxCallback );
     }
   }
   return 0;
}
void BLE_TX_RX_IRQHandler(void)
{
   RADIO_IRQHandler();
}
```

The user callback, RxCallback(), is defined in order to re-schedule another reception and retrieve the information if a packet has been received as follows:

```
uint8_t RxCallback(ActionPacket* p, ActionPacket* next)
{
  /* Check if the RX action is ended */
  if( (p->status & BLUE_STATUSREG_PREVTRANSMIT) == 0) {
    /* Check if a packet with a valid CRC is received */
    if((p->status & BLUE_INTERRUPT1REG_RCVOK) != 0) {
      /* Retrieve the information from the packet */
      // p->data contains the data received: header field | length field | data field
      // p->rssi
      // p->timestamp_receive
    }
    /* Check if a RX timeout occurred */
    else if( (p->status & BLUE_INTERRUPT1REG_RCVTIMEOUT) != 0) {
    }
    /* Check if a CRC error occurred */
    else if ((p->status & BLUE_INTERRUPT1REG_RCVCRCERR) != 0) {
    }
  }
  /* Triggers the next reception */
  rx_flag = 1;
  return TRUE;
}
```

4.1.4 RX example with no ActionPacket (STM32CubeWB0 SW package)

In the example below, the radio is programmed to go to RX state periodically. The delay between each RX operation is 9 ms. This ensures that after the first good reception, the RX device wakes up always 1 ms before the TX device (configured in TX example) starts to send the packet (guard time).

```
uint8_t rx_flag == 1;

int main(void)
{
  uint8_t packet[MAX_PACKET_LENGTH];

  MX_RADIO_Init();
  MX_RADIO_TIMER_Init();

  /* Set the Network ID */
  HAL_RADIO_SetNetworkID(0x88DF88DF);

  uint8_t map[] = CFG_RF_CHANNEL_MAP;
  HAL_RADIO_SetChannelMap(0, map);

  HAL_RADIO_SetChannel(0, channel, 0);

  /* Infinite loop */
  while (1)
  {

    HAL_RADIO_TIMER_Tick();
    if(rx_flag == 1)
    {
      HAL_RADIO_ReceivePacket(22, 9000, packet, 20000, MAX_PACKET_LENGTH, HAL_RADIO_Callback)
;
    }
  }

}
```

To manage RX packets, the user can implement a set of four standard callbacks.

The structure, `RxStats`, contains information about the RSSI and timestamp of the received packet.

```
void HAL_RADIO_CallbackRcvOk(RxStats_t* rxDataStats)
{
   rx_flag = 1;
}

void HAL_RADIO_CallbackRcvTimeout(RxStats_t* rxDataStats)
{
   rx_flag = 1;
}

void HAL_RADIO_CallbackRcvError(RxStats_t* rxDataStats)
{
   rx_flag = 0;
}

void HAL_RADIO_CallbackRcvEncryptErr(RxStats_t *rxDataStats)
{
   rx_flag = 1;
}
```

4.2 ActionPacket approach

The most flexible way is to declare a number of ActionPackets, according to the actions that must be taken by the radio. Then the user fills these structures with the description of the operations to execute. For each ActionPacket, the API `RADIO_SetReservedArea()` must be called in order to initialize the information of the ActionPacket itself.

To start the execution of an ActionPacket, the API `RADIO_MakeActionPacketPending()` has to be called. After this, the application could:

- Make sure that another ActionPacket is called, by linking the ActionPackets together and then decide which ActionPacket executes within the condition routine.
- Reactivate the radio execution by calling again the API `RADIO_MakeActionPacketPending()`.

Note: Refer to Section 3.2: API architecture for the APIs list and the naming convention used in the context of STSW-BNRGLP-DK SW package and STM32CubeWB0 SW package.

All further actions are handled in interrupt mode as for the HAL layer approach, but in this case, the user handles two callback functions:

- Condition routine: `condRoutine()`
  It provides the result of the current ActionPacket, and it returns TRUE or FALSE. Depending on this, the next ActionPacket linked to the current one is selected from two possibilities:
  - `next_true ActionPacket1`
  - `next_false ActionPacket2`

  The purpose of this mechanism is to differentiate the next action of the scheduler. For example the condition routine in an RX action can decide:
  - To schedule the `next_true ActionPacket`, if the packet received is good (`ActionPacket1`)
  - To schedule the `next_false ActionPacket`, if the packet received is not good (`ActionPacket2`)
- Data routine: `dataRoutine()`
  It provides information as data received or transmitted, RSSI, timestamps and others. Besides, it is intended to modify the transmit data for the next packet based upon the last received data. This could be used to modify the packet data to be sent.

The goal of the multiple callbacks is to enable the user to access to the total performance of the radio, by avoiding time criticality bottlenecks. The goal is to bundle time critical aspects in the condition routine, and have the rest of the framework to be non-time-critical.

One benefit is that the framework forces the user to split code over smaller routines, which leads to more structured programming.

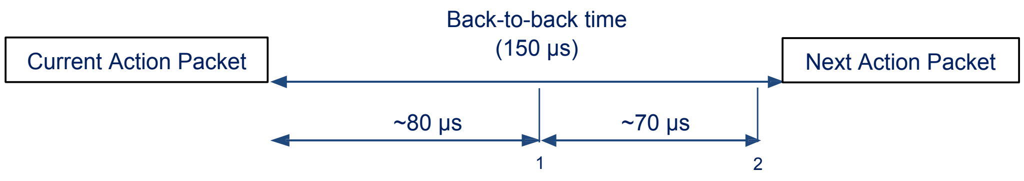
The most time critical operations are performed in the `condRoutine()`. Here the next activity is chosen according to some criteria and then patched on the fly. It is obvious that this must occur before the radio controller starts its operation, otherwise the patch is ignored.

The maximum execution time of the condition routine is about the back-to-back time less the time that the radio spends for the RF setup. For the moment, it can be considered that the radio in a back-to-back scenario needs 70 µs to set up. Then if the back-to-back time is 150 µs, the `condRoutine()` has about 80 µs to set the pointer to the next activity.

Therefore, the `dataRoutine()` has about the remaining time to modify, for example, the data to send.

The figure below summarizes the timing of the different callbacks.

Figure 4. Callback timing



1: Before this time, the condition routine must end.

2: Before this time, the updating of the next packet must be completed.

4.2.1 TX example with ActionPacket (STSW-BNRGLP-DK SW package)

Below is an example where the radio is programmed to send a packet periodically, with a time between two consecutive packets of 10 ms. Each packet contains 3 bytes of data. The `ActionPacket` structure is used to define this operation.

