BT-Gear Family: GR661X Series

Software Programming Guide

Printed in Taiwan 2020

Gear Radio Electronics Corp.
Rm. 417, Innovation Incubation Center., No.101, Sec. 2, Guang-fu Rd., East Dist., Hsinchu City 300, Taiwan, 30013

Gear Radio Electronics Corp.'s home page can be found at:
http://www.gearradio.net

Revision History

Revision	Issue Date	Comments	Author
1.0	October 08, 2020	Version 1	Eric Peng
1.1	October 27, 2020	Remove no use word	Eric Peng

Table of Contents

List of Figures

1. Introduction

This document introduces gear radio software architecture and introduces application layer basically API. The chapter 3 shows how App Task operation mechanism and guide user how to use it.

1-1 Acronyms and abbreviations

Acronym or abbreviation	Writing out in full	Comments
MMI	Man Machine Interface	

Table 1.1-1 Acronyms and abbreviations

2. Task Architecture

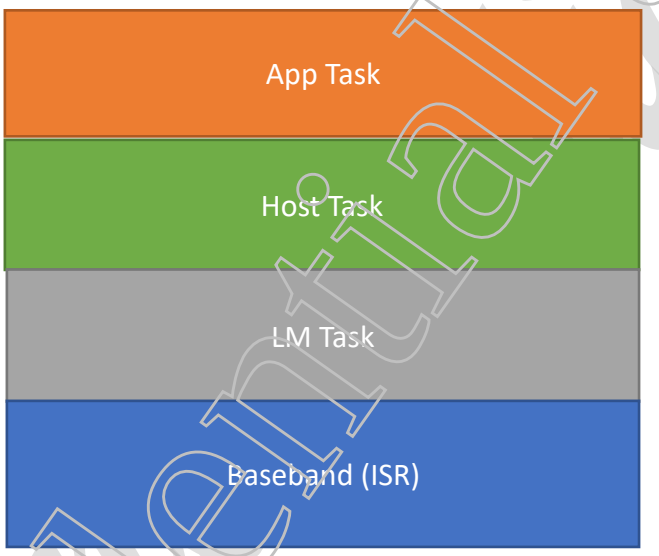


Figure 2.1 generic task architecture

GR661x software uses freeRTOS, and the Figure 2-1 show the mainly tasks corresponding Bluetooth specification system architecture.

2-1 App Task

This task handles all MMI commands and events, such as key events, audio control commands, and media events, etc. The Application Layer chapter introduces more detail how it works.

2-2 Host Task

The Host is a logical entity, the Bluetooth almost profiles and protocols are included in this task. And the Host is below app task and above the Host Controller interface (HCI). This task almost functions are protected.

2-3 LM Task

The LM (Link Manager) is responsible for the creation, modification and release of logical links, as well as the update of parameters related to physical links between devices. This task function is NOT open.

2-4 Baseband

This block is the baseband functions, it is responsible for link control and many RF related control. This part is NOT open.

3. App Task

App Task controls MMI and its actions could be configurated by the configuration tool. The application task also includes many sub-tasks (includes the task handler and its own variables), and the sequence to execute the sub-task handlers could be scheduled by the generation of message in order.

There are 2 functions: initial function and handler function, next section describes how they work.

The timer and the debug print log usually be used while developing. In App Task, the timer is designed and merge into message data “MessageSendLater”. And the “Print Log” section introduces how to print log in app task.

