UM2379

User manual

The BlueNRG-1, BlueNRG-2 radio driver

Introduction

This document describes the BlueNRG-1, BlueNRG-2 radio low level driver, which provides access to the BlueNRG-1 and BlueNRG-2 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.

The document content is valid both for the BlueNRG-1 and BlueNRG-2 devices. Any reference to BlueNRG-1 device is also valid for the BlueNRG-2 device. Any specific difference is highlighted whenever it is needed.

1 BlueNRG-1, BlueNRG-2 radio operation

The BlueNRG-1, BlueNRG-2 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:

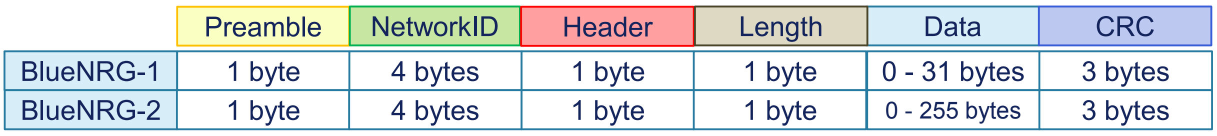
- Packet encryption
- Communication timing
- Sleep management

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

2 Data packet format

There is only one packet format used in the BlueNRG-1 and BlueNRG-2, it is shown below.

Figure 1. Packet format



	Preamble	NetworkID	Header	Length	Data	CRC
BlueNRG-1	1 byte	4 bytes	1 byte	1 byte	0 - 31 bytes	3 bytes
BlueNRG-2	1 byte	4 bytes	1 byte	1 byte	0 - 255 bytes	3 bytes

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

- Preamble is not user-accessible; its length is 1 byte.
- 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 equals
  - 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 read this value from a received packet.

  The maximum value of the length field is 31 for the BlueNRG-1 and 255 for the BlueNRG-2, 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. If the radio works in the Bluetooth advertising channels (37, 38 and 39), then the received data field is limited to 37 bytes for the BlueNRG-1 and 43 bytes for the BlueNRG-2. The table below contains a summary about the length field.

Table 1. Values in bytes for the length field

	Data channels	Data channels with encryption	Advertising channels	Advertising channels with encryption
BlueNRG-1	31	27	37	27
BlueNRG-2	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.

- CRC is not user-accessible. It is used to identify corrupted packets. Its length is 3 bytes. The user can configure the initial value for the CRC calculation, except in the advertising channels where the initial value must be 0x555555.

Figure 2. Packet format with encryption enabled

	Preamble	NetworkID	Header	Length	Data	MIC	CRC
BlueNRG-1	1 byte	4 bytes	1 byte	1 byte	0 - 27 bytes	4 bytes	3 bytes
BlueNRG-2	1 byte	4 bytes	1 byte	1 byte	0 - 251 bytes	4 bytes	3 bytes

