BL600 e-BoB Part 5

Bluetooth Low Energy Module
- SPI port & digital/analog converter
- Android application

By **Jennifer Aubinais** (France) elektor@aubinais.net

Following on from the BL600's I²C port we looked at last time, we're going to take a look here at its serial interface. To do this, we're going to be using a digital/analog converter, itself fitted with an SPI port. This will be an opportunity to produce your own Bluetooth application.

Out of the BL600's impressive arsenal, we're going to take a look here at its serial interface, referred to as SPI. To do this, we're going to be using a 16-bit digital/analog converter (DAC), itself fitted with an SPI port. The joint information will be an opportunity to produce your own Bluetooth application. Obviously, it's not our aim to give a course in development under Android, but the article gives all the elements you need for your phone and the BL600 to be able to get along well together. If there is enough demand, perhaps one day we'll present here an Android application devoted entirely to our SPI device.
Note: the tools and commands used in this article have already been presented in the previous articles in this series. [1].

SPI port and DAC

The LTC1655L DAC from Linear Technology [2] can be powered at 3.3 V, and if we use the internal voltage reference, it operates with no other components (**Figure 1**). In this case, its output voltage is limited to 0–2.5 V. The value we want this output voltage to take will be sent to the DAC's serial port by our BL600 e-Bob's SPI port (pin 10, MOSI = Master Output, Slave Input and pin 12, CLK) (**Figure 2**). The DAC also has a serial output that we've connected to the input of the e-BoB's SPI port (pin 11, MISO = Master Input, Slave Output), which will let us check our circuit is working. Using a voltmeter connected to the DAC output (pin 7), we'll measure its output voltage and adjust the drive so as to obtain a voltage of e.g. 1 V.

Our BL600 module and the DAC are now connected. Now let's take a look at the BL600's SmartBASIC program. Here's what happens in the *LTC1655L.sb* program (**Listing 1**), downloadable from the Elektor Magazine website [4]:

a) Open SPI

The first step is to open the SPI port and handle its return code using a simple *if* condition.

```
Rc = SpiOpen(0,1000000,0,handle)
```

- **nMode** = 0: CPOL (*Clock Polarity*) to 0 and CPHA (*Clock Phase*) to 0

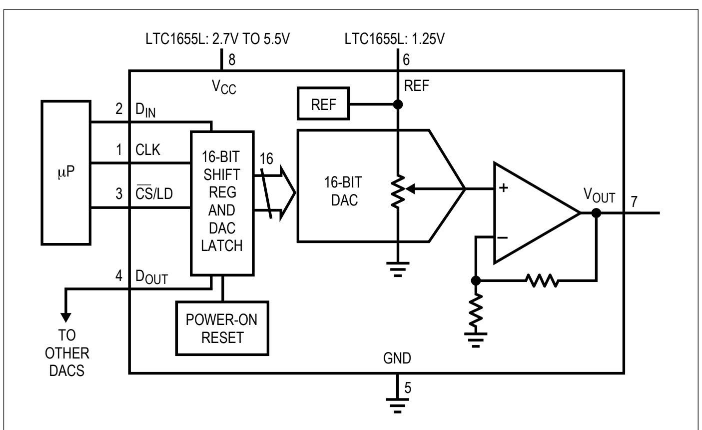


Figure 1. Internal circuit of the LTC1655L analog/digital converter (ADC) used to illustrate communication via the SPI interface.

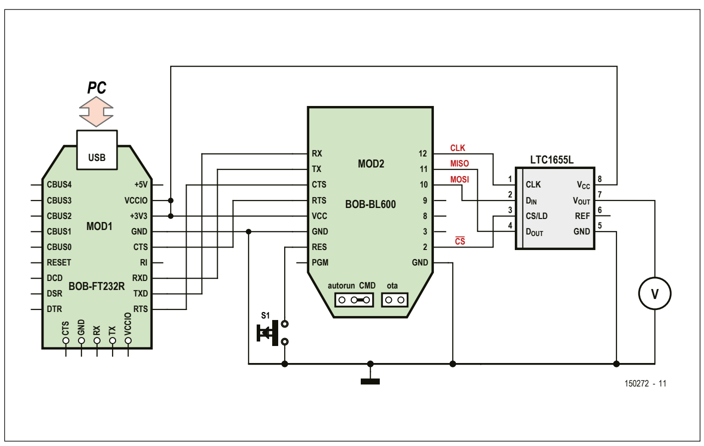


Figure 2. Circuit of the BL600 using the LTC1655L. The use of the FT232 e-BoB was described in the previous articles.

