QN902x Software Developer's Guide

Version 1.4

Table of Contents

1. Introduction

1.1 Purpose

This document specifies the Quintic QN902x Bluetooth Low Energy (BLE) technical details which the software developers need to know. Quintic QN902x BLE solution offers a complete Software Development Kit (SDK) to develop various single-mode BLE applications. This document serves as a guide for the users to develop application program based on Quintic QN902x.

1.2 References

[1] Bluetooth Specification Version 4.0
[2] ARMv6-M Architecture Reference Manual
[3] Cortex-M0 R0P0 Technical Reference Manual
[4] QN9020 API Programming Guide

1.3 Definitions, Symbols and Abbreviations

ACI	Application Control Interface
API	Application Program Interface
APP	Application
ATT	Attribute Protocol
BLE	Bluetooth Low Energy
BSP	Board Support Package
GAP	Generic Access Profile
GATT	Generic Attribute Profile
HCI	Host Control Interface
ISP	In-system Program
LL	Link Layer
L2CAP	Logical Link Control and Adaptation Protocol
PHY	Physical
SMP	Security Manager Protocol
SoC	System on Chip

2. QN902x BLE Software Platform

2.1 Software Architecture

Quintic QN902x is an Ultra Low Power SoC (System-on-Chip) solution designed for Bluetooth Low Energy standard, which combines an ARM Cortex-M0 processor, 96kB ROM, 64kB SRAM, 128kB FLASH, 2.4GHz RF transceiver and a full range of peripherals.

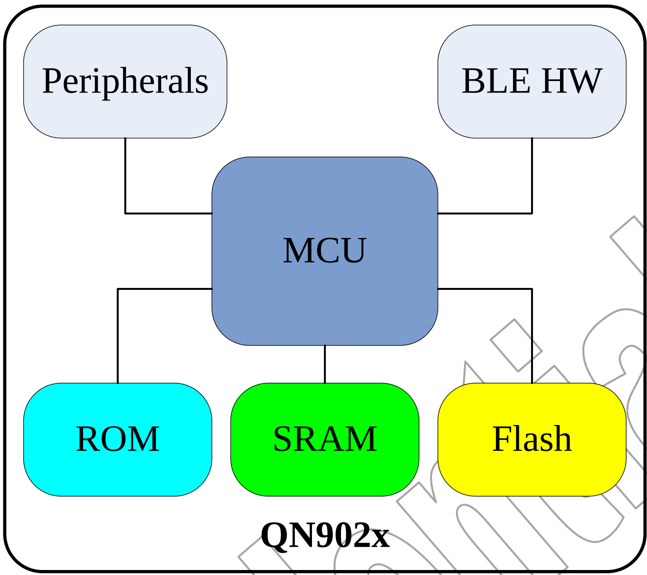


Figure 1 QN902x Hardware Architecture

The software platform of QN902x consists of two main parts: Firmware and Application project. All codes should be executed from internal SRAM and ROM. The code executed in the ROM is called Firmware. The code executed in the SRAM is called Application project. The FLASH can be used to store the application program and the user data which should be saved when system is power-down.

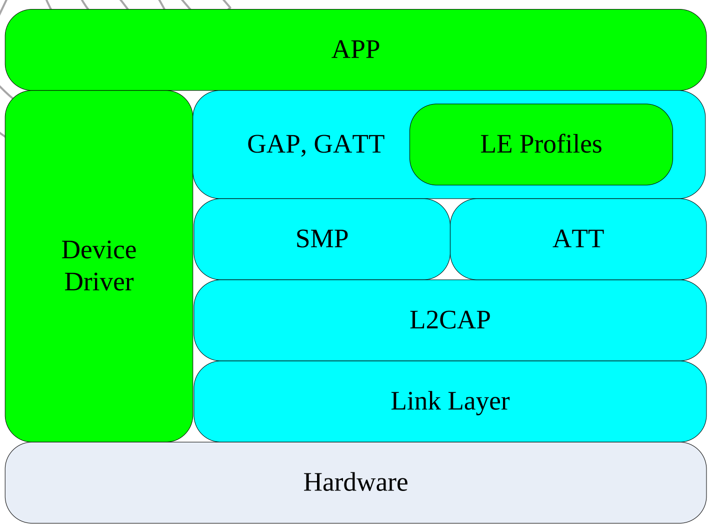


Figure 2 Software Architecture

The software platform is also considered as five major sections: Kernel, BLE protocol stack, Profiles, Device drivers and Application. The Firmware in the ROM contains the Kernel and BLE protocol stack which are only provided as APIs. The Application project in the SRAM contains Profiles, Device drivers and Application which are provided as full source code.

2.2 Working Mode

Quintic QN902x provides the most flexible platform for wireless applications, which supports three working modes: Wireless SoC Mode, Network Processor Mode and Controller Mode.

2.2.1 Wireless SoC Mode

In the Wireless SoC Mode the link layer, host protocol, profiles and application all run on the QN902x as a single chip solution. This is the work mode that the application samples are used.

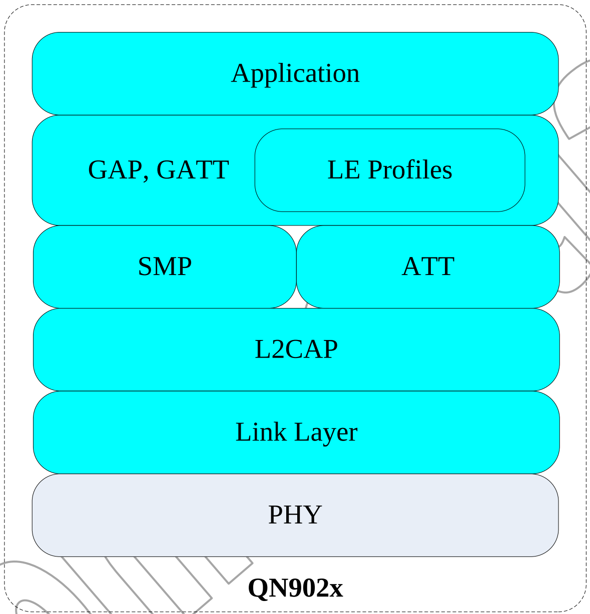


Figure 3 Wireless SoC Mode

2.2.2 Network Processor Mode

In the Network Processor Mode the link layer, host protocols and profiles run on the QN902x, and the application runs on the external microcontroller or PC. These two components communicate via ACI (Application Control Interface), which are provided on QN902x.

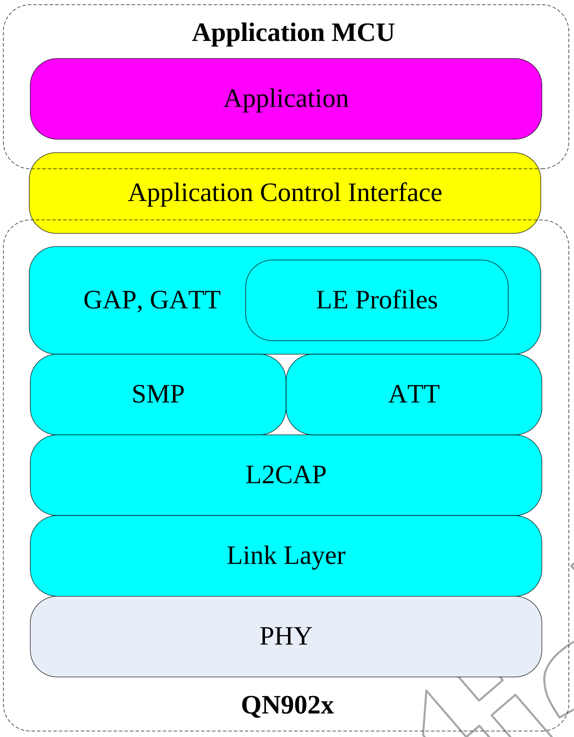


Figure 4 Network Processor Mode

2.2.3 Controller Mode

In the Controller Mode only the link layer runs on the QN902x. The host protocol, profiles and application all run on the external microcontroller or PC. These two components communicate via HCI. Generally this mode is not used in the product design except for the product testing.

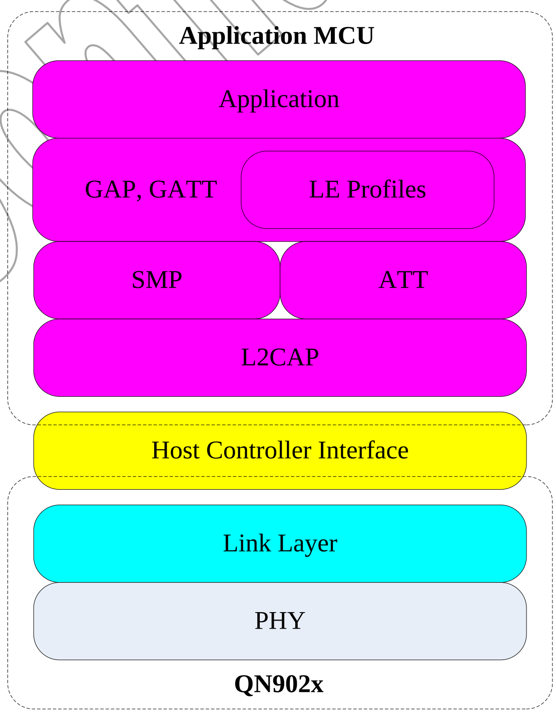


Figure 5 Controller Mode

3. Operation System

3.1 Overview

The BLE protocol stack is composed of several protocol layers which have their own state machines and will exchange messages with each other. In order to support data and message exchange between different protocol layers, a very basic message driven kernel is implemented. It is a small and efficient Real Time Operating System, offering the features of event service, exchange of messages, memory management and timer functionality.

The kernel provides an event service used to defer actions. In the interrupt handler only the critical handling is performed and less critical handling is done in an event handler that is scheduled under interrupt.

Quintic's BLE protocol is composed of LL, L2CAP, SMP, ATT, GATT, GAP, Profiles and APP Layer. Each protocol layer may be divided into a number of sub-layers. These layers have these own state machine which is managed by an individual task. The kernel defines task descriptor to help each task to manage their state machine and message processing handler.

The kernel manages a message queue which saves all of the messages sent by task. And the kernel is responsible for distributing messages in the message queue. It will find the appropriate message handler based on the message ID and execute the handler. When there is no message in the message queue, the kernel enters into idle state.

The kernel provides basic memory management which is similar to C standard malloc/free function. These memory management functions are only implemented for kernel message, kernel timer and ATTS database. So the user will use message API, timer API and ATTS API instead of using memory management functions directly. This kernel memory management module needs its own heap to control. So the application should arrange an available memory space to memory management module. The heap size is based on application design. If you want to know how to determine the heap size, please refer to chapter 7.

The kernel provides timer services for tasks. Timers are used to reserve a message, delay some time and then send the message.

In this chapter, we just introduce the basic concept of the kernel. For more details about how to add application tasks into the kernel and how to execute the kernel in the application, please refer to chapter 7.

3.2 Events

When the scheduler is called in the main loop of the background, the kernel checks if the event field is non-null, gets the one with highest priority and executes the event handlers for which the corresponding event bit is set.

There are total 32 events, and the highest priority events are used by BLE stack. So users have 24 events could be used in the application. The following snippet is a pseudo code of event scheduler.

```
    // Get event field
    field = ke_env.evt_field;

    while (field)
    {
            // Find highest priority event set
            event = co_clz(field);

             // Execute corresponding handler
            (ke_evt_hdlr[event])();

             // Update the event field
            field = ke_env.evt_field;
    }
```

Table 1 API for Event

API	Description
ke_evt_set	Set one or more events in the event field.
ke_evt_clear	Clear one or more events in the event field.
ke_evt_callback_set	Register one event callback.

3.3 Messages

A message is the entity that is exchanged between two tasks. Message can load any type data as message parameter and can be any size. The structure of the message contains:

hdr	List header for chaining.
hci_type	Type of HCI data (used by the HCI only, user do not need to fill it).
hci_offset	Offset of the HCI data in the message (used by the HCI only, user do not need to fill it).
hci_len	Length of the HCI traffic (used by the HCI only, user do not need to fill it).
id	Message identifier.
dest_id	Destination task identifier.
src_id	Source task identifier.
param_len	Parameter embedded structure length.
param	Parameter embedded structure.

A message is identified by a unique ID composed of the task type and an increasing number. The following Figure 6 illustrates how the message ID is constituted.

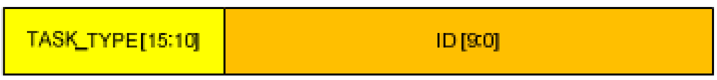


Figure 6 Message ID constitution

The element 'dest_id' is destination task unique identifier. Refer chapter 3.3 for details of task id.

The element 'src_id' is source task unique identifier. Refer chapter 3.3 for details of task id.

The element 'param' in the message structure contains message content. This field is defined by each message and shall have different content and length. The message sender is responsible for filling this field.

Transmission of messages is done in 3 steps. First the message structure must be allocated in the kernel heap by the sender task calling the function 'ke_msg_alloc()' or the macro 'KE_MSG_ALLOC' which is a convenient wrapper for ke_msg_alloc(). In order to store the message data conveniently, the pointer of the element 'param' in the message structure will be returned. Second, the user will fill the message parameter which pointer is returned by ke_msg_alloc(). Third, call ke_msg_send() to pushed message into the kernel 's message queue. The function ke_msg_send() will signal the kernel that there is at least one message in message queue by setting message exist flag.

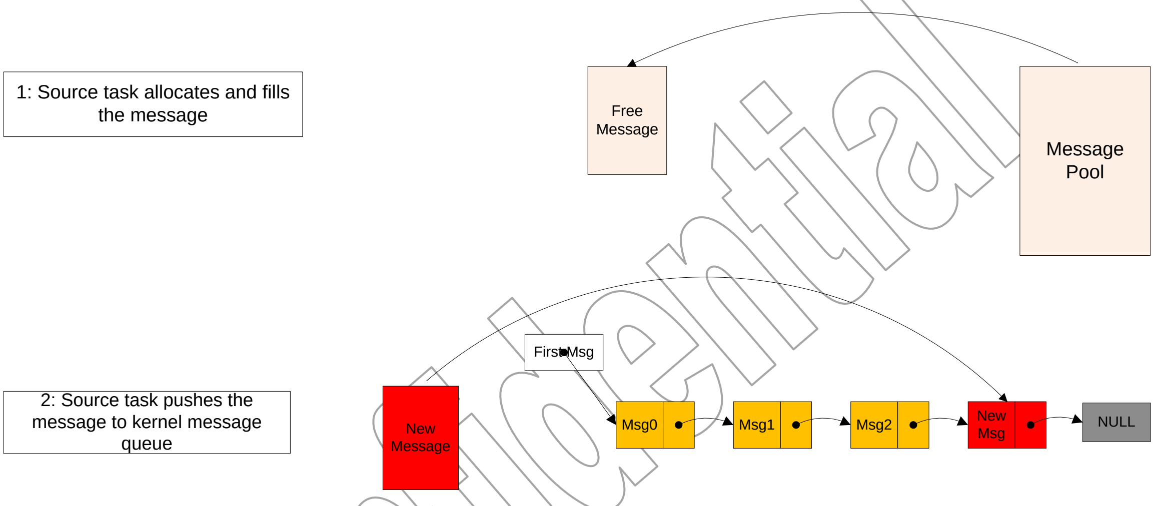


Figure 7 Message Allocation and Transmission

The following table lists a brief description of all message APIs. For detailed usage, please refer to the file 'ke_msg.h' in the example.

Table 2 API for Message Allocation and Transmission

API	Description
ke_param2msg	Convert a parameter pointer to a message pointer.
ke_msg2param	Convert a message pointer to a parameter pointer.
ke_msg_alloc	This function allocates memory for a message that has to be sent. The memory is allocated dynamically on the heap and the length of the variable parameter structure has to be provided in order to allocate the correct size.
ke_msg_send	Send a message previously allocated with any ke_msg_alloc()-like functions. The kernel will take care of freeing the message memory. Once the function have been called, it is not possible to access its data anymore as the kernel may have copied the message and freed the original memory.
ke_msg_send_front	Send a message and insert at the front of message queue.
ke_msg_send_basic	Send a message that has a zero length parameter member. No allocation is required as it will be done internally.

ke_msg_forward	Forward a message to another task by changing its destination and source tasks IDs.
ke_msg_free	Free allocated message.

3.4 Tasks

One task is defined by its task type and task descriptor. The task type is a constant value defined by the kernel and this value is unique for each task. The Table 3 lists all available task types in the QN902x. Only the profile task type can be arranged to the profile determined by application design, the other task types are fixed.

Table 3 Task Type Definition

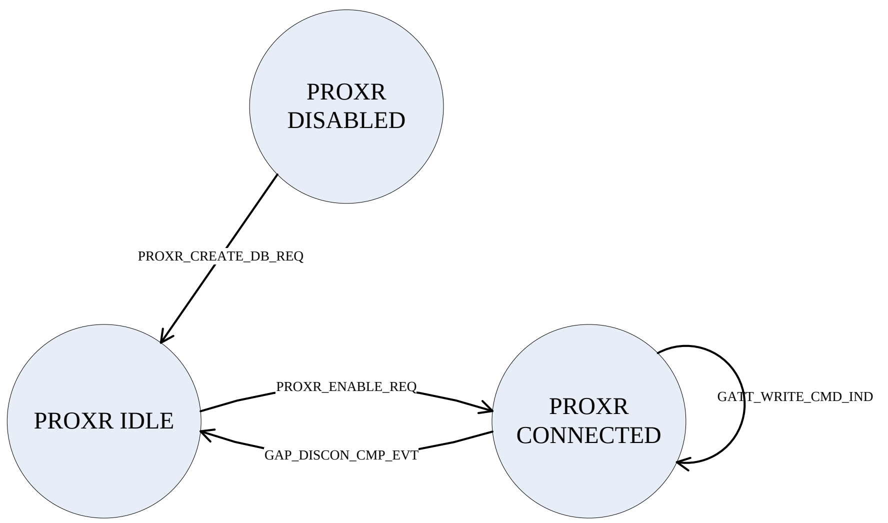
Task Type	Description
0~3	**Link layer tasks**
4~5	**L2CAP tasks**
6~7	**SMP tasks**
8~10	**ATT tasks**
11	**GATT task**
12	**GAP task**
13~20	**Profile tasks**
21	**Application task**

Each task has its own state machine and message processing handler that are saved in the task descriptor. The kernel keeps a pointer to each task descriptor that is used to handle the scheduling of the messages transmitted from a task to another one.

The structure of task descriptor contains:

state_handler	The messages that it is able to receive in each of its states
default_handler	The messages that it is able to receive in the default state
state	The current state of each instance of the task
state_max	The number of states of the task
idx_max	The number of instances of the task

The proximity reporter is used as an example to illustrate how to construct a task descriptor. The following figure is the state machine of proximity reporter task.



From the above figure Proximity Reporter task has three states. The arrows show the messages that will be processed in each state. The following snippet is created based on state machine.

state_handler

```
/// Disabled State handler definition.
const struct ke_msg_handler proxr_disabled[] =
{
    {PROXR_CREATE_DB_REQ,     (ke_msg_func_t) proxr_create_db_req_handler },
};

/// Idle State handler definition.
const struct ke_msg_handler proxr_idle[] =
{
    {PROXR_ENABLE_REQ,        (ke_msg_func_t) proxr_enable_req_handler },
};

/// Connected State handler definition.
const struct ke_msg_handler proxr_connected[] =
{
    {GATT_WRITE_CMD_IND,      (ke_msg_func_t) gatt_write_cmd_ind_handler},
};

/// Specifies the message handler structure for every input state.
const struct ke_state_handler proxr_state_handler[PROXR_STATE_MAX] =
{
    [PROXR_DISABLED]      = KE_STATE_HANDLER(proxr_disabled),
    [PROXR_IDLE]          = KE_STATE_HANDLER(proxr_idle),
    [PROXR_CONNECTED]     = KE_STATE_HANDLER(proxr_connected),
};
```

default_handler

The message GAP_DISCON_CMP_EVT is put in the default state handler. That means this message can be processed in any states.

```
/// Default State handlers definition
const struct ke_msg_handler proxr_default_state[] =
{
    {GAP_DISCON_CMP_EVT,      (ke_msg_func_t) gap_discon_cmp_evt_handler},
};

/// Specifies the message handlers that are common to all states.
const struct ke_state_handler proxr_default_handler = KE_STATE_HANDLER(proxr_default_state);
```

state

```
/// Defines the place holder for the states of all the task instances.
ke_state_t proxr_state[PROXR_IDX_MAX];
```

state_max

```
/// Possible states of the PROXR task
enum
{
    /// Disabled State
    PROXR_DISABLED,
    /// Idle state
    PROXR_IDLE,
    /// Connected state
    PROXR_CONNECTED,

    /// Number of defined states.
    PROXR_STATE_MAX
};
```

idx_max

Proximity Reporter which works as a server role is only one instance.

```
/// Maximum number of Proximity Reporter task instances
#define PROXR_IDX_MAX                    (1)
```

PROXR task descriptor

```
// Register PROXR task into kernel
void task_proxr_desc_register(void)
{
    struct ke_task_desc task_proxr_desc;

    task_proxr_desc.state_handler = proxr_state_handler;
    task_proxr_desc.default_handler=&proxr_default_handler;
    task_proxr_desc.state = proxr_state;
    task_proxr_desc.state_max = PROXR_STATE_MAX;
    task_proxr_desc.idx_max = PROXR_IDX_MAX;

    task_desc_register(TASK_PROXR, task_proxr_desc);
}
```

The following Table 4 lists a brief description of all task APIs. For detailed usage, please refer to the file 'ke_task.h'.

Table 4 API for Task Management

API	Description
ke_state_get	Retrieve the state of a task.
ke_state_set	Set the state of the task identified by its task id.
ke_msg_discard	Generic message handler to consum message without handling it in the task.
ke_msg_save	Generic message handler to return KE_MSG_SAVED without handling it in the task.
task_desc_register	Register task description into kernel.

3.5 Message Scheduler

The message scheduler provides a mechanism to transmit one or more messages to a task. The message scheduler is called in one event handler. The following snippet is a pseudo code of message scheduler.

```
void ke_task_schedule(void)
{
    while(1)
    {
        // Get a message from the queue
        msg = ke_queue_pop(queue_msg);

        if (msg == NULL) break;

        // Retrieve a pointer to the task instance data
        func = ke_task_handler_get(msg->id, msg->dest_id);

        // Call the message handler
        if (func != NULL)
        {
            msg_status = func(msg->id, ke_msg2param(msg), msg->dest_id, msg->src_id);
        }
        else
        {
            msg_status = KE_MSG_CONSUMED;
        }

        switch (msg_status)
        {
        case KE_MSG_CONSUMED:
            // Free the message
            ke_msg_free(msg);
            break;

        case KE_MSG_NO_FREE:
            break;
```

```
            case KE_MSG_SAVED:
                // The message has been saved
                // Insert it at the end of the save queue
                ke_queue_push(queue_saved, msg);
                break;
            } // switch case
        }
}
```

When the scheduler is executed, it checks if the message queue is not empty, finds the corresponding message handler, and executes the handler. The scheduler will take care of freeing processed message. If the message cannot be consumed by the destination task at this time, the message handler will return status 'KE_MSG_SAVED'. Then the scheduler holds this message in the saved message queue until the task state changes. When one task state is changed, the kernel looks for all of the messages destined to this task that have been saved and inserts them into message queue again. These messages will be scheduled at the next scheduling round.