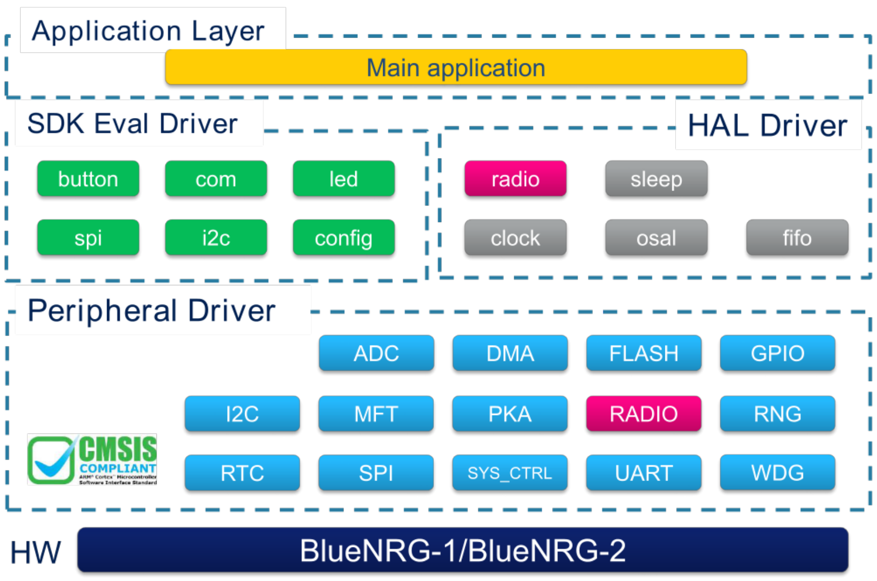
3.1 Description

The radio low level driver consists of four files:

- BlueNRG1_Radio.h and BlueNRG1_Radio.c are inside the Peripheral Driver folder
- hal_radio.h and hal_radio.c are inside the HAL Driver folder

The figure below shows the layers of a common application using the radio low level driver.

Figure 3. Framework of an application (radio low level included)



3.2 API architecture

The radio low level driver interface provides a set of APIs (file BlueNRG1_radio.c) which allows the following functions to be addressed :

- Radio initialization
- Encryption
- Communication channel management
- Set of network ID, CRC initial value, power level
- Test commands (tone)

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_StopActivity()
- RADIO_StartTone()
- RADIO_StopTone()

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 with 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 table ActionTag
WakeupTime	IN	Contains the wake-up time in microseconds if it is relative. It should not be more than 24 bits if it is absolute. It only applies if TIMER_WAKEUP flag is set in ActionTag
ReceiveWindowLength	IN	Sets RX window size in microseconds. Applicable only for RX actions
*next_true	IN	Pointer to next ActionPacket if condRoutine() returns TRUE
*next_false	IN	Pointer to next ActionPacket if condRoutine() returns FALSE

Parameter name	Input/output	Summary
(*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 us.
(*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 period of 512 kHz clock. To convert it to microseconds it should be multiplied by (1000/512) It has a wrapping value equals to (2^24)/512000
status	OUT	The status register with the information on the action. Details on the status in the table status field description
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	TIMESTAMP_POSITION	This bit sets where the position of the timestamp is taken, at the beginning of the packet or at the end of it. RX only. 0: end of the packet 1: beginning of the packet
6	RESERVED	RESERVED
5	RELATIVE	It determines if the WakeupTime field of the ActionPacket is considered as absolute time or relative time to the current. 0: absolute (not suggested) 1: relative
4	INC_CHAN	This bit activates automatic channel increment. The API RADIO_SetChannel()(1) sets the value of the increment. 0: no increment 1: automatic increment
3	RESERVED	RESERVED
2	TIMER_WAKEUP	The bit determines if the action (RX or TX) is going to be executed based on the back-to-back time or based on the WakeupTime. 0: based on the back-to-back time (default 150 μs). 1: based on the WakeupTime

Bit	Name	Description
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. User should set this bit only if TIMER_WAKEUP is set to 1

1. *In the advertising channels, the frequency hopping is limited to 1 hop.*

The table below describes the different bits of the status field in the ActionPacket.

Table 4. Status field description

Bit name	Bit position	Description
IRQ_RCV_OK	31	The packet is received, and the CRC is valid
IRQ_CRC_ERR	30	The packet is received with CRC error or timeout error (1)
RESERVED	29:27	RESERVED
IRQ_TIMEOUT	26	No packet received within the defined RX time window
RESERVED	25	RESERVED
IRQ_DONE	24	Requested action (TX or RX) has been executed
IRQ_ERR_ENC	23	Encryption error. The packet received has an error in the MIC field
IRQ_TX_OK	22	The packet has been sent successfully
RESERVED	20:19	RESERVED
BIT_TX_MODE	18	Previous ActionPacket was a TX action
RESERVED	17:0	RESERVED

1. *It is set when there is an CRC error or timeout error. So, to know whether it is a pure CRC error, IRQ_CRC_ERR and IRQ_TIMEOUT should be checked together.*

4 How to write an application

There are two ways to write an application: the former is based on the HAL layer composed mainly of four APIs, and the latter based on the use of the ActionPacket data structure.