- **nClockHz** = 1000000: clock frequency
- **nCfgFlags** = 0: must be set to 0.
- **nHandle** = if the return code is 0, this variable can then be used to read, write, and close the SPI interface.

rc = the return code must be zero to be able to continue. If this is not the case, we convert its value into a character string via the *SPRINT* function using the option *INTEGER.H* and display it. Then we stop the program (*STOP*).

b) Writing and reading the SPI bytes

The aim is to obtain 1 V at the DAC output. We shall also read the data we've sent – this will be a good exercise.

- **Calculation for an output voltage of 1 V**
  The DAC's output voltage range of 0–2.5 V is covered using 16-bit words (2^{16}) from 0 to 65,536. Hence its resolution is 2.5/65,536 ≈ 38 µV. Thus the value to be sent in order to obtain 1 V will be (1/2.5) × 65,536 ≈ 26,214 (= 6666 in hexadecimal, written as 0x6666).
- **Writing the two bytes**
  As the function for sending data via the SPI port only accepts strings of ASCII characters, the digital values to be sent will need to be converted. Now the DAC expects a 16-bit word, so we shall have to convert the digital value 26,214 into two hexadecimal values that will form the character string 0x66-0x66, i.e. *ff* in ASCII. By sending this string, we obtain a voltage of 1 V at the DAC output.

```
Rc = SpiReadWrite(stWrite$,stRead$)
```

- **stWrite$** = character string of the data to be written
- **stRead$** = character string of the read data, the same length as the **stWrite$** character string.
- **rc =** The return code must be 0..

- **Reading from the SPI port**
  The SPI port is synchronous (full duplex), we can write to it and read from it at the same time. Our function to write data to the SPI port is also a read function. And since the DAC's serial output is looped to the e-BoB's MISO input and the DAC sends back the received data, we receive an echo of the data sent during the previous operation to write to our DAC.

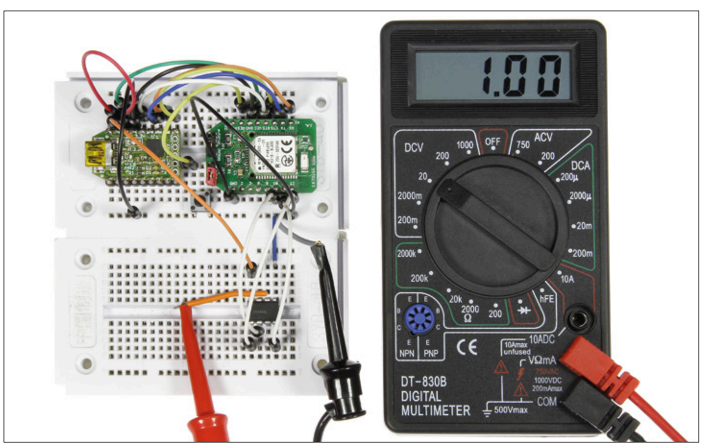


Figure 3. Photo of the experimental assembly on a prototyping board.