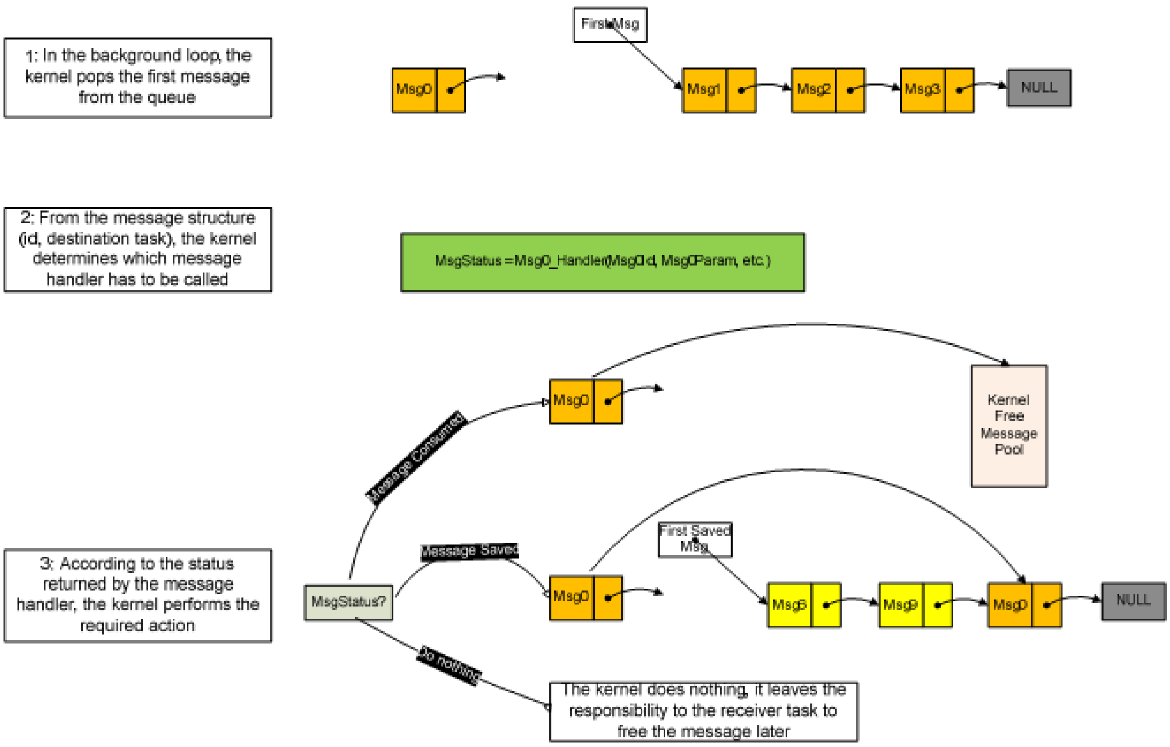


Figure 8 Scheduling Algorithm

The following Table 5 lists a brief description of all scheduler APIs. For detailed usage, please refer to the document 'QN9020 API Programming Guide'.

Table 5 API for Scheduler

API	Description
ke_schedule	Run scheduler.

3.6 Timer Scheduling

The kernel provides timer services including start and stop timer. The timer runs by absolute time counter. Timers are implemented by the mean of a reserved queue of delayed messages, and timer messages do not have parameters.

Time is defined as duration. The minimal step is 10ms. The minimal duration is 20ms and the maximal duration is 300s.

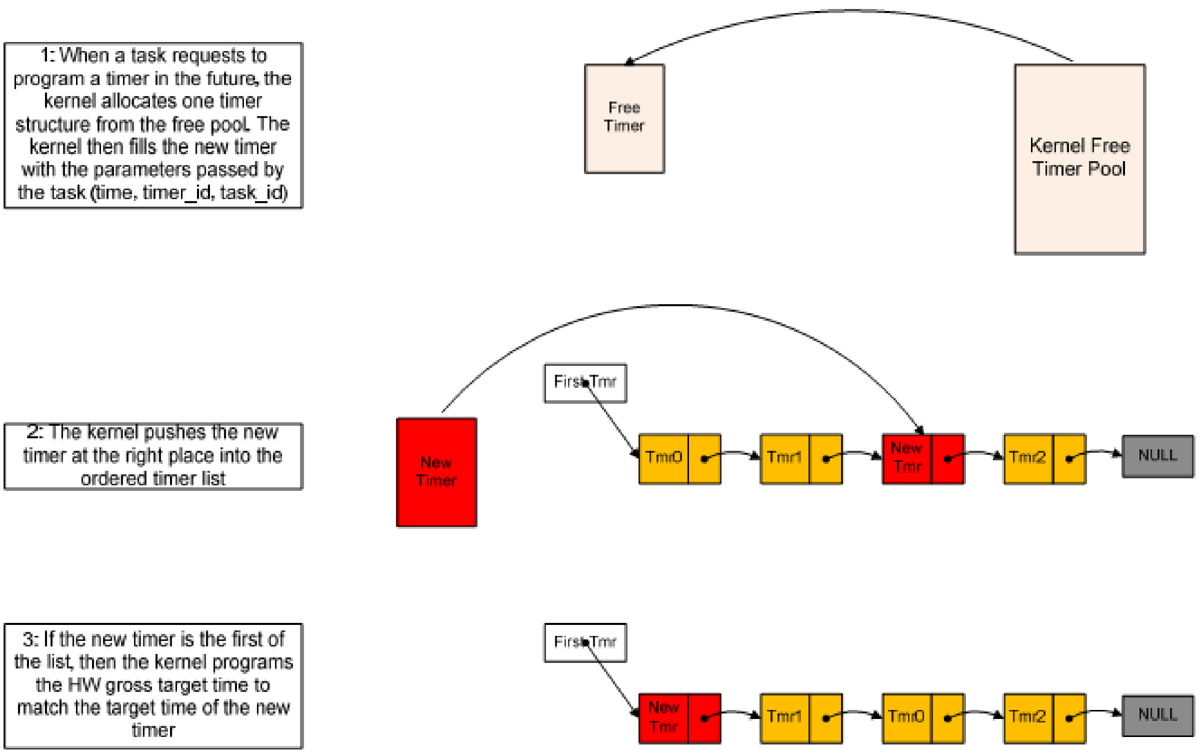


Figure 9 Timer Setting Procedure

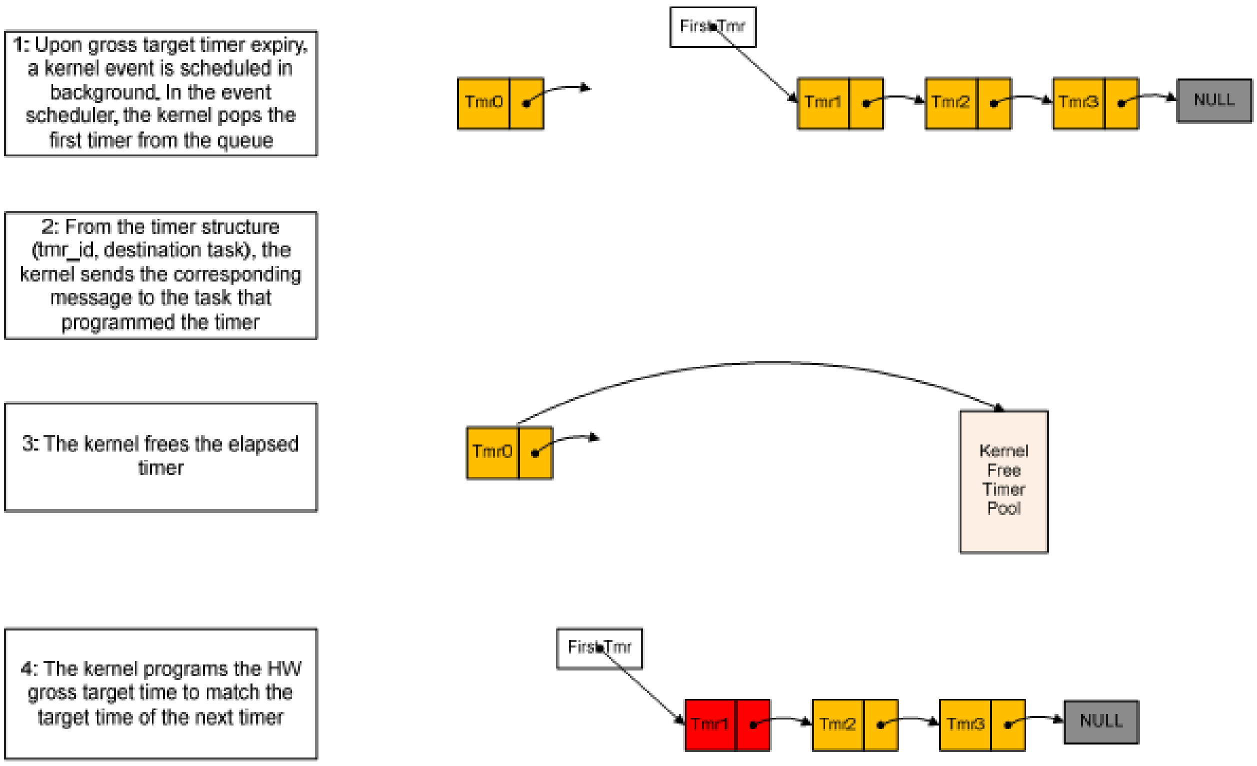


Figure 10 Timer Expiry Procedure

The following Table 6 lists a brief description of all timer APIs. For detailed usage, please refer to the document 'QN9020 API Programming Guide'.

Table 6 Timer API Definition

API	Description
ke_timer_set	The function first cancel the timer if it is already exist, then it creates a new one. The timer can be one-shot or periodic, i.e. it will be automatically set again after each trigger.
ke_timer_clear	This function search for the timer identified by its id and its task id. If found it is stopped and freed, otherwise an error message is returned.

3.7 Include Files

In order to use the services offered by the kernel the user should include the following header files:

Table 7 Include Files

File	Description
ke_msg.h	Contains the definition related to message scheduling and the primitives called to allocate, send or free a message.
ke_task.h	Contains the definition related to kernel task management
ke_timer.h	Contains the primitives called to create or delete a timer.
ble.h	Contains the definition related to scheduler.
lib.h	Contains the scheduler declaration.

4. BLE Protocol Stack

The BLE protocol stack architecture is illustrated in following Figure 11.

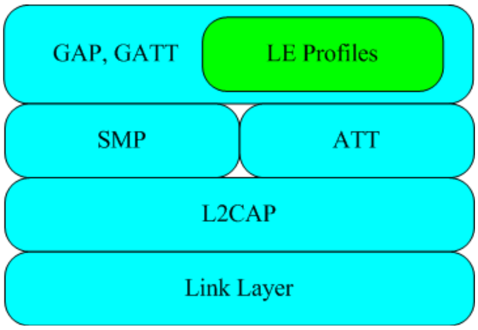


Figure 11 BLE Stack Architecture

4.1 Link Layer (LL)

Link Layer lies above the physical layer and interfaces with host layer protocols. Link layer is in charge of following features.

- Packet scheduling over the air.
- Link establishment and maintenance. The LE LL transport and L2CAP logical link between two devices is set up using the Link Layer Protocol. The Link Layer provides orderly delivery of data packets. No more than one Link Layer channel exists between any two devices. The LE LL always provides the impression of full-duplex communication channels. The LE LL performs data integrity checks and resends data until it has been successfully acknowledged or a timeout occurs. The LE LL also maintains the link supervision timeout.
- Frequency hopping calculation.
- Packet construction and recovery. The LE LL follows the little endian format to create the packet sent over the air. Only the CRC is transmitted by most significant bit first.
- Encryption and decryption.
- Link Control procedures (connection update, channel map update, encryption, feature exchange, version exchange, and termination).
- Device filtering policy applies based on device white list.

Link Layer has five possible states which are controlled by a state machine describing the operation of the link layer. The Link Layer state machine allows only one state to be active at a time.

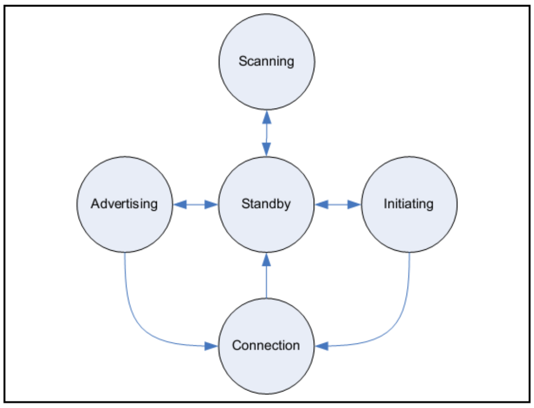


Figure 12 LL State Diagram

- **Standby**

Device has no activity. Do not transmit or receive any packet.

- **Advertising**

Device sends advertising packets
Device can receive scan request, and return scan response
Device can accept connection request

- **Scanning**

Device waits to receive advertising packets
Device can respond with scan request, and wait for scan response

- **Initiating**

Device waits to receive advertising packet from a specific device, and responds with connection request

- **Connection**

Device exchanges data. Two roles are defined, master role and slave role. When entered from the Initiating State, the Connection State shall be in the Master Role. When entered from the Advertising State, the Connection State shall be in the Slave Role.

In Wireless SoC mode and Network Processor mode, the application shall not exchange message with link layer directly. In controller mode, the application in the host processor could use standard HCI interface to communication with link layer. For details about controller mode, please refer to chapter 9.

4.2 Logical Link Control and Adaptation Protocol (L2CAP)

L2CAP lies above the link layer and interfaces with higher layer protocols. L2CAP in BLE operates only in basic mode and uses fixed channels. In the fixed channel type, only BLE signaling, security management protocol and Attribute protocol channels shall only be used.

L2CAP Layer is in charge of following features:

- Provide connection-oriented data services to upper layer protocols.
- Provide a mean to set or change the connection parameters of the data link.
- Support protocol and/or channel multiplexing (fixed channels – ATT, SMP, SIGNAL).
- The application shall not exchange message with link layer directly.

4.3 Security Manager Protocol (SMP)

The Security Manager Protocol (SMP) is in charge of BLE secure communication issues including encrypted links, identity or private addresses resolution and signed unencrypted messages. The functionalities of the SMP are enforced by clearly specified pairing and key distribution methods, and the protocol that is to be respected for their correct implementation.

- **Interface with APP**

The Application is keeper of keys which are necessary during the pairing/encrypting procedure, so all key requests are received and answered by the Application. Messages exchanged between the SMP and the APP list in the following table. The SMP tasks have handlers for these messages sent by APP. The APP task should implement handlers for these message sent by SMP. All of these message and parameter structures are defined in smpc_task.h and smpm_task.h. Please refer to the document 'QN9020 API Programming Guide' for more details.

Table 8 SMPM Message

Message	Direction	Description	Parameters	Response
SMPM_SET_KEY_REQ	APP → SMPM	Set the device keys that are unique for the device and not connection dependent.	struct smpm_set_key_req	SMPM_SET_KEY_CFM
SMPM_SET_KEY_CFM	SMPM → APP	Respond to the Application to its SMPM_SET_KEY_REQ, informing it that saving the key values was done.	struct smpm_set_key_cfm	

Table 9 SMPC Message

Message	Direction	Description	Parameters	Response
SMPC_START_ENC_REQ	APP → SMPC	Encrypt a link with a peer using known bonding information from a previous connection when pairing and bonding occurred.	struct smpc_start_enc_req	SMPC_SEC_STARTED_IND
SMPC_SEC_STARTED_IND	SMPC → APP	Response the status of a security procedure.	struct smpc_sec_started_ind	
SMPC_TK_REQ_IND	SMPC → APP	Request for TK.	struct smpc_tk_req_ind	SMPC_TK_REQ_RSP
SMPC_TK_REQ_RSP	APP → SMPC	Response for TK request.	struct smpc_tk_req_rsp	
SMPC_LTK_REQ_IND	SMPC → APP	Request for LTK.	struct smpc_ltk_req_ind	SMPC_LTK_REQ_R

				SP
SMPC_LTK_REQ_RSP	APP → SMPC	Response for LTK request.	struct smpc_ltk_req_rsp	
SMPC_IRK_REQ_IND	SMPC → APP	Request for IRK.	struct smpc_irk_req_ind	SMPC_IRK_REQ_RSP
SMPC_IRK_REQ_RSP	APP → SMPC	Response for IRK request.	struct smpc_irk_req_rsp	
SMPC_CSRK_REQ_IND	SMPC → APP	Request for CSRK.	struct smpc_csrk_req_ind	SMPC_CSRK_REQ_RSP
SMPC_CSRK_REQ_RSP	APP → SMPC	Response for CSRK request.	struct smpc_csrk_req_rsp	
SMPC_KEY_IND	SMPC → APP	Indicate the value of received bonding information from peer device.	struct smpc_key_ind	
SMPC_CHK_BD_ADDR_REQ_IND	SMPC → APP	Request to check if the address exists in application.	struct smpc_chk_bd_addr_req	SMPC_CHK_BD_ADDR_REQ_RSP
SMPC_CHK_BD_ADDR_REQ_RSP	APP → SMPC	Inform that the Bluetooth address that was requested to be checked has been found or not.	struct smpc_chk_bd_addr_rsp	
SMPC_TIMEOUT_EVT	SMPC → APP	SMP timeout event. Inform application to disconnect.	struct smpc_timeout_evt	

4.4 Attribute Protocol (ATT)

Attribute Protocol lies above L2CAP and interfaces with L2CAP and GATT. The Attribute Protocol is used to read and write the attribute values from an attribute database of a peer device, called the attribute server. To do this, firstly the list of attributes in the attribute database on the attribute server shall be discovered. Once the attributes have been found, they can be read and written as required by the attribute client. The application shall not exchange message with Attribute Protocol directly.

4.5 Generic Attribute Profile (GATT)

The GATT profile is designed to be used by an application or other LE profiles, so that a client can communicate with a server. The server contains a number of attributes, and the GATT Profile defines how to use the Attribute Protocol to discover, read, write and obtain indications of these attributes, as well as configuring broadcast of attributes.

The GATT of QN902x has complete and substantial support of the LE GATT (Core 4.0):

- Two Roles–client and server
- Configuration Exchange

- Service Discovery
- Characteristic Discovery
- Reading Characteristic
- Writing Characteristic
- Indicating Characteristic
- Notifying Characteristic
- Profile Interface

4.5.1 Interface with APP/PRF

Messages exchanged between the GATT and the APP/PRF list in the following tables. The GATT task has handlers for these messages sent by APP/PRF. The APP/PRF task should implement handlers for these message sent by GATT. All of these message and parameter structures are defined in gatt_task.h. Moreover it is recommended that the users check the document 'QN9020 API Programming Guide' for GATT APIs. This document can further provide information on GATT interface (e.g. data structures, message calling).

4.5.2 Generic Interface

The generic GATT interface includes commands and events common to GATT server and client.

Table 10 Generic Interface Message

Message	Direction	Description	Parameters	Response
GATT_CMP_EVT	GATT → PRF/APP	Complete event for GATT operation. This is the generic complete event for GATT operations.	struct gatt_cmp_evt	
GATT_TIMEOUT_EVT	GATT → PRF/APP	Timeout notification.		
GATT_READ_ATTRIBUTE_REQ	PRF/APP → GATT	Read an attribute element in local attribute server database.	struct gatt_read_attribute_req	GATT_READ_ATTRIBUTE_CMP_EVT
GATT_READ_ATTRIBUTE_CMP_EVT	GATT → PRF/APP	Complete event for read an attribute element in local attribute server database.	struct gatt_read_attribute_cmp_evt	
GATT_WRITE_ATTRIBUTE_REQ	PRF/APP → GATT	Write an attribute element in local attribute server database.	struct gatt_write_attribute_req	GATT_WRITE_ATTRIBUTE_CMP_EVT
GATT_WRITE_ATTRIBUTE_CMP_EVT	GATT → PRF/APP	Complete event for write an attribute element in local attribute server database.	struct gatt_write_attribute_cmp_evt	
GATT_RESOURCE_ACCESS_REQ	GATT → APP	Inform upper layer that a peer device request access of database resources.	struct gatt_resource_access_req	GATT_RESOURCE_ACCESS_RSP

GATT_RESOURCE_ACCESS_RSP	APP → GATT	When the response is received by GATT, peer device is able to access attribute database.	struct gatt_resource_access_rsp	

4.5.3 Configuration

This is intended for setting the Maximum Transmission Unit (MTU) of the link for GATT transactions. The client and the server will exchange this information to inform the peer of their sending bandwidth.

Table 11 Configuration Message

Message	Direction	Description	Parameters	Response
GATT_EXC_MTU_REQ	APP → GATT	Inform the peer device of the MTU size. This is an optional GATT transaction to make. If the MTU is set to the default value (23 bytes) specified in the specification, there is no need to send this command.	struct gatt_exc_mtu_req	

4.5.4 Service Discovery

Discovery of services exposed by the GATT server to the GATT client is an important interface for the GATT. Once the primary services are discovered, additional information can be accessed including characteristic and relationship discovery. The GATT provides means for the user to discover the services by group type and by UUID.

Table 12 Service Discovery Message

Message	Direction	Description	Parameters	Response
GATT_DISC_SVC_REQ	PRF/APP → GATT	Discover services exposed by peer device in its attribute database.	struct gatt_disc_svc_req	GATT_DISC_SVC_ALL_CMP_EVT GATT_DISC_SVC_ALL_128_CMP_EVT GATT_DISC_SVC_BY_UUID_CMP_EVT GATT_DISC_SVC_INCL_CMP_EVT GATT_DISC_CMP_EVT
GATT_DISC_SVC_ALL_CMP_EVT	GATT → PRF/APP	Complete event for discovery of all services.	struct gatt_disc_svc_al	

		This event will contain the list of services discovered from the attribute database of the peer.	l_cmp_evt	
GATT_DISC_SVC_ALL_128_CMP_EVT	GATT → PRF/APP	Complete event for discovery all services using 128-bit UUID.	struct gatt_disc_svc_all_128_cmp_evt	
GATT_DISC_SVC_BY_UUID_CMP_EVT	GATT → PRF/APP	Complete event for discovery of a specific service identified by UUID.	struct gatt_disc_svc_by_uuid_cmp_evt	
GATT_DISC_SVC_INCL_CMP_EVT	GATT → PRF/APP	Complete event for discovery of included services. The incl_list is represented in a union list because an entry in the include list may be represented in a 128-bit UUID	struct gatt_disc_svc_incl_cmp_evt	
GATT_DISC_CMP_EVT	GATT → PRF/APP	Complete event for GATT discovery operation.	struct gatt_disc_cmp_evt	

4.5.5 Characteristic Discovery

The GATT of QN902x provides the means to discover characteristic present in the Attribute database of the GATT server. The search interface can take different parameters, giving the user the ability to tailor its characteristic search. Some of these parameters include range handles, UUID search pattern and all characteristic group discoveries.