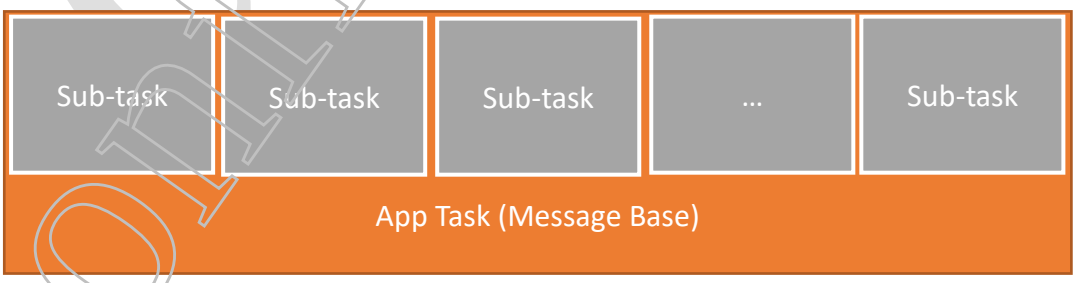


Figure 3.1 sub-task in App Task

3-1 SubTask

3-1-1 Sub Task Initialization Function

The sub-task usually exists one initial function and one handler. Initial function will be executed once while App Task creation.

In this function, developers are able to set/get its own variables or states.

And APP Task uses message to achieve scheduling, control, timer application. The initial function must define the subtask handler if developers want to receive any message.

Because the message uses the task pointer to assign which task to execute, in the sub-task initial function, developers can assign task pointer to the sub-task handler.

In other words, other sub-tasks want to send message to this sub-task, it need to point to this task's handler. Following is the example how sub-task declares its task handler and send message from others sub-task.

Ex.

Header File

```
typedef struct
{
    TaskData    task;
    …
} TestTaskData;

extern void TestInit ( void );
extern TestTaskData gTestTask;
```

C file

```
void TESTInit ( void )
{
    gTestTask.task.handler = TestMessageHandler;
    …
}

void TestMessageHandler ( Task pTask, MessageId pId, Message pMessage )
{
    …
}
```

Others sub-task want to send messages to the test sub-task, it needs to point to the test handler.

```
MessageSend((Task)&gTestTask, messageID ,NULL);
```

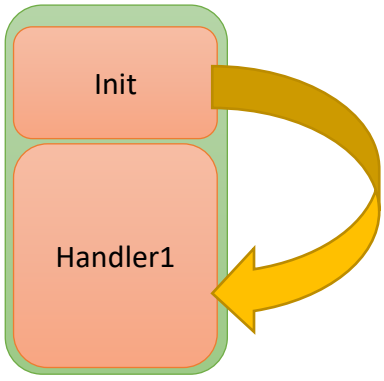
3-1-2 Sub Task Handler

After initial function assigns the handler function, the sub-tasks can send messages to each sub-task. Usually, handler function uses switch case to process different message due to one handler may receive many different messages.

Developers can use this mechanism to achieve many applications.

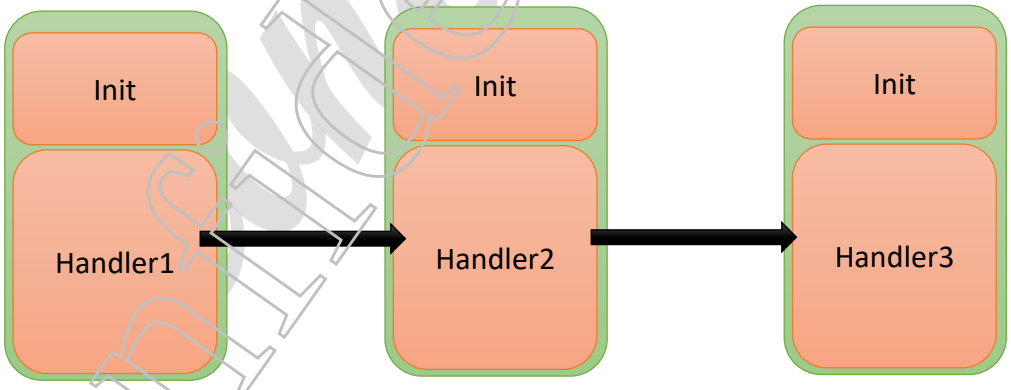
Following examples show some applications.