4.1 HAL layer approach

The simplest way is to use a set of APIs provided in HAL radio driver (file hal_radio.c), 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 send back (ACK)

In this contest, the user does not need to use the ActionPacket to configure the operations of the radio, but it is requested a pointer to a user callback, 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 ISR Blue_Handler(), that has the maximum priority among the other ISR functions of the radio.

4.2 TX example with HAL layer

Below 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 not used directly in this example, but it is used through the APIs of the HAL layer.

```
uint8_t send_packet_flag = 1;
uint8_t packet[5];
int main(void)
{
  SystemInit();

  /* Radio configuration - HS start up time, external LS clock enable, whitening e
nable */
  RADIO_Init(1, 0, NULL, ENABLE);
  /* 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) {
    /* If ready, a new action is scheduled */
    if(send_packet_flag == 1) {
      send_packet_flag = 0;
      /* Schedule the action with the parameter - channel, wakeupTime, relative ti
me, dataCallback */
      HAL_RADIO_SendPacket( 22, 10000, 1, packet, TxCallback );
   }
  }
  return 0;
}


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

The user callback, TxCallback(), is defined in order to re-schedule another send packet action as follows:

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

4.3 RX example with HAL layer

Below 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 the Section 4.2 TX example with HAL layer) 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 not used directly in this example, but it is used through the APIs of the HAL layer.

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

int main(void)
{
  SystemInit();

  /* Radio configuration - HS start up time, external LS clock enable, whitening e
nable */
  RADIO_Init(1, 0, NULL, ENABLE);
  /* Set the Network ID */
  HAL_RADIO_SetNetworkID(0x88DF88DF);

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

      /* Schedule the action with the parameter - channel, wakeupTime, relative ti
me, RX timeout, timestamp, dataCallback */
      HAL_RADIO_ReceivePacket ( 22, 9000, 1, packet, 20000, 0, RxCallback );
    }
  }
 return 0;
}


void Blue_Handler(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 & BIT_TX_MODE) == 0) {
    /* Check if a packet with a valid CRC is received */
    if((p->status & IRQ_RCV_OK) != 0) {
      /* Retrieve the information from the packet */
      // p->data contains the data received: header field | length field | data fi
eld
      // p->rssi
      // p->timestamp_receive
    }
    /* Check if a RX timeout occurred */
    else if( (p->status & IRQ_TIMEOUT) != 0) {
    }
    /* Check if a CRC error occurred */
    else if ((p->status & IRQ_CRC_ERR) != 0) {
    }
  }
  /* Triggers the next reception */
  rx_flag = 1;
  return TRUE;
}
```

4.4 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 ActionPacket itself.

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

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

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 between 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.

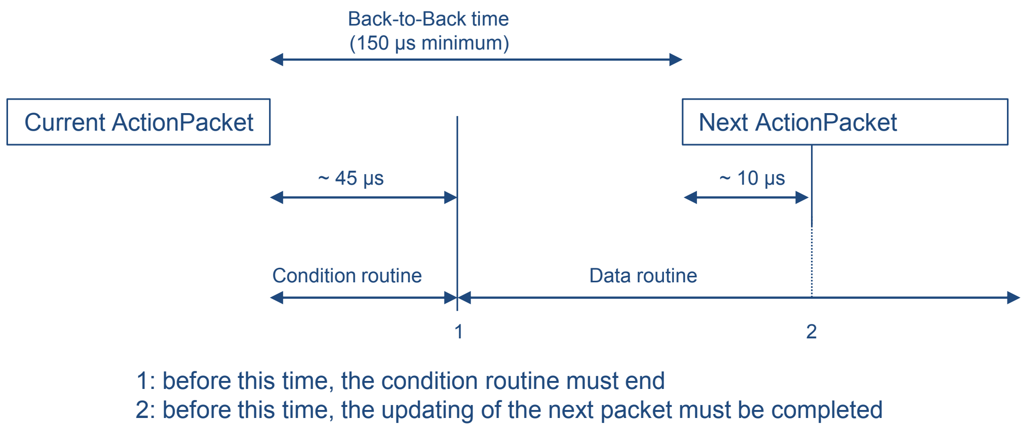
The goal of the multiple callbacks is to enable the user to access to the total performance of the radio, by avoiding time criticality bottlenecks. The goal is to bundle time critical aspects in the condition routine, and have the rest of the framework to be non-time-critical.