Table 13 Characteristic Discovery Message

Message	Direction	Description	Parameters	Response
GATT_DISC_CHAR_REQ	PRF/APP → GATT	Discover characteristics exposed by peer device in its attribute database.	struct gatt_disc_char_req	GATT_DISC_CHAR_ALL_CMP_EVT/GATT_DISC_CHAR_ALL_128_CMP_EVT/GATT_DISC_CHAR_BY_UUID_CMP_EVT/ GATT_DISC_CHAR_BY_UUID_128_CMP_EVT
GATT_DISC_CHAR_ALL_CMP_EVT	GATT → PRF/APP	Complete event for discovery all	struct gatt_disc_char_	

		characteristics exposed by peer device in its attribute database.	all_cmp_evt	
GATT_DISC_CHAR_ALL_128_CMP_EVT	GATT → PRF/APP	Complete event for discovery all characteristics using 128-bit UUID.	struct gatt_disc_char_all_128_cmp_evt	
GATT_DISC_CHAR_BY_UUID_CMP_EVT	GATT → PRF/APP	Complete event for discover specific characteristics exposed by peer device in its attribute database.	struct gatt_disc_char_by_uuid_cmp_evt	
GATT_DISC_CHAR_BY_UUID_128_CMP_EVT	GATT → PRF/APP	Complete event for discovery characteristic by UUID.	struct gatt_disc_char_by_uuid_128_cmp_evt	
GATT_DISC_CHAR_DESC_REQ	PRF/APP → GATT	Discover all characteristics within a given range of element handle.	struct gatt_disc_char_desc_req	GATT_DISC_CHAR_DESC_CMP_EVT/ GATT_DISC_CHAR_DESC_128_CMP_EVT
GATT_DISC_CHAR_DESC_CMP_EVT	GATT → PRF/APP	Complete event for discovery of characteristic descriptors within a specified range.	struct gatt_disc_char_desc_cmp_evt	
GATT_DISC_CHAR_DESC_128_CMP_EVT	GATT → PRF/APP	Complete event for discovery of characteristic descriptors within a specified range. 128-bit UUID is used.	struct gatt_disc_char_desc_128_cmp_evt	

4.5.6 Read and Write Characteristics

The GATT of QN902x provides a way for a peer characteristic to be read and written. More than just reading and writing, it has a ready interface to modify or read characteristic with different format or length.

Table 14 Read and Write Message

Message	Direction	Description	Parameters	Response
GATT_READ_CHAR_REQ	PRF/APP → GATT	Read a characteristic from peer attribute database.	struct gatt_read_char_req	GATT_READ_CHAR_RESP GATT_READ_CHAR_LONG_RESP GATT_READ_CH

				AR_MULTI_RESP GATT_READ_CHAR_LONG_DESC_RESP
GATT_READ_CHAR_RESP	GATT → PRF/APP	Read characteristic response. This will contain the value of the attribute handle element which is being queried from the read characteristic request.	struct gatt_read_char_resp	
GATT_READ_CHAR_LONG_RESP	GATT → PRF/APP	Read long characteristic response.	struct gatt_read_char_long_resp	
GATT_READ_CHAR_MULT_RESP	GATT → PRF/APP	Read multiple characteristics response.	struct gatt_read_char_mult_resp	
GATT_READ_CHAR_LONG_DESC_RESP	GATT → PRF/APP	Read long characteristic descriptor response.	struct gatt_read_char_long_desc_resp	
GATT_WRITE_CHAR_REQ	PRF/APP → GATT	Write a characteristic to peer attribute database.	struct gatt_write_char_req	GATT_WRITE_CHAR_RESP
GATT_WRITE_CHAR_RESP	GATT → PRF/APP	Write characteristic response.	struct gatt_write_char_resp	
GATT_WRITE_RELIABLE_REQ	PRF/APP → GATT	Write reliable characteristic to peer attribute database.	struct gatt_write_reliable_req	GATT_WRITE_RELIABLE_RESP
GATT_WRITE_CHAR_RELIABLE_RESP	GATT → PRF/APP	Write reliable characteristic response.	struct gatt_write_reliable_resp	
GATT_CANCEL_WRITE_CHAR_RESP	GATT → PRF/APP	Cancel write characteristic response.	struct gatt_cancel_write_char_resp	
GATT_EXECUTE_WRITE_CHAR_REQ	PRF/APP → GATT	Send an attribute execute reliable write request to peer. This is used when automatic sending of execute write reliable to peer is turned off. The command can either ask the peer to execute all the reliable writes performed	struct gatt_execute_write_char_req	GATT_CMP_EVT

		earlier, or cancel the write operation.		
GATT_WRITE_CMD_IND	ATTS → PRF	Write command indication.	struct gatt_write_cmd_ind	

4.5.7 Notify and Indication Characteristics

Characteristics can be notified and indicated. These actions originate from GATT server. Notification would not expect attribute protocol layer acknowledgement. Unlikely indication would expect a confirmation from GATT client.

Table 15 Notify and Indication Message

Message	Direction	Description	Parameters	Response
GATT_NOTIFY_REQ	PRF/APP → GATT	Notify characteristic. The GATT server does not wait for any attribute protocol layer acknowledgement.	struct gatt_notify_req	GATT_NOTIFY_CMP_EVT
GATT_NOTIFY_CMP_EVT	GATT → PRF/APP	Complete event for notification.	struct gatt_notify_cmp_evt	
GATT_INDICATE_REQ	PRF/APP → GATT	Indicate characteristic.	struct gatt_indicate_req	
GATT_HANDLE_VALUE_NOTIF	GATT → PRF/APP	Inform that a notification is received.	struct gatt_handle_value_notif	
GATT_HANDLE_VALUE_IND	GATT → PRF/APP	Inform that an indication is received.	struct gatt_handle_value_ind	
GATT_HANDLE_VALUE_CFM	GATT → PRF/APP	Inform that a confirmation is received.	struct gatt_handle_value_cfm	

4.5.8 Profile Interface

Interface for the profiles or higher layer is necessary to have efficient connection to GATT.

Table 16 Profile Interface Message

Message	Direction	Description	Parameters	Response
GATT_SVC_REG2PRF_REQ	PRF/APP → GATT	Register a SVC for indications, notifications or confirms forward	struct gatt_svc_reg2prf_req	
GATT_SVC_UNREG	PRF/APP	Unregister a SVC for	struct	

2PRF_REQ	→ GATT	indications, notifications	gatt_svc_unreg2 prf_req	

4.6 Generic Access Profile (GAP)

The Generic Access Profile (GAP) defines the basic procedures related to discovery of Bluetooth devices and link management aspects of connecting to Bluetooth devices. Furthermore, it defines procedures related to the use of different LE security levels. This profile describes common format requirements for parameters accessible on the user interface level.

The GAP of QN902x has complete and substantial support of the LE GAP (Core 4.0):

- Four Roles–central, peripheral, broadcaster and scanner
- Broadcast and Scan
- Modes–Discovery, Connectivity, Bonding
- Security with Authentication, Encryption and Signing
- Link Establishment and Detachment
- Random and Static Addresses
- Privacy Features
- Pairing and Key Generation
- BR/EDR/LE combination support ready

The GAP of QN902x supports all defined GAP roles.

- **Broadcaster**

This is a device that sends advertising events, and shall have a transmitter and may have a receiver. This is also known as Advertiser.

- **Observer**

This is a device that receives advertising events, and shall have a receiver and may have a transmitter. This is also known as Scanner.

- **Peripheral**

This is any device that accepts the establishment of an LE physical link using any of the specified connection establishment procedure in the Core specification. When the device is operating on this role, it will assume the Slave role of the link layer connection state. This device shall have both a transmitter and a receiver.

- **Central**

This is any device that initiates the establishment of a physical link. It shall assume the Master role of the link layer connection state. Similarly with the peripheral, this device shall have both a transmitter and a receiver.

4.6.1 Interface with APP

Messages exchanged between the GAP and the APP list in the following table. The GAP task has handlers for these messages sent by APP. The APP task should implement handlers for these message sent by GAP. All of these message and parameter structures are defined in gap_task.h. Moreover it is recommended that the user check the document 'QN9020 API Programming Guide' for GAP APIs. This document can further provides information on GAP interface (e.g. data structures, message calling).

4.6.2 Generic Interface

The generic GAP interface includes commands which are device setup and information gathering related control. These commands are available for any BLE GAP role.

Table 17 Generic Interface Message

Message	Direction	Description	Parameters	Response
GAP_RESET_REQ	APP → GAP	Reset the BLE stack including the link layer and the host.	None	GAP_RESET_REQ_CMP_EVT
GAP_RESET_REQ_CMP_EVT	GAP → APP	Complete event for device driver.	struct gap_event_common_cmd_complete	
GAP_SET_DEVNAME_REQ	APP → GAP	Set the device name as seen by remote device.	struct gap_set_devname_req;	GAP_SET_DEVNAME_REQ_CMP_EVT
GAP_SET_DEVNAME_REQ_CMP_EVT	GAP → APP	Complete event for set the device name.	struct gap_event_common_cmd_complete	
GAP_READ_VER_REQ	APP → GAP	Read the version information of the BLE stack.	None	GAP_READ_VER_REQ_CMP_EVT
GAP_READ_VER_REQ_CMP_EVT	GAP → APP	Complete event for read the version information of the BLE stack.	struct gap_read_ver_req_cmp_evt	
GAP_READ_BDADDR_REQ	APP → GAP	Read the Bluetooth address of the device.	None	GAP_READ_BDADDR_REQ_CMP_EVT
GAP_READ_BDADDR_REQ_CMP_EVT	GAP → APP	Complete event for read the Bluetooth address of the device.	struct gap_read_bdaddr_req_cmp_evt	
GAP_SET_SEC_REQ	APP → GAP	Set security level of the device. It is advisable to set the security level as soon as the device starts.	struct gap_set_sec_req	GAP_SET_SEC_REQ_CMP_EVT
GAP_SET_SEC_REQ_CMP_EVT	GAP → APP	Complete event for set security level.	Struct gap_set_sec_req_cmp_evt	
GAP_READY_EVT	GAP → APP	Inform the APP that the GAP is ready.	None	
GAP_ADD_KNOWN_ADDR_IND	GAP → APP	Indicate address to remember as known device.	struct gap_add_known_a	

			ddr_ind	

4.6.3 Device Mode Setting

The mode setting interface is used to put the device in a GAP specific mode.

Table 18 Device Mode Setting Message

Message	Direction	Description	Parameters	Response
GAP_SET_MODE_REQ	APP → GAP	Set the device mode.	struct gap_set_mode_req	GAP_SET_MODE_REQ_CMP_EVT
GAP_SET_MODE_REQ_CMP_EVT	GAP → APP	Complete event for set mode request.	struct gap_event_common_cmd_complete	

4.6.4 White List Manipulation

The white list manipulation interface pertains to commands that control the white list components and information inside the BLE device. Control includes adding and removing of the addresses in the lists –public and private. Reading of white list size is included in this sub group.

Table 19 White List manipulation Message

Message	Direction	Description	Parameters	Response
GAP_LE_RD_WLST_SIZE_REQ	APP → GAP	Read the white list size of the local device.	None	GAP_LE_RD_WLST_SIZE_CMD_CMP_EVT
GAP_LE_RD_WLST_SIZE_CMD_CMP_EVT	GAP → APP	Complete event for read the white list size of the local device.	struct gap_rd_wlst_size_cmd_complete	
GAP_LE_ADD_DEV_TO_WLST_REQ	APP → GAP	Add device to white list.	struct gap_le_add_dev_to_wlst_req	GAP_LE_ADD_DEV_TO_WLST_REQ_CMP_EVT
GAP_LE_ADD_DEV_TO_WLST_REQ_CMP_EVT	GAP → APP	Complete event for add device to white list.	Struct gap_event_common_cmd_complete	
GAP_LE_RMV_DEV_FRM_W	APP →	Remove	struct	GAP_LE_RMV_DEV

LST_REQ	GAP	device from white list.	gap_le_rmv_dev_frm_wlst_req	_FRM_WLST_REQ_CMP_EVT
GAP_LE_RMV_DEV_FRM_WLST_REQ_CMP_EVT	GAP → APP	Complete event for remove device from white list	Struct gap_event_common_cmd_complete	

4.6.5 LE Advertisement and Observation

LE advertising mode allows a device to send advertising data in a unidirectional connectionless manner. In a similar fashion, the LE scanning mode allows a device to receive advertising data. For a broadcaster, scanning is not possible. Advertising on the other hand is not possible for a Scanner. Central and Peripheral devices shall support both advertising and scanning features.

Table 20 Advertisement and Observation Message

Message	Direction	Description	Parameters	Response
GAP_ADV_REQ	APP → GAP	Start or stop data broadcast.	struct gap_adv_req	GAP_ADV_REQ_CMP_EVT
GAP_ADV_REQ_CMP_EVT	GAP → APP	Complete event for start or stop data broadcast.	struct gap_event_common_cmd_complete	
GAP_SCAN_REQ	APP → GAP	Set scanning parameters and start or stop observing.	struct gap_scan_req	GAP_SCAN_REQ_CMP_EVT
GAP_SCAN_REQ_CMP_EVT	GAP → APP	Complete event for set scanning parameters and start or stop observing.	struct gap_event_common_cmd_complete	
GAP_ADV_REPORT_EVT	GAP → APP	Indicate advertising report and data.	struct gap_adv_report_evt	

4.6.6 Name Discovery and Peer Information

The name discovery procedure is used to retrieve the name of the peer device. Normally this is performed when the name of the device is not acquired either from limited discovery procedure or general discovery procedure. Once the name of the peer device is discovered by GATT client of the local device (read by Characteristic UUID), the connection may be terminated.

Table 21 Name Discovery and Peer Information Message

Message	Direction	Description	Parameters	Response
GAP_NAME_REQ	APP → GAP	Find out the user friendly name of peer device.	struct gap_name_req	GAP_NAME_REQ_CMP_EVT
GAP_NAME_REQ_CMP_EVT	GAP → APP	Complete event for name request. The name of the romote device will be returned.	struct gap_name_req_cmp_evt	
GAP_RD_REM_VER_INFO_REQ	APP → GAP	Read version information of peer device.	struct gap_rd_rem_ver_info_req	GAP_RD_REM_VER_INFO_CMP_EVT
GAP_RD_REM_VER_INFO_CMP_EVT	GAP → APP	Complete event for read remote version information.	struct gap_rd_rem_ver_info_cmp_evt	
GAP_LE_RD_REMOTE_FEAT_REQ	APP → GAP	Read remote features.	struct gap_le_rd_rem_used_feats_req	GAP_LE_RD_REMOTE_FEAT_REQ_CMP_EVT
GAP_LE_RD_REMOTE_FEAT_REQ_CMP_EVT	GAP → APP	Complete event for read remote features.	struct gap_le_rd_remote_feat_req_cmp_evt	

4.6.7 Device Discovery

This device discovery interface searches for devices within range, with consideration on specific parameters. There are three types of inquiry:

- 0x00 General Inquiry - The advertising data must have the flag option and its bit set to GEN DISC
- 0x01 Limited Inquiry - The advertising data must have the flag option and its bit set to LIM DISC
- 0x02 Known Device Inquiry - Received advertising data will be filtered by GAP and will return to the upper layer only those devices which are known and recognized by the device.

Table 22 Device Discovery Message

Message	Direction	Description	Parameters	Response
GAP_DEV_INQ	APP	Search devices	struct gap_dev_inq_req	GAP_DEV_INQ_REQ_C

_REQ	→ GAP	within range.		MP_EVT GAP_DEV_INQ_RESULT_EVT GAP_KNOWN_DEV_DISC_RESULT_EVT
GAP_DEV_INQ_REQ_CMP_EVT	GAP → APP	Complete event of device search.	struct gap_event_common_cmd_complete	
GAP_DEV_INQ_RESULT_EVT	GAP → APP	Return result of the inquiry command.	struct gap_dev_inq_result_evt	
GAP_KNOWN_DEV_DISC_RESULT_EVT	GAP → APP	Return known device result of the inquiry command.	struct gap_known_dev_disc_result_evt	

4.6.8 Connection Establishment and Detachment

The connection modes and procedures allow a device to establish a link to another device. Only central and peripheral devices can perform connection and disconnection procedure.

Table 23 Connection Establishment and Detachment Message

Message	Direction	Description	Parameters	Response
GAP_LE_CREATE_CONN_REQ	APP → GAP	Establish LE connection. This is initiated by central device, which will become the master of the link.	struct gap_le_create_conn_req	GAP_LE_CREATE_CONN_REQ_CMP_EVT
GAP_LE_CREATE_CONN_REQ_CMP_EVT	GAP → APP	Complete event for LE create and cancel connection establishment.	struct gap_le_create_conn_req_cmp_evt	
GAP_LE_CANCEL_CONN_REQ	APP → GAP	Cancel LE connection establishment. This is initiated when there is a currently existing connection request attempt by the central device to a peripheral device.	None	GAP_CANCEL_CONN_REQ_CMP_EVT
GAP_CANCEL_CONN_REQ_CMP_EVT	GAP → APP	Complete event of cancel LE connection establishment.	struct gap_event_common_cmd_complete	

GAP_DISCON_REQ	APP →GAP	Destroy an existing LE connection.	struct gap_discon_req	GAP_DISCON_CMP_EVT
GAP_DISCON_CMP_EVT	GAP → APP/PRF	Complete event for LE connection detachment.	struct gap_discon_cmp_evt	

4.6.9 Random Addressing

The use of random address is to tighten security of the LE transactions. The GAP provides an interface to generate random address (static, resolvable or non-resolvable addresses) and to set the random address to the link layer.

Table 24 Random Addressing Message

MessageRandom Addressin	Direction	Description	Parameters	Response
GAP_SET_RANDOM_ADDR_REQ	APP → GAP	Set random address in link layer.	struct gap_set_random_addr_req	GAP_SET_RANDOM_ADDR_REQ_CMP_EVT
GAP_SET_RANDOM_ADDR_REQ_CMP_EVT	GAP → APP	Complete event of set random address command.	struct gap_set_random_addr_req_cmp_evt	

4.6.10 Privacy Setting

The privacy feature of the LE GAP provides a level of protection which makes it harder for an attacker to track a device over a period of time. All roles have privacy implementation specific as mandated by the Core specification.

Table 25 Privacy Setting Message

Message	Direction	Description	Parameters	Response
GAP_SET_RECON_ADDR_REQ	APP → GAP	Set reconnection address command. This will be set by the central device to the reconnection address attribute of the peripheral device.	struct gap_set_recon_addr_req	GAP_SET_RECON_ADDR_REQ_CMP_EVT
GAP_SET_RECON_ADDR_REQ_CMP_EVT	GAP →APP	Complete event for set reconnection address command.	struct gap_set_recon_addr_req_cmp_evt	
GAP_SET_PRIVACY_REQ	APP → GAP	Set privacy feature of the device.	struct gap_set_privacy_req	GAP_SET_PRIVACY_REQ_CMP_EVT
GAP_SET_PRIVACY_REQ_CMP_EVT	GAP → APP	Complete event for set privacy feature of the device.	struct gap_event_common_cmd_complete	

GAP_SET_PH_PRIVACY_REQ	APP → GAP	Set the privacy settings of the peer peripheral device. This will cause a sending of characteristic write attribute request to the peer, to change the value of the privacy flag in the attribute database.	struct gap_set_ph_privacy_req	GAP_SET_PH_PRIVACY_REQ_CMP_EVT
GAP_SET_PH_PRIVACY_REQ_CMP_EVT	GAP → APP	Complete event for privacy setting of the peripheral device.	struct gap_event_common_cmd_complete	

4.6.11 Pair and Key Exchange

Pairing allows two linked devices to exchange and store security and identity information for building a trusted relationship. To start the creation of a trusted relationship, either the central or peripheral will issue bonding request. Bonding should occur only when these devices which intended to pair are in bondable mode. During bonding, set of parameters are exchanged and scrutinized for compatibility between these devices. Input and Output capabilities, authentication requirements, key distribution parameters are some of the values which are shared and exchanged during the initial portion of the pairing procedure. If privacy is enabled, IRKs of both devices should be exchanged. At any time, pairing procedure can be aborted (due to insufficient bonding requirements, etc).

Table 26 Link Security Status

Value	Link Status	Description
0x00	GAP_LK_SEC_NONE	No security imposed in the link
0x01	GAP_LK_UNAUTHENTICATED	Link is un-authenticated
0x02	GAP_LK_AUTHENTICATED	Link is authenticated
0x04	GAP_LK_AUTHORIZED	Link is authorized
0x08	GAP_LK_BONDED	Bonding information exists

Table 27 Pair and Key Exchange Message

Message	Direction	Description	Parameters	Response
GAP_BOND_REQ	APP → GAP	Initiate bonding procedure. The bonding request can originate from either central or peripheral.	struct gap_bond_req	GAP_BOND_REQ_CMP_EVT

GAP_BOND_REQ_CMP_EVT	GAP → APP	Complete event for bonding command. This will return the status of the operation, if the bonding procedure has been successful.	struct gap_bond_req_cmp_evt	
GAP_BOND_RESP	GAP → APP	Answer to bond request from peer device. This message will contain bonding information like input/output capabilities, authentication requirements, key distribution preferences, etc.	struct gap_bond_resp	
GAP_BOND_REQ_IND	GAP → APP	Indicate bonding request from peer. The application needs to send GAP_BOND_RESP to the GAP to indicate response to the bonding request.	struct gap_bond_req_ind	
GAP_BOND_INFO_IND	APP → GAP	Retrieve bonding information. The application informs the GAP on the bonding status of the device.	struct gap_bond_info_ind	

4.6.12 Parameter Update

The connection parameter update is a way for a connected device to change the link layer connection parameters set up during connection establishment. This procedure is only available for central and peripheral devices. When central wants to update the connection parameters of the link, it will directly send a message to the link layer. If the peripheral wants to change the connection parameters of the link, it needs to send the request via L2CAP and inform the central device about the desired parameters. The central will decide on whether to accept the connection parameters. Furthermore, it will be the central which will send the request to its link layer block, for the update of the connection parameters of the link.

Table 28 Parameter Update Message

Message	Direction	Description	Parameters	Response
GAP_PARAM_UPDATE_REQ	APP → GAP	Change the current connection parameter. The peripheral is the only one that can send this request. The central device, upon receiving this request will decide if it will accept the request of not. The central will eventually change the current connection parameters by issuing	struct gap_param_update_req	GAP_PARAM_UPDATE_RESP

		GAP_CHANGE_PARAM_REQ.		
GAP_PARAM_UPDATE_RESP	GAP → APP	Indicate parameter update response from peer device.	struct gap_param_update_resp	
GAP_PARAM_UPDATE_REQ_IND	GAP → APP	Indicate parameter update request from peer device.	struct gap_param_update_req_ind	GAP_CHANGE_PARAM_REQ
GAP_CHANGE_PARAM_REQ	APP → GAP	Master sends parameter update change. This command is sent in two occasions: 1. Device receives connection parameter update from slave, 2. Device wants to change the current connection parameters.	struct gap_change_param_req	GAP_CHANGE_PARAM_REQ_CMP_EVT
GAP_CHANGE_PARAM_REQ_CMP_EVT	GAP → APP	Complete event for master which sent parameter update change.	struct gap_change_param_req_cmp_evt	

4.6.13 Channel Map Update

The LE controller of the master may receive some channel classification data from the host and can trigger performing channel update. This will ensure that the channels for use in the Bluetooth transactions are available and not used by WLAN or other technologies simultaneously with the controller.

Table 29 Channel Map Update Message

Message	Direction	Description	Parameters	Response
GAP_CHANNEL_MAP_REQ	APP → GAP	Central role can either read or update the current channel map. Peripheral role can only read it.	struct gap_channel_map_req;	GAP_CHANNEL_MAP_CMP_EVT
GAP_CHANNEL_MAP_CMP_EVT	GAP → APP	Complete event of the channel map update operation.	struct gap_channel_map_cmp_evt;	

4.6.14 RSSI

Table 30 RSSI Message

Message	Direction	Description	Parameters	Response
GAP_READ_RSSI_REQ	APP → GAP	Read RSSI value.	struct gap_read_rssi_req	GAP_READ_RSSI_REQ_CMP_EVT
GAP_READ_RSSI_REQ_CMP_EVT	GAP → APP	Complete event for read RSSI value.	struct gap_read_rssi_req_cmp_evt	

4.7 Include Files

In order to use the services offered by the BLE protocol the user should include the following header files:

Table 31 Include Files

File	Description
llc_task.h	Contains LLC data structure quoted by GAP.
llm_task.h	Contains LLM data structure quoted by GAP.
smpc.h	Contains the definition related to security manager protocol.
smpc_task.h	Contains the message definitions and message parameters which exchange between SMPC task and APP task.
smpm_task.h	Contains the message definitions and message parameters which exchange between SMPM task and APP task.
attc_task.h	Contains the definitions quoted by GAP.
attm.h	Contains ATT defines and data structures quoted by upper layer.
attm_cfg.h	Contains the definitions quoted by upper layer.
atts.h	Contains ATT defines and data structures quoted by upper layer.
atts_db.h	Contains the primitives called to operate ATT database.
atts_util.h	Contains utility functions for ATT.
gatt.h	Contains the definition related to GATT.
gatt_task.h	Contains the message definitions and message parameters which exchange between GATT task and APP/PRF task.
gap.h	Contains the definition related to GAP.
gap_cfg.h	Contains configurable value related to GAP.
gap_task.h	Contains the message definitions and message parameters which exchange between GAP task and APP task.
co_bt.h	Contains common BLE defines.
co_error.h	Contains error codes in BLE messages.
co_list.h	Contains definitions related to list management.
co_utils.h	Contains common utilities.