Ex. Send a message from initial function



```
Void TESTInit(void)
{
      gTestTask.task.handler = TestMessageHandler;
      …
      StatusA = getStatus;
      MessageSend((Task)&gTestTask, AAA ,NULL);
      …
}

void TestMessageHandler ( Task pTask, MessageId pId, Message pMessage )
{
      If(pId == AAA)
      {
            // receive message AAA
      }
}
```

Ex. Scheduling function, message A, B and C generated from different sub-task and execution sequentially



Developers also can use message to achieve timer effect, please reference Timer Application.

3-2 Message

The message consists of task pointer, message id, and data pointer. The message.h shows the related message API. All sub-tasks in App Task can use these APIs to achieve sequence or periodically send message.

3-2-1 Member of Message

TASK POINTER OF MESSAGE

The task pointer is used to point the sub-task handler. The message loop uses this information to call corresponding handler.

MESSAGE ID

The message identification number usually used to indicate the different events. Developers can define its own events such as button_press_event, button_release_event.

There are many message ids already defined, usually, developers use switch case to process different message id in handler of subtask.

```
void Handler ( Task pTask, MessageId pId, Message pMessage )
{
    switch(pId)
    {
        case AAA:
            ...
        break;
        case BBB:
            ...
        break;
            ...
    }
}
```

DATA POINTER

The message can carry some data while sending message. Developers can input NULL parameter if there is no data want to send. Correspondingly, developers need

allocate a memory and fill fit data, and call MessageSend API let this data point as the 3rd input parameter.

Following examples show send a message and data pointer is point to a 10 bytes data.

```
Header File:
    typedef struct
    {
        uint8_t data[10];
    }PACKED DATA_S;
==================================================
void Handler ( Task pTask, MessageId pId, Message pMessage )
{
    TaskMsg_t taskMsg = osTaskMsg_Alloc( 2, 10 );
    DATA_S *pMsg = osTaskMsg_GetDataFieldOrigin(taskMsg);

    memset(&pMsg->data, 0x00, 10);
    MessageSend(&theSink.task, AAA, taskMsg);
}
```

Developers also can create a new sub-task, and control, communicate with others sub-task by message. So, following sections introduce basically message APIs.

The message.h describes all message APIs in detail, there are some examples in the following.

```
void Handler1 ( Task pTask, MessageId pId, Message pMessage )
{
    switch(pId)
    {
        case AAA:
            MessageSend((Task)& Task2, BBB ,NULL);
            // send message BBB to task 2
        break;
    }
}
void Handler2 ( Task pTask, MessageId pId, Message pMessage )
{
    switch(pId)
    {
        case BBB:
            // receive message BBB
            MessageSend((Task)& Task3, CCC ,NULL);
            // send message CCC to task 3
        break;
    }
}
void Handler3 ( Task pTask, MessageId pId, Message pMessage )
{
    switch(pId)
    {
        case CCC:
            // receive message CCC
```

3-2-2 MessageSend

ProtoType: void MessageSend(Task task, MessageId id, void* message);

It used to be sent a message to the corresponding task immediately. The message will be passed to free after delivery.

Case 1: Without Message Data – input NULL pointer

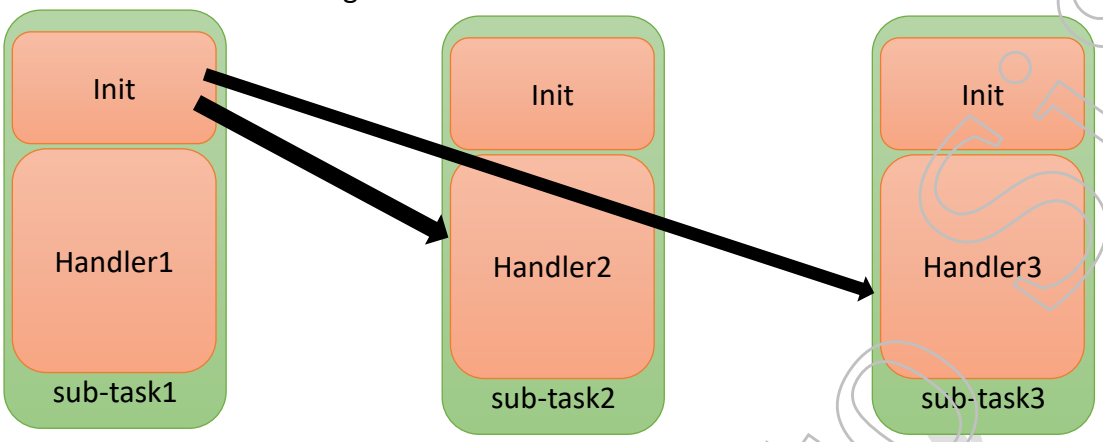
```
MessageSend((Task)&gTestTask, AAA ,NULL);
```