```
int8_t packet[5];
ActionPacket txAction;

int main(void)
{
  HAL_VTIMER_InitType VTIMER_InitStruct =
  {HS_STARTUP_TIME, INITIAL_CALIBRATION,
  CALIBRATION_INTERVAL};

  /* System initialization function */
  if (SystemInit(SYSCLK_64M, BLE_SYSCLK_32M)!=SUCCESS) {
     /* Error during system clock configuration take appropriate action */
     while(1);
  }
  /* Radio configuration*/
  RADIO_Init();
  /* Timer Init */
  HAL_VTIMER_Init(&VTIMER_InitStruct);
  /* Set the channel (22) and the channel increment (0) */
  RADIO_SetChannel(STATE_MACHINE_0, 22, 0);
  /* Sets of the Network ID and the CRC initial value */
  RADIO_SetTxAttributes(STATE_MACHINE_0,0x88DF88DF,    0x555555);
  /* Set the RF output power at max level */
  RADIO_SetTxPower(MAX_OUTPUT_RF_POWER);
  packet[0] = 1;   /* Header field */
  packet[1] = 3;   /* Length field */
  packet[2] = 2;   /* Data */
  packet[3] = 3;
  packet[4] = 4;
  /* Builds ActionPacket (txAction) for sending a
  single packet and schedules
  the next ActionPacket as itself (txAction) */
  txAction.StateMachineNo = STATE_MACHINE_0;
  /* Make a TX action with time relative to Wakeup
  Timer and enable the PLL calibration */
  txAction.ActionTag = RELATIVE | TIMER_WAKEUP | TXRX |
  PLL_TRIG;
  /* 10 ms before operation */
  txAction.WakeupTime = 10000;
  /* Pointer to the data to send */
  txAction.data = packet;
  /* Pointer to next ActionPacket: txAction */
  txAction.next_true = &txAction;
  /* Do nothing */
  txAction.next_false = NULL_0;
  /* Condition routine for selecting next ActionPacket*/
  txAction.condRoutine = conditionRoutine;
  /* Data routine called after conditionRoutine */
  txAction.dataRoutine = dataRoutine;
  /* Records the ActionPacket information */
  RADIO_SetReservedArea(&txAction);
  /* Execute the ActionPacket */
  RADIO_MakeActionPacketPending(&txAction);
  while(1) {
    HAL_VTIMER_Tick();
  }
  return 0;
}

void BLE_TX_RX_IRQHandler(void)
{
  RADIO_IRQHandler();
}
```

The condition callback triggers the execution of next scheduled `ActionPacket` (`txAction`), while the data callback is not used in this case, but must be defined anyway.

```
uint8_t conditionRoutine(ActionPacket* p)
{
  /* Check if the TX action is ended */
  if( p ->status & BLUE_STATUSREG_PREVTRANSMIT) != 0) {
  }
  /* The TRUE schedules the next_true action: txAction */
  return TRUE;
}

uint8_t dataRoutine(ActionPacket* p,  ActionPacket* next)
{
  return TRUE;
}
```

4.2.2 TX examples with ActionPacket (STM32CubeWB0 SW package)

In the example below, the radio is programmed to send a packet periodically, with a time between two consecutive packets of 10 ms. Each packet contains 3 bytes of data. The `ActionPacket` structure is used to define this operation.

```
uint8_t DataLen = 20;
uint16_t data_val = 0;
int main(void)
{
  MX_RADIO_Init();
  MX_RADIO_TIMER_Init();

  /* Build packet */
  packet[0] = 0x02;      /* Header */
  packet[1] = DataLen;   /* Length */

  for(uint8_t i = 0; i < (DataLen-15); i++) {
    packet[i+2] = i + data_val;
  }
  data_val++;

  /* Channel map configuration */
  uint8_t map[5]= {0xFF,0xFF,0xFF,0xFF,0xFF};
  HAL_RADIO_SetChannelMap(0, &map[0]);

  /* Setting of channel */
  HAL_RADIO_SetChannel(0, 22, 0);

  /* Sets of the NetworkID and the CRC.*/
  HAL_RADIO_SetTxAttributes(0, 0x88DF88DF, 0x555555);

  /* Configures the transmit power level */
  HAL_RADIO_SetTxPower(MAX_OUTPUT_RF_POWER);

  /* Build Action Packet */
  actPacket.StateMachineNo = 0;
  actPacket.ActionTag = RELATIVE | TIMER_WAKEUP | TXRX | PLL_TRIG;
  actPacket.WakeupTime = 10000;
  actPacket.MaxReceiveLength = 0;                    /* Not applied for TX */
  actPacket.data = packet;                                  /* Data to send */
  actPacket.next_true = &actPacket;                  /* Pointer to the next Action Packet*/
  actPacket.next_false = NULL_0;                     /* Null */
  actPacket.condRoutine = conditionRoutine;          /* Condition routine */
  actPacket.dataRoutine = dataRoutine;               /* Data routine */

  /* Call this function before execute the action packet */
  HAL_RADIO_SetReservedArea(&actPacket);

  /* Call this function for the first action packet to be executed */
  HAL_RADIO_MakeActionPacketPending(&actPacket)

  while (1)
  {
    HAL_RADIO_TIMER_Tick();
  }
}
```

The condition callback triggers the execution of next scheduled `ActionPacket` (`txAction`), while the data callback modify data to be sent in next frame.