5. Bootloader

The QN902x contains a tiny bootloader which is capable of:

1. Downloading an application via UART/SPI and programming the application image to the flash in QN902x.
2. Downloading an application to internal SRAM via UART/SPI and executing the application directly.
3. Loading an application located in Flash to internal SRAM and executing the application.

There are two modes in the bootloader (ISP Mode and Load Mode). The ISP mode assures correct information download and the Load Mode is responsible for correctly loading application from Flash to internal SRAM. When the QN902x is powered on or reset, the bootloader is activated firstly. It looks for a connection command from UART and SPI interface for a while to determine which mode to go.

If the connection command is found, the bootloader enters into Program Mode. It starts the ISP command parser which is implemented to process host commands (see chaptor 5.4.3 for details), and then the host could send the ISP commands to download application, to verify correction of downloaded application, to branch to the application entry point and to finish other features provided by bootloader.

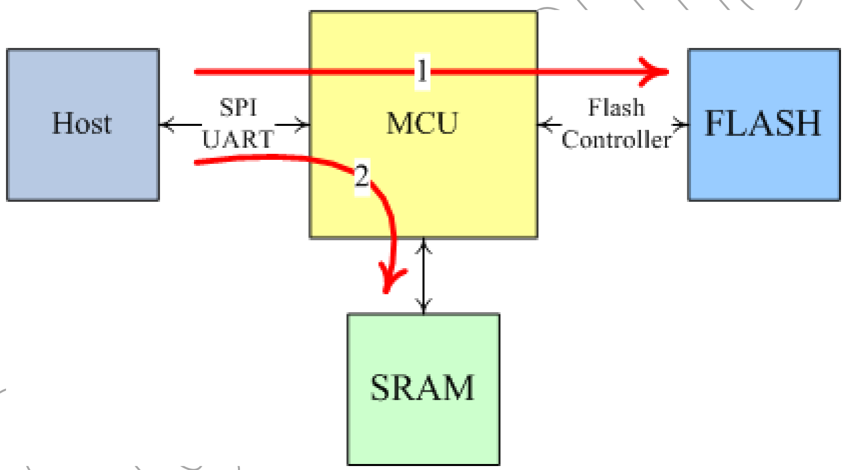


Figure 13 ISP mode

Branch 1: Bootloader downloads APP image to flash.
Branch 2: Bootloader downloads APP image to SRAM.

If no connection command found from UART and SPI interface, the bootloader enters into Load Mode. The bootloader copies application stored in the flash to internal SRAM, and then jump to the address of application entry point in the SRAM.

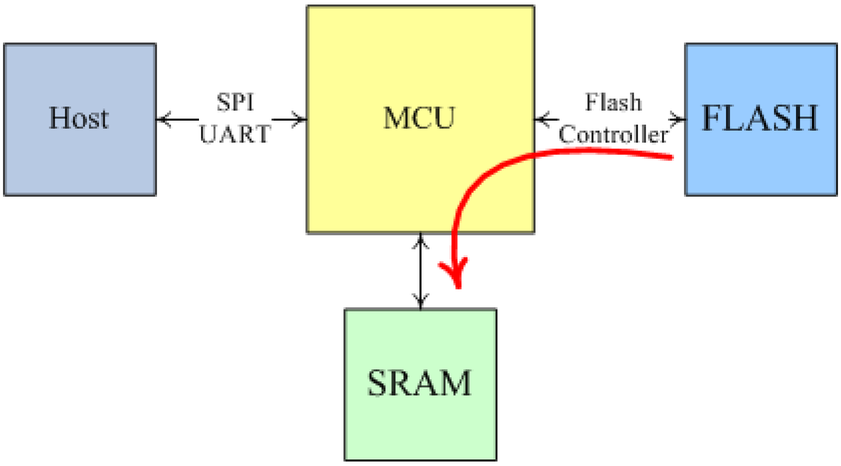


Figure 14 Load Mode

5.1 Flash Arrangement

Quintic QN902x has a 128K bytes flash. The flash is divided into four parts (NVDS area, bootloader information area, application area and NVDS backup area).

The NVDS area occupies 4k bytes Flash space from address 0x0 to 0xfff, and NVDS backup area occupies 4k bytes Flash space from address 0x1e000 to 0x1ffff. The details refer to Chapter 6.

The bootloader information area occupies 256 bytes Flash space from address 0x1000 to 0x10ff. This area stores some important information needed by the bootloader, such as storage address of application, application size and so on. This area is prohibited storing application.

The application area is from address 0x1100 to the end of the Flash. The starting address of the application is not fixed, which is easily configured by ISP command.

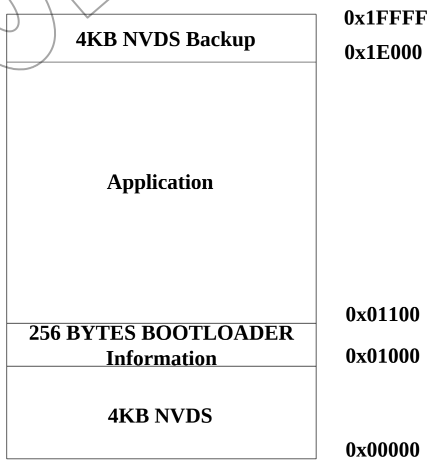


Figure 15 Flash Address Map

5.2 Peripherals Used in the Bootloader

- GIPIO
- TIMER0
- UART0
- SPI1
- Flash Controller

Bootloader uses above peripherals, therefore application should set these peripherals in a correct state. Although the pin-mux feature allows that the peripheral interface can be mapping to different pin, bootloader use fixed pin for UART0 and SPI1. QN902x's PIN0_0 and PIN1_7 are used as UART ISP interface; PIN1_0, PIN1_1, PIN1_2 and PIN1_3 are used as SPI ISP interface.

5.3 Program Protection

All of the data and code in the Flash are encrypted. Even if the cracker can bypass the flash controller in the QN902x and directly access the Flash, they cannot read the correct data. In order to prevent from stealing code through SWD interface, the QN902x provides Flash and SRAM lock feature. After PROTECT_CMD is sent to bootloader, the bootloader will shut down the way SWD accessing Flash and SRAM.

Notes:

1. In the developing phase the SWD interface is used to debug. Do not set PROTECT_CMD.
2. The application executed in the SRAM can always access Flash and SRAM.

5.4 ISP Protocol Description

5.4.1 ISP Interface Requirements

The QN902x provides two kinds of interface for interactive ISP protocol PDU, which are UART and SPI. The frame format requirement as below:

- **UART Frame Format**
  1bit start + 8bit data + 1bit stop
- **SPI Frame Format**
  8 bit data width
  MSB transmitting first
  Mode 0(CPOL: 0, CPHA: 0)

5.4.2 ISP PDU Format

UART and SPI interface both use the same ISP PDU format when downloading application. The format of PDU is defined as below:

Host TX	HeadCode	Command	Data Length(3 bytes)	Data(N)	CRC(2bytes)
BL Confirm	Confirm				
BL EXE Result	Result				

- Head Code is the first byte of a PDU, which is transmitted first. The Head Code is defined 0x71.
- Date Length is byte number of the Data field. Do not include CRC field.
- Data is the payload transmitting in the PDU and it must be word align.
- CRC is the result of 16-bits CRC which covers Command field, Data Length field and Data field. The CRC16 polynomial is X16+X12+X5+X0.
- Confirm is sent by bootloader to acknowledge whether the PDU has been received correctly. 0x01 means the correct reception otherwise send 0x02 to host.
- Result is sent by bootloader when the command PDU needs execution result return. 0x03 means the command is executed successfully; otherwise 0x04 is sent to host.

5.4.3 ISP Commands

All of the ISP commands are listed in below table.

Table 32 ISP Commands

CMD Code	UART Commands	Functions
0x33	B_C_CMD	Build connection with bootloader.
0x34	SET_BR_CMD	Set UART baud rate used in ISP mode.
0x35	SET_FLASH_CLK_CMD	Set clock frequency used by QN902x's flash.
0x36	RD_BL_VER_CMD	Read bootloader version.
0x37	RD_CHIP_ID_CMD	Read the chip number of QN902x.
0x38	RD_FLASH_ID_CMD	Read flash ID of QN902x.
0x39	SET_APP_LOC_CMD	Set application routine download location, internal SRAM or Flash.
0x3A	SETUP_FLASH_CMD	Set the flash operation commands.
0x3B	SET_ST_ADDR_CMD	Set the start address of Read, Program, Erase and Verify commands.
0x3C	SET_APP_SIZE_CMD	Set the application size.
0x3E	SET_APP_CRC_CMD	Set the CRC result of verifying application.
0x40	SET_APP_IN_FLASH_ADDR_CMD	Set the starting address of application storage location.
0x42	SE_FLASH_CMD	Sector erase flash
0x43	BE_FLASH_CMD	Block erase flash
0x44	CE_FLASH_CMD	Chip erase flash
0x45	PROGRAM _CMD	Download.
0x46	RD_CMD	Read NVDS.
0x47	VERIFY_CMD	Verify the application.
0x48	PROTECT_CMD	Enter into protect mode.
0x49	RUN_APP_CMD	Run application.
0x4A	REBOOT_CMD	Reboot system. (software reset)
0x4B	WR_RANDOM_DATA_CMD	Write a random number to Bootloader.

0x4C	SET_APP_IN_RAM_ADDR_CMD	Set the starting address of application location in the SRAM.
0x4D	SET_APP_RESET_ADDR_CMD	Set the address of application entry point.

The procedure of building connection with bootloader

Host TX	B_C_CMD
BL Confirm	Confirm

This format of B_C_CMD is different from the other ISP commands. There is only one byte in this command, no Head Code and Data Length, Data and CRC. The host continuously sends connection command until receive the confirmation which represents that the bootloader entered into ISP mode.

The procedure of setting UART baud rate

Host TX	0x71	SET_BR_CMD	4	V0~V7	V8~V15	V16~V23	V24~V31	CRC
BL Confirm	Confirm							

The payload in SET_BR_CMD is one word data which represent the value of UART0 baud rate register. Refer to QN902x data sheet for details. After SET_BR_CMD is confirmed the new baud rate will be used for Subsequent PDU exchange.

The procedure of setting Flash clock

Host TX	0x71	SET_FLASH_CLK_CMD	4	C0	C1	C2	C3	CRC
BL Confirm	Confirm							
BL EXE Result	EXE Result							

The payload in SET_FLASH_CLK_CMD is one word data which represent the Flash clock frequency and the unit is Hz. The range of clock frequency is from 100000Hz to 16000000Hz. Bootloader will return an execution result when it has already set the Flash clock.

The procedure of reading bootloader version number

Host TX	0x71	RD_BL_VER_CMD	0	CRC				
BL Confirm	Confirm							
BL Return	0x71	RD_BL_VER_CMD	4	Ver0	Ver1	Ver2	Ver3	CRC

When bootloader receives RD_BL_VER_CMD command, firstly it returns confirmation, and then returns a PDU including bootloader version information.

The procedure of reading Chip ID

Host TX	0x71	RD_CHIP_ID_CMD	0	CRC				
BL Confirm	Confirm							
BL Return	0x71	RD_CHIP_ID_CMD	4	ID0	ID1	ID2	ID3	CRC

When bootloader receives RD_CHIP_ID_CMD command, firstly it returns confirmation, and then returns a PDU including QN902x Chip ID.

The procedure of reading Flash ID

Host TX	0x71	RD_FLASH_ID_CMD	0	CRC				
BL Confirm	Confirm							
BL Return	0x71	RD_FLASH_ID_CMD	4	ID0	ID1	ID2	ID3	CRC

When bootloader receives RD_FLASH_ID_CMD command, firstly it returns confirmation, and then returns a PDU including Flash ID.

The procedure of setting application location (Flash or SRAM)

BL Return	0x71	SET_CD_LOC_CMD	4	L0	L 1	L 2	L 3	CRC
BL Confirm	Confirm							

The command SET_CD_LOC_CMD indicates where the following code download, Flash or SRAM. The payload in SET_FLASH_CLK_CMD is one word data which represent the location of following code. 0 means SRAM, 1 means Flash. The default download location is Flash in the bootloader.

The procedure of setting Flash operation commands

Host TX	0x71	SETUP_FLASH_CMD	8	CMD0	…	CMD7	CRC
BL Confirm	Confirm						
BL EXE Result	EXE Result						

When the internal Flash is not existent, the Flash can be connected outside of the chip. In this case the type of Flash may be variety, and the Flash operation commands may be a little different from each other. The command SETUP_FLASH_CMD is used to set proper Flash operation code. Then the bootloader can access the Flash correctly.

The payload meaning is shown as below table.

CMD Index	Flash Commands
CMD0	RDSR_CMD: Read Status Register
CMD1	WREN_CMD: Write Enable
CMD2	SE_CMD: Sector Erase
CMD3	BE_CMD: Block Erase
CMD4	CE_CMD: Chip Erase
CMD5	DPD_CMD: Deep Power Down
CMD6	RDPD_CMD: Release Deep Power Down
CMD7	Reserved

Bootloader will return an execution result when Bootloader has executed this command.

The procedure of setting the starting address of Read, Program and Erase command

Host TX	0x71	SET_ST_ADDR_CMD	4	A0~A7	A8-A15	A16-A23	A24-A31	CRC
BL Confirm	Confirm							

Host must set the operation address once before Reading, Programming and Erasing operations. Bootloader will increase address automatically. So it is no necessary to set operation address before the next command of the same operation.

The procedure of setting application size

Host TX	0x71	SET_APP_SIZE_CMD	4	S0	S1	S2	S3	CRC
BL Confirm	Confirm							
BL EXE Result	EXE Result							

The payload in SET_APP_SIZE_CMD is one word data which represent the size of the application. This number will be saved in the flash for next loading and also for CRC calculation.

The procedure of setting the CRC result of application

Host TX	0x71	SET_APP_CRC_CMD	4	C0	C1	C2	C3	CRC
BL Confirm	Confirm							
BL EXE Result	EXE Result							

The payload in SET_APP_CRC_CMD is one word data which represent the CRC result of the application. This number will be compared with the CRC which is calculated by bootloader and then complete the verification procedure.

The procedure of setting the starting address of application storage location in the Flash

Host TX	0x71	SET_APP_IN_FLASH_ADDR_CMD	4	A0	A1	A2	A3	CRC
BL Confirm	Confirm							
BL EXE Result	EXE Result							

The application shall not occupy the NVDS area and the bootloader information area, so the address must be bigger and equal than 0x1100. Bootloader will return an execution result when Bootloader has executed this command.

The procedure of Flash sector erasing

Host TX	0x71	SE_FLASH_CMD	4	SN0	SN 1	SN 2	SN 3	CRC
BL Confirm	Confirm							
BL EXE Result	EXE Result							

The payload in SE_FLASH_CMD is one word data which represent the number of section will be erased. The address of the first section is set by SET_ST_ADDR_CMD. The size of each is fixed to 4KB.

The procedure of Flash block erasing

Host TX	0x71	BE_FLASH_CMD	8	BS0	BS1	BS2	BS 3
BN0	BN1	BN2	BN3	CRC			
BL Confirm	Confirm						
BL EXE Result	EXE Result						

The payload in BE_FLASH_CMD is two words data. The first word represents the size of each block and the second word represents the number of block will be erased. The unit of block size is byte. The address of the first block is set by SET_ST_ADDR_CMD.

The procedure of Flash chip erasing

Host TX	0x71	CE_FLASH_CMD	0	CRC
BL Confirm	Confirm			
BL EXE Result	EXE Result			

This command is used to erase all of the Flash.

The procedure of Programming

Host TX	0x71	PROGRAM _CMD	<=256	Data	CRC
BL Confirm	Confirm				
BL EXE Result	EXE Result				

The programming length must be word align and less and equal than 256. When programming the Flash, the starting address is set by SET_ST_ADDR_CMD. When programming the SRAM, the starting address is set by SET_APP_IN_RAM_ADDR_CMD and SET_ST_ADDR_CMD.

The procedure of reading information in the NVDS

Host TX	0x71	RD_CMD	4	L0	L1	L2	L3	CRC
BL Confirm	Confirm							
BL Return	0x71	RD_FLASH_CMD	<=256	Data	CRC			

The length must be word align and less and equal than 256. Bootloader will return an execution result when Bootloader has executed this command. The starting address of NVDS is set by SET_ST_ADDR_CMD.

The procedure of verifying downloaded application

Host TX	0x71	VERIFY_CMD	0	CRC
BL Confirm	Confirm			
BL EXE Result	EXE Result			

Bootloader will calculate the CRC of downloaded application and compare with the CRC set by SET_APP_CRC_CMD . Then bootloader returns the result of comparison.

The procedure of entering into protection mode

Host TX	0x71	PROTECT_CMD	0	CRC
BL Confirm	Confirm			
BL EXE Result	EXE Result			

When the protection mode is set, SWD will not access Flash and SRAM after rebooting.

The Procedure of running application

Host TX	0x71	RUN_APP_CMD	0	CRC
BL Confirm	Confirm			

When host transmits this command to bootloader, the bootloader will quit the ISP mode and goes to execute QN902x application.

The procedure of Rebooting QN902x

Host TX	0x71	REBOOT_CMD	0	CRC
BL Confirm	Confirm			

When host transmits this command to Bootloader, QN902x will be rebooted by bootloader.

The procedure of writing a 32bit random number to bootloader

Host TX	0x71	WR_RANDOM_DATA_CMD	4	D0	D1	D2	D3	CRC
BL Confirm	Confirm							
BL EXE Result	EXE Result							

Bootloader will return an execution result when bootloader has executed this command.

The procedure of setting the starting address of application location in the SRAM

Host TX	0x71	SET_APP_IN_RAM_ADDR_CMD	4	A0	A1	A2	A3	CRC
BL Confirm	Confirm							
BL EXE Result	EXE Result							

The payload in SET_APP_IN_RAM_ADDR_CMD is one word data which represent the starting address of application will be downloaded to SRAM. The default address is 0x10000000 in the bootloader.

The procedure of setting the address of application entry point

Host TX	0x71	SET_APP_RESET_ADDR_CMD	4	A0	A1	A2	A3	CRC
BL Confirm	Confirm							
BL EXE Result	EXE Result							

The payload in SET_APP_RESET_ADDR_CMD is one word data which represent the entry point of application. After loading application to SRAM, the bootloader will jump to this address to execute the application. The default value of application reset address is set 0x100000D4 in the bootloader. If this address is not configured, bootloader will jump to default address.

5.4.4 ISP Program Flow Diagram

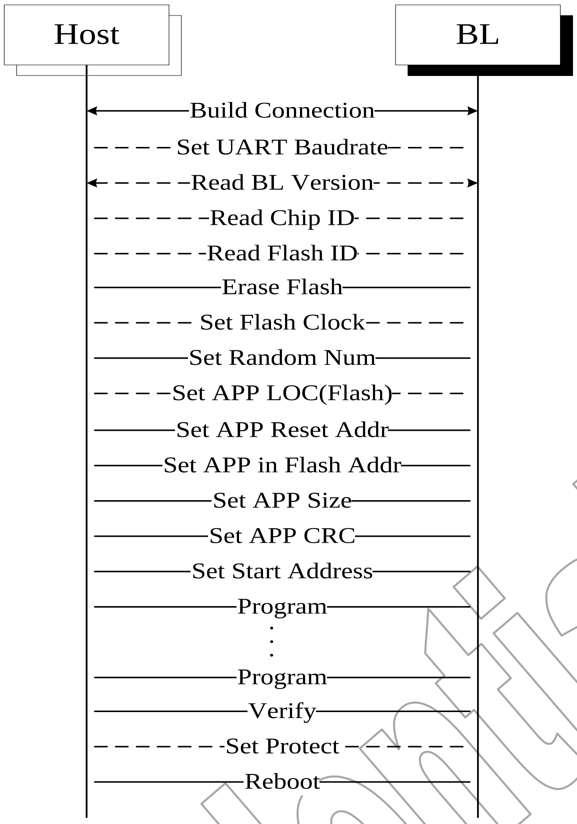


Figure 16 Download an application to Flash

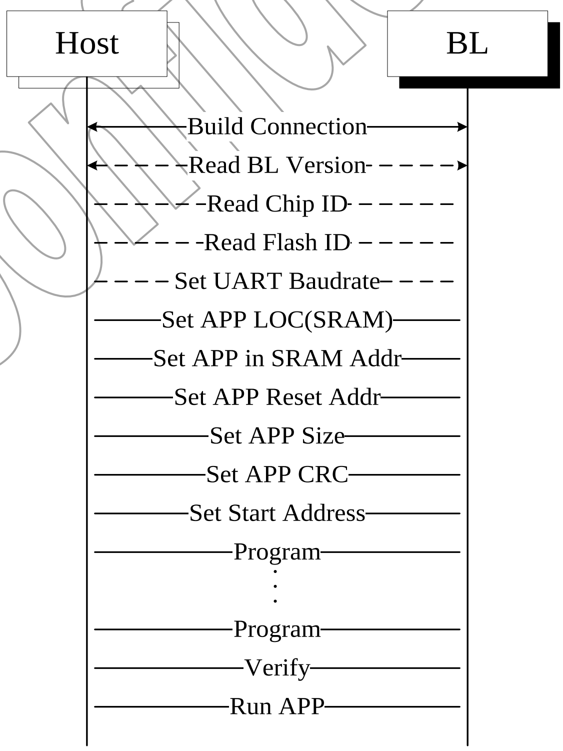


Figure 17 Download an application to SRAM

6. NVDS

The QN902x provides a normalized way to access Non Volatile Data Storage (NVDS). The NVDS is used for storing information which should be saved when chip loses power. The Non Volatile Memory could be FLASH or E2PROM. For now the NVDS driver only supports the internal FLASH as NVDS.

The information in the NVDS is consisted of several TAGs. Each TAG records one type of information and contains ID field, Status field, Length field and Value field. The following NVDS APIs are provided for developer.

Table 33 API for NVDS

API	Description
nvds_get	Look for a specific tag and return the value field of the tag.
nvds_del	Look for a specific tag and delete it. (Status set to invalid)
nvds_lock	Look for a specific tag and lock it. (Status lock bit set to LOCK)
nvds_put	Add a specific tag to the NVDS.

Due to the physical characteristics of the FLASH, Erasing must be done before writing. In order to prevent the information already existed in the Flash lost, the information in the FLASH shall be stored in the SRAM first, and then erase FLASH, then write the correct information to the FLASH. The length of NVDS data is dependent on application design. In order to provide a buffer for NVDS driver storing temporary data, the software developer should allocate an array and pass the address and size to NVDS driver using function 'plf_init()'.

