UART configuration for Bluetooth and connect with android app.

by Manav Meghpara, Parth Paghdal, Danieva Paraiso
Last update: Apr 12, 2024

This document guides the user through

1. Configure UART on BBG to read Bluetooth data.
2. Reading Bluetooth data via a C program
3. Implement android app to connect with BLE module.

Table of Contents

1. Bluetooth module (HM10) setup on BBG 2
   1.1 Wiring module to UART port 2
   1.2 Updating Device Tree for UART 2
2. Reading Bluetooth data via a C program 4
   2.1 Configure serial port using termios.h 4
3. Implement android app to connect with BLE module 5
   3.1 App permissions in and checking 5
   3.2 Scan for Bluetooth devices 5
   3.3 Connect and send data using GATT server 5

Formatting

1. Commands for the host Linux's console are show as:
   ```
   (host)$ echo "Hello PC world!"
   ```
2. Commands for the target (BeagleBone) Linux's console are shown as:
   ```
   (bbg)$ echo "Hello embedded world!"
   ```
3. Almost all commands are case sensitive.

1. Bluetooth module (HM10) setup on BBG

For Bluetooth module, we need to connect to any one **UART** port on BeagleBone. For our purpose, we can connect to UART 4. Each UART will consist of two channels, one is Receive (**Rx**) and other is Transmit (**Tx**). The pin layout for UART 4 Rx is **P9.11** and for Tx is **P9.13**.

1.1 Wiring module to UART port

To connect the BLE module to a UART port we need to understand the pin layout on Bluetooth module. This picture clearly shows all the pins for a BLE module.

Pins	**Description**
VCC	Connect this to a 1.8 V or 3.3 V power source on BeagleBone
GND	Connect this to ground pin
TXD	Connect this to Rx pin of UART 4 (P9.11)
RXD	Connect this to Tx pin of UART 4 (P9.13)

1.2 Updating Device Tree for UART

For our use, we must configure UART to detect the port as a serial port connection.
Linux must be told what hardware is connected to the CPU. It learns this at boot up using a Device Tree (file is a .DTB for the device tree binary). The boot loader (UBoot) detects what capes are installed (or configured) and sets up the device tree for the kernel to use.

Do the following just once (per board).

1. Backup `uEnv.txt`

   The `uEnv.txt` file is critical to controlling how UBoot starts the system. We will change it to load the UART config; however, we must first take a backup copy to recover from some errors.

   ```
   (bbg)$ cd /boot
   (bbg)$ sudo cp uEnv.txt uEnv-BeforeUART.txt
   ```

   - **WARNING:** Corrupting the uEnv.txt file may cause the BeagleBone to be unbootable, and hence must be reflashed. Be careful editing this file!

2. Edit `uEnv.txt` to load UART configuration.
    - Edit `uEnv.txt`:
      `(bbg)$` **`sudo nano /boot/uEnv.txt`**
    - Find the section titled:
      `###Additional custom capes`
    - Change it to be:
      ```
      ###Additional custom capes
      uboot_overlay_addr4=/lib/firmware/BB-UART4-00A0.dtbo
      #uboot_overlay_addr5=<file5>.dtbo
      #uboot_overlay_addr6=<file6>.dtbo
      #uboot_overlay_addr7=<file7>.dtbo
      ```
3. Reboot the target

If you corrupted uEnv.txt file please follow the professor's guide on recovery.

To verify and check if Bluetooth is working properly, install **'LightBlue'** app and look for the Bluetooth device name. Connect to it and the flashing Red Led on module should be stable. After that follow these steps.

1. Read the UART driver file for data. In our case it's UART 4 so use ttyS4 file.
    - `(bbg)$` **`cat /dev/ttyS4`**
2. Now send a message from the phone to the Bluetooth module using the app.
3. The File will read the values and print on the screen.

Troubleshooting -

1. If you can't read data, try to write data using 'echo' and see if the problem persists.
2. If nothing works, check the wiring for the Receive and Transmit channel.
3. Finally, check again if the pins are configured to UART from the uEnv.txt file.

2. Reading Bluetooth data via a C program

To access Bluetooth data seamlessly, we engage with the UART driver file which requires us to configure the serial port within C. To accomplish this, we use the termios.h library which provides a set of functions and data structures for controlling terminal I/O interfaces.