```
uint8_t conditionRoutine(ActionPacket* p)
{
  if( (p->status & BLUE_INTERRUPT1REG_DONE) != 0)
  {
    if(p->status & BLUE_STATUSREG_PREVTRANSMIT)
    {
      return TRUE;
    }
  }
  return FALSE;
}

uint8_t dataRoutine(ActionPacket* p,  ActionPacket* next)
{
  for(uint8_t i = 0; i < (DataLen-15); i++)
  {
    packet[i+2] = i + data_val;
  }
  data_val++;

  return TRUE;
}
```

4.2.3 RX example with ActionPacket (STSW-BNRGLP-DK SW package)

Below is an example where the radio is programmed to go to RX state periodically. The delay between each RX operation is 9 ms. This ensures that after the first good reception, the RX device wakes up always 1 ms before the TX device (configured in Section 4.2.1: TX example with ActionPacket (STSW-BNRGLP-DK SW package)) starts to send the packet (guard time). The RX timeout is 20 ms. This value is big enough to ensure that at least one packet should be received.

The `ActionPacket` structure is used to define this operation.

```
ActionPacket rxAction;

int main(void)
{
    HAL_VTIMER_InitType VTIMER_InitStruct =
    {HS_STARTUP_TIME, INITIAL_CALIBRATION,
    CALIBRATION_INTERVAL};

    if (SystemInit(SYSCLK_64M,BLE_SYSCLK_32M)!= SUCCESS) {
       /* Error during system clock configuration
       take appropriate action */
       while(1);
    }

    /* Radio configuration*/
    RADIO_Init();
    /* Timer Init */
    HAL_VTIMER_Init(&VTIMER_InitStruct);
    /* Set the channel (22) and the channel increment (0) */
    RADIO_SetChannel(STATE_MACHINE_0, 22, 0);
    /* Sets of the Network ID and the CRC initial value */
    /* RX timeout 20 ms*/
    RADIO_SetGlobalReceiveTimeout(20000);
    rxAction. MaxReceiveLength = 255;
    RADIO_SetTxAttributes(STATE_MACHINE_0, 0x88DF88DF, 0x555555);
    /* Builds ActionPacket (rxAction) to make a reception and schedules
    the next ActionPacket at itself (rxAction) */
    rxAction.StateMachineNo = STATE_MACHINE_0;
    /* Make a RX action with time relative to Wakeup Timer and enable the PLL calibration */
    rxAction.ActionTag = RELATIVE | TIMER_WAKEUP | PLL_TRIG;
    /* 9 ms before operation */
    rxAction.WakeupTime = 9000;
    /* Pointer to the array where the data are received */
    rxAction.data = packet;
    /* Pointer to next ActionPacket: rxAction */
    rxAction.next_true = &rxAction;
    /* Do nothing */
    rxAction.next_false = NULL_0;
    /* Condition routine for selecting next ActionPacket*/
    rxAction.condRoutine = conditionRoutine;
    /* Data routine called after conditionRoutine : RSSI, RX timestamps,
    data received or data modification before next transmission*/
    rxAction.dataRoutine = dataRoutine;
    /* Records the ActionPacket information */
    RADIO_SetReservedArea(&rxAction);
    /* Execute the ActionPacket */
    RADIO_MakeActionPacketPending(&rxAction);
    while(1) {
       HAL_VTIMER_Tick();
    }
    return 0;
}
void BLE_TX_RX_IRQHandler(void)
{
  RADIO_IRQHandler();
}
```

The condition callback triggers the execution of next scheduled `ActionPacket` (`rxAction`), while the data callback is not used in this case, but must be defined anyway.

```
uint8_t conditionRoutine(ActionPacket* p)
{
  /* Check if the RX action is ended */
  if( (p->status & BLUE_STATUSREG_PREVTRANSMIT) == 0) {
    /* Check if a packet with a valid CRC is received */
    if((p->status & BLUE_STATUSREG_RCVOK) != 0) {
    }
    /* Check if a RX timeout occurred */
    else if( (p->status & BLUE_INTERRUPT1REG_RCVTIMEOUT) != 0) {
    }
    /* Check if a CRC error occurred */
    else if ((p->status & BLUE_INTERRUPT1REG_RCVCRCERR) != 0) {
    }
  }
  /* Triggers the next reception */
  return TRUE;
}

uint8_t dataRoutine(ActionPacket* p,  ActionPacket* next)
{
  /* Check if the RX action is ended */
  if( (p->status & BLUE_STATUSREG_PREVTRANSMIT) == 0) {
    /* Check if a packet with a valid CRC is received */
    if((p->status & BLUE_STATUSREG_RCVOK) != 0) {
      /* Retrieve the information from the packet */
      // p->data contains the data received: header field | length field | data field
      // p->rssi
      // p->timestamp_receive
    }
    /* Check if a RX timeout occurred */
    else if( (p->status & BLUE_INTERRUPT1REG_RCVTIMEOUT) != 0) {
    }
    /* Check if a CRC error occurred */
    else if ((p->status & BLUE_INTERRUPT1REG_RCVCRCERR) != 0) {
    }
  }
  return TRUE;
}
```

4.2.4 RX example with ActionPacket (STM32CubeWB0 SW package)

In the example below, the radio is programmed to go to RX state periodically. The delay between each RX operation is 9 ms. This ensures that after the first good reception, the RX device wakes up always 1 ms before the TX device (configured in TX example) starts to send the packet (guard time). The RX timeout is 20 ms. This value is big enough to ensure that at least one packet should be received.

The `ActionPacket` structure is used to define this operation.

```
int main(void)
{
  MX_RADIO_Init();
  MX_RADIO_TIMER_Init();

  /* Channel map configuration */
  uint8_t map[5]= {0xFF,0xFF,0xFF,0xFF,0xFF};
  HAL_RADIO_SetChannelMap(0, &map[0]);

  /* Setting of channel*/
  HAL_RADIO_SetChannel(0, 22, 0);

  /* Sets of the NetworkID and the CRC.*/
  HAL_RADIO_SetTxAttributes(0, 0x88DF88DF, 0x555555);

  /* Configures the transmit power level */
  HAL_RADIO_SetTxPower(MAX_OUTPUT_RF_POWER);

  /* Seting RX timeout*/
  HAL_RADIO_SetGlobalReceiveTimeout(20000);

