Embedded Simply Blue Application Note

Texas Instruments
Application Note 1711
Sebastien Mathieu
September 2007
Revised February 2013

Embedded Simply Blue Application Note

1.0 Introduction

SB_Custom has been created to give an embedded example of a Simply Blue device (LMX9830 or LMX9838 based) communicating with a 16 bits microprocessor chip.

Note that those files are not complete and can not be compiled and tested as is. The purpose of it is to give some guidelines for an embedded world and simplify the development process.

In this document, the term “Host” refers to the 16 bits microprocessor platform which role is to initialize and manage the communication with the National Bluetooth module (LMX9820A, LMX9830 or LMX9838 based). The term “Bluetooth module” or “module” refers to the National Bluetooth module (LMX9820A, LMX9830 or LMX9838 based).

2.0 SB_Custom files overview

SB_Custom contains the following files with their respective description:

- sbappli_custom.c:

This file is the top level file of the project and contains the basic framework to create an embedded application using uCOS-II as Operating System. All high level initializations such as UART communication, host hardware initialization and module initialization have to be done in this file.

- sbopcodes_custom.h:

A listing and definition of every command opcode of the SimplyBlue command interface is done in this file. The SimplyBlue command interface allows sending and receiving commands from/to the module in order to easier the communication with the module. Please refer to the Software User’s Guide to get more information.

- sb_custom.c:

This file implements the command interface on the host. Construction of a command, sending of a command to the module, command parser and wrapper are all included in this file.

- sb_custom.h:

This header file contains the definition and important information for the sb_custom.c file.

3.0 SimplyBlue Command construction

3.1 UART PROTOCOL PRINCIPLES

The Bluetooth module can be controlled by simple commands on the UART interface. The host should send those commands using the UART interface to set up the Bluetooth module. The commands have to be sent within a special package format. The following sections describe the format of the command set packages.

3.1.1 Framing

The connection is considered “Error free”. But for packet recognition and synchronization, some framing is used.
All packets sent in both directions are constructed after the following model:

Table 1. Package Framing

Start delimiter	Packet Type identification	Op code	Data length	Check-sum	Packet Data	End delimiter
1 byte	1 byte	1 byte	2 bytes	1 byte	<Data length> bytes	1 byte
	|-------------- Checksum		---------------|			

3.1.2 Start delimiter

The start delimiter indicates the Bluetooth module the beginning of a new package. The “STX” char is used as start delimiter.

STX = 0x02

3.1.3 Packet type identification

This byte identifies the type of packet. The following types are valid:

Table 2. Packet Type Identification

Code	Packet Type	Description
0x52 'R'	Request (REQ)	A request sent to the Bluetooth module. All request are answered by exactly one confirm.
0x43 'C'	Confirm (CFM)	The Bluetooth modules confirm to a request. All request are answered by exactly one confirm.
0x69 'i'	Indication (IND)	Information sent from the Bluetooth module, that is not a direct confirm to a request.
0x72 'r'	Response (RES)	An optional response to an indication. This is used to respond to some type of indication messaged.

All other values are reserved.

3.1.4 Opcode

The opcode is a command specifier. Each command is represented by a one byte identifier. The complete list of command opcode can be found in Annex "Command Opcode".

3.1.5 Data length

Number of bytes in the "Packet data" area. The maximum size is 333 bytes.

3.1.6 Packet data

The data fields hold binary data; hence both 0x02 (=STX) and 0x03 (=ETX) are allowed as data.

3.1.7 Checksum

This is a simple Block Check Character (BCC) checksum of the bytes from "Packet type" to, and including, "data length". The BCC checksum is calculated as the low byte of the sum of all bytes.

E.g. if the sum of all bytes are 0x3724, the checksum is 0x24.

3.1.8 End delimiter

The "ETX" char is used as end delimiter.
ETX = 0x03

3.1.9 Retransmission

The connection is considered "Error free", hence no need for implementing time-outs and retransmissions.

3.1.10 Flow control

A transparent data-mode is supported for RFCOMM communication. When using this transparent mode, full hardware handshake is needed.