The execution time of the condition routine should never exceed 45 µs.

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

The figure below summarizes the timing of the different callbacks.

Figure 4. Callback timing


Back-to-Back time
(150 µs minimum)
Current ActionPacket
Next ActionPacket
~ 45 µs
~ 10 µs
Condition routine
Data routine
1
2
1: before this time, the condition routine must end
2: before this time, the updating of the next packet must be completed

4.5 TX example with ActionPacket

Below 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.

```
uint8_t packet[5];
ActionPacket txAction;

int main(void)
{
  SystemInit();

  /* Radio configuration - HS start up time, external LS clock enable, whitening e
nable */
  RADIO_Init(1, 0, NULL, ENABLE);

  /* 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, 0);
  /* 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 calibr
ation */
  txAction.ActionTag = RELATIVE | TIMER_WAKEUP | TXRX | PLL_TRIG;
  /* 10 ms before operation */
  txAction.WakeupTime = 10000;
  /* Not available for TX */
  txAction.ReceiveWindowLength = 0;
  /* 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) {
  }
 return 0;
}


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

The condition callback triggers the execution of next schedule 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 & BIT_TX_MODE) != 0) {

  }
  /* The TRUE schedules the next_true action: txAction */
  return TRUE;
}

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

4.6 RX example with ActionPacket

Below 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 4.5 TX example with ActionPacket) 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.

```
uint8_t packet[MAX_PACKET_LENGTH];
ActionPacket rxAction;

int main(void)
{
  SystemInit();

  /* Radio configuration - HS start up time, external LS clock enable, whitening e
nable */
  RADIO_Init(1, 0, NULL, ENABLE);

  /* 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, 0);

  /* Builds ActionPacket (rxAction) to make a reception and schedules the next Act
ionPacket at itself (rxAction) */
  rxAction.StateMachineNo = STATE_MACHINE_0;

  /* Make a RX action with time relative to Wakeup Timer and enable the PLL calibr
ation */
  rxAction.ActionTag = RELATIVE | TIMER_WAKEUP | PLL_TRIG;
  /* 9 ms before operation */
  rxAction.WakeupTime = 9000;
  /* RX timeout 20 ms*/
  rxAction.ReceiveWindowLength = 20000;
  /* 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 receiv
ed or data modification before next transmission*/
  rxAction.dataRoutine = dataRoutine;

  /* Records the ActionPacket information */
  RADIO_SetReservedArea(&rxAction);
  /* Execute the ActionPacket */
  RADIO_MakeActionPacketPending(&rxAction);