Component List

Semiconductors
IC1 = LTC1655L

Miscellaneous
S1 = pushbutton
MOD1 = FT232 e-BoB, ready assembled, # 110553-91 [7]
MOD2 = BL600 e-BoB, ready assembled, # 140270-91 [7]
Or printed circuit board # 140270-1 [7]

c) Close SPI

To finish, we close the SPI port

```
SpiClose(nHandle)
```

- **nHandle** = value created by SPIOpen

Android application

The moment has come to set about programming an application under Android. On the BL600 manufacturer's website [3], you will find the source code for the *Toolkit* application which includes the following services:

Listing 1

```
//--------------------------------------------------
DIM rc
DIM handle
DIM stWrite$, stRead$
//--------------------------------------------------
rc = GpioSetFunc(2,2,0)    // pin 2 at low
//--------------------------------------------------
rc=SpiOpen(0,1000000,0,handle)
if rc!= 0 then
print "\nFailed to open SPI with error code
";integer.h' rc
else
print "\nSPI open success"
endif
//--------------------------------------------------
GpioWrite(2,0) // pin 2 at low
stWrite$ = "\66\66" : stRead$=""
rc = SpiReadWrite(stWrite$, stRead$)
if rc!= 0 then
print "\nFailed to ReadWrite"
else
print "\nWrite = 0x";strhexize$(stWrite$)
endif
//--------------------------------------------------
stRead$=""
rc = SpiReadWrite(stWrite$, stRead$)
if rc!= 0 then
print "\nFailed to ReadWrite"
else
  print "\nRead = 0x";strhexize$(stRead$)
endif
//--------------------------------------------------
GpioWrite(2,1) // pin 2 at high
SpiClose(handle)  //close the port
SpiClose(handle)  //no harm done doing it again
print "\nCLOSE SPI\n"
//--------------------------------------------------
```

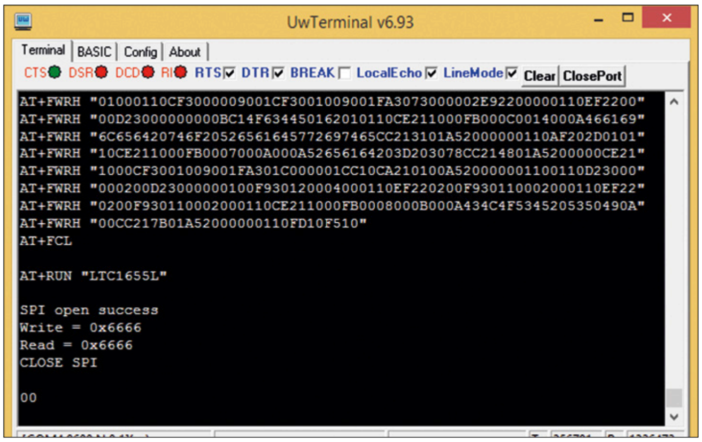
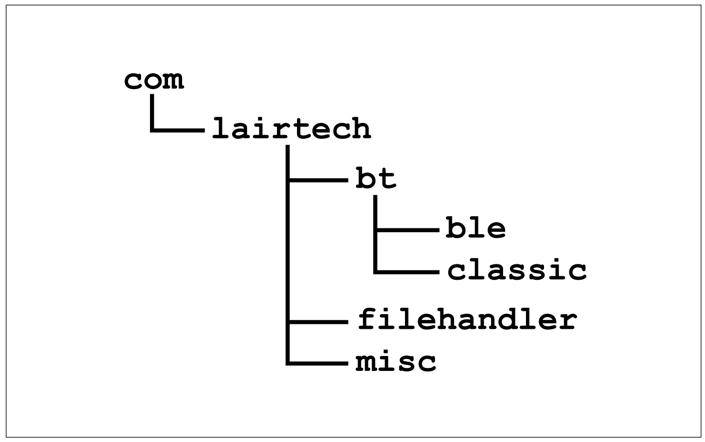


Figure 4. If the voltmeter displays a voltage of 1 V, the LTC1655L.sb program has run successfully.



The tree structure in the library.

- BPM (blood pressure)
- HRM (heart rate)
- *Proximity*
- HTM (medical thermometer) used with the temperature sensor in the previous article.
- Serial (UART)
- OTA (Over The Air)
- Batch

  We are offering you for download [4] the extremely simplified source code for an application using just the UART service. We have used *Android Studio*, available under Windows, MAC OS and Linux [5]

The manufacturer has created a library *laird_library_ver.0.18.1.1.jar* in order to speed up development of Android applications in normal Bluetooth and Bluetooth Low Energy.