  /* Build Action Packet */
  actPacket.StateMachineNo = 0;
  actPacket.ActionTag = RELATIVE | TIMER_WAKEUP | PLL_TRIG;
  actPacket.WakeupTime = 9000;
  actPacket.MaxReceiveLength = 255;
  actPacket.data = packet;                          /* Point to memory where store data */
  actPacket.next_true = &actPacket;                 /* Pointer to the next Action Packet*/
  actPacket.next_false = NULL_0;                    /* Null */
  actPacket.condRoutine = conditionRoutine;         /* Condition routine */
  actPacket.dataRoutine = dataRoutine;              /* Data routine */

  /* Call this function before execute the action packet */
  HAL_RADIO_SetReservedArea(&actPacket);

  /* Call this function for the first action packet to be executed */
  HAL_RADIO_MakeActionPacketPending(&actPacket)

  while (1)
  {
    HAL_RADIO_TIMER_Tick();
  }
}
```

```

uint8_t conditionRoutine(ActionPacket* p)
{
  if( (p->status & BLUE_STATUSREG_PREVTRANSMIT) == 0)
  {
  /* Reception ends with no errors */
     if((p->status & BLUE_INTERRUPT1REG_RCVOK) != 0)
  {
    return TRUE;
  }
  /* Reception ends with timeout */
  else if((p->status & BLUE_INTERRUPT1REG_RCVTIMEOUT) != 0){}
    /* Reception ends with errors */
  else if((p->status & BLUE_INTERRUPT1REG_RCVCRCERR) != 0){}
  }
  return FALSE;
}
```