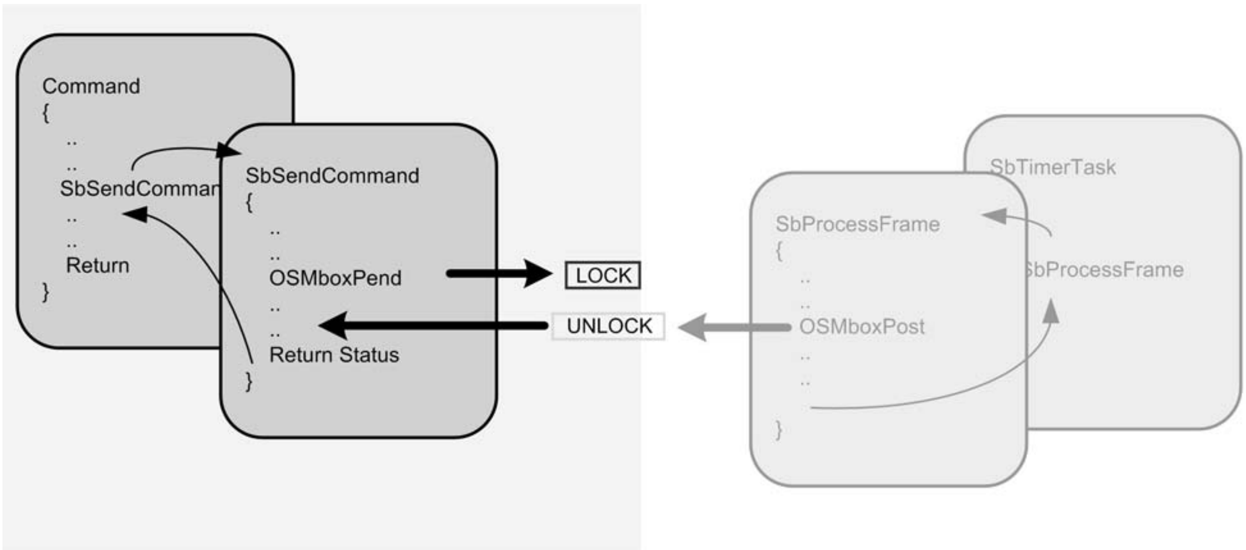
When not in transparent mode, the protocol principle of REQ-CFM, limits the need of buffer capacity. As IND's can come out of REQ-CFM sequence, and is unconfirmed, the user device has to be able to read these data fast enough / have enough buffer capacity.

3.1.11 Byte Order

The byte order of the protocol is Little Endian, if nothing else is specified.

4.0 Command Wrapper

The Command Wrapper mechanism is "packing" the command bytes together and sending the command through UART to the module. One part of the Wrapping is done in the function specific to the command and the other part is done in the function SbSendCommand. Once the command is ready to be sent, the function will send it over UART interface to the module and wait for a status OK. This is done through the message box OSMboxPend(SbDevInfo.SbCmdMbox, BTCORE_CALLBACK_TIMEOUT, &err). This message box will block the function SbSendCommand until the status is received from the receiving function.



The Command Wrapper function SbSendCommand is taking the command pointer and the command size as input parameters. This function fills out:

- Start delimiter
- Packet type identification
- Checksum
- End delimiter

The rest of the command (Opcode, Data Length and Packet Data) should be filled previously in the specific command function calling this wrapper.

For example, the user wants to send the reset command to the module. The function SbResetDevice will first create and allocate the command buffer in the memory, and fill:

- Opcode
- Data Length
- Packet Data

For reference, see the following example code.

```
SBStatus_T SbResetDevice(void)
{
   uint16 payloadlen;
   uint8  SbCommand[7];

   payloadlen = 0x0000;
   SbCommand[2] = RESET;
   SbCommand[3] = (payloadlen & 0x00FF); // payload size is stored
   SbCommand[4] = (payloadlen >> 8);     // in little endian fashion

   return SbSendCommand(SbCommand, 7 + payloadlen);
}
SBStatus_T SbSendCommand(uint8* SbCommand, uint16 Size)
{
   uint8  err;
   void   *msg;
   uint16 checksum;

   SbCommand[0]    = STX;
   SbCommand[1]    = REQ;
   checksum        = SbCommand[1] + SbCommand[2] + SbCommand[3] + SbCommand[4];
   SbCommand[5]    = checksum % 256;
   SbCommand[Size - 1] = ETX;

   if (usart_tx(SB_UART_PORT, SbCommand, Size) == Size) {
        msg = OSMboxPend(SbDevInfo.SbCmdMbox, BTCORE_CALLBACK_TIMEOUT, &err);
      if (err == OS_NO_ERR) {
         if ((uint32)msg == SBSTATUS_OK) {
            return SBSTATUS_OK;
         }
         else {
            return SBSTATUS_ERROR;
         }
      }
      else {
         return SBSTATUS_TIMEOUT;
      }
   }
   else {
      return SBSTATUS_UART_INCOMPLETE_TRANSFER;
   }
}
```