Case 2: With Message Data – input a data pointer

Ex. Allocate 6 Bytes (sizeof(BDADDR_S)), get BD_ADDR and the data pointer as the 3rd parameter of MessageSend API

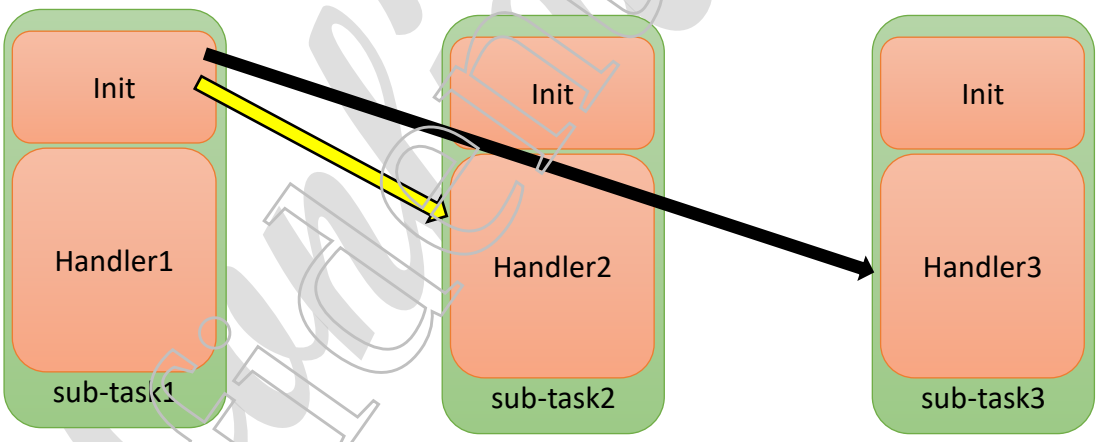
```
Header File:
    typedef struct
    {
        BDADDR_S bdaddr;
    }PACKED DATA_S;
=====================================================
void Handler ( Task pTask, MessageId pId, Message pMessage )
{
    TaskMsg_t taskMsg = osTaskMsg_Alloc( 2, sizeof(BDADDR_S) );
    DATA_S *pMsg = osTaskMsg_GetDataFieldOrigin(taskMsg);

    memcpy(&pMsg->bdaddr, pBdAddr, sizeof(BDADDR_S));
    MessageSend(&theSink.task, AAA, taskMsg);
}
```