When write operation is used, the erase operation could be executed. If the power is lost, when the NVDS area is erasing, the system configuration and user configuration will be lost. So one 4k flash area at the end of the Flash is used to backup information in the NVDS to prevent losing configuration.

6.1 BLE Stack TAG

Following NVDS TAGs are used in program. These TAGs are recommended for setting. If these TAGs do not exist in the NVDS, the BLE stack will use the default value or the value set by application.

Table 34 BLE Stack TAG

TAG Name	TAG ID	TAG Length (Bytes)	Description	Default Value
NVDS_TAG_BD_ADDRESS	0x1	6	Local Bluetooth address	0x087CBE000001
NVDS_TAG_DEVICE_NAME	0x2	32	Device name	"Quintic BLE"
NVDS_TAG_LPCLK_DRIFT	0x3	2	Radio drift	100ppm
NVDS_TAG_EXT_WAKEUP_TIME	0x4	2	External wakeup duration.	900us
NVDS_TAG_OSC_WAKEUP_TIME	0x5	2	Oscillator wakeup duration.	900us
NVDS_TAG_TK_TYPE	0xb	1	TK type	False
NVDS_TAG_TK_KEY	0xc	6	TK	'111111'

NVDS_TAG_IRK_KEY	0xd	16	IRK	01 5F E8 B4 56 07 8E 22 18 A6 7C E1 E4 BA 99 A5
NVDS_TAG_CSRK_KEY	0xe	16	CSRK	02 45 30 DA 3A FC 81 48 F1 0D AD 2E 91 9D 57 7B
NVDS_TAG_LTK_KEY	0xf	16	LTK	02 45 30 DA 3A FC 81 48 F1 0D AD 2E 91 9D 57 7B
NVDS_TAG_XCSEL	0x10	1	XTAL capacitance loading selection	0x2f
NVDS_TAG_TEMPERATURE_OFFSET	0x11	4	The offset of temperature sensor	-200
NVDS_TAG_ADC_INT_REF_SCALE	0x12	4	ADC internal reference scale	1000
NVDS_TAG_ADC_INT_REF_VCM	0x13	4	ADC internal reference vcm	500

6.2 Include Files

In order to use the services offered by the NVDS driver the user should include the following header files:

File	Description
nvds.h	Contains the declarations of NVDS driver.

7. Application Development

7.1 Available hardware resource for APP

7.1.1 CPU

An ultra-low-power ARM Cortex-M0 microcontroller is available in QN902x, which is a full 32-bit processor that incorporates Thumb-2 technology with many benefits of energy efficiency and code density. The ARM Cortex-M0 also contains a nested vector interrupt controller which is useful for interrupt latency, and the interrupt service routines are able to be coded directly as C functions. For further information on ARM Cortex-M0, please visit www.arm.com.

7.1.2 Memory

There are four types of memory in QN902x. (ROM, SRAM, peripheral registers and NVDS if on-chip flash exists)

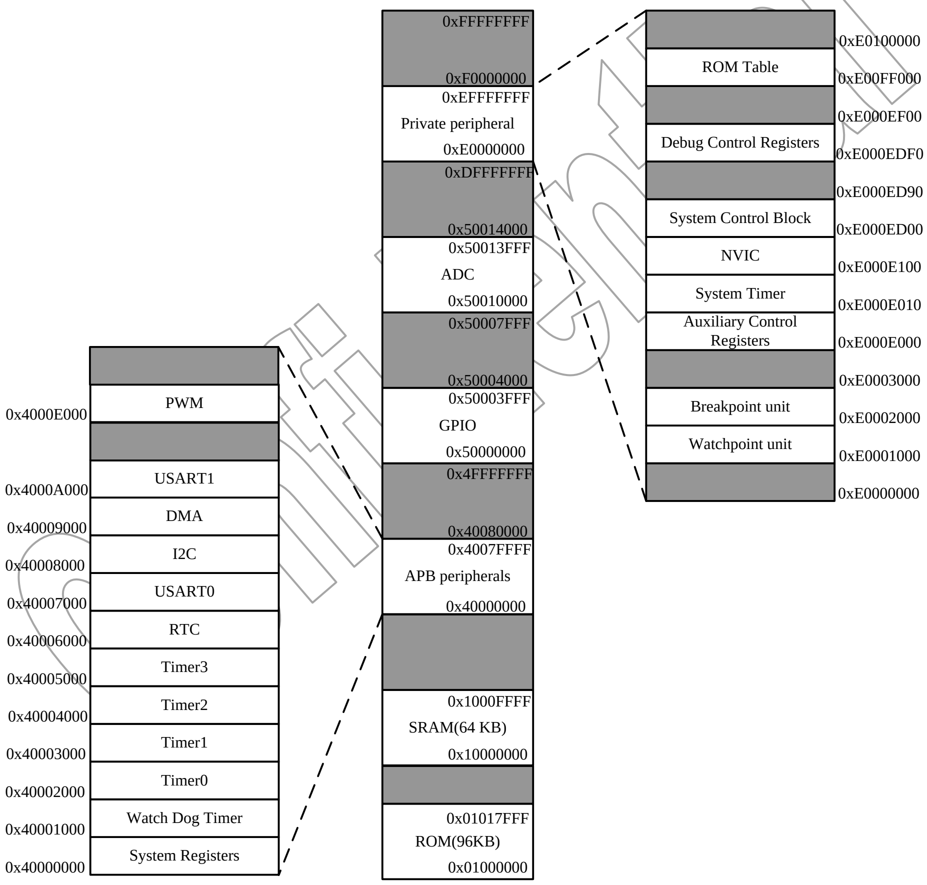


Figure 18 Memory Map

The internal ROM is located at 0x01000000. When system is power up, the address 0x0 is also mapped to the first address of ROM. The ROM is used for storing QN902x firmware's code and RO data. The firmware is composited of BLE protocol stack, bootloader and some device drivers. The application can use the firmware functionality by calling API functions provided by Quintic. For details please refer to chapter 4.

The internal SRAM is located at 0x10000000. After REMAP_BIT is set, the address 0x0 is mapped to the first address of SRAM. The SRAM is used for storing QN902x firmware's RW data and application's code and data. Although the internal SRAM is total 64k bytes, the application does not use all of SRAM space. Because firmware's RW data also needs to take up 0x32e0 bytes of the SRAM space, which is located at 0x1000cd20 in the SRAM. The application shall not use this part of SRAM. The stack of firmware is also in the SRAM. But the stack space of firmware is obsolete, when the program jumps to the entry point of application. So this space is available for application.

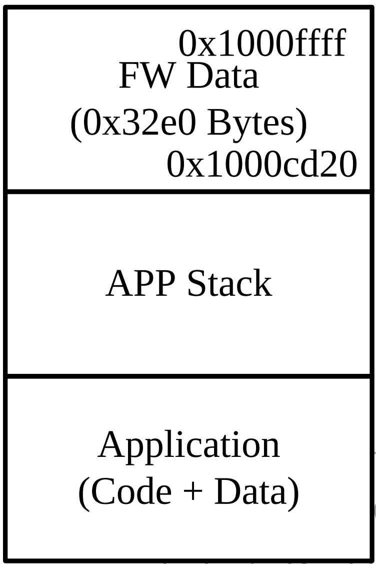


Figure 19 Internal SRAM Map

The peripheral registers are used for configuring and controlling peripheral device such as GPIO, UART, SPI, ADC and so on.

The Non Volatile Data Storage (NVDS) is used for storing information which is still valid when the chip loses power. For details please refer to chapter 6.

7.1.3 Peripheral

In addition to the ARM Cortex-M0 processor, internal ROM and SRAM, QN902x also integrates a wide range of peripherals. Such as UART, SPI, IIC, Flash Controller, Timer, PWM, RTC, Watchdog Timer, GPIO Controller, DMA controller, ADC, Comparator. For figuring out how to use these peripheral devices, please get the detail information from chapter 7.7.

7.1.4 Interrupt Controller

The Nested Vectored Interrupt Controller (NVIC) is an integral part of Cortex-M0, which is tightly coupling to the CPU. The NVIC supports 32 vectored interrupt and 4 programmable interrupt priority levels. External interrupts need to be enabled before being used. If an interrupt is not enabled, or if the processor is already running another exception handler with same or higher priority, the interrupt request will be stored in a pending status register. The pended interrupt request can be triggered when the priority level allowed for example, when a higher-priority interrupt handler has been completed and returned. The NVIC can accept interrupt request signals in the form of logic level, as well as an interrupt pulse (with a minimum of one clock cycle). The NVIC supports stacking and unstacking processor status automatically. Do not need ISR to handles this. The NVIC supports vectored interrupt entry. When an interrupt occurs, the NVIC automatically locates the entry point of the interrupt service routine from a vector table in the memory. For detailed information about NVIC, please refer to "ARM® Cortex™-M0 Technical Reference Manual" and "ARM® v6-M Architecture Reference Manual".

Table 35 lists all of the interrupt sources in QN902x. Each peripheral device may have one or more interrupts lines to the NVIC. Each line may represent more than one interrupt sources. For more information on how the interrupts are handled inside the QN902x, please refer to chapter 7.7.

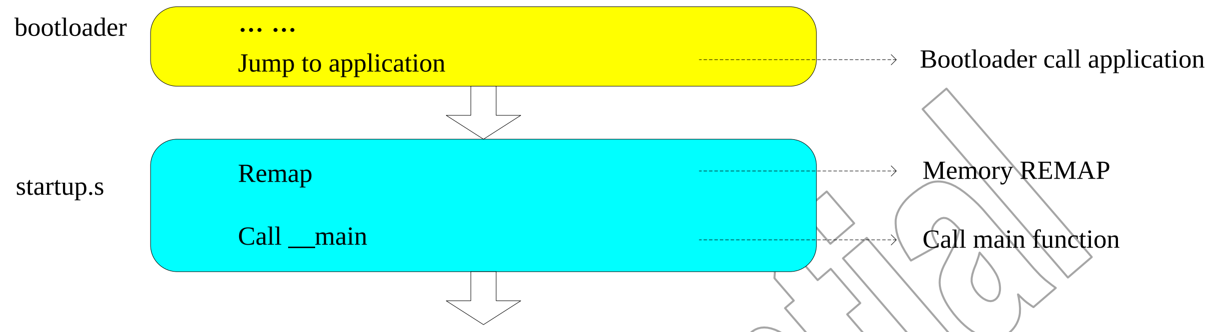
Table 35 Interrupt Vector Define

Interrupt Number	Source	Function
0	GPIO	**GPIO interrupt**
1	Comparator 1	**Comparator 1 interrupt**
2	Comparator 2	**Comparator 2 interrupt**
3	BLE Hardware	**BLE stack interrupt**
4	RTC	**RTC capture interrupt**
5	BLE Hardware	**Exit sleep mode and enable oscillator**
6	RTC	**RTC second interrupt**
7	ADC	**ADC interrupt**
8	DMA	**DMA interrupt**
9	Reserved	
10	UART 0	**UART0 TX ready interrupt**
11	UART 1	**UART0 RX interrupt**
12	SPI 0	**SPI0 TX ready interrupt**
13	SPI 0	**SPI0 RX interrupt**
14	UART 1	**UART1 TX ready interrupt**
15	UART 1	**UART1 RX interrupt**
16	SPI 1	**SPI1 TX ready interrupt**
17	SPI 1	**SPI1 RX interrupt**
18	I2C	**I2C interrupt**
19	Timer 0	**Timer 0 interrupt**
20	Timer 1	**Timer 1 interrupt**
21	Timer 2	**Timer 2 interrupt**
22	Timer 3	**Timer 3 interrupt**
23	Watch dog timer	**Watchdog timer interrupt**
24	PWM CH0	**PWM channel 0 interrupt**
25	PWM CH1	**PWM channel 1 interrupt**
26	Calibration	**Calibration interrupt**
27	Reserved	
28	Reserved	
29	Tuner	**Tuner RX control interrupt (unused in 902x-B2)**

30	Tuner	**Tuner TX control interrupt (unused in 902x-B2)**
31	Tuner	**Tuner setting interrupt (unused in 902x-B2)**

7.2 Application Execution Flow

When the bootloader has already prepared application runtime environment (copy the Application to correct SRAM location), it sets PC to the entry point of Reset Handler of the Application. Since then the Application obtains the fully control of the QN902x. The figure 20 illustrates the Application execution flow. For the details please see the chapter 7.2.1~7.2.11 as below.




```
int main(void)
{
    int ble_sleep_st, usr_sleep_st;

    // DC-DC
    dc_dc_enable(QN_DC_DC_ENABLE);

    // QN platform initialization
    plf_init(QN_POWER_MODE, __XTAL, QN_32K_RCO, nvds_tmp_buf, NVDS_TMP_BUF_SIZE);

    // System initialization, user configuration
    SystemInit();

    // Profiles register
    prf_register();

    // BLE stack initialization
    ble_init((enum WORK_MODE)QN_WORK_MODE, QN_HCI_UART, QN_HCI_UART_RD,
             QN_HCI_UART_WR, ble_heap, BLE_HEAP_SIZE, QN_BLE_SLEEP);

    set_max_sleep_duration(QN_BLE_MAX_SLEEP_DUR);

    // initialize APP task
    app_init();
```

```
    // initialize user setting
    usr_init();

    // sleep configuration
    sleep_init();
    wakeup_by_sleep_timer(XTAL32);

    GLOBAL_INT_START();

    while(1)
    {
        ke_schedule();

        // Checks for sleep have to be done with interrupt disabled
        GLOBAL_INT_DISABLE_WITHOUT_TUNER();

        // Obtain the status of the user program
        usr_sleep_st = usr_sleep();

        // If the user program can be sleep, deep sleep or clock off then check
        // ble status
        if(usr_sleep_st >= PM_IDLE)
        {
            // Obtain the status of ble sleep mode
            ble_sleep_st = ble_sleep(usr_sleep_st);

            // Check if the processor clock can be gated
            if(((ble_sleep_st==PM_IDLE) || (usr_sleep_st==PM_IDLE))
                && (usr_sleep_st!=PM_ACTIVE))
            {
                enter_sleep(SLEEP_CPU_CLK_OFF,
                             WAKEUP_BY_ALL_IRQ_SOURCE,
                            NULL);
            }
            // Check if the processor can be power down
            else if((ble_sleep_st==PM_SLEEP)
                   && (usr_sleep_st==PM_SLEEP))
            {
                enter_sleep(SLEEP_NORMAL,
                            (WAKEUP_BY_OSC_EN | WAKEUP_BY_GPIO),
                            sleep_cb);
            }
            // Check if the system can be deep sleep
            else if((usr_sleep_st==PM_DEEP_SLEEP)
                    && (ble_sleep_st==PM_SLEEP))
```

```
            {
                enter_sleep(SLEEP_DEEP,
                            WAKEUP_BY_GPIO,
                            sleep_cb);
            }
        }

        // Checks for sleep have to be done with interrupt disabled
        GLOBAL_INT_RESTORE_WITHOUT_TUNER();
    }
}
```

Figure 20 Application Execution Flow

7.2.1 Startup (Remap)

When QN902x is power up, it always boots at address 0x0. Therefore, in the initial state of the system, it is necessary to ensure that the correct code exists at the address 0x0 and the space mapping to address 0 is non-volatile memory (ROM). In QN902x the ROM space is mapped from address 0x0 to 0x17fff when system is power up.

The vector table of Cortex-M0 exceptions and interrupts is required being at the fixed memory space from address 0x0 to 0xef, which is in the ROM space when system is power up. It means the exception and interrupt handler cannot be changed and an interrupt forwarding mechanism should be provided if the developers want to add their own interrupt handler. The interrupt forwarding method is not intuitive for developer, and will increase interrupt response time. The memory remap is provided by QN902x instead of interrupt forwarding. REMAP allows the user to lay out the internal SRAM bank to address 0x0. At this time, the interrupt vector table of application project is placed at the address 0x0, which is available for Cortex-M0.

One decoder is provided for every AHB Master Interface. The decoder offers each AHB Master several memory mappings. In fact, depending on the product, each memory area may be assigned to several slaves. Thus it is possible to boot at the same address while using different AHB slaves (internal RAM or internal ROM). Regarding master, two different slaves are assigned to the memory space decoded at address 0x0: one for internal SRAM, one for internal ROM. The QN902x provides SYS_REMAP_BIT in System Boot Mode Register (SYS_MODE_REG) that performs remap action. At reset SYS_REMAP_BIT = 0, the internal ROM is lay out at address 0x0. When SYS_REMAP_BIT is set to 1, the internal SRAM is lay out at address 0x0.

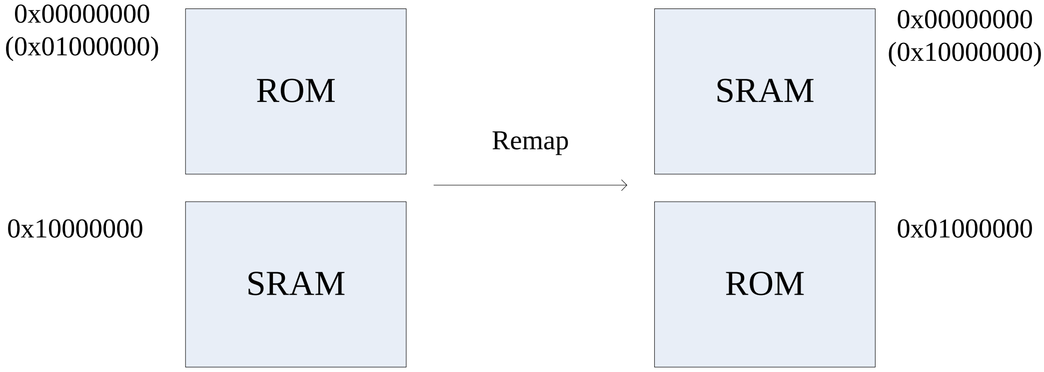


Figure 21 Remap Action

Notes:

Memory blocks that are not affected by SYS_REMAP_BIT can always be seen at their specified base addresses. The base address of ROM is 0x01000000. The base address of SRAM is 0x10000000. See the complete memory map presented in Figure 21.

The application is linked from address 0x0. If one absolute jump instruction is executed before REMAP, the program will jump to ROM space which is lay out at address 0x0 at that moment. So the REMAP must be finished before any absolute jump being executed. Adding REMAP at the beginning of application is recommended. In the sample code, REMAP is executed at the beginning of RESET handler.

7.2.2 DC-DC Configuration

Depending on circuit design of application product to enable or disable the DC-DC.

7.2.3 BLE Hardware Initialization

At the beginning of main function the subroutine 'plf_init()' is invoked to finish the initialization of RF, modem and BLE related hardware. When this function is returned, the BLE hardware and physical layer shall be ready.

There are four parameters required by this function. The parameters' function and optional setting is described by following table.

Table 36 Hardware Initialization

Parameter	Function	Optional Value
pw_mode	Configure which power mode the BLE hardware uses. Two types of power mode can be selected. Developer should choose the mode based on product design.	NORMAL_MODE : chip consumes lower power than HIGH_PERFORMANCE mode, but the performance is not good at HIGH_PERFORMANCE mode. HIGH_PERFORMANCE : Chip has higher performance and consumes more power.
xtal	Configure which frequency the external crystal the system used. Two types of external crystal can be	16000000UL: 16MHz XTAL 32000000UL: 32MHz XTAL

	selected. The selection based on product design and the software needs to know which type crystal is used.	
clk_32k	Configure which 32k clock is used.	0: XTAL 1: RCO
nvds_tmp_buf	The start address of temporary buffer used by NVDS driver.	The pointer of unsigned char array. NULL: No buffer for NVDS.
nvds_tmp_buf_len	The length of temporary buffer used by NVDS driver.	4096 : 4096 bytes buffer for NVDS. 0 : No buffer for NVDS.

7.2.4 Initialize System

In the subroutine 'SystemInit()', the software developer should set each module's clock, configure IO and initialize all of used peripheral.

These configurations rely on the user's system design. The module which is not used in your product should be gated to reduce power consumption. And the clock should be set to the lowest available value. It is helpful to reduce the power consumption of clock MUX.

The IO configuration tool is provided by Quintic to reduce development effort. For the details please refer to the document 'Driver_tools_manual'.

7.2.5 Register Profiles Functions into BLE Stack

GAP task will invoke two profile subroutines, 'prf_init()' and 'prf_dispatch_disconnect()'. Before BLE kernel running, The function 'prf_register()' shall complete the registration of these two profile subroutines into BLE stack.

7.2.6 Initialize BLE Stack

The function 'qn_ble_init()' is provided by Quintic to finish the BLE software configuration and initialization. Within this function, the BLE heap which memory space is offered by developer needs to be initialized. The initialization routine for each layer of BLE stack is called.

Five parameters should pass to this function, and the function and optional setting of these parameters is described by following table.

Table 37 BLE Stack Initialization

Parameter	Function	Optional Value
mode	Indicate which work mode is selected by the developer. Three modes can be selected. The details refer to chapter 2.2.	SOC_MODE: Wireless SoC Mode NP_MODE: Network Processor Mode HCI_MODE: Controller Mode
port	Configure which UART/SPI interface is	QN_UART0

	used for ACI and HCI.	QN_UART1 QN_SPI0 QN_SPI1
hci_read	Configure the UART/SPI read operation API.	The function 'uart_read()'/'spi_read()' in the driver.
hci_write	Configure the UART/SPI write operation API	The function 'uart_write()'/spi_write()' in the driver.
ble_heap_addr	The BLE heap is allocated by software developer. And this parameter tells BLE stack where the starting address of the heap is.	Available address in the SRAM.
ble_heap_size	Indicate the size of the heap which will is used in BLE stack	Available value.
sleep_enable	Configure whether the BLE stack can enter into sleep mode	True: BLE stack sleep is allowable. False: BLE stack sleep is disallowed.

The BLE protocol stack needs a block of memory to dynamically allocate message and attribute database which is called BLE heap. The BLE heap would not only depend on the maximum number of possible LE links, but also with the actual LE profiles to support. In QN902x the heap size is not fixed at a maximum possible value that will result in a waste of memory. So the allocation of protocol heap is implemented in the application, and then the function 'qn_ble_init()' is called to configure heap address.

There is a way to determine the heap requirements depending on certain parameters (profiles, number of links, etc). In the reference code, app_config.h contains the BLE heap size definition.

#define BLE_HEAP_SIZE (BLE_DB_SIZE + 300 + 256 * BLE_CONNECTION_MAX)

BLE_DB_SIZE is added in the equation to take care of the dynamic allocation of attribute database. When the profile is used, the database size of this profile is added.

The messages and timers are all allocated from the BLE heap. More messages and timers are used means more BLE heap is needed. If the free memory in the BLE heap is not enough and the memory allocation fails, the kernel will automatically issue a software reset. You can confirm this software reset from the bit1 in debug information register (0x1000fffc). When memory allocation failure occurs, developer should increase BLE heap size.

7.2.7 Set Maximum BLE Sleep Duration

The maximum BLE sleep duration is used by BLE stack. When the BLE is inactive, the BLE sleep timer will be configured as the setting value. Otherwise when the BLE stack is active, the actual sleep duration is revised based on application scenario (advertising interval, connection interval...).
The unit of the parameter 'duration' is 625us, and the value of the parameter shall be less than 209715199.

7.2.8 Initialize Application Task

Application task is responsible for either processing message from the BLE stack and device driver, or constructing and sending the message to BLE stack. The application initialization function will prepare the APP task environment and register APP task into the kernel.

7.2.9 Sleep initialization

Initialize configuration for sleep mode.

7.2.10 Run Scheduler

After application task initialization, in order to use the kernel, there should be a call to 'ke_schedule()' in forever loop at the end of the main function. This routine checks the message queue, and implement message deliver.