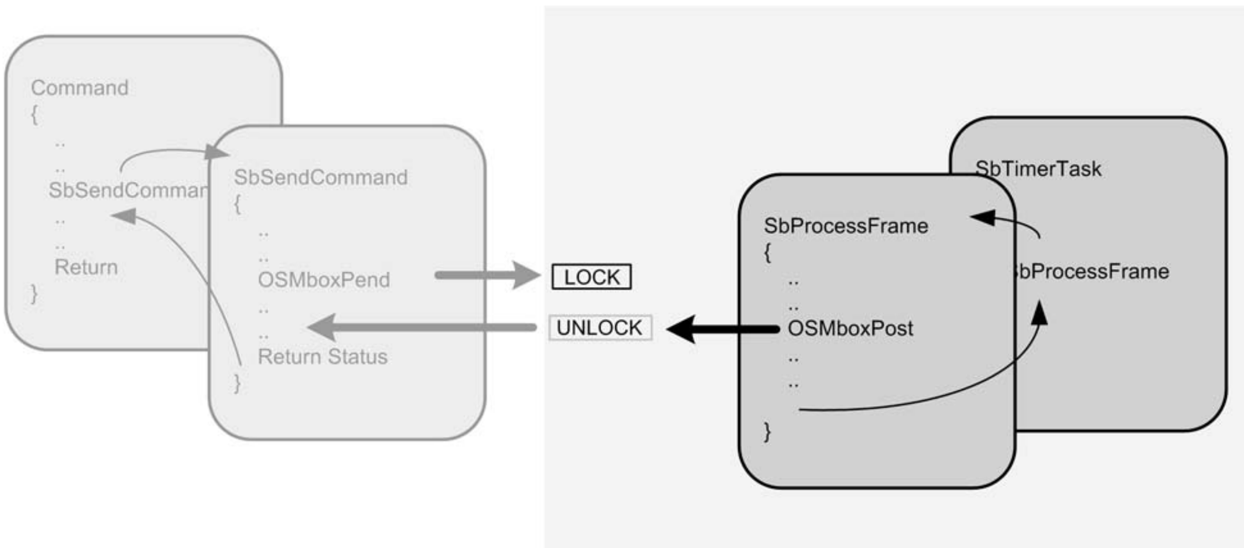
5.0 Command Parser

The Command Parser is in charge of analyzing an incoming frame to determine which command and information have been received from the module, and what is the action to take.

This Embedded Simplyblue application is based on multi-tasking programming, using uCOS RTOS core. In this application is running two tasks in parallel. The first one is dealing with the actions to take, and mainly sending commands to the modules and the second one is receiving and analyzing the incoming frames.

In the code the function receiving the frames to be analyzed is called SbTimerTask. As soon as the host receives a Byte on the UART port, this function will check if the command received is valid, and this byte per byte. If one byte is not expected, it will return an invalid result and proceed another frame.

Once the command has been completely received, this function will call the SbProcessFrame function which is the real Command Parser. If the message coming from the module is a confirmation to a command previously sent, the status will be given back through the message box OSMboxPost(SbDevInfo.SbCmdMbox, (void*)status);