```
uint8_t dataRoutine(ActionPacket* p,  ActionPacket* next)
{
  /* Event is a reception */
  if( (p->status & BLUE_STATUSREG_PREVTRANSMIT) == 0)
  {
    /* Reception ends with no errors */
    if((p->status & BLUE_INTERRUPT1REG_RCVOK) != 0)
    {
      if( (p->status & BLUE_INTERRUPT1REG_ENCERROR)!= 0){}
    }
    /* Reception ends with timeout */
    else if((p->status & BLUE_INTERRUPT1REG_RCVTIMEOUT) != 0){}
    /* Reception ends with errors */
    else if((p->status & BLUE_INTERRUPT1REG_RCVCRCERR)!= 0) {}
  }
  return TRUE;
}
```

5 Proprietary over-the-air (OTA) firmware

This section describes the BlueNRG-LP, BlueNRG-LPS, and STM32WB0 series devices proprietary over-the-air (OTA) firmware upgrade based on the radio low-level driver, which provides access to BlueNRG-LP, BlueNRG-LPS, and STM32WB0 series devices in order to send and receive packets without using the Bluetooth link layer.

This section describes two roles: server and client.

The former node is in charge of sending over-the-air a binary image to the client node.

The latter node acts as a reset manager program choosing, which application to run: the OTA client application that communicates with the server node in order to get the binary image and update its flash memory with it; or the application loaded previously (with OTA or in another way).

5.1 OTA server application

The OTA server application is in charge of sending over-the-air a binary image to the client node. The image is acquired through the UART port by using the YMODEM communication protocol.

The server state machine is implemented by using two state machine routines:

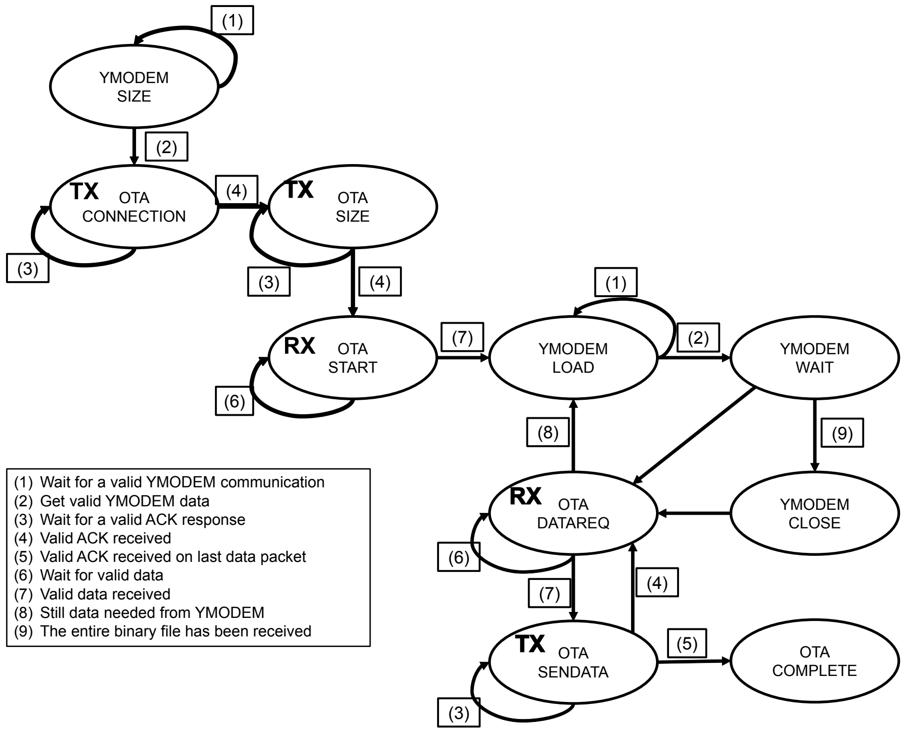
- A state machine dedicated to the YMODEM protocol
- A state machine dedicated to the OTA protocol

A token is used to decide which state machine runs.

5.1.1 OTA server state machine

The following diagram shows the state machine implemented for the OTA server node including both OTA protocol and YMODEM protocol.

Figure 5. OTA server state diagram


YMODEM SIZE
TX OTA CONNECTION
TX OTA SIZE
RX OTA START
YMODEM LOAD
YMODEM WAIT
RX OTA DATAREQ
YMODEM CLOSE
TX OTA SENDATA
OTA COMPLETE
(1) Wait for a valid YMODEM communication
(2) Get valid YMODEM data
(3) Wait for a valid ACK response
(4) Valid ACK received
(5) Valid ACK received on last data packet
(6) Wait for valid data
(7) Valid data received
(8) Still data needed from YMODEM
(9) The entire binary file has been received

There are 4 YMODEM states:

- **SIZE**: it is the first YMODEM communication where the user provides the size of the binary file.
- **LOAD**: it is the main state where up to 1 Kbyte of data (binary image) is received and stored in RAM memory.
- **WAIT**: it is a transient state where it is checked if the entire binary image has been received or not.

- **CLOSE**: it is the state in charge of closing the YMODEM communication once the entire binary image has been received.

Any communication issue through the UART YMODEM is handled with a general abort of the application. The application starts over.

There are 6 OTA states:

- **CONNECTION**: once the size of the binary image file has been received (YMODEM SIZE state), a new firmware update sequence is started, so the server starts to send packets periodically, “connection packet”, in order to show its availability to make a connection. If an ACK response to the “connection packet” is received, then the connection with the client is considered established.
- **SIZE**: during this state, the server sends a “size packet” to the client, showing the size of the binary image that it can send.
- **START**: it is the state where the client says to the server to start the over-the-air firmware update. The client can send a “start packet” or a “not start packet” according to the size of the binary image received during the SIZE state. If the server receives a “start packet”, then the OTA firmware update starts. Otherwise, the server goes to CONNECTION state looking for a next connection.
- **DATAREQ**: the server receives the number of the packets to send (sequence number) from the client.
- **SENDATA**: the server sends the requested (through the sequence number) packet to the client. All the data packets have the same length, but the last one that can have a reduced length. If the data request are still not acquired, the server goes to YMODEM LOAD state and gets the next part of the binary image.
- **COMPLETE**: when the client has acknowledged the last data packet, then the entire binary image has been transferred and the OTA operation is completed.

The RF communication during the OTA operations is managed through re-transmission and a certain number of retries is preprogrammed. If the number of retries during a single state goes through the maximum number of retries configured, then the OTA operation is aborted and the application starts over.

5.1.2 OTA server packet frame

The packet frame used is based on the packet format of the radio low-level driver framework

Figure 6. **Packet frame format**

Preamble	NetworkID	Header	Length	Data	CRC
1 byte	4 bytes	1 byte	1 byte	variable	3 bytes

The header of the packet is used to provide the information about the state where actually the server operates, while the data of the packet provides the information such as the size of the binary image or the data block of the binary image. The NetworkID can be configured by the user, and must be the same both for the server and for the client.

The packet list sent by the server is the following:

- **Connection packet**: it does not contain the data field. The header is set to 0xA0.

Figure 7. **Connection packet**

Preamble	NetworkID	Header	Length	CRC
1 byte	4 bytes	0xA0	0x00	3 bytes

- **Size packet**: it contains 4 bytes of data with the information about the size of the binary image in MSB first.

Figure 8. **Size packet**

Preamble	NetworkID	Header	Length	Data	CRC
1 byte	4 bytes	0xB0	0x04	Image size [4 bytes]	3 bytes

- **Start ACK packet**: it is the response packet used to acknowledge the start packet of the client.

Figure 9. **Start ACK packet**

Preamble	NetworkID	Header	Length	CRC
1 byte	4 bytes	0xC0	0x00	3 bytes

- **Data request ACK packet**: it is the response packet used to acknowledge the data request packet of the client.

Figure 10. **Data request ACK packet**

Preamble	NetworkID	Header	Length	CRC
1 byte	4 bytes	0xD0	0x00	3 bytes

- **Send data packet**: it is the packet with a block of data from the binary image. The sequence number is used to synchronize both the client and the server, and it is used for mechanism of re-transmission.

Figure 11. **Send data packet**

Preamble	NetworkID	Header	Length	Data		CRC
1 byte	4 bytes	0xE0	variable	Seq. num. [2 bytes]	Image [variable]	3 bytes

5.2 OTA client application

The OTA client application is a reset manager application that at the start-up takes the decision to “jump” to the user application or activate the client OTA firmware update application.

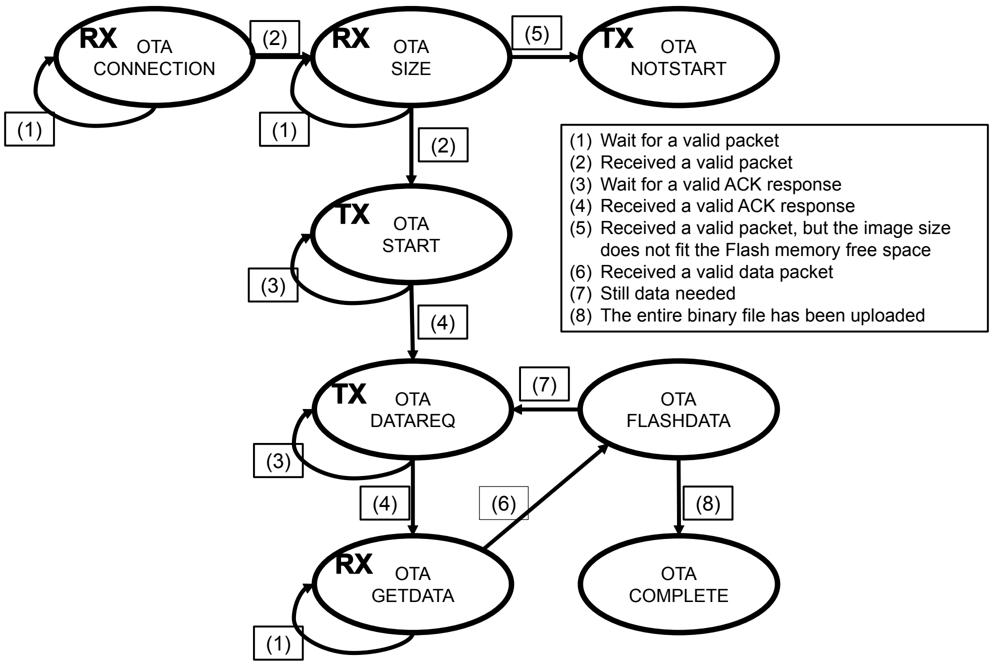
The client OTA firmware update application is implemented through a state machine that manages the OTA protocol and loads in the user flash the binary image acquired.

The OTA client application memory size is below the 8 Kbytes of flash memory.

5.2.1 OTA client state machine

The following diagram shows the state machine for the OTA protocol implemented for the OTA client node.

Figure 12. **OTA client state diagram**


RX OTA CONNECTION
RX OTA SIZE
TX OTA NOTSTART
TX OTA START
TX OTA DATAREQ
OTA FLASHDATA
RX OTA GETDATA
OTA COMPLETE
(1)
(2)
(3)
(4)
(5)
(6)
(7)
(8)
(1) Wait for a valid packet
(2) Received a valid packet
(3) Wait for a valid ACK response
(4) Received a valid ACK response
(5) Received a valid packet, but the image size does not fit the Flash memory free space