Ex. Send same message to other tasks from Initial function



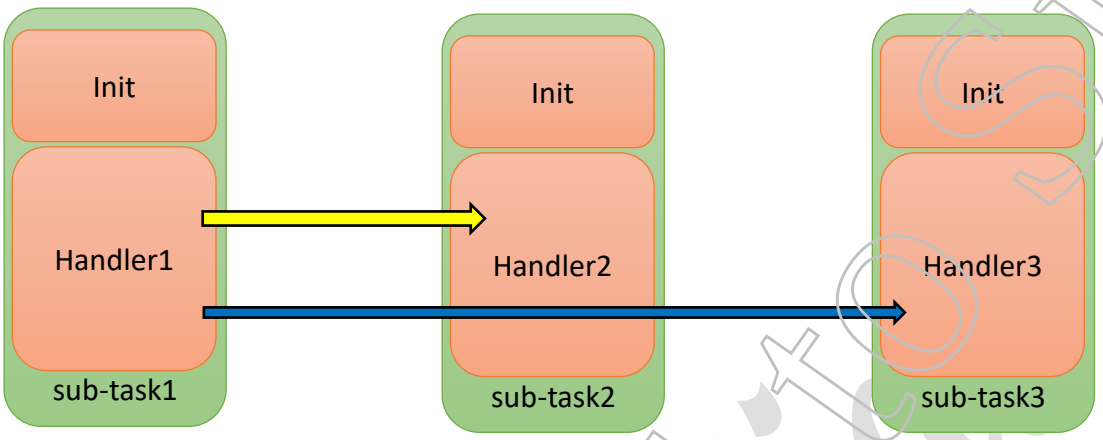

```
static void handler1 ( Task pTask, MessageId pId, Message pMessage )
{
    MessageSend((Task)&gTask2, AAA ,NULL);
    MessageSend((Task)&gTask3, AAA ,NULL);
    …
}
```

Ex. Send different message to other tasks from Initial function




```
static void handler1 ( Task pTask, MessageId pId, Message pMessage )
{
    …
    MessageSend((Task)&Task2, AAA ,NULL);
    MessageSend((Task)&Task3, BBB ,NULL);
    …
}
```

Ex. Handler1 sends messages to handler2 and handler3



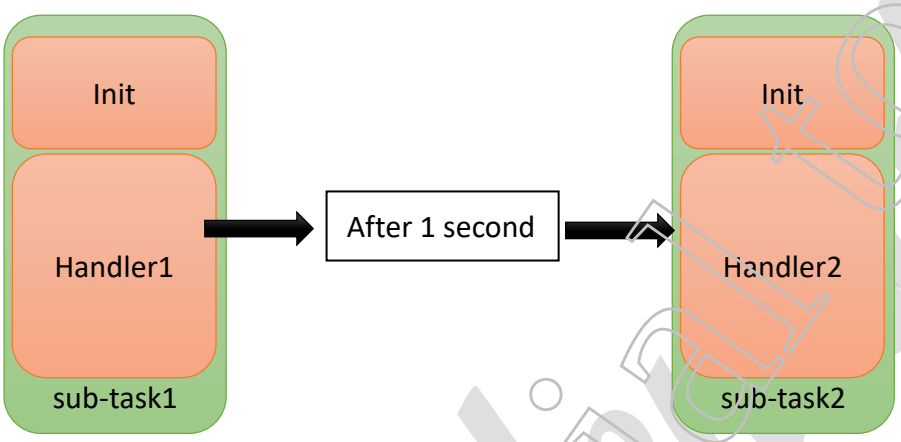

```
static void handler1 ( Task pTask, MessageId pId, Message pMessage )
{
        …
        MessageSend((Task)&gTask2, AAA ,NULL);
        MessageSend((Task)&gTask3, BBB ,NULL);
        …
}
```

3-2-3 MessageSendLater

Prototype: void MessageSendLater(Task task, MessageId id, void *message, uint32_t delay);

This API is similar to MessageSend and add a timer condition. The delay unit is 1.25ms.

Ex. Handler 1 send a message to handler 2, and handler 2 receive this message after 1 second




```
static void handler1 ( Task pTask, MessageId pId, Message pMessage )
{
    …
    MessageSendLater(Task2, AAA ,NULL, 800);
}

static void handler2 ( Task pTask, MessageId pId, Message pMessage )
{
    switch(pId)
    {
        case AAA:
            // Receive after 1 second
        break;
    }
```

MessageSendLater API also could achieve Timer effect. Please reference "Timer Application" section.