7.2.11 Sleep Mode

QN902x has four power modes: active mode, idle mode (CPU clock off mode), sleep mode and deep sleep mode. See the following table to obtain detailed difference.

Table 38 Processor Power Mode

Processor Mode	Processor Status	Processor Clock	Processor Power	Wakeup Source	Modules can be disabled	Modules must be power on
Processor active mode	Active	Yes	Yes	/	Timer, UART, SPI, 32k clock, Flash controller, GPIO, ADC, DMA, BLE, PWM.	40MHz oscillator, Bandgap, IVREF, VREG of analog, VREG of digital.
Processor clock off mode	Inactive	No	Yes	All interrupts	Timer, UART, SPI, 32k clock, Flash controller, GPIO, ADC, DMA, BLE, PWM.	40MHz oscillator, Bandgap, IVREF, VREG of analog, VREG of digital.
Processor sleep mode	Inactive	No	No	GPIO, comparator, BLE sleep timer (32k clock shall be exist).	/	32k XTAL/RCO, retention MEM, Comparator if used.
Processor deep sleep mode	Inactive	No	No	GPIO, comparator.	/	Retention MEM, Comparator if used.

In the active mode and idle mode, the clock of digital modules (Timer, UART, SPI, PWM …) can be enabled or disabled independently. The power of analog modules which have independent power domain can also be enabled or disabled by application.
In the idle mode, the clock of processor is gated and all of the interrupts can wakeup system.
In the sleep mode, the interrupts of GPIO, comparator and BLE sleep timer can wake up the system.
In the deep sleep mode, only the interrupts of GPIO and comparator can wakeup system. 32k clock is power off, so BLE stack does not work in the deep sleep mode.

Power mode setting depends on user setting, peripherals' status and BLE's status.
The user power mode setting is managed by user who can use API sleep_set_pm() to set power mode user want the system to be. The macro CFG_DEEP_SLEEP in the usr_config.h defines the initial value of user's power mode setting.
The peripherals' status is managed by drivers. Actually the driver knows when the peripherals cannot enter into sleep mode. For example when the ADC is sampling, the ADC driver knows the system cannot enter into sleep mode at this time. User does not take care of the peripherals' status.
The status of BLE is managed by BLE stack. User can invoke API ble_sleep() to obtain the status of BLE. If the macro CFG_BLE_SLEEP is not defined in the usr_config.h, ble_sleep() will not enter into sleep mode.

In the main loop the program obtain sleep status by two APIs (usr_sleep/ble_sleep). Program determines power mode setting based on these two statuses. Active mode has the highest priority. It means if any one status is active mode, the system should stay in the active mode. You can see all possible combination in the table.

USR / BLE	Active (MCU)	Idle (CLOCK OFF)	Sleep (POWR DOWN)	Deep Sleep
Active	Active	Active	Active	Active
Idle	Active	Idle	Idle	Idle
Sleep	Active	Idle	Sleep	Deep Sleep

When the system is waked up from sleep mode, all register setting of the peripherals are lost and the peripherals need to be reconfigured.

Sleep API

sleep_int()
Set which modules will be power down when system enter into sleep mode.

set_max_sleep_duration()
This function sets the maximum sleep duration of BLE sleep timer. When the BLE stack works, the actual sleep duration will be revised based on BLE stack's requirement. Otherwise the BLE stack does not work, the sleep duration will be the configured value.
The unit of the parameter is 625us and the value of the parameter should be less than 209715199.The maximum sleep duration is about 36hours 16minutes.

sleep_set_pm()
User sets the highest level of power mode.

sleep_get_pm()
User gets the highest level of power mode.

usr_sleep()
Obtain the power mode allowed by user and peripherals.

ble_sleep()
Get the power mode allowed by BLE stack.

enter_sleep()
Set power mode.

sleep_cb()
Restore setting of system, BLE and peripherals in the system wakeup procedure.

save_ble_setting()
When the BLE hardware is really power off, all of the register setting is lost. When the power is on, the register value will be default value. So it is important to save necessary register setting before power down to restore system in a working state.
The peripheral setting will be also lost when system is in the sleep mode. It is necessary to save the setting of peripheral before entering into sleep mode. Or the software can reinitialize the peripheral when system exit sleep mode.

restore_ble_setting()
Restore the saved setting of BLE hardware.

enable_ble_sleep()
This function allows or disallows BLE hardware going to Sleep Mode.

ble_hw_sleep()
This function checks BLE hardware sleep states. When the return value is TRUE, it means BLE hardware is sleeping. When the return value is FALSE, it means BLE hardware is not in the sleep status.

sw_wakeup_ble_hw ()
This function forcibly wakeups BLE hardware.

reg_ble_sleep_cb()
This function provides a way to register two optional callbacks into the firmware. The first callback is invoked before firmware runs decision algorithm of BLE entering sleep. If the return of callback is FALSE, the procedure of checking whether the BLE hardware goes into sleep mode will be break. BLE hardware will not be allowed to enter sleeping. Otherwise the BLE hardware is going to enter sleep mode.
The second callback is invoked after BLE hardware exits sleep mode.

7.3 User Configuration

In order to simplify the development, Quintic provides a user configuration file (usr_config.h) to customize QN902x's application setting and behavior.

Chip Version
The chip feature and firmware may have diference in the different QN902x version. Application needs to know which chip version is used.

Work Mode
QN902x is a very flexible BLE chip. It supports a variety of work modes. Define different macros (CFG_WM_SOC, CFG_WM_NP, CFG_WM_HCI) to set to the appropriate work mode. These macros are mutually exclusive and cannot be defined simultaneously.

Easy ACI
Support easy ACI interface. See EACI document for details.

Easy API
Support easy API interface. See EAPI document for details.

Local Name
Generally the local name in the advertising packet is obtained from the NVDS. Once the device name tag is not available in the NVDS, BLE stack will use the name string defined by macro 'CFG_LOCAL_NAME'. The string is considered to be local name passed to stack, and be added in the advertising packet.

DC-DC Enable
If the DC-DC is used, the macro 'CFG_DC-DC' shall be defined.

32k RCO
If the 32k RCO is used, the macro 'CFG_32K_RCO' shall be defined.

NVDS Write Support
If the application will use 'nvds_put()', this macro shall be defined.

Test mode control pin
The developer can use this macro to select one pin that controls application to go test mode or the work mode defined by the work mode macro.

Memory retention
Define which memory banks need to be retention in sleep mode.

Deep sleep support
If the macro 'CFG_DEEP_SLEEP' is defined, the system is allowed enter into deep sleep mode.

BLE sleep
If the macro 'CFG_BLE_SLEEP' is defined, the BLE stack is allowed enter into sleep mode.

Maximum sleep duration
This macro defines the maximum sleep duration of BLE sleep timer.

UART Interface for Transport Layer
Define which UART interface is used for ACI or HCI.

UART Interface for Debugging
Define which UART interface is used for debug information output.

Debugging Output API
Define which printf() function is used in the debugging. If the macro 'CFG_STD_PRINTF' is defined, the function printf() in the library is used. Otherwise the quintic's implementation of printf() is used.

Terminal Menu
When the macro 'CFG_DEMO_MENU' is defined, user can use a terminal menu tree to control QN902x.

Debug Information
If the macro 'CFG_DBG_PRINT' and 'CFG_DBG_TRACE' are defined, more debug and trace information can be sent out though UART interface.
If the macro 'CFG_DBG_INFO' is defined, some diagnostic information will be saved in the debug information register (0x1000fffc).

Maximum number of connections
The QN902x fully supports 8 links simultaneously, but BLE heap size and some data structure size in application depend on maximum links. There is no need to define 8 links when QN902x runs on peripheral role which requires only one connection. Please specify maximum number of connections based on product design.

GAP Role
The GAP of QN902x supports ALL defined GAP roles (broadcaster, observer, peripheral and central). With different roles the required message handler is different. In order to optimize the code size of application, please specify the GAP role based on product design.

Bluetooth Address Type
Device uses public address or random address.

ATT Role
The ATT role should be supported by Application Task.

Profiles Configuration
The QN902x supports 8 profiles simultaneously. The application shall define the profiles used in the product. In the reference code, usr_config.h contains enabled profiles definition. Define the profile macros to include

profile source code. For example, a central device enables three profiles (HTPC, BLPC, HRPC). The following macros shall be defined.

```
#define CFG_PRF_HTPC
#define CFG_PRF_BLPC
#define CFG_PRF_HRPC
```

Furthermore the task ID of profile shall be defined. The available task id for profiles is from 13 to 20. The different profile used in one device shall occupy a different task id. For example, the task id should be defined for previous example.

```
#define TASK_HTPC          TASK_PRF1
#define TASK_BLPC          TASK_PRF5
#define TASK_HRPC          TASK_PRF7
```

7.4 Application Task

The Application Task implements the feature specified by the protocol and interacts with other protocol layers and profiles in the BLE stack.

The kernel task is defined by its task type and task descriptor (See 3.3). The task type of Application task is 21. The task descriptor of Application task needs to be filled by developer. In the initialization of Application task, the task_app_desc is registered into kernel using the subroutine 'task_desc_register()'.

After inserting the Application task descriptor the program can start kernel scheduler.

7.4.1 APP_TASK API Description

In order to help developers understand how to develop application task, Quintic provides message handler for all of the messages sent to application task and all of the functions which packed the message will send to other tasks. Quintic calls it APP_TASK API. It is very easy to develop application based on APP_TASK API. After understanding how the APP_TASK API works, the developer can cut these APIs for your design and get the optimized program size.

For the details of APP_TASK API please refer to the document 'QN9020 API Programming Guide'.

7.5 Quintic BLE Profiles

A profile defines an optimal setting for a particular application. Quintic provides all of the profiles' source code for developer's reference. This chapter describes the non-standard interface of Quintic Proximity Profile implementation. The description is for a better understanding of the user that needs to develop higher application interface the profiles.

The Proximity profile defines the behavior when a device moves away from a peer device so that the connection is dropped or the path loss increases above a preset level, causing an immediate alert. This alert can be used to notify the user that the devices have become separated.

The Proximity Reporter shall be a GATT server and must have an instance of the Link Loss service (LLS), and optionally both the Immediate Alert (IAS) and the Tx Power service (TPS). The two last ones must be used together, if one is missing, the other one should be ignored.

The Proximity Monitor shall perform service discovery to get peer device's service and characteristic. The following message and API is used for enabling the Monitor role of the Proximity profile. The Application sends it and it contains the connection handler for the connection this profile is activated, the connection type and the previously cached discovered LLS, IAS and TPS details on peer.

Message and API

Proximity Monitor Message	Direction	APP API
PROXM_ENABLE_REQ	APP→PROXM	app_proxm_enable_req
PROXM_ENABLE_CFM	PROXM→APP	app_proxm_enable_cfm_handler

The connection type may be 0 = Connection for discovery or 1 = Normal connection. This difference has been made and Application would handle it in order to not discover the attributes on the Reporter at every connection, but do it only once and keep the discovered details in the Monitor device between connections. If it is a discovery type connection, the LLS, IAS and TPS parameters are useless, they will be filled with 0's. Otherwise it will contain pertinent data which will be kept in the Monitor environment while enabled.

The LLS allows the user to set an alert level in the Reporter, which will be used by the reporter to alert in the corresponding way if the link is lost. The disconnection must not come voluntarily from one of the two devices in order to trigger the alert.

Link loss service requirements

Characteristic	Req.	Properties	Permissions	Descriptors
Alert level	M	Read Write	None	

The following message and API is used to set and get an LLS Alert Level.

Proximity Monitor Message	Direction	APP API
PROXM_RD_ALERT_LVL_REQ	APP→PROXM	app_proxm_enable_req
PROXM_RD_CHAR_RSP	PROXM→APP	app_proxm_enable_cfm_handler
PROXM_WR_ALERT_LVL_REQ	APP→PROXM	app_proxm_wr_alert_lvl_req
PROXM_WR_CHAR_RSP	PROXM→APP	app_proxm_wr_char_rsp_handler

The following message and API is used by the Reporter role to request the Application to start the alert on the device considering the indicated alert level.

Proximity Reporter Message	Direction	APP API
PROXR_ALERT_IND	PROXR→APP	app_proxr_alert_ind_handler

The IAS allows the user to set an immediate alert level based on path loss computation using the read Tx Power Level and RSSI monitored on received packets. According to the alert level set in IAS, the Reporter will start alerting immediately.

Immediate alert service requirements

Characteristic	Req.	Properties	Permissions	Descriptors
Alert level	M	Write without Response	None	

The following message and API is used to set an IAS Alert Level.

Proximity Monitor Message	Direction	APP API
PROXM_WR_ALERT_LVL_REQ	APP→PROXM	app_proxm_wr_alert_lvl_req
PROXM_WR_CHAR_RSP	PROXM→APP	app_proxm_wr_char_rsp_handler

The following message and API is used by the Reporter role to request the Application to start the alert on the device considering the indicated alert level.

Proximity Reporter Message	Direction	APP API
PROXR_ALERT_IND	PROXR→APP	app_proxr_alert_ind_handler

The TPS allows the user to read the Tx Power Level for the physical layer. The value is used by the Monitor to continuously evaluate path loss during the connection, and decide to trigger/stop an alert based on path loss going over/under a set threshold in the Monitor application.

Tx power service requirements

Characteristic	Req.	Properties	Permissions	Descriptors
Tx power level	M	Read	None	

The following message and API is used for reading the tx power level in TPS Tx Power Level Characteristic.

Proximity Monitor Message	Direction	APP API
PROXM_RD_TXPW_LVL_REQ	APP→PROXM	app_proxm_rd_txpw_lvl_req
PROXM_RD_CHAR_RSP	PROXM→APP	app_proxm_rd_char_rsp_handler

7.6 Application Samples

Quintic offers a range of BLE examples which cover the application of the profiles and services provided by Quintic. These examples can help software developers to understand the BLE stack, profiles and how to develop application. The developers will be able to develop their own products after a little modification of these examples.

7.6.1 Directory Structure

This section provides information about the directory structure of Quintic BLE examples.

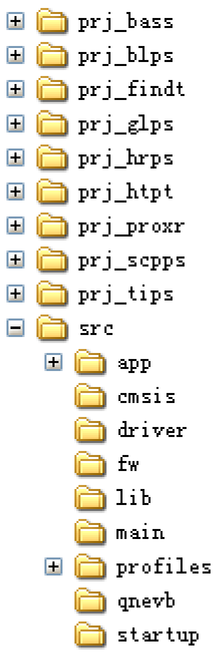


Table 39 Directory Elaboration

Block	Description
prj_xxx	This fold contains project file and a fold includes configuration files and user design files.
src	This contains BLE stack configuration file, driver configuration file, system setup file and user design file.
src	This fold contains BLE stack related files, application files and drivers.
app	This contains files which supported BLE application task development. The file 'app_xxx_task.c' process the message from other tasks in the stack, and the file 'app_xxx.c' construct the messages to send to other tasks. The file'app_task.c' gathers all of the messages the application task should take care.
cmsis	This contains CMSIS compatible processor and peripheral access layer files for QN902x.
driver	This contains device driver for QN902x.
fw	This contains all of firmware feature header file which will be used by application.
lib	This contains BLE hardware and software initialization library.
main	This contains application main file.
profiles	This contains all of the profiles' source code.
qnevb	This block contains support files for Quintic evaluation board.
startup	This contains startup file for application project.

7.6.2 Proximity Reporter

In the following chapter proximity reporter is used to illustrate how to use these examples.

1. Project

The project in the fold 'prj_proxr' is a Proximity Reporter example. Developer can open the project file 'prj_proxr.uvproj' in the Keil IDE. The project structure is shown below. We have already learned the features of Proximity Reporter in the previous chapter. In this section we will describe the function of each file in the project and why these file should be picked to support Proximity Reporter. Though this example, the developers could learn how to choose file to create their own product.

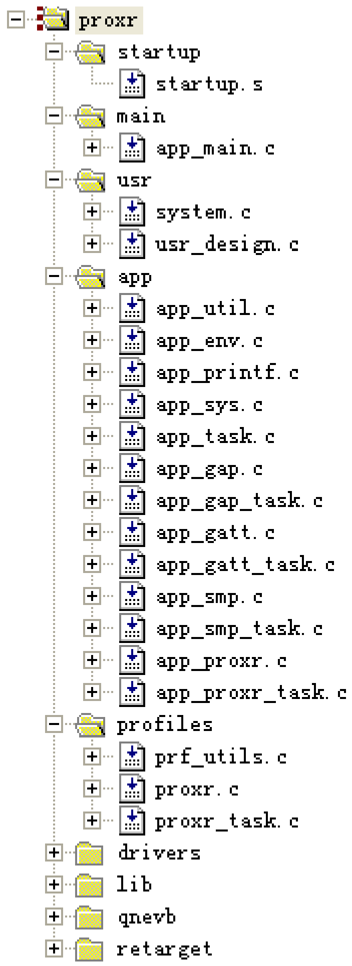


Table 40 File Elaboration

File	Description
BLE/src/startup/startup.s	This file is the entry of the project, which responsible for allocating the stack and heap, setting the interrupt vector and initializing the system library. After completion of these functions jump to main function. This file is a must for every project.
BLE /src/main/app_main.c	This file contains main function which handles all the modules' initialization and main loop. This file is a must for every project.
BLE/prj_proxr/keil/src/system.c	This file contains system setting including sub-module's clock, IO configuration and peripherals. Sub-module used for each product may not be the same, so the developer should modify this file to meet the design requirements.
BLE/prj_proxr/keil/src/usr_desig n.c	Product-related design file. In this example it is the application how interpret starting/stopping alert. The developer can use different device components to show alert, such as Buzz, LED and so on. It is

	recommended to put developer design file in one fold, but not required.
BLE/src/app/app_util.c	This file contains application utility API. This file is a must for every project.
BLE/src/app/app_env.c	This file contains initialization function of application task and record important information in the application task. This file is a must for every project.
BLE/src/app/app_printf.c	This file provide a debug way.
BLE /src/app/app_sys.c	This file contains peripheral related functions in the application task.
BLE /src/app/app_task.c	This file gathers all of the messages the application task should take care.
BLE/src/app/app_gap.c	This file constructs messages to GAP task and will be a part of application task. This file is a must for every project.
BLE/src/app/app_gap_task.c	This file takes care of message from GAP task and will be a part of application task. This file is a must for every project.
BLE/src/app/app_gatt.c	This file constructs messages to GATT task and will be a part of application task. If the product plays central role or peripheral role, this file is a must. It is not necessary to include this file when product plays broadcaster role or observer role.
BLE/src/app/app_gatt_task.c	This file takes care of message from GATT task and will be a part of application task. If the product plays central role or peripheral role, this file is a must. It is not necessary to include this file when product plays broadcaster role or observer role.
BLE/src/app/app_smp.c	This file constructs messages to SMP task and will be a part of application task. If the security feature is support in the product, this file is a must. Otherwise it is no need to include.
BLE /src/app/app_smp_task.c	This file takes care of message from SMP task and will be a part of application task. If the security feature is support in the product, this file is a must. Otherwise it is no need to include.
BLE/src/app/app_proxr.c	This file constructs messages to proximity reporter task and will be a part of application task.
BLE/src/app/app_proxr_task.c	This file takes care of message from proximity reporter task and will be a part of application task.
BLE/src/profiles/prf_utils.c	This file is implementation of Profile Utilities. As long as any profiles are used, this file should be included.
BLE/src/profiles/prox/proxr/proxr.c	The file implements the features of Proximity Reporter role in the proximity profile. It is necessary to contain profile source code when the profile is used.
BLE/src/profiles/prox/proxr/proxr_task.c	The file implements the features of Proximity Reporter role in the proximity profile.
BLE/src/driver/uart.c	In the example UART interface is used to output debug information, so the UART driver should be included in the project. Whether the device driver is contained in the project depends on the product design requirements. And also the driver may be modified by developer. This file is just a reference design.

BLE/src/driver/gpio.c	In the example GPIO is used to control LED and button.
BLE/src/driver/sleep.c	Sleep API.
BLE/src/driver/syscon.c	System clock API.
BLE/src/pwm.c	In the example PWM is used to control buzzer.
BLE/src/lib/keil/qn9020b2_lib_peripheral.lib	The library of BLE hardware and software initialization API. This file is a must for every project.
BLE/src/qnevb/led.c	This file provides LED control for Quintic EVB. In the proximity reporter the LEDs are used to show link status. The developer should design your own BSP to replace these files.
BLE/src/qnevb/button.c	This file provides button control for Quintic EVB. In the proximity reporter the buttons are used to start/stop advertising and stop alert. The developer should design your own BSP to replace these files.
BLE/src/qnevb/buzz.c	This file provides buzzer control for Quintic EVB. In the proximity reporter buzzer is used to show alert. The developer should design your own BSP to replace these files.

2. User Configuration

BLE stack configuration (usr_config.h)

Application runs on QN902x, so it is Soc Mode. The example of Proximity Reporter plays peripheral role, so the connection number is one. And the application should be compiled with proximity server only. The following macro shall be defined in the 'usr_config.h'.

- **#define CFG_WM_SOC**
- **#define CFG_PERIPHERAL**
- **#define CFG_CON 1**
- **#define CFG_PRF_PXPR**
- **#define CFG_TASK_PXPR TASK_PRF1**

Driver configuration (driver_config.h)

Only GPIO, PWM and UART are used in the Proximity Reporter example. The developer should enable corresponding driver in the 'driver_config.h'.

System setup (system.c)

At the beginning all peripheral's clock are disabled, and the clock will be enabled when the driver initialization is invoked.

Set system clock source, system clock and BLE. In the demo, external 16MHz XTAL is use. CPU runs on 8MHz and BLE also runs on 8MHz. Some APIs for setting clock configuration are in the file 'syscon.c', which can be used to change clock in the application.

The IOs are recommended as GPIO except UART and PWM which are used in this project.

Initialize all of the peripherals used in this project.

3. Message Flow

In the proximity reporter example, the developer should understand three basic procedures (initialization, advertising, connection and profile operation). The following figures describe message sequence between APP_TASK and stack in these procedures, and also indicates related application handler. Through the following illustrations the developer can learn that the APP_TASK is how to interact with stack task, how to start advertising, how to obtain connection information and how to work with profiles.