Documentation is available with the library download. There are three classes:

- `BluetoothAdapterWrapper Class`
- Initialization of the Bluetooth
- Verifies if Bluetooth is present
- Detects Bluetooth peripherals
- `BluetoothAdapterWrapperCallback Interface`
- Detects, stops Bluetooth devices
- Handles Call-Backs
- `BleDeviceBase Abstract Class`
- Access to the Bluetooth methods for initializing the connection to the peripheral. For example: `connect`, `isConnected` etc.

In our program, you'll find the tree structure in the library.

Listing 2

```
<LinearLayout
     android:orientation="vertical"
     android:layout_width="match_parent"
     android:layout_height="match_parent"
     android:layout_below="@+id/textView"
     android:layout_alignParentStart="true"
     android:layout_above="@+id/btnSend">

     <ScrollView
          android:layout_width="match_parent"
          android:layout_height="match_parent"
          android:id="@+id/scrollViewVspOut"
          android:background="#ffffffff">

          <TextView
               android:layout_width="wrap_content"
               android:layout_height="wrap_content"
               android:textSize="20sp"
               android:id="@+id/valueVspOutTv" />
     </ScrollView>
 </LinearLayout>
```

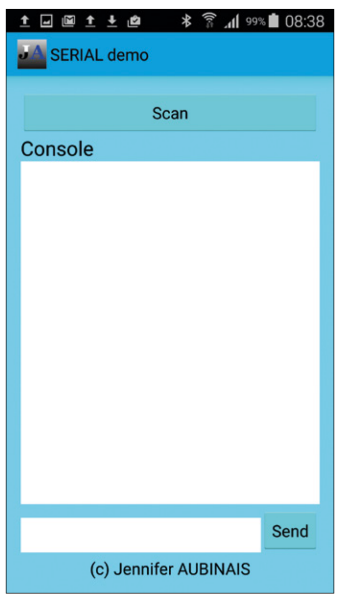


Figure 5. As simple as possible: the screen for our Serial application.

Figure 6. Display of the *mDialogFoundDevices Dialog* class with two peripherals.

To thrive, connected object networks need low power consumption wireless communication. This breakout board for the Bluetooth BLE module is the dream accessory for exploring the IoT.

The program

The complete program can be downloaded from the Elektor site [4]. The first step is to create your project. As *Package name* we've chosen: `com.ja.serial` (where ja are the author's initials)

Screen layout: To start off, let's keep it simple: two buttons *btnScan* and *btnSend*, an area for displaying the received data using scrolling *scrollViewVspOut* and *valueVspOutTv*, and the text entry box *valueVspInputEt*. And that's all (**Figure 5**). Example of *activity_main.xml* for the received data display area with scrolling (**Listing 2**).

Permissions: The *AndroidManifest.xml* file allows our program to access the Bluetooth. If you forget this, it will cause an error in the application that is difficult to identify.

```
<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
```

Declaration in the program: We're going to import different *classes* with no changes to the original sources into the following directories:

- `com.ja.bt.ble.vsp`
- `FifoAndVspManager.java`
- `FifoAndVspUiBindingCallback.java`
- `FileAndFifoAndVspManager.java`
- `VirtualSerialPortDevice.java`
- `VirtualSerialPortDeviceCallback.java`
- `com.ja.serial.bases`
- `BaseActivityUiCallback.java`
- `com.ja.serial.serialdevice`
- `SerialManager.java`
- `SerialManagerUiCallback.java`
- `com.ja.serial`
- `ListFoundDevicesHandler.java`

Importing classes from the manufacturer's library is described in **Listing 3**.

Given that the two classes *SerialManagerUiCallback* and *BluetoothAdapterWrapperCallback* are implemented, you'll find all their classes specified in our program. These can be recognized by the presence