  while(1) {
  }
 return 0;
}
```

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

The condition callback triggers the execution of next schedule 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 & BIT_TX_MODE) == 0) {
    /* Check if a packet with a valid CRC is received */
    if((p->status & IRQ_RCV_OK) != 0) {
    }
    /* Check if a RX timeout occurred */
    else if( (p->status & IRQ_TIMEOUT) != 0) {
    }
    /* Check if a CRC error occurred */
    else if ((p->status & IRQ_CRC_ERR) != 0) {
    }
  }
  /* Triggers the next reception */
  return TRUE;
}

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

5 The BlueNRG-1, BlueNRG-2 proprietary over-the-air (OTA) firmware

This section describes the BlueNRG-1, BlueNRG-2 proprietary over-the-air (OTA) firmware upgrade based on the radio low-level driver, which provides access to the BlueNRG-1, BlueNRG-2 devices in order to send and receive packets without using the Bluetooth link layer.

This chapter 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 other 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:

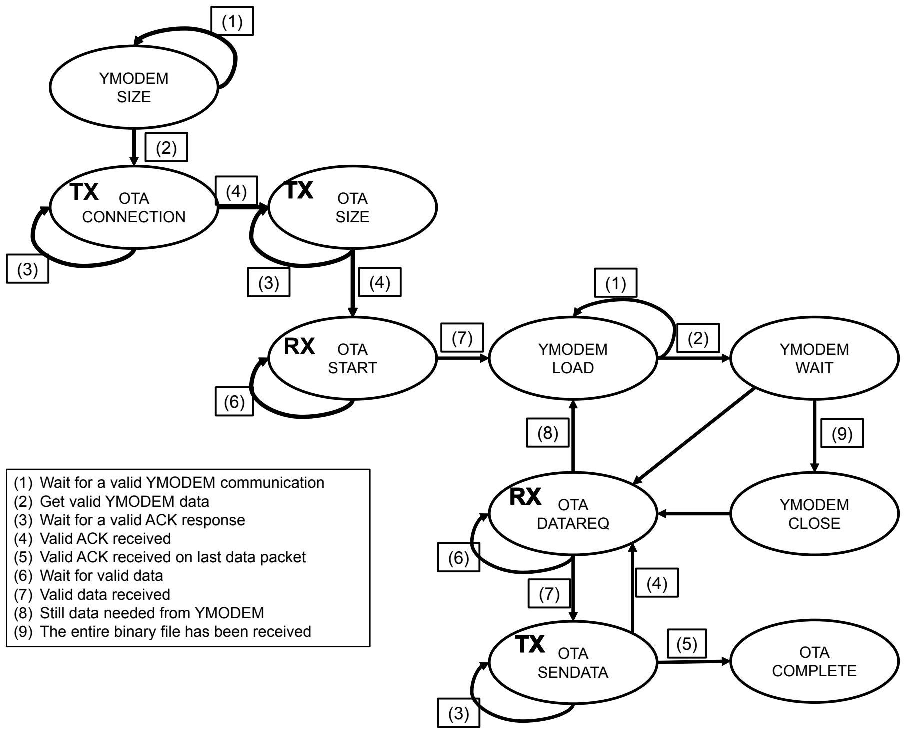
- A state machine dedicated to the YMODEM protocol
- A state machine dedicated to the OTA protocol

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

The YMODEM states are 4:

- **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 kB 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.

The OTA states are 6:

- **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 periodically packets, “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 not still be 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.

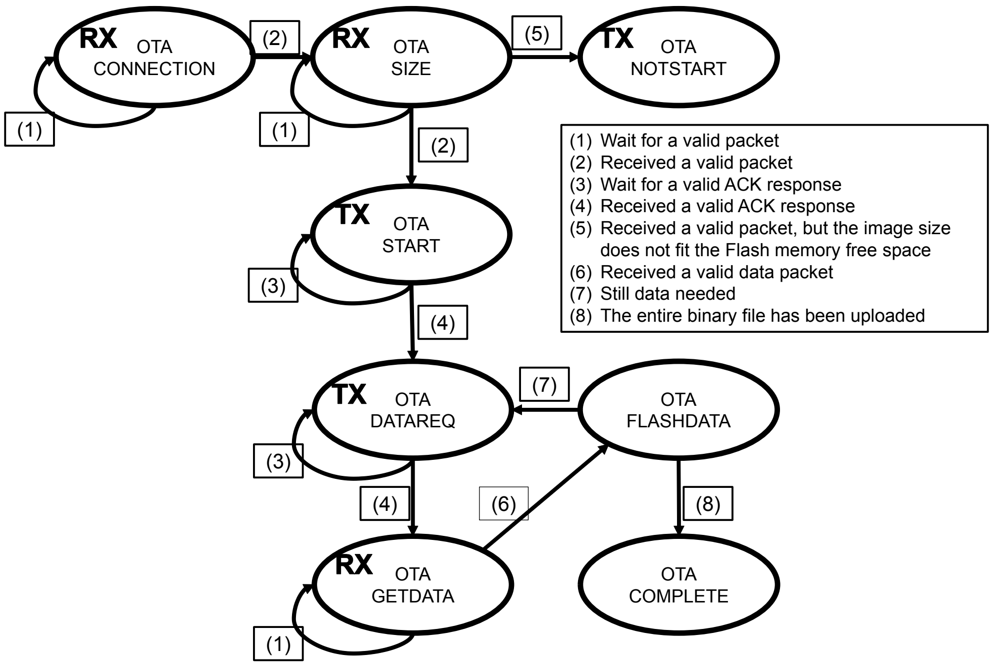
The client OTA firmware update application is implemented through a state machine that manages the OTA protocol and loads in the user Flash the binary image acquired.

The OTA client application memory size is below the 8 kB 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

The OTA states are 8:

- **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 makes the connection with the server established.