2.1 Configure serial port using termios.h

In the following function we open the UART driver file and use the struct termios struct tty to manipulate terminal attributes like baud rate, parity, and control flow which allows us to facilitate communication between peripheral devices.

```
int uart_init(const char* UART_DEVICE){
    int uartFileDescriptor = open(UART_DEVICE, O_RDWR);
    if (uartFileDescriptor < 0) {
        perror("Unable to open UART device");
        return -1;
    }
    struct termios tty;
    // Set serial port parameters
    memset(&tty, 0, sizeof(tty));
    if (tcgetattr(uartFileDescriptor, &tty) != 0) {
        perror("Error getting serial port attributes");
        close(uartFileDescriptor);
        return -1;
    }
    cfsetospeed(&tty, BAUD_RATE);
    cfsetispeed(&tty, BAUD_RATE);
    tty.c_cflag |= (CLOCAL | CREAD); // Enable receiver and set local mode
    tty.c_cflag &= ~CSIZE;
    tty.c_cflag |= CS8; // 8 data bits
    tty.c_cflag &= ~PARENB; // No parity
    tty.c_cflag &= ~CSTOPB; // 1 stop bit
    tty.c_cflag &= ~CRTSCTS; // Disable hardware flow control
    tty.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG); // Raw input
    tty.c_iflag &= ~(IXON | IXOFF | IXANY); // Disable software flow control
    tty.c_oflag &= ~OPOST; // Raw output
    tty.c_cc[VMIN] = 1; // Read at least 1 character
    tty.c_cc[VTIME] = 5; // Timeout in 0.5 seconds
    if (tcsetattr(uartFileDescriptor, TCSANOW, &tty) != 0) {
        perror("Error setting serial port attributes");
        close(uartFileDescriptor);
        return -1;
    }
    return uartFileDescriptor;
}
```

3. Implement android app to connect with BLE module

To use Bluetooth Low Energy and connect with android app. We must use different functions which is different from classical Bluetooth connection.

3.1 App permissions in and checking

We need to have given permissions in our android app to connect to Bluetooth devices.
This is the list of permissions needed to connect. (Add these in AndroidManifest file)

```
<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.BLUETOOTH_CONNECT" />
<uses-permission android:name="android.permission.BLUETOOTH_SCAN" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
```

In the main activity if the permissions are granted and then scan for Bluetooth devices.

```
if(ActivityCompat.checkSelfPermission(this, android.Manifest.permission.BLUETOOTH_SCAN )!=
PackageManager.PERMISSION_GRANTED)
{
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.S) {
        requestPermissions(arrayOf( android.Manifest.permission.BLUETOOTH_SCAN), 1)
    }
}
```

3.2 Scan for Bluetooth devices

Now, to scan the Bluetooth devices, we need a ScanCallback() function that gets the result of scan.
The code below shows the steps to do this -

```
private fun scanLeDevice(){
    bluetoothAdapter = BluetoothAdapter.getDefaultAdapter()
    bluetoothLeScanner = bluetoothAdapter.bluetoothLeScanner

    if (!scanEnd){
        bluetoothLeScanner.startScan(leScanCallback)
    }
    else{
        bluetoothLeScanner.stopScan((leScanCallback))
    }
}

private val leScanCallback: ScanCallback = object : ScanCallback() {
    override fun onScanResult(callbackType: Int, result: ScanResult) {
        super.onScanResult(callbackType, result)
        if(!scanEnd){
            if(result.device.name == "Slave") {
                Log.i(TAG, "onScanResult: "+result.device.address+":"+result.device.name)
                gatt= result.device.connectGatt(this@MainActivity, false, gattCallback)
                Thread.sleep(700)
                Toast.makeText(this@MainActivity, "Connected!", Toast.LENGTH_SHORT).show()
            }

        }
    }
}
```

3.3 Connect and send data using GATT server

Now, to connect with a desired bluetooth device, we can use device.name or device.address of the device and match it.
After connecting to a Gatt server, we can use BluetoothGattCallback() for all kinds of activities such as

onServicesDiscovered() and read/write functions.
Below is a sample code to do just that -

1. First connect to a gatt server

```
gatt= result.device.connectGatt(this@MainActivity, false, gattCallback)
```

2.Get the callback to discoverServices

```
private val gattCallback: BluetoothGattCallback = object : BluetoothGattCallback() {
    @SuppressLint("MissingPermission")
    override fun onConnectionStateChange(gatt: BluetoothGatt, status: Int, newState: Int) {
        // Handle connection state changes here
        if(status == BluetoothGatt.GATT_SUCCESS) {
            if (newState == BluetoothProfile.STATE_CONNECTED) {
                gatt.discoverServices()
            }
        }
    }
```

3.Now use the onServiceDiscovered() function to set the services and characteristics.
To find the service and characteristic UUID, use the app ‘LightBlue’ and connect to Bluetooth. It will show all the details of the module.

```
override fun onServicesDiscovered(gatt: BluetoothGatt?, status: Int) {
        super.onServicesDiscovered(gatt, status)
        if (status == BluetoothGatt.GATT_SUCCESS) {
            val serviceUuid = UUID.fromString("0000FFE0-0000-1000-8000-00805F9B34FB")
            val characteristicUuid = UUID.fromString("0000FFE1-0000-1000-8000-00805F9B34FB")
            val service = gatt?.getService(serviceUuid)
            if (service != null) {
                characteristic = service.getCharacteristic(characteristicUuid)
                if (characteristic != null) {
                    // ... send data using the characteristic

                }

            }
        }
}
```

4. To send the data across Bluetooth we use this characteristic variable. The code below show how to do this -

```
val data = ("Hello\r\n").toByteArray()
characteristic.setValue(data)
gatt.writeCharacteristic(characteristic);
```