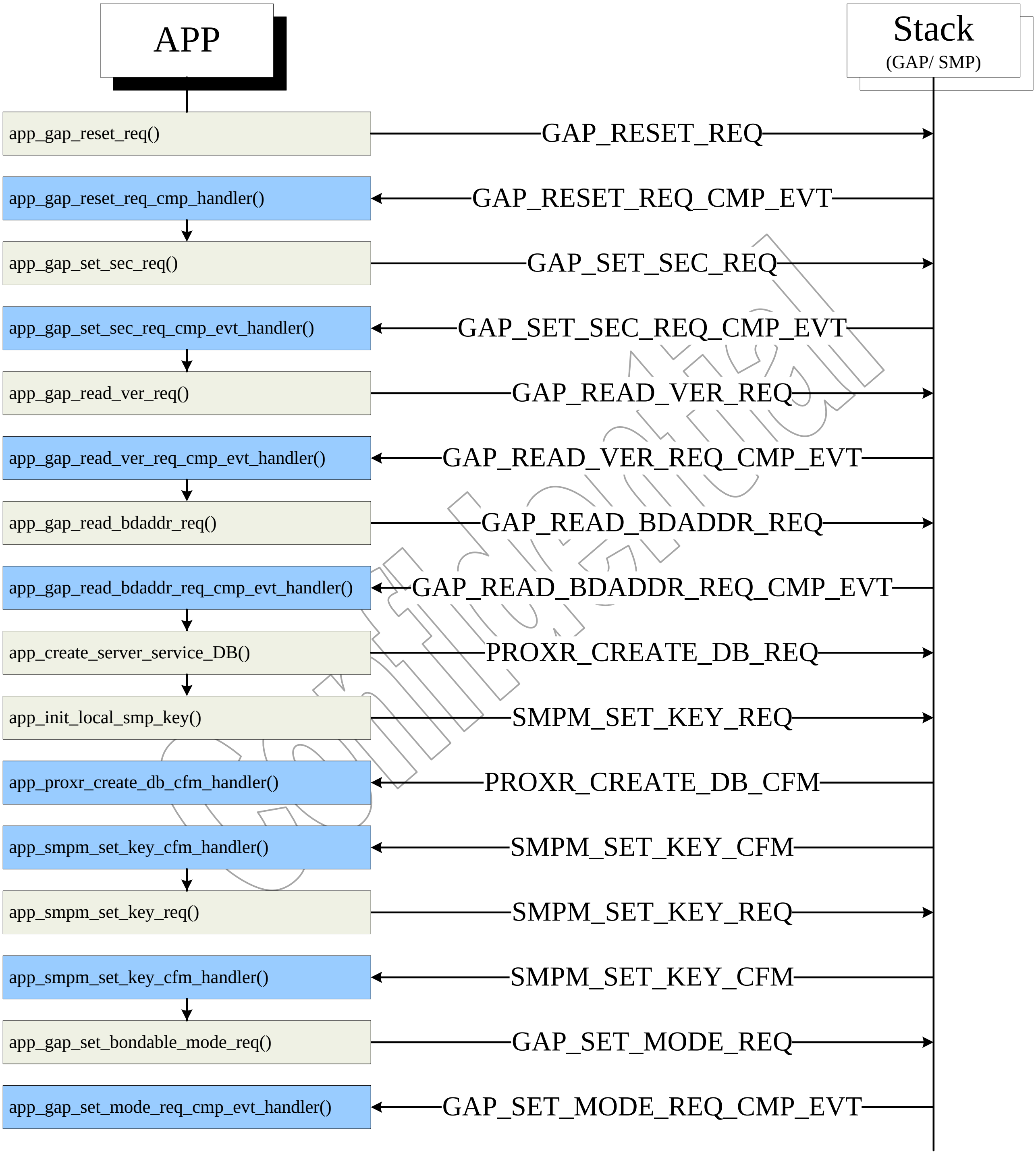


Figure 22 Application Initialization

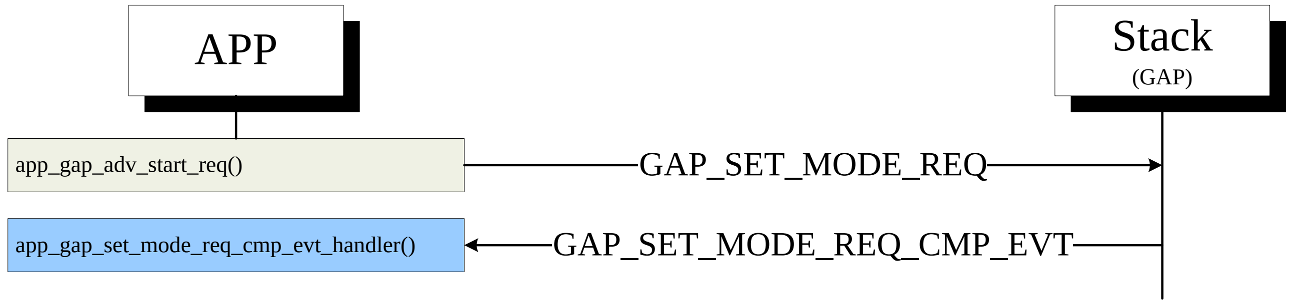


Figure 24 Application Start Advertising

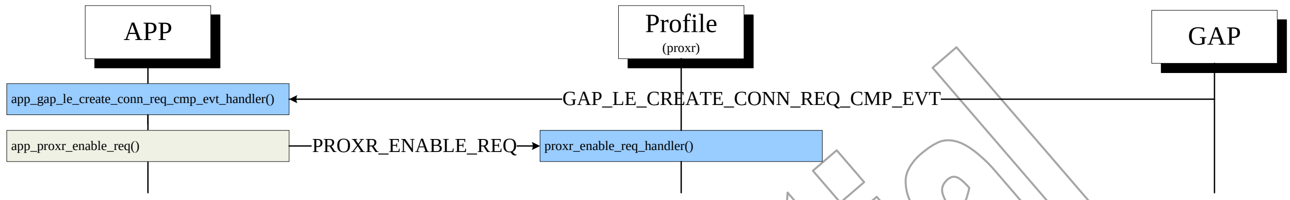


Figure 25 Connection

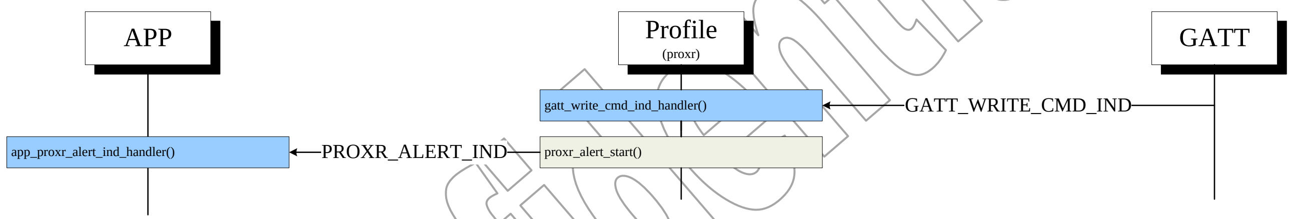


Figure 236 Write Alert

4. User Design

The file 'usr_design.c' shows how to used peripherals in the BLE application.
Button 4 is used to start/stop advertising.
Button 5 is used to stop alert.
LED1 is used to show link status.
Buzzer is used to show alert.

7.7 Device Driver

The Device Driver of the QN902x provides an interface of abstraction between the physical hardware and the application. System-level software developers can use the QN902x driver to do the fast application software development, instead of using the register level programming, which can reduce the total development time significantly.

This document only contains the brief driver descriptions including the followings: System Controller Driver, GPIO Driver, UART Driver, SPI Driver, I2C Driver, Timer Driver, RTC Driver, Watch Dog Timer Driver, PWM Driver, DMA Driver, ADC Driver, Analog Driver, Sleep Driver, Serial Flash Driver and RF Driver.

Please refer to QN902x driver API reference guide for details. In the driver programming guide, a description, usage and an illustrated example code are provided for each driver API. The full driver samples and driver source codes can be found in the QN902x software release package. The example of QN902x device driver provided in QN902x software package based on QN902x EVB and it may be modified for developing with a different hardware platform.

7.7.1 Device Driver File Structure

The source code of QN902x driver has five groups of files related to driver: cmsis, driver, lib, startup, config, the files in these groups are listed as follow:

- cmsis: CMSIS defines for a Cortex-M Microcontroller System.
  - core_cm0.c: CMSIS Cortex-M0 Core Peripheral Access Layer Source File.
  - core_cm0.h: CMSIS Cortex-M0 Core Peripheral Access Layer Header File.
  - core_cmInstr.h: CMSIS Cortex-M Core Instruction Access Header File.
  - core_cmFunc.h: CMSIS Cortex-M Core Function Access Header File.
  - QN9020.h: CMSIS compatible Cortex-M0 Core Peripheral Access Layer Header File for QN9020.
  - driver_QN9020.h: This file defines many inline functions which are used to read/write access to system registers.
- driver: QN9020 driver source code.
  - adc.h: ADC driver header file.
  - adc.c: ADC driver source file.
  - analog.h: Analog driver header file.
  - analog.c: Analog driver source file.
  - dma.h: DMA driver header file.
  - dma.c: DMA driver source file.
  - gpio.h: GPIO driver header file.
  - gpio.c: GPIO driver source file.
  - i2c.h: I2C driver header file.
  - i2c.c: I2C driver source file.
  - pwm.h: PWM driver header file.
  - pwm.c: PWM driver source file.
  - rtc.h: RTC driver header file.
  - rtc.c: RTC driver source file.
  - syscon.h: System controller driver header file.
  - syscon.c: System controller driver source file.
  - serialflash.h: Serial flash driver header file.
  - serialflash.c: Serial flash driver source file.
  - sleep.h: Sleep driver header file.
  - sleep.c Sleep driver source file.
  - spi.h: SPI driver header file.
  - spi.c: SPI driver source file.
  - timer.h: Timer driver header file.
  - timer.c: Timer driver source file.

- uart.h: UART driver header file.
- uart.c: UART driver source file.
- wdt.h: Watchdog timer driver header file.
- wdt.c: Watchdog timer driver source file.
- fw_func_addr.h: ROM driver API point address.
- nvds.h: NVDS driver header file.

- lib: Contains library of calibration, RF driver, and platform initial API.
  - qn902xbx_lib_lite.lib: Include calibration and RF driver.
  - lib.h: Header file of lib.
  - calibration.h: Calibration driver header file.
  - qnrf.h: RF driver header file.

- startup: QN9020 startup code.
  - startup.s: startup code of QN9020.

- config: QN9020 driver configurations.
  - driver_config.h: Driver configuration for QN9020, please refer to section Driver Configurations for details.
  - system.c: QN9020 system setup and initial configuration source file.
  - system.h: QN9020 system setup and initial configuration header file.

7.7.2 Driver Configuration

QN902x Driver example contains one configuration files: driver_config.h which defines driver status (enable or disable), realization method (interrupt or polling), which driver to use (driver source code or driver burned in ROM), driver callback status (enable or disable) and driver work mode (for example, I2C module work in MASTER or SLAVE mode). All the configurations can be modified by user, the following is an example of how to configure UART driver:

CONFIG_ENABLE_DRIVER_UART: This macro can be set to TRUE or FALSE, means to enable or disable UART driver, only if this macro value is TRUE, the other macros related to UART have meanings.

CONFIG_UART0_TX_DEFAULT_IRQHANDLER: This macro used to enable or disable UART0 TX default interrupt request handler, can be set to TURE or FALSE. If the macro defines to FALSE, user can rewrite a new handler to replace the default handler. This macro will be effective under the condition of UART driver is enabled and UART0 TX interrupt is enabled.

CONFIG_UART0_TX_ENABLE_INTERRUPT: Define this macro to TRUE to enable UART0 TX interruption, otherwise, UART0 data will be transmitted via polling.

CONFIG_ENABLE_ROM_DRIVER_UART: This macro set to TRUE means to use driver burned in ROM, all the UART APIs become to function pointer which point to ROM address and driver configurations are fixed, otherwise, the UART source code will be used, and user can modify them.

UART_CALLBACK_EN: This macro means enable or disable UART callback.

UART_BAUDRATE_TABLE_EN: This macro means enable or disable UART baud rate parameters table, if the macro define to FALSE, baud rate will be set by formula calculation.

7.7.3 System Controller Driver

QN902x System Controller is responsible for controlling Reset Management Unit (RMU), Clock Management Unit (CMU) and Power Management Unit (PMU). The following functions are included in these units:

- syscon_set_sysclk_src(), this function is used to set system clock source.
- syscon_set_ahb_clk(), this function is used to set AHB clock.
- syscon_set_apb_clk(), this function is used to set APB clock.
- syscon_set_timer_clk(), this function is used to set TIMER clock.
- syscon_set_usart_clk(), this function is used to set USART clock.
- syscon_set_ble_clk(), this function is used to set BLE clock.
- syscon_get_reset_cause(), this function is used to get system reset cause.
- syscon_enable_transceiver(), this function is used to enable or disable transceiver, contains BLE clock setting and REF PLL power setting.
- clock_32k_correction_init(), this function is used to initialize 32K clock correction.
- clock_32k_correction_enable(), this function is used to enable 32K clock correction.
- clock_32k_correction_cb(), this function will be called after 32K clock correction finish.
- clk32k_enable(), this function is used to enable 32K clock.
- memory_power_off(), this function is used to set memory power off.
- clk32k_power_off(), this function is used to set 32K clock power off.
- syscon_set_xtal_src(), this function is used to set XTAL clock source type.

7.7.4 GPIO Driver

QN902x has up to 31 General Purpose I/O pins can be shared with other function pins, it depends on the pin mux configuration. The main functions of GPIO driver list as follow:

- gpio_init(), this function is used to initialize callback function pointer and enable GPIO NVIC IRQ.
- gpio_read_pin(), this function is used to get a specified GPIO pin's level.
- gpio_write_pin(), this function is used to set level high(1) or low(0) to a specified GPIO pin.
- gpio_set_direction(), this function is used to set direction(input or output) of a GPIO pin.
- gpio_read_pin_field(), this function is used to read a set of GPIO pins's level.
- gpio_write_pin_field(), this function is used to write a set of GPIO pins's level.
- gpio_set_direction_field(), this function is used to set direction (input or output) of a set of GPIO pins.
- gpio_toggle_pin(), this function is used to set a specified GPIO pin to the opposite level that is currently applied.

- gpio_set_interrupt(), this function is used to configure a specified GPIO pin's interrupt.
- gpio_enable_interrupt(), this function is used to enable a specified GPIO pin's interrupt.
- gpio_disable_interrupt(), this function is used to disable a specified GPIO pin's interrupt.
- gpio_pull_set(), this function is used to set a specified GPIO pin's mode.
- gpio_wakeup_config(), this function is used to configure wakeup GPIO pin.
- gpio_sleep_allowed(), this function is used to check the GPIO module sleep is allowed or not
- gpio_clock_on(), this function is used to enable clock of GPIO module.
- gpio_clock_off(), this function is used to disable clock of GPIO module.
- gpio_reset(), this function is used to reset GPIO module.

7.7.5 UART Driver

QN902x have two configurable full-duplex UART ports, each UART port support hardware flow control and baud rate is up to 2MHz while UART clock is 16MHz, the UART module performs a serial-to-parallel conversion on data characters received from the peripheral, and a parallel-to-serial conversion on data characters received from the CPU, QN902x UART driver contains APIs to realize these operation. The main functions are described as follow:

- uart_init(), this function is used to initialize UART, it consists of baud-rate, parity, data-bits, stop-bits, over sample rate and bit order, the function is also used to enable specified UART interrupt, and enable NVIC UART IRQ.
- uart_read(), this function is used to read Rx data from RX FIFO and the data will be stored in buffer, as soon as the end of the data transfer is detected, the callback function is executed.
- uart_write(), this function is used to write data into TX buffer to transmit data by UART, as soon as the end of the data transfer is detected, the callback function is executed.
- uart_printf(), print a string to specified UART port.
- uart_finish_transfers(), waiting for specified UART port transfer finished.
- uart_flow_on(), enable specified UART port hardware flow control.
- uart_flow_off(), disable specified UART port hardware flow control.
- uart_rx_enable(), enable or disable specified UART RX port
- uart_tx_enable(), enable or disable specified UART TX port.
- uart_clock_on(), this function is used to enable clock of UART module.
- uart_clock_off(), this function is used to disable clock of UART module.
- usart_reset(), this function is used to reset UART and SPI module.

7.7.6 SPI Driver

The Serial Peripheral Interface (SPI) is a synchronous serial data communication protocol which operates in full duplex mode. Devices communicate in master/slave mode with 4-wire bi-direction interface. QN902x contain 2 sets of SPI controller performing a serial-to-parallel conversion on data received from a peripheral device, and a parallel-to-serial conversion on data transmitted to a peripheral device. Each SPI set can drive up to 2 external peripherals. It also can be driven as the slave device when the slave mode is enabled. The main SPI driver APIs are described as follow:

- spi_init(), this function is used to initialize SPI, it consists of bit rate, transmit width, SPI mode, big/little endian, MSB/LSB first, master/salve, the function is also used to enable specified SPI interrupt, and enable NVIC SPI IRQ.
- spi_read(), this function is used to read Rx data from RX FIFO and the data will be stored in buffer, as soon as the end of the data transfer or a buffer overflow is detected, the callback function is called.
- spi_write(), this function is to write data into TX buffer to transmit data by SPI, as soon as the end of the data transfer is detected, the callback function is called.
- spi_clock_on(), this function is used to enable clock of SPI module.
- spi_clock_off(), this function is used to disable clock of SPI module.

7.7.7 I2C Driver

I2C is bi-directional serial bus with two wires that provides a simple and efficient method of data exchange between devices. The I2C standard is a true multi-master bus including collision detection and arbitration that prevents data corruption if two or more masters attempt to control the bus simultaneously. For QN902x, I2C device could act as master or slave and I2C driver can help user to use I2C functions easily. The main function list as follow:

- i2c_init(), this function is used to initialize I2C in master mode, SCL speed is up to 400KHz, the function is also used to enable I2c interrupt, and enable NVIC I2C IRQ.
- i2c_read(), this function is used to complete a I2C read transaction from start to stop. All the intermittent steps are handled in the interrupt handler while the interrupt is enabled. Before this function is called, the read length, write length, I2C master buffer, and I2C state need to be filled, please refer to i2c_byte_read(). As soon as the end of the data transfer is detected, the callback function is called.
- i2c_write(), this function is used to complete a I2C write transaction from start to stop. All the intermittent steps are handled in the interrupt handler while the interrupt is enabled. Before this function is called, the read length, write length, I2C master buffer, and I2C state need to be filled, please refer to i2c_byte_write(). As soon as the end of the data transfer is detected, the callback function is called.
- i2c_byte_read(), read a byte data from slave device, the data address is 8 bits.
- i2c_byte_write(), write a byte data to a 8 bits address of slave device.
- i2c_byte_read2(), read a byte data from slave device, the data address is 16 bits.
- i2c_byte_write2(), write a byte data to a 16 bits address of slave device.
- i2c_nbyte_read (), read n byte data from slave device, the data address is 8 bits.
- i2c_nbyte_write(), write n byte data to a 8 bits address of slave device.
- i2c_nbyte_read2 (), read n byte data from slave device, the data address is 16 bits.
- i2c_nbyte_write2(), write n byte data to a 16 bits address of slave device.

7.7.8 Timer Driver

QN902x have two 32-bit timers Timer0/1, and two 16-bit timers Timer2/3. All the Timers support four operation modes, which allow user to easily implement a counting scheme. The Timers can perform functions like frequency measurement, event counting, interval measurement, clock generation, delay timing, and so on. The Timers also can generate an interrupt signal upon timeout, or provide the current value of count during operation, and support external count and capture functions. The main Timer driver APIs are listed as follow:

- timer_init(), this function is used to initialize the timer modules.
- timer_config(), this function is used to configure timer to work in timer mode, and set timer pre-scaler and top count number.
- timer_pwm_config(), this function is used to configure timer to work in PWM mode, and set timer pre-scaler, period, and pulse width.
- timer_capture_config(), this function is used to configure timer to work in capture mode, and set input capture mode, timer pre-scaler, count or event number.
- timer_enable(), this function is used to enable or disable the specified timer.
- timer_delay(), this function is used to do precise time delay.
- timer_clock_on(), this function is used to enable clock of timer module.
- timer_clock_off(), this function is used to disable clock of timer module.
- timer_reset(), this function is used to reset timer module.

7.7.9 RTC Driver

QN902x Real Time Clock (RTC) module provides user the real time and calendar message, the RTC real time based on external or internal low power 32 KHz clock, and its main functions are listed as follow:

- rtc_init(), initial RTC environment variable.
- rtc_time_set(), this function is used to set RTC date, time and install callback function.
- rtc_time_get(), this function is used to get current RTC time.
- rtc_correction(), this function is used to correct RTC time after CPU wakeup.
- rtc_clock_on(), this function is used to enable clock of RTC module.
- rtc_clock_off(), this function is used to disable clock of RTC module.
- rtc_reset(), this function is used to reset RTC module.

7.7.10 Watchdog Timer Driver

The purpose of Watchdog Timer (WDT) is to perform a system reset after the software running into a problem. This prevents system from hanging for an infinite period of time. The main functions of QN902x WDT driver are listed as follow:

- wdt_init(), this function is used to set WDT work mode and WDT time-out interval.
- wdt_set(), this function is used to set WDT time-out interval.
- wdt_clock_on(), this function is used to enable clock of WDT module.
- wdt_clock_off(), this function is used to disable clock of WDT module.
- wdt_reset(), this function is used to reset WDT module.

7.7.11 PWM Driver

QN902x PWM module provides two channels with programmable period and duty cycle. The main functions of PWM driver are listed as follow:

- pwm_init(), this function is used to initialize the specified PWM channel.
- pwm_config(), this function is used to configure PWM pre-scaler, period, and pulse width.
- pwm_enable(), this function is used to enable/disable the specified PWM channel.
- pwm_clock_on(), this function is used to enable clock of PWM module.
- pwm_clock_off(), this function is used to disable clock of PWM module.

7.7.12 DMA Driver

QN902x contains a single channel DMA controller, which support 4 types transfer mode: memory to memory, peripheral to memory, memory to peripheral, peripheral to peripheral. The main functions of DMA driver are listed as follow:

- dma_init(), this function is used to clear callback pointer and enable DMA NVIC IRQ.
- dma_memory_copy(), this function is used to transfer data from memory to memory by DMA.
- dma_tx(), this function is used to transfer data from memory to peripheral by DMA.
- dma_rx(), this function is used to transfer data from peripheral to memory by DMA.
- dma_transfer(), this function is used to transfer data from peripheral to peripheral by DMA.
- dma_abort(), this function is used to abort current DMA transfer, and usually used in undefined transfer length mode.
- dma_clock_on(), this function is used to enable clock of DMA module.
- dma_clock_off(), this function is used to disable clock of DMA module.
- dma_reset(), this function is used to reset DMA module.

7.7.13 ADC Driver

QN902x contain an up to 12 bits resolution successive approximation analog-to-digital converter (SAR A/D converter) with 12 input channels. It takes about 20 ADC clock cycles to convert one sample, and the maximum input clock to ADC is 16MHz. The A/D converter supports multi operation modes and can be started by 4 types of trigger source. The main functions of ADC driver are listed as follow:

- adc_init(), this function is used to set ADC module work clock, resolution, trigger mode and interrupt.
- adc_compare_init(), this function is used to initialize ADC window comparator.
- adc_decimation_enable(), this function is used to enable ADC decimation.
- adc_read(), this function is used to read specified ADC channel conversion result in specified mode.
- adc_clean_fifo(), this function is used to clear ADC data FIFO.
- adc_enable(), this function is used to enable or disable ADC module.
- adc_clock_on(), this function is used to enable clock of ADC module.
- adc_clock_off(), this function is used to disable clock of ADC module.

- adc_reset(), this function is used to reset ADC module.
- adc_power_on(), this function is used to power on ADC module.
- adc_power_down(), this function is used to power down ADC module.
- adc_buf_in_set(), this function is used to set ADC buffer input source.
- adc_buf_gain_set(), this function is used to set ADC buffer gain stage, and only available at the input mode with buffer driver.
- adc_offset_get(), this function is used to get ADC offset for conversion result correction, and should be called after ADC initialization and buffer gain settings.
- ADC_SING_RESULT_mV(), this function is used to calculate ADC single mode voltage value.
- ADC_DIFF_RESULT_mV(), this function is used to calculate ADC differential mode voltage value.

7.7.14 Analog Driver

QN902x analog circuit contains: clock generator, two comparators, ADC, battery monitor, brown out detector, temperature sensor, RF, power and reset modules. Please refer to system controller driver for how to control clock generator, power and reset modules, refer to ADC driver for how to use ADC, and refer to RF driver for how to set frequency, the rest modules are described in this section and main functions of analog driver are listed as follow:

- comparator_init(), this function is used to initialize specified analog comparator, and to register callback function.
- comparator_enable(), this function is used to enable or disable specified analog comparator.
- battery_monitor_enable(), this function is used to enable or disable battery monitor.
- brown_out_enable(), this function is used to enable or disable brown out detector.
- temp_sensor_enable(), this function is used to enable or disable temperature sensor.
- get_reset_source(), this function is used to get system reset cause.