(6) Received a valid data packet
(7) Still data needed
(8) The entire binary file has been uploaded

There are 8 OTA states:

- **CONNECTION**: it is the starting state for the client. It looks for a valid “connection packet” coming from the server. If the “connection packet” is received, then an ACK response is sent back. This action establishes the connection with the server.
- **SIZE**: during this state, the client gets the “size packet” from the server.
- **START**: it is the state where the client sends the “start packet” to the server. This causes the OTA firmware update to start. The “start packet” is sent only if the size of the binary image fits the user flash memory.
- **NOTSTART**: the size of the binary image does not fit the user flash memory of the client (it is too big). Therefore, the OTA firmware update cannot start.
- **DATAREQ**: the client sends to the server the number of the data packets it needs. This is calculated considering the size of the binary image and the maximum number of bytes that the server can send (defined initially both for the client and server)
- **SENDATA**: the clients gets the data packet requested
- **FLASHDATA**: in this state the data from the data packet are stored inside a buffer and once the size of the buffer is greater or equal to the page size, all the buffer is actually written in the user flash memory. This operation is done also once the last block of the binary image has been received.
- **COMPLETE**: once the entire binary image has been uploaded in the user flash memory, the OTA operation is completed.

The RF communication during the OTA operation is managed through re-transmission in order to have a certain number of retries. If the number of retries during a single state goes through the maximum number of retries configured, then the OTA operation is aborted and the application starts over.

5.2.2 OTA client packet frame

The packet frame used is based on the packet format of the radio low-level driver framework.

Figure 13. Packet frame format (client)

Preamble	NetworkID	Header	Length	Data	CRC
1 byte	4 bytes	1 byte	1 byte	variable	3 bytes

The header of the packet is used to provide the information about the state where the client actually operates, while the data of the packet is used to request a specific block only of the binary image. The NetworkID can be configured by the user, and must be the same both for the server and for the client.

The packet list sent by the client is the following:

- **Connection ACK packet**: it is the response to the connection packet of the server.

Figure 14. Connection ACK packet

Preamble	NetworkID	Header	Length	CRC
1 byte	4 bytes	0xA0	0x00	3 bytes

- **Size ACK packet**: it is the response to the size packet of the server.

Figure 15. Size ACK packet

Preamble	NetworkID	Header	Length	CRC
1 byte	4 bytes	0xB0	0x00	3 bytes

- **Start packet**: it is used to start OTA operation.

Figure 16. Start packet

Preamble	NetworkID	Header	Length	CRC
1 byte	4 bytes	0xC0	0x00	3 bytes

- **NotStart packet**: it is used to not start OTA operation.

Figure 17. Not start packet

Preamble	NetworkID	Header	Length	CRC
1 byte	4 bytes	0xF0	0x00	3 bytes

- **Data request packet**: it is used to request a specific data packet to the server.

Figure 18. **Data request packet**

Preamble	NetworkID	Header	Length	Data	CRC
1 byte	4 bytes	0xD0	0x02	Seq. num. [2 bytes]	3 bytes

- **Send data ACK packet**: it is used to ACK the data coming from the server.

Figure 19. **Send data ACK packet**

Preamble	NetworkID	Header	Length	CRC
1 byte	4 bytes	0xE0	0x00	3 bytes

5.3 OTA firmware upgrade scenario

Hereafter, a scenario for an OTA firmware upgrade operation.

The scenario is made up of two devices. In the context of BlueNRG-LP, BlueNRG-LPS SW package (STSW-BNRGLP-DK) framework, the following applications are used:

- The server running the application RADIO_OTA_ResetManager, configuration OTA_Server_Ymodem.
- The client running the application RADIO_OTA_ResetManager, configuration OTA_Client.

In the context of STM32CubeWB0 SW package, the following demonstration applications are used:

- The server running the demonstration example RADIO_otaServerYmodem.
- The client running the demonstration example RADIO_otaClient.

The firmware application is located in the related device SW packages under the peripheral examples, *RADIO* folder.

The steps to follow to have the OTA firmware upgrade are listed below:

1. Power up the BlueNRG-LP, BlueNRG-LPS, or STM32CubeWB0 board and its respective demonstration examples for OTA client, and OTA server with YMODEM.
   The OTA_Server_Ymodem board must be plugged with the USB cable to a PC.
2. Open the COM port of the board with the OTA server with YMODEM examples with a serial terminal program such as TeraTerm or similar.
3. Select the transfer mode of a file with YMODEM standard.
   In TeraTerm, this can be done by opening the menu File, then select transfer, then YMODEM and press send.
4. Select the binary image to be uploaded (*.bin* file). Note that the binary image must be generated as explained in Section 5.4: How to add the OTA client function.
5. Once the YMODEM transfer has been completed, the OTA firmware upgrade of the client board is also completed.

Inside the released DK, an example application also containing two configurations reserved to the OTA client functionality, can be found. The example application is the TX and RX applications, which demonstrates a point-to-point communication by using the radio low-level driver. In the context of BlueNRG-LP, BlueNRG-LPS SW package (STSW-BNRGLP-DK) framework, the RADIO_TxRx MIX example is used with the following configurations:

- TX_Use_OTA_ResetManager
- RX_Use_OTA_ResetManager

In the context of STM32CubeWB0 SW package, the following demonstration applications are used:

- RADIO_otaTx
- RADIO_otaRx

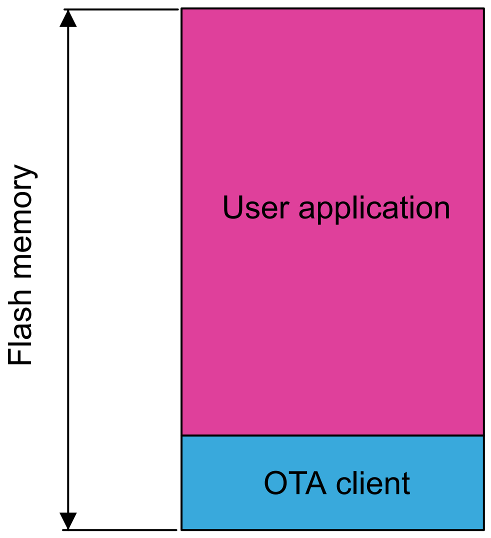
For these two configurations, the steps explained in Section 5.4: How to add the OTA client function have been applied. The button PUSH2 on STEVAL kits, or B2 on NUCLEO kits, is used to run the OTA Client functionality, in order to update the respective image.

5.4 How to add the OTA client function

In order to integrate the OTA client functionality into an existing application, the user must follow the steps below:

1. Reserve the required number of pages of the flash memory for the client application. The linker symbol “MEMORY_FLASH_APP_OFFSET” can be used for this scope as follows:
   MEMORY_FLASH_APP_OFFSET ="offset_value" where offset_value = 0x3000 in the context of STM32CubeWB0 SW package, and 0x2800 in the context of BlueNRG-LP, BlueNRG-LPS SW package).
2. In the context of BlueNRG-LP, BlueNRG-LPS SW package (STSW-BNRGLP-DK), include the files “*radio_ota.c*” and “*radio_ota.h*” in the application project. These files are located in *\Library \BLE_Application\OTA*. In the context of STM32CubeWB0 SW package “*radio_ota.c*” and “*radio_ota.h*” are provided within the user demonstration *RADIO_otaTx* or *RADIO_otaRx* folders
3. Define a trigger to be used to jump from the user application to the OTA client application. This trigger is used to call the function `OTA_Jump_To_Reset_Manager()`.

Figure 20. OTA client flash memory layout



DT58365

6 How to optimize radio throughput

The ActionPacket framework offers a high degree of flexibility. It also allows for code optimization in terms of throughput and latency.

This section shows different areas of optimization using the following application scenario: sending 100 packets of 10 bytes each in the shortest time.

No ActionPacket implementation

The simplest way to implement this scenario is to use the approach without ActionPacket by calling `HAL_RADIO_SendPacket()` for all packets and setting the wakeup time to 270 µs.

Example:

```
int main(void)
{
/* Set the Network ID */
  HAL_RADIO_SetNetworkID(0x88DF88DF);
  /* Configures the transmit power level */
  HAL_RADIO_SetTxPower(0x18);
  /* Build packet */
  sendData[0] = 0x02;
  sendData[1] = PAYLOAD_LEN;   /* Length position is fixed */
  sendData[2] = (uint8_t)packet_counter;
  for(uint8_t i = 0; i < PAYLOAD_LEN; i++)
  {
    sendData[3+i] = 0x00 + i;
  }
  sendNewPacket = TRUE;
  while(1)
 {
    HAL_RADIO_TIMER_Tick();

    if( sendNewPacket == TRUE )
    {
      sendNewPacket = FALSE;
      ret =  HAL_RADIO_SendPacket(channel, 270, sendData, HAL_RADIO_Callback);
    }
 }
}
```

Only the `HAL_RADIO_CallbackTxDone()` callback is required. It implements the checks for the next transmission and the payload update operations.

Example:

```
void HAL_RADIO_CallbackTxDone(void)
{
  packet_counter++;
  if(packet_counter <= MAX_NUM_PACKET)
  {
    /* Update payload */
    sendData[2] = packet_counter;
    for(uint8_t i = 0; i < PAYLOAD_LEN; i++)
    {
      sendData[3+i] = sendData[3+i] + 0x01;
    }
    sendNewPacket = TRUE;
   }
}
```

This implementation measures a sequence length of 45 ms.

ActionPacket implementation

This scenario can also be implemented using the ActionPacket approach. This removes the overhead caused by the ActionPacket definition code. The execution of an action can be divided into two phases:

1. Define where ActionPacket is configured and written in memory using the `HAL_RADIO_SetReservedArea()` function
2. Schedule ActionPacket using the `HAL_RADIO_MakeActionPacketPending()` function

If during the sequence the only change is the frame payload, it is possible to call only `HAL_RADIO_MakeActionPacketPending()` in while loop. This way the declaration is done only once.

In this example, the condition routine always returns true because there is no other path, and the data routine is used to upload payload and check all frames.

`HAL_RADIO_TIMER_SetRadioCloseTimeout()` can be called after `HAL_RADIO_MakeActionPacketPending()` to generate the closest possible action event.

Example:

```
int main(void)
{
/* Set the Network ID */
  HAL_RADIO_SetNetworkID(0x88DF88DF);
  /* Configures the transmit power level */
  HAL_RADIO_SetTxPower(0x18);

  aPacket.StateMachineNo = 0;
  aPacket.ActionTag = RELATIVE | TXRX | PLL_TRIG;
  aPacket.WakeupTime = 270;
  aPacket.MaxReceiveLength = 0; /* does not affect for Tx */
  aPacket.data = sendData;
  aPacket.next_true = NULL_0;
  aPacket.next_false = NULL_0;
  aPacket.condRoutine = TxCondRoutineSingle;
  aPacket.dataRoutine = Callback;

  HAL_RADIO_SetReservedArea(&aPacket);

  /* Build packet */
  sendData[0] = 0x02;
  sendData[1] = PAYLOAD_LEN;   /* Length position is fixed */
  sendData[2] = (uint8_t)packet_counter;
  for(uint8_t i = 0; i < PAYLOAD_LEN; i++)
  {
    sendData[3+i] = 0x00 + i;
  }
  sendNewPacket = TRUE;
  while(1)
 {
    HAL_RADIO_TIMER_Tick();

    if( sendNewPacket == TRUE )
    {
      sendNewPacket = FALSE;
      ret =  HAL_RADIO_MakeActionPacketPending(&aPacket);
      HAL_RADIO_SetRadioCloseTimeout();
    }
 }
}
```

This implementation measures a sequence length of 38 ms with a time saving of 16%.

The `TxCondRoutineSingle()` is implemented as follow:

```
static uint8_t TxCondRoutineSingle(ActionPacket *p)
{
  return TRUE;
}

uint8_t Callback(ActionPacket* p, ActionPacket* next)
{
  /* Event is a reception */
  /* Operation done */
  if ((p->status & BLUE_INTERRUPT1REG_DONE) != 0)
  {
    /* Event is a transmission */
    if ((p->status & BLUE_STATUSREG_PREVTRANSMIT) != 0)
    {
      packet_counter++;

      if(packet_counter != MAX_NUM_PACKET)
      {
          sendData[2] = (uint8_t)packet_counter;
          for(uint8_t i = 0; i < PAYLOAD_LEN; i++)
          {
            sendData[3+i] = sendData[3+i] + 0x01;
          }
          sendNewPacket = TRUE;
      }
    }
  }
  return TRUE;
}
```

Skip PLL calibration

The PLL calibration can be skipped if:

- the channel selected for transmission is not the first one
- the channel selected is the same as the previous action
- there is no channel change during the sequence

Together with `HAL_RADIO_TIMER_SetRadioCloseTimeout()` 40 µs can be saved for each transfer.

Example:

```
aPacket.StateMachineNo = 0;
aPacket.ActionTag = RELATIVE | TXRX;
aPacket.WakeupTime = 270;
aPacket.MaxReceiveLength = 0; /* does not affect for Tx */
aPacket.data = sendData;
aPacket.next_true = NULL_0;
aPacket.next_false = NULL_0;
aPacket.condRoutine = TxCondRoutineSingle;
aPacket.dataRoutine = Callback;
```

In this example, the sequence length is reduced to 34 ms.

Schedule action by condition routine and HAL_RADIO_ActionPacketIsr()

For further optimisation, configure ActionPacket to point to itself when sending another packet, and to NULL when sending all packets.

Example:

```
int main(void)
{
/* Set the Network ID */
  HAL_RADIO_SetNetworkID(0x88DF88DF);
  /* Configures the transmit power level */
  HAL_RADIO_SetTxPower(0x18);
  HAL_RADIO_SetBackToBackTime(85);

  aPacket.StateMachineNo = 0;
  aPacket.ActionTag = RELATIVE | TXRX | PLL_TRIG;
  aPacket.WakeupTime = wakeup_time;
  aPacket.MaxReceiveLength = 0; /* does not affect for Tx */
  aPacket.data = sendData;
  aPacket.next_true = &aPacket;
  aPacket.next_false = NULL_0;
  aPacket.condRoutine = TxCondRoutineBurst;
  aPacket.dataRoutine = TxCallbackBurst;

  HAL_RADIO_SetReservedArea(&aPacket);;
   /* Build packet */
  sendData[0] = 0x02;
  sendData[1] = PAYLOAD_LEN;   /* Length position is fixed */
  sendData[2] = (uint8_t)packet_counter;
  for(uint8_t i = 0; i < PAYLOAD_LEN; i++)
  {
    sendData[3+i] = 0x00 + i;
  }
  startNewSequence = TRUE;
  while(1)
 {
    HAL_RADIO_TIMER_Tick();

    if( startNewSequence == TRUE )
    {
      startNewSequence = FALSE;
      ret =  HAL_RADIO_MakeActionPacketPending(&aPacket);
    }
 }
}
```

In this case, ActionPacket ISR schedules the new event according to back-to-back time, which can be less than wakeup time.

The code to select the path is in the condition routine, while the code to update the frame is in the data routine.

The `TxCondRoutineBurst ()` and `TxCallbackBurst(ActionPacket ()` are implemented as follows:

```
static uint8_t TxCondRoutineBurst(ActionPacket* p)
{
  if ((p->status & BLUE_INTERRUPT1REG_DONE) != 0)
  {
    if ((p->status & BLUE_STATUSREG_PREVTRANSMIT) != 0)
    {
      packet_counter++;
      if(packet_counter < MAX_NUM_PACKET)
      {
        return TRUE;
      }
    }
  }
  return FALSE;
}

uint8_t TxCallbackBurst(ActionPacket* p, ActionPacket* next)
{
  /* Operation Done */
  if ((p->status & BLUE_INTERRUPT1REG_DONE) != 0)
  {
    /* RADIO TX Operation */
    if ((p->status & BLUE_STATUSREG_PREVTRANSMIT) != 0)
    {
      if(packet_counter < MAX_NUM_PACKET)
      {

         sendData[2] = (uint8_t)packet_counter;
         for(uint8_t i = 0; i < PAYLOAD_LEN; i++)
        {
          sendData[3+i] = sendData[3+i] + 0x01;
        }
      }
    }
  }
  return TRUE;
}
```

In this example, setting back-to-back time (B2B_TIME) to 85 µs and skipping PLL calibration completes the sequence in 24 ms, saving an additional 30% over previous optimisation and 47% over no ActionPacket approach.

Table 5. Configuration summary

Parameter	Value
N_PACKET_SENT	100
PAYLOAD LENGHT	10 byte
WAKEUP_TIME	270 µs
B2B_TIME	85 µs

Table 6. Optimization results

Implementation	PLL Calibration	Sequence lenght
HAL	ON	45 ms
HAL (SetCloseTimeout)	ON	40 ms
ActionPacket (SetCloseTimeout)	ON	38 ms
ActionPacket (SetCloseTimeout)	OFF	34 ms
ActionPacket ISR	ON	33 ms
ActionPacket ISR	OFF	24 ms

Revision history

Table 7. **Document revision history**

Date	Version	Changes
20-Jul-2020	1	Initial release.
29-Mar-2021	2	Added Table 4. Status_table. Updated Table 3. ActionTag field description, Section 4.1.1: TX example with no ActionPacket (STSW-BNRGLP-DK SW package) and Section 4.1.3: RX example with no ActionPacket (STSW-BNRGLP-DK SW package).
06-Apr-2022	3	Updated Section Introduction, Section 3.1: Description, Section 3.2: API architecture, Section 4.1: No ActionPacket approach and Section 5.3: OTA firmware upgrade scenario. Added the BlueNRG-LPS references throughout the document.
19-Jun-2024	4	General update of the scope of the document to include STM32WB0 series. Updates impacting document title, and all sections.
17-Oct-2024	5	Added Section 6: How to optimize radio throughput .

Contents

List of tables

List of figures