See the following source code:

```
void SbProcessFrame(void)
{
    int    i;
    uint16 ServiceResponseStart=0;
    uint32 status;

    switch (SbEvent.bOpcode)
    {
        case GAP_DEVICE_FOUND:                      /* to be implemented */ return or break;
        case SPP_LINK_ESTABLISHED:                  /* to be implemented */ return or break;
        case SPP_INCOMING_LINK_ESTABLISHED:  /* to be implemented */ return or break;
        case SPP_LINK_RELEASED:                     /* to be implemented */ return or break;
        case GAP_ESTABLISH_SCO_LINK:                /* to be implemented */ return or break;
        case GAP_RELEASE_SCO_LINK:                  /* to be implemented */ return or break;
        case SPP_SEND_DATA:                         /* to be implemented */ return or break;
        case SPP_INCOMING_DATA:                     /* to be implemented */ return or break;
        case SDAP_SERVICE_BROWSE:                   /* to be implemented */ return or break;
        case SDAP_SERVICE_REQUEST:                  /* to be implemented */ return or break;
        case MODULE_READY:
            if ( COMMON_MboxWaitingTasks( SbDevInfo.SbCmdMbox ) != 0 ) {
                OSMboxPost(SbDevInfo.SbCmdMbox, (void*)1);
            }
            /* At this point, the Simply Blue device is reset and ready to work */
            return;
        default:
            break;
    }

    if (SbEvent.bType == CFM) {
        //signal that sb command is completed
        status = ((!SbEvent.pPayload[0]) ? SBSTATUS_OK : SbEvent.pPayload[0]);
        OSMboxPost(SbDevInfo.SbCmdMbox, (void*)status);
    }
}
```

6.0 Transparent mode

Simply Blue devices have this in-built specificity called the command interpreter which allows the user to send predefined commands to the module. In case the user wants to use the device as a true cable replacement, the module will have to be switched to the so called transparent mode. Once in transparent mode, the module will not interpret the commands anymore and will just forward any bytes received as is, as a pure cable.

To get more information about the transparent mode, please refer to “Texas Instruments: “LMX9830” or “LMX9838 Software User’s Guide””.

An example of a function setting the module into transparent mode is detailed below.

The function gets the local port number corresponding to the link to be switched to transparent mode. If the command has been successful, the transparent flag will be set to give the information that the transparent mode is now active, and the function will return the status OK If the command was not successful the function will return the status ERROR.

```
SBStatus_T SbEnterTransparentMode(uint8 LocalPortNo)
{

   int16 payloadlen;
   uint8 SbCommand[8];
   uint8 err;
   void  *msg;

   payloadlen = 0x0001;
   SbCommand[2] = SPP_TRANSPARENT_MODE;
   SbCommand[3] = (payloadlen & 0x00FF); // payload size is stored
   SbCommand[4] = (payloadlen >> 8);        // in little endian fashion
   SbCommand[6] = LocalPortNo;

   if (SbSendCommand(SbCommand, 7 + payloadlen) == SBSTATUS_OK) {
      Transparent_flag = TRUE;                  // transparent mode is now active
      return SBSTATUS_OK;
   }

   return SBSTATUS_ERROR;
}
```

7.0 UART Break

Once the Bluetooth module is in transparent mode, the only way to go back to command mode and get control access over the module, is to send a UART Break. As defined in the UART specification, a UART Break is a contiguous transmission of “0” (space) for a certain length of time. The CCITT “blue book” specification states that the time duration for this is larger than 2M+3 bit time (where M is the character length). After the break sequence, another 2M+3 bit time consisting of the contiguous transmission of “1” (mark) is required to start the next character.