3-2-4 MessageCancelAll

Prototype: uint16_t MessageCancelAll(Task task, MessageId id);

This API is used to delete all particular task message in message queue if match the input parameters.

Ex. handler 1 use MessageSendLater API send message to handler 2 but handler 3 cancel it before this message time expire. If the execute order is handler 1,3 then the handler 2 will not receive this message.

```
static void handler1 ( Task pTask, MessageId pId, Message pMessage )
{
      …
      MessageSendLater((Task)& Task2, AAA ,NULL, 800);
      …
}

static void handler2 ( Task pTask, MessageId pId, Message pMessage )
{
      //Not Receive message AAA
}
static void handler3 ( Task pTask, MessageId pId, Message pMessage )
{
      …
      MessageCancelAll((Task)& Task2, AAA);
      …
}
```

Ex. Handler 1 send 3 messages AA, BB, and CC to handler 2, but handler 3 cancel these messages if the execute order is handler 1,3

```
static void handler1 ( Task pTask, MessageId pId, Message pMessage )
{
    …
    MessageSend((Task)& Task2, AA ,NULL);
    MessageSend((Task)& Task2, BB ,NULL);
    MessageSendLater((Task)& Task2, CC ,NULL, 800);
    …
}

static void handler2 ( Task pTask, MessageId pId, Message pMessage )
{
    //Not Receive message AA, BB, and CC
}
static void handler3 ( Task pTask, MessageId pId, Message pMessage )
{
    …
    MessageCancelAll((Task)& Task2, AA);
    MessageCancelAll((Task)&Task2, BB);
    MessageCancelAll((Task)&Task2, CC);
    …
}
```

3-2-5 MessageCancelAllWithBdaddr

Prototype: uint16_t MessageCancelAllWithBdaddr(Task task, MessageId id, BDADDR_S *pbdaddr);

This API is similar to the MessageCancelAll, and this API also checks bdaddr condition. And the bd_address is be included in the message data point.

Ex. Handler 1 send a message with bd address to Handler 2 after 1 second, but Handler 3 cancel it. The execute order is 1,3

```
static void handler1 ( Task pTask, MessageId pId, Message pMessage )
{
        …
        TaskMsg_t taskMsg = osTaskMsg_Alloc(sizeof(uint16_t), sizeof(BDADDR_S));

        memcpy(osTaskMsg_GetDataFieldOrigin(taskMsg),pBdAddr,sizeof(BDADDR_
        S));
        MessageSendLater(task2, AAA, taskMsg, 800);
        …
}

static void handler2 ( Task pTask, MessageId pId, Message pMessage )
{
        //Not Receive message AAA
}
static void handler3 ( Task pTask, MessageId pId, Message pMessage )
{
        …
        MessageCancelAllWithBdaddr(task2, AAA, pBdAddr);
        …
}
```

3-2-6 MessageBlock / MessageUnblock

Prototype: void MessageBlock(Task task, MessageId id);
void MessageUnblock(Task task, MessageId id);

The MessageBlock is used to block particular message. And if there is message match the condition, it will be put into block queue until it be unblocked.

The MessageUnblock is used to unblock particular message if it exists in block queue. If the message pop from block queue it will be put into message queue.

Ex. Handler 1 case DD executes block message AA and send message BB to handler 2, handler 2 receive message BB and send message AA and CC to Handler 1,

and handler 1 only receive message CC. Handler 3 unblock message AA, then handler 1 receive message AA. The execute order is handler 1,2,1,3,1

```
static void handler1 ( Task pTask, MessageId pId, Message pMessage )
{
    switch(pId)
    {
        case AA:
            // execute order 5
        Break;
        case CC:
            // execute order 3
        break;
        case DD:
            // execute order 1
            MessageBlock(task1, AA);
            MessageSend(task2, BB);
        Break;
    }
}

static void handler2 ( Task pTask, MessageId pId, Message pMessage )
{
    switch(pId)
    {
        case BB:
            // execute order 2
            MessageBlock(task1, AA);
            MessageSend(task1, CC);
        Break;
    }
}
static void handler3 ( Task pTask, MessageId pId, Message pMessage )
{
    // execute order 4
    MessageUnBlock(task1, AA);
    ...
}
```