- **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 makes the OTA firmware update 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 equals 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 actually the client 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.

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

The firmware application is located in the DK folder \Project\BlueNRG1_Periph_Examples\RADIO \OTA_ResetManager.

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

1. Power up both the BlueNRG-1 boards and their respective applications OTA_Server_Ymodem and OTA_Client.

   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_Ymodem configuration 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 TxRx, which demonstrates a point-to-point communication by using the radio low-level driver. The configurations are:

- TX_Use_OTA_ResetManager.
- RX_Use_OTA_ResetManager.

For these two configurations, the steps explained in Section 5.4 How to add the OTA client function have been applied. The push button PUSH2 of the STEVAL-IDB007Vx is used to run the OTA Client functionality, in order to update the respective image.

The example is located in \Project\BlueNRG1_Periph_Examples\RADIO\TxRx.

5.4 How to add the OTA client function

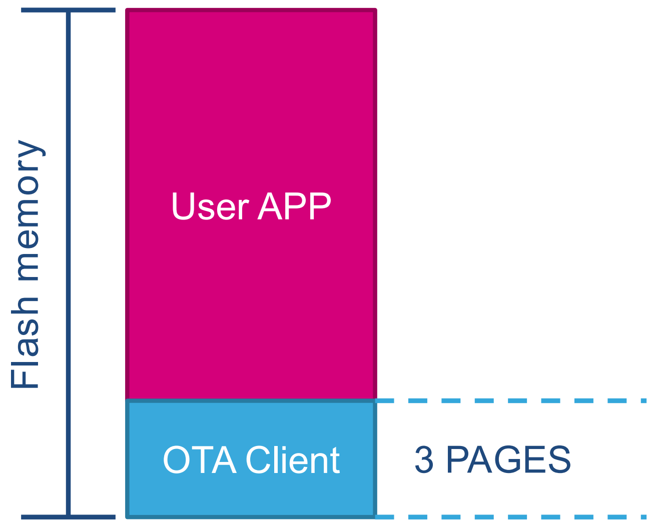
In order to integrate the OTA client functionality to an existing application, the user must follow the steps below:

1. Reserve the first three pages (8 kB) 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 =0x2000

   The example application TxRx, configurations TX_Use_OTA_ResetManager or RX_Use_OTA_ResetManager can be used as reference.
2. Include the file “radio_ota.c” and “radio_ota.h” to the application project. These files are located in \Library \BLE_Application\OTA. The files contain the API OTA_Jump_To_Reset_Manager() used for setting the RAM variable named ota_sw_activation and then reset the system.
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



Revision history

Table 5. **Document revision history**

Date	Version	Changes
21-Jun-2018	1	Initial release.

Contents

List of tables

List of figures