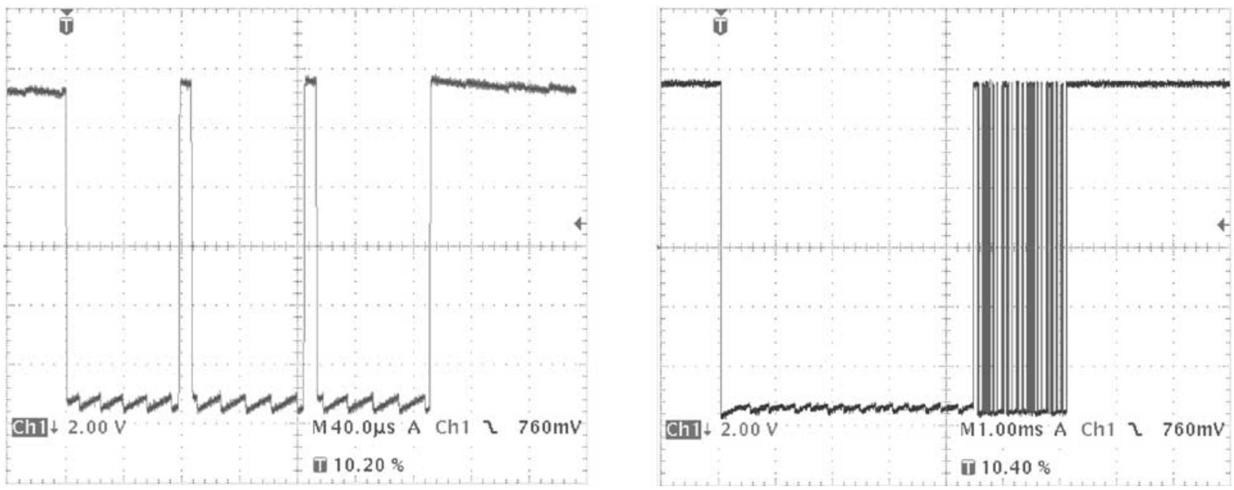


Figure 1. Difference between a Standard 0 transmission and BREAK signal

Figure 1 shows the difference between the signal of a normal 0 and the BREAK signal. The left picture shows the signalling of 3 Zeros at 115.2kbit/s. Each character is started and ended with a start bit and a Stop bit. The normal length of 1 byte is therefore about 86.8µs (1startbit + 8bit data + 1stopbit).

The picture on the right shows a BREAK signalled by the Bluetooth module after a released link. The signal is held low for over 4 ms. Theoretical minimum value for a BREAK at this speed would be about 165µS.

An example of how could be implemented this UART Break functionality in embedded environement, is detailed below.

The function protects the execution core from an eventual OS interrupt. The transmit line TX of the UART is pulled down for about 10 ms to cover the worst case. Of course this time could be computed depending on the UART baudrate to get a more accurate time. Then the transmit line TX of the UART is raised again to finish the UART Break.

```
void SbSendUartBreak(void)
{
    uint32 ticks,new_tick=0;
    usart_t* usart = usart_tab[SB_UART_PORT];

    OS_ENTER_CRITICAL();
    UMDSL1 |= UBRK;                    // asserts TX line to 0
    ticks = OSTimeGet();
    while(OSTimeGet() < (ticks+1));    // wait for 10 ms (1 tick)
    UMDSL1 &= (~UBRK);                 // asserts TX line to 1
    Transparent_flag = FALSE;          // command mode active
    OS_EXIT_CRITICAL();
}
```

8.0 Annex

8.1 COMMAND OPCODE

Table 3. Opcode Values

Opcode	Value
GAP_INQUIRY	0x00
GAP_DEVICE_FOUND	0x01
GAP_REMOTE_DEVICE_NAME	0x02
GAP_READ_LOCAL_NAME	0x03
GAP_WRITE_LOCAL_NAME	0x04
GAP_READ_LOCAL_BDA	0x05
GAP_SET_SCANMODE	0x06
GAP_GET_FIXED_PIN	0x16
GAP_SET_FIXED_PIN	0x17
GAP_GET_PIN	0x75
GAP_GET_SECURITY_MODE	0x18
GAP_SET_SECURITY_MODE	0x19
GAP_REMOVE_PAIRING	0x1B
GAP_LIST_PAIRED_DEVICES	0x1C
GAP_ENTER_SNIFF_MODE	0x21
GAP_EXIT_SNIFF_MODE	0x37
GAP_ENTER_PARK_MODE	0x38
GAP_EXIT_PARK_MODE	0x39
GAP_ENTER_HOLD_MODE	0x3A
GAP_SET_LINK_POLICY	0x3B
GAP_GET_LINK_POLICY	0x3C
GAP_POWER_SAVE_MODE_CHANGED	0x3D
GAP_ACL_ESTABLISHED	0x50
GAP_ACL_TERMINATED	0x51
GAP_SET_AUDIO_CONFIG	0x59
GAP_GET_AUDIO_CONFIG	0x5A
GAP_ESTABLISH_SCO_LINK	0x5D
GAP_RELEASE_SCO_LINK	0x5E
GAP_MUTE_MIC	0x5F
GAP_SET_VOLUME	0x60
GAP_GET_VOLUME	0x61
GAP_CHANGE_SCO_PACKET_TYPE	0x62
SPP_SET_PORT_CONFIG	0x07
SPP_GET_PORT_CONFIG	0x08
SPP_PORT_CONFIG_CHANGED	0x09