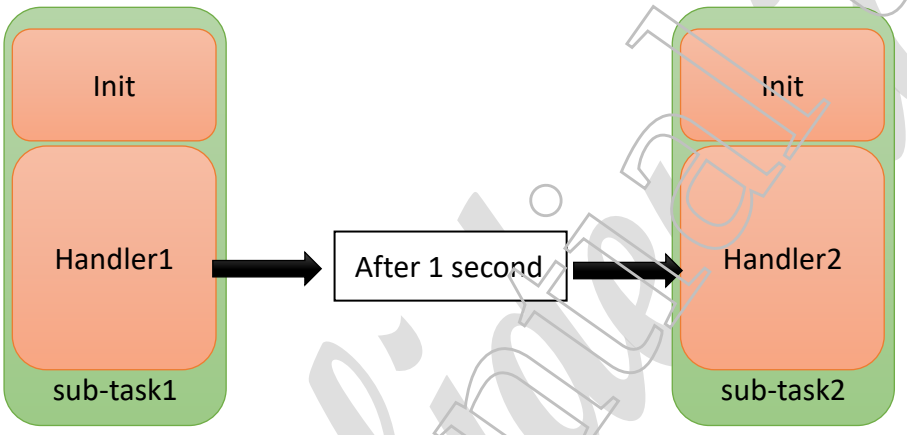
3-3 Timer Application

App Task has no Timer API, but the "MessageSendLater" API can used to be a timer. Below examples are 3 different timer usage conditions.

3-3-1 create a timer and wait time expire

i. Create a message by MessageSendLater.
ii. Wait time expire.
iii. Message loop function pops this message and call the corresponding task to execute this message.

Ex. Handler 1 send a message to handler 2, and handler 2 receive this message after 1 second



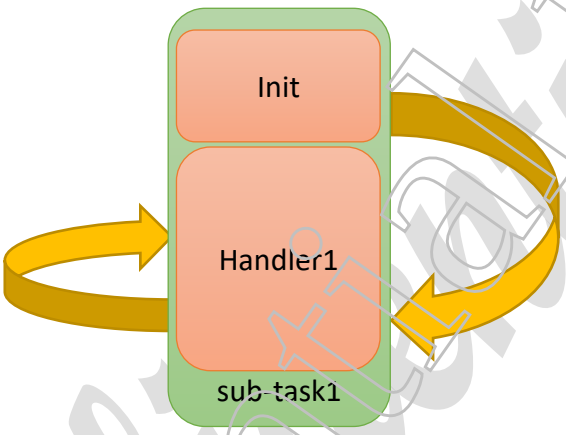

```
static void handler1 ( Task pTask, MessageId pId, Message pMessage )
{
      ...
      MessageSendLater(Task2, AAA ,NULL, 800);
}

static void handler2 ( Task pTask, MessageId pId, Message pMessage )
{
      switch(pId)
      {
          case AAA:
              // Receive after 1 second
          break;
      }
```

3-3-2 create a periodic timer

i. Create a message by MessageSendLater.
ii. Wait time expire.
iii. Message loop function pops this message and call the corresponding task to execute this message.
iv. Create a same message by MessageSendLater. And goto step ii.

If the purpose is periodic execute handler 1, handler 1 can always send the same message to itself like a system tick.

Ex. The initial function of subtask 1 send a timer 100ms message AA to handler 1, and handler 1 send this message again.




```
void subtask1Init ( void )
{
    MessageSendLater(Task1, AAA ,NULL, 80);
}
static void handler1 ( Task pTask, MessageId pId, Message pMessage )
{
    Switch(pId)
    {
        Case AAA:
            // Execute every 100 ms
            MessageSendLater(Task1, AAA ,NULL, 80);
        Break;
    }
```

3-4 Print Log

To print log, following conditions must be met.