7.7.15 Sleep Dirver

In QN902x, three sleep modes are defined according to cortex-M0 low power modes: CPU clock gating mode, CPU deep clock gating mode, CPU sleep mode. The main driver functions are listed as follow:

- sleep_init(), this function is used to initialize system sleep mode.
- enter_sleep(), this function is used to enable system to enter sleep status.
- wakeup_by_gpio(), this function is used to enable system wakeup by GPIO.
- wakeup_by_analog_comparator(), this function is used to enable system wakeup by analog comparator.
- wakeup_by_sleep_timer(), this function is used to enable system wakeup by sleep timer.
- sleep_cb(), callback function of wakeup.
- enter_low_power_mode(), this function is used to set MCU entering into low power mode.
- exit_low_power_mode(), this function is used to set MCU exiting from low power mode.
- restore_from_low_power_mode(), this function is used to set MCU restoring from low power mode.

7.7.16 Serial Flash Driver

QN902x contains a Serial Flash Controller, which has mainly 2 functions: access serial flash (erase/read/write) and boot from serial flash (copy code from serial flash to RAM and then to execute). The main functions of serial flash driver are listed as follow:

- read_flash_id(), this function is used to read serial flash ID, which consists of 3 or 4 bytes depends on difference vendor.
- chip_erase_flash(), this function is used to erase entire serial flash.
- sector_erase_flash(), this function is used to erase serial flash sector.
- block_earse_flash(), this function is used to erase serial flash block.
- read_flash(), this function is used to read data from serial flash.
- write_flash(), this function is used to write data to serial flash.

7.7.17 RF Driver

The API of RF driver are listed as follow, please refer to qnrf.h for function prototype:

- rf_enable_sw_set_freq(), this function is used to enable software to set radio frequency.
- rf_set_freq(), this function is used to set radio frequency.
- rf_enable(), this function is used to set RF working mode.
- rf_tx_power_level_set(), this function is used to set RF TX power level.
- rf_tx_power_level_get(), this function is used to get RF TX power level.

8. Network Processor

It is called Network Processor Mode when the link layer, host protocols and profiles run on the QN902x and the application executes on an external microcontroller or PC (See Figure 22). These two components communicate via ACI (Application Control Interface) over UART/SPI interface.

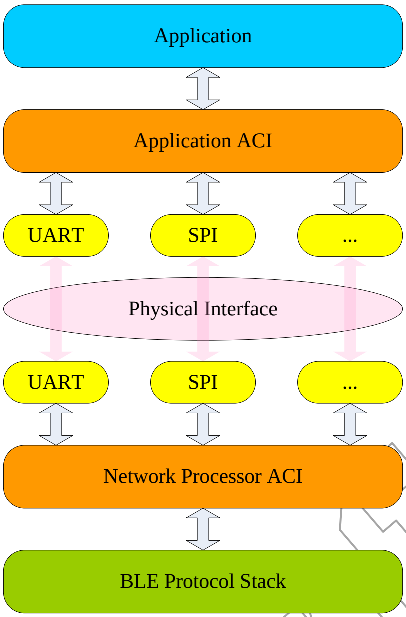


Figure 24 ACI Driver Interface

The main goal of the transport layer between the application and the BLE protocol stack is transparency. The hardware interface is independent of the message passing over the transport layer. This allows the message API or hardware driver to be upgraded without affecting each other.

The physical transport layer for ACI is the same as Controller mode. The reception and sending of these interface messages is adapted from the HCI module. Flow control with RTS/CTS is used to prevent UART buffer overflow. If the buadrate of UART is less than or equal 9600, UART flow control could not be used.

Table 41 Uart Settings

Setting	Value
Data	8 bits
Parity	No parity
Stop bit	1 stop bit
Flow control	RTS/CTS

The following setting is used for ACI SPI Transport Layer.

Table 42 SPI Settings

Setting	Value
Mode	Slave
Width	8 bits

For UART/SPI transport, the HCI uses a single byte at the beginning of the packet to identify the type of the packet (Command, Event, ACL or SCO data packets). In order to identify the Application Control Interface messages, the choice was made to use such a byte, different from those 4. There was no need to have a complex classification of messages like in HCI; the same byte (**0x05**) is used for receiving and sending messages.
All of the message between application and protocol stack will be exchanged over ACI interface. In order to not have additional processing of sent and received messages, the structure of the kernel messages is directly used in the application to format the packet to be sent to the BLE protocol stack in the QN902x. (See chapter 4)
When kernel messages from protocol stack are destined to an Application Layer, the source and destination task identifiers allow the Kernel to know that the message it is redirecting should go to a task that exists outside the QN902x, thus requiring to be sent through ACI. A message handler in the ACI is called and then the kernel message is transformed into an ACI packet and sent through the interface to external micro controller, and then free.

When an ACI packet is received through the physical interface, according to its header information it will cause an allocation of a kernel message with a corresponding source and destination task identifiers and a corresponding length of parameter structure. The kernel message parameter structure will be filled with the unpacked ACI parameter values. This kernel message is then delivered to the kernel to be sent to the appropriate destination task and processed.

8.1 ACI PDU Format

This section describes the non-standard Application Control Interface (ACI) and explains the format of the interface packets passed between the application through the physical interface with the QN902x running in Network Processor mode.

Table 43 ACI PDU Format

Items	Packet Type	MSG_ID	DEST_ID	SRC_ID	LEN	PAYLOAD
Length(bytes)	1	2	2	2	2	By LEN
Description	0x05	Message ID	Destination task identifier	Source task identifier	Length of payload field	Message Parameter

The transport function was adapted to add the 0x05 byte as Packet Type before all packets. The Application Control Interface message contains all of the information included in a kernel message (See chapter 3.3). The message structures are described in chapter 3.3 which helps understanding how to construct ACI message (where to pad a packet that is built to map onto a kernel message, what task identifiers to use in the packet, etc.) by application. The QN902x process data in ACI message using little endian, so transfer LSB first for every field in ACI message.

Task Identifiers are simple shifted indexes for tasks that are instantiated only once. But for those that are instantiated per connection, they are re-indexed with the connection index e.g.: (TASK_SMPC <<8) + connection_index. When a kernel message is received outside the QN902x, the task identifier will be indexed with the connection index which must thus be known in the application handling sending and receiving messages to the right tasks. Generally in the first messages corresponding to a connection, the index is present so it can be recovered and used in a tool to build task identifiers for sending requests, etc.

8.2 ACI Message Example

A simple illustration of ACI message GAP_LE_CREATE_CON_REQ sent by Application Task is as below.

Packet Type : QN ACI (0x05)

MSG_ID : GAP_LE_CREATE_CONN_REQ (0x3006)

DEST_ID : GAP_TASK (0x000c)

SRC_ID : APP_TASK (0x0015)

LEN : The length of struct gap_le_create_conn_req (0x001A)

PAYLOAD : Parameter of the GAP_LE_CREATE_CONN_REQ message

```
struct gap_le_create_conn_req
{
    /// LE connection command structure
    struct llm_le_create_con_cmd create_cnx;
};

///LLM LE Create Connection Command parameters structure
struct llm_le_create_con_cmd
{
    ///Scan interval
    uint16_t        scan_intv;
    ///Scan window size
    uint16_t        scan_window;
    ///Initiator filter policy
    uint8_t         init_filt_policy;
    ///Peer address type - 0=public/1=random
    uint8_t         peer_addr_type;
    ///Peer BD address
    struct          bd_addr peer_addr;
    ///Own address type - 0=public/1=random
    uint8_t         own_addr_type;
    ///Minimum of connection interval
    uint16_t        con_intv_min;
    ///Maximum of connection interval
    uint16_t        con_intv_max;
    ///Connection latency
    uint16_t        con_latency;
    ///Link supervision timeout
    uint16_t        superv_to;
    ///Minimum CE length
    uint16_t        ce_len_min;
    ///Maximum CE length
    uint16_t        ce_len_max;
};
```

Message Parameter

Scan Interval: 0x0640

Scan Window: 0x0320

Filter policy: 0

Peer Address Type: 0

Peer Address: 01:01:01:BE:7C:08

Connection Interval Minimum: 0x00A0
Connection Interval Maximum: 0x00A0
Latency: 0x0000
Supervision Timeout: 0x01F4
Minimum CE Length: 0x0000
Maximum CE Length: 0x0140

Data on UART

<UART>TX:[05:06:30:0C:00:15:00:1A:00:40:06:20:03:00:00:01:01:01:BE:7C:08:00:00:A0:00:A0:00:00:00:F4:01:00:00:40:01]

9. Controller Mode

It is called Controller Mode when only the link layer runs on the QN902x. The host protocol, profiles and application all execute on an external microcontroller. (See Figure 27) These two components communicate via HCI. The HCI provides uniform command method of accessing controller capabilities.

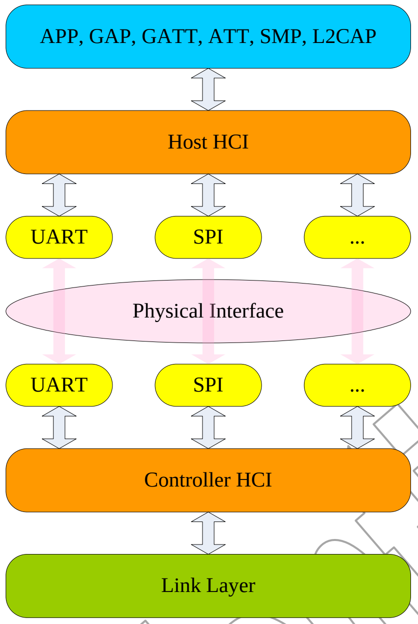


Figure 25 HCI Driver Interface

To control the Link Layer below the HCI, a hardware transport layer is needed. The UART/SPI are available for transferring HCI command, event and data in the QN902x. The following setting is used for HCI UART Transport Layer. Flow control with RTS/CTS is used to prevent UART buffer overflow. If the buadrate of UART is less than or equal 9600, UART flow control could not be used.

Table 44 UART Settings

Setting	Value
Data	8 bits
Parity	No parity
Stop bit	1 stop bit
Flow control	RTS/CTS

9.1 HCI PDU Format

There are 3 types of packet that can be exchanged over the HCI.

- Command (from host to controller)
- Event (from controller to host)
- Data (both directions)

HCI Command Packets can only be sent to the Controller. The Length depends of the command type.

Figure 26 HCI Command Packet Format

HCI Data Packets can be sent both to and from the Controller. Connection Handles are used to identify logical channels between the Host and LE Controller. Connection Handles are assigned by the LE Controller when a new logical link is created, using the LE Connection Complete event. No Broadcast Handle for LE. The flag field indicates if the L2CAP or LL has fragmented the Data or not. The Length depends of the number of data to transmit.

Figure 27 HCI ACI Packet Format

HCI Event Packets can only be sent from the Controller. The Length depends of the number of parameters to return.

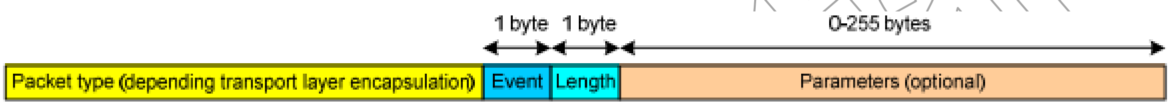


Figure 28 HCI Event Packet Format

Table 45 Uart Transport Layer

HCI packet type	Packet Indicator
0x01	HCI commands
0x02	HCI data
0x03	Reserved
0x04	HCI events

9.2 Supported Commands and Events

Generic Events:

- Command Complete Event.
- Command Status Event.
- Hardware Error Event.

Device Setup:

- Reset Command.
- Controller Flow Control:
- Number of Completed Packets Event.
- LE Read Buffer Size Command

Controller Information:

- Read Local Version Information Command.
- Read Local Supported Commands Command.
- Read Local Supported Features Command.
- Read BD_ADDR Command.
- LE Read Local Supported Features Command.
- LE Read Supported States Command.

Controller Configuration:

- LE Set Advertise Enable Command.
- LE Set Advertising Data Command.
- LE Set Advertising Parameters Command.
- LE Set Random Address Command.
- LE Set Scan Response Data Command.

Device Discovery:

- LE Advertising Report Event.
- LE Set Scan Enable Command.
- LE Set Scan Parameters Command.

Connection Setup:

- Disconnect Command.
- Disconnection Complete Event.
- LE Connection Complete Event.
- LE Create Connection Cancel Command.
- LE Create Connection Command.

Remote Information:

- Read Remote Version Information Command.
- Read Remote Version Information Complete Event.
- LE Read Remote Used Features Command.
- LE Read Remote Used Features Complete Event.

Connection State:

- LE Connection Update Command.
- LE Connection Update Complete Event.

Quality of Service:

- Flush Command.

- Flush Occurred Event.

Physical Links:

- LE Set Host Channel Classification Command.

Host Flow Control:

- Host Buffer Size Command.
- Set Event Mask Command.
- Set Controller To Host Flow Control Command.
- Host Number Of Completed Packets Command.
- Data Buffer Overflow Event.
- LE Add Device To White List Command.
- LE Clear White List Command.
- LE Read White List Size Command.
- LE Remove Device from White List Command.
- LE Set Event Mask Command.

Link Information:

- Read Transmit Power Level Command.
- Read RSSI Command.
- LE Read Advertising Channel TX Power Command.
- LE Read Channel Map Command.

Authentication and Encryption:

- Encryption Change Event.
- Encryption Key Refresh Complete Event.
- LE Encrypt Command.
- LE Long Term Key Request Event.
- LE Long Term Key Request Reply Command.
- LE Long Term Key Request Negative Reply Comma
- LE Rand Command.
- LE Start Encryption Command.

Testing:

- LE Receiver Test Command.
- LE Transmitter Test Command.
- LE Test End Command.

Direct Test mode:

- LE Test Status Event.
- LE Test Packet Report Event.

Quintic Vender Command:

- **LE_QN_NVDS_GET_CMD**

OGF	OCF	Parameter Length	Command Parameters	Return Parameters
0x3F	0x09	0x2	TAG ID(2)	Status(1) length(1) data(<=128)

The LE_QN_NVDS_GET_ CMD command is used by the Host to get a specific tag in the NVDS.

- **LE_QN_NVDS_PUT_CMD**

OGF	OCF	Parameter Length	Command Parameters	Return Parameters
0x3F	0x0a	<=0x83	TAG ID (2) length(1) data(<=128)	Status(1)

The LE_QN_NVDS_PUT_CMD command is used by the Host to add a specific tag to the NVDS.

- **LE_QN_REG_RD_CMD**

OGF	OCF	Parameter Length	Command Parameters	Return Parameters
0x3F	0x30	0x4	Register address(4)	Status(1) Register address(4) Register value(4)

The LE_QN_REG_RD_CMD command is used by the Host to get the value of the specific register.

- **LE_QN_REG_WR_CMD**

OGF	OCF	Parameter Length	Command Parameters	Return Parameters
0x3F	0x31	0x8	Register address(4) Register value(4)	Status(1) Register address(4)

The LE_QN_SET_BD_ADDR_CMD command is used by the Host to set the value of the specific register.

- **LE_QN_SET_BD_ADDR_CMD**

OGF	OCF	Parameter Length	Command Parameters	Return Parameters
0x3F	0x32	0x6	BD address	Status

The LE_QN_SET_BD_ADDR_CMD command is used by the Host to set the LE Bluetooth Device Address in the Controller.

- **LE_QN_SET_TYPE_PUB_CMD**

OGF	OCF	Parameter Length	Command Parameters	Return Parameters
0x3F	0x33	0x0		Status

The LE_QN_SET_TYPE_PUB_CMD command is used by the Host to set the LE Bluetooth Device Address Type to Public Address in the Controller.

- **LE_QN_SET_TYPE_RAND_CMD**

OGF	OCF	Parameter Length	Command Parameters	Return Parameters
0x3F	0x34	0x0		Status

The LE_QN_SET_TYPE_RAND_CMD command is used by the Host to set the LE Bluetooth Device Address Type to Random Address in the Controller.

9.3 Direct Test Mode

One of the main purposes of QN902x controller mode is to provide a solution for testing transceiver performance. The Direct Test Mode is used to control the device under test and to provide a report to the tester. The controller mode of QN902x offers complete test commands and events used in the Direct Test Mode which can be set up over HCI.

Supported Commands and Events for Direct Test Mode:

- Reset Command.
- BLE Receiver Test Command.
- BLE Transmitter Test Command.
- BLE Test End Command.
- BLE Test Status Event.
- BLE Test Packet Report Event.

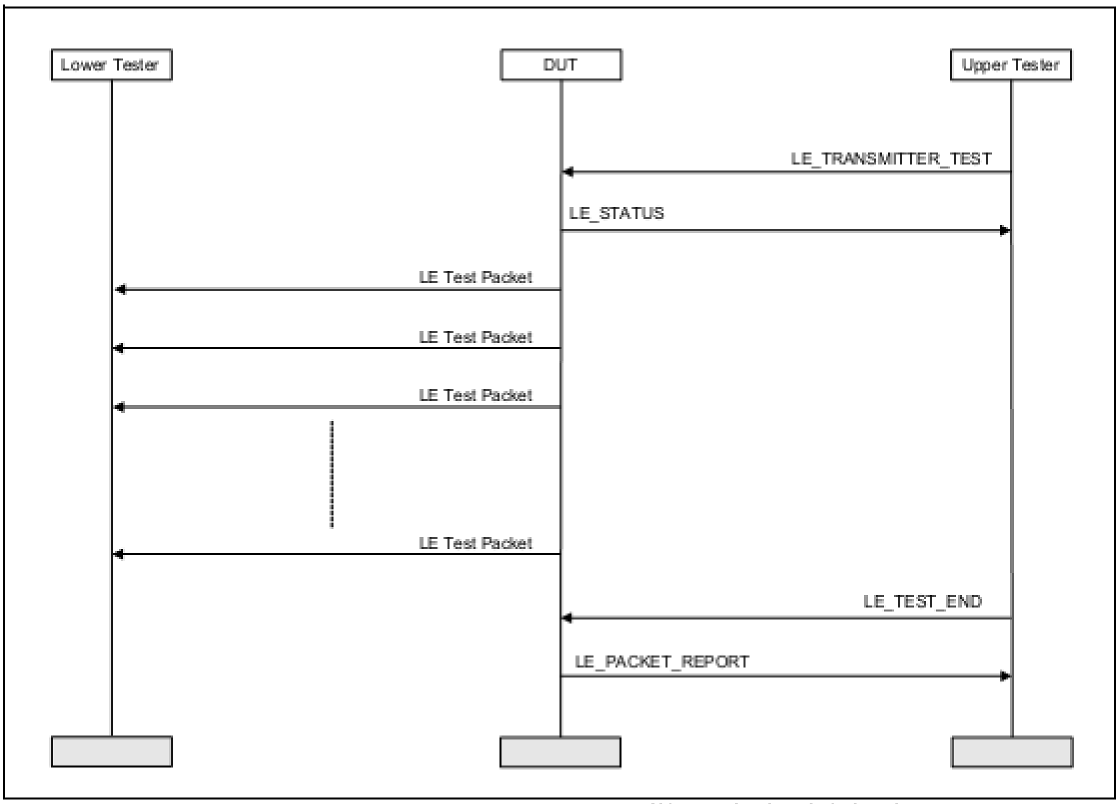


Figure 29 Transmitter Test MSC

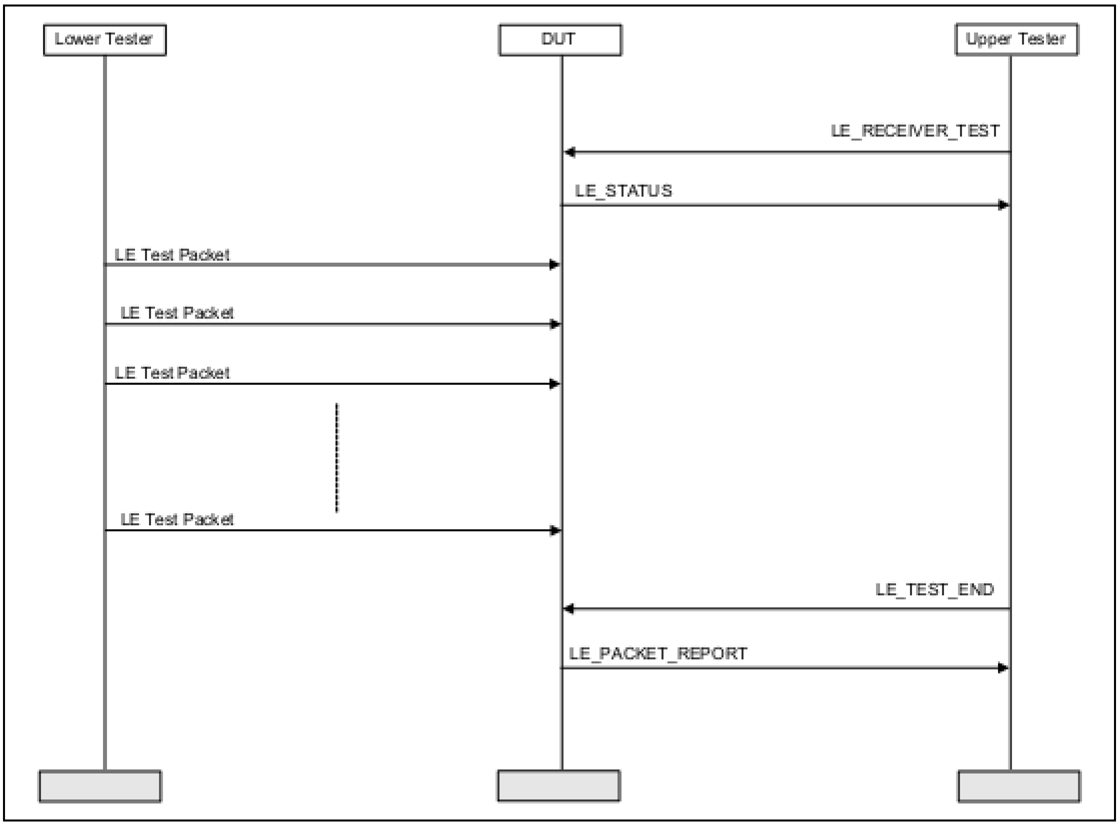


Figure 30 Receiver Test MSC

The switch of work mode is very easy, so it is convenient to integrate Direct Test Mode in the real product. When the product needs to be tested, the application can switch the work mode to Controller mode and the product is controlled by tester over HCI. When the testing is finished, the application can switch work mode back.

Release History

REVISION	CHANGE DESCRIPTION	DATE
0.1	Initial release	2013-01-30
0.2	Update chapter of application samples and ACI message example	2013-03-07
0.3	Added abbreviations. Added pseudo code of message scheduler Updated NVDS TAGs.	2013-03-28
0.4	Updated application initialization flow and device driver.	2013-04-25
0.5	Updated chapter of application samples and device driver.	2013-05-17
0.6	Updated to firmware version v18.	2013-07-10
0.7	Updated to SDK v0.9.2.	2013-08-21
0.8	Updated to SDK v0.9.6.	2013-10-28
0.9	Updated the figure of initialization flow.	2013-11-22
1.0	Updated to SDK v0.9.8.	2013-12-18
1.1	Updated to SDK v1.0.0	2014-02-11
1.2	Updated to SDK v1.2.0	2014-04-08
1.3	Updated flash arrangements	2014-06-03
1.4	Updated table number	2014-07-07