Table 3. Opcode Values

Opcode	Value
SPP_ESTABLISH_LINK	0x0A
SPP_LINK_ESTABLISHED	0x0B
SPP_INCOMMING_LINK_ESTABLISHED	0x0C
SPP_RELEASE_LINK	0x0D
SPP_LINK_RELEASED	0x0E
SPP_SEND_DATA	0x0F
SPP_INCOMING_DATA	0x10
SPP_TRANSPARENT_MODE	0x11
SPP_CONNECT_DEFAULT_CON	0x12
SPP_STORE_DEFAULT_CON	0x13
SPP_GET_LIST_DEFAULT_CON	0x14
SPP_DELETE_DEFAULT_CON	0x15
SPP_SET_LINK_TIMEOUT	0x57
SPP_GET_LINK_TIMEOUT	0x58
SPP_PORT_STATUS_CHANGED	0x3E
SPP_GET_PORT_STATUS	0x40
SPP_PORT_SET_DTR	0x41
SPP_PORT_SET_RTS	0x42
SPP_PORT_BREAK	0x43
SPP_PORT_OVERRUN_ERROR	0x44
SPP_PORT_PARITY_ERROR	0x45
SPP_PORT_FRAMING_ERROR	0x46
SDAP_CONNECT	0x32
SDAP_DISCONNECT	0x33
SDAP_CONNECTION_LOST	0x34
SDAP_SERVICE_BROWSE	0x35
SDAP_SERVICE_SEARCH	0x36
SDAP_SERVICE_REQUEST	0x1E
SDAP_ATTRIBUTE_REQUEST	0x3F
CHANGE_LOCAL_BDADDRESS	0x27
CHANGE_NVS_UART_SPEED	0x23
CHANGE_UART_SETTINGS	0x48
SET_PORTS_TO_OPEN	0x22
GET_PORTS_TO_OPEN	0x1F
RESTORE_FACTORY_SETTINGS	0x1A
STORE_CLASS_OF_DEVICE	0x28
FORCE_MASTER_ROLE	0x1D

Table 3. Opcode Values

Opcode	Value
READ_OPERATION_MODE	0x49
WRITE_OPERATION_MODE	0x4A
SET_DEFAULT_LINK_POLICY	0x4C
GET_DEFAULT_LINK_POLICY	0x4D
SET_EVENT_FILTER	0x4E
GET_EVENT_FILTER	0x4F
SET_DEFAULT_LINK_TIMEOUT	0x55
GET_DEFAULT_LINK_TIMEOUT	0x56
SET_DEFAULT_AUDIO_CONFIG	0x5B
GET_DEFAULT_AUDIO_CONFIG	0x5C
SET_DEFAULT_LINK_LATENCY	0x63
GET_DEFAULT_LINK_LATENCY	0x64
SET_CLOCK_FREQUENCY	0x67
GET_CLOCK_FREQUENCY	0x68
SET_PCM_SLAVE_CONFIG	0x74
ENABLE_SDP_RECORD	0x29
DELETE_SDP_RECORDS	0x2A
STORE_SDP_RECORD	0x31
RESET	0x26
Bluetooth module_READY	0x25
TEST_MODE	0x24
WRITE_ROM_PATCH	0x47
READ_RSSI	0x20
RF_TEST_MODE	0x4B
DISABLE_TL	0x52
TL_ENABLED	0x53
HCI_COMMAND	0x65
AWAIT_INITIALIZATION_EVENT	0x66
ENTER_BLUETOOTH_MODE	0x66
SET_CLOCK_AND_BAUDRATE	0x69
SET_GPIO_WPU	0x6B
GET_GPIO_STATE	0x6C
SET_GPIO_DIRECTION	0x6D
SET_GPIO_OUTPUT_HIGH	0x6E
SET_GPIO_OUTPUT_LOW	0x6F
READ_NVS	0x72
WRITE_NVS	0x73

9.0 Bibliography

[1] Texas Instruments: “LMX9830” or “LMX9838 Software User’s Guide”