1. USB-to-UART transfer IC

The GR demo board already mounts FTDI USB-to-UART IC, it transfers UART signal to USB signal. Develops also can directly connect UART pin to this kind transfer IC to print log.

2. FW enable debug print definition.

In config.h the definition configDEBUG_Log must set to 1.

```
#define configDEBUG_Log 1
```

3. FW call grDbgPrint API

In SDK, use grDbgPrint function to print debug message. The grDbgPrint is similar printf function, user can use %d and %x to print out variables.

```
grDbgPrint(GRDBG_APP, "Test: status = %d", status);

grDbgPrint(GRDBG_AUTOTEST, "TEST, bdaddr:%x%x", (*((uint32_t*)(&pBdAddr->Nap)) & 0xFFFF), (*((uint32_t*)(&pBdAddr->Lap))) & 0xFFFFFFFF);
```

4. Gear Radio also provides LogTool.exe to receive debug print log.

Please contact your HW or SW contact windows to get it.

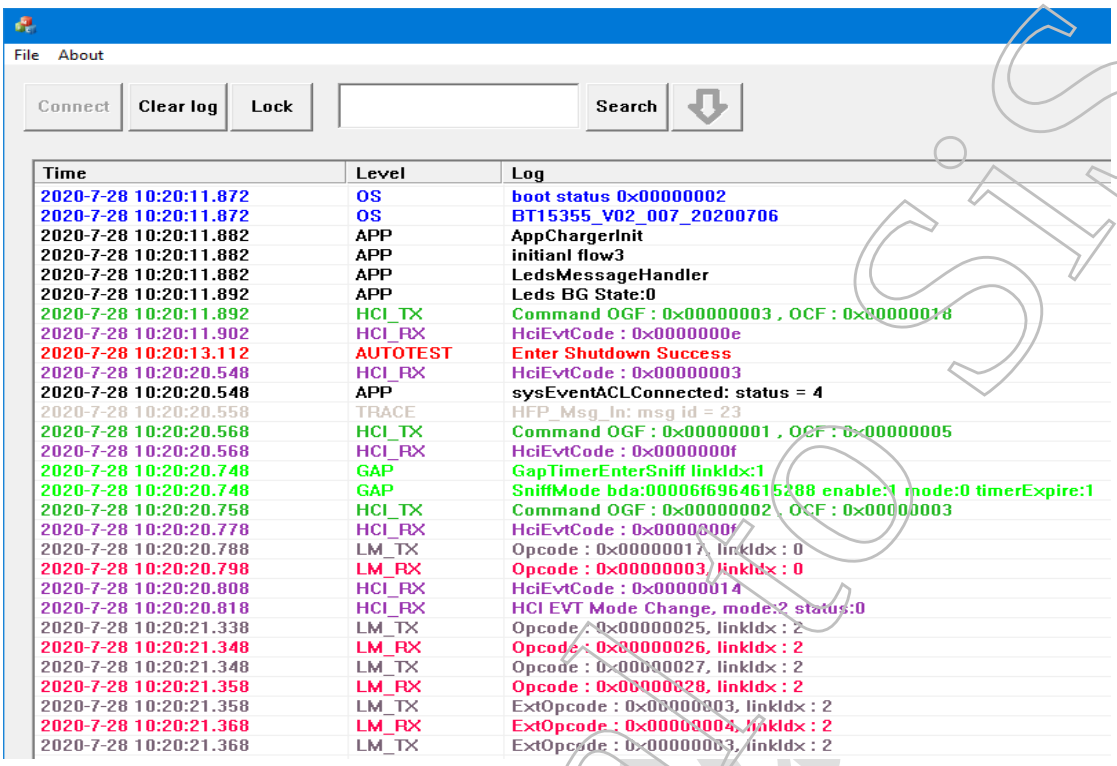


Figure 3.2 LogTool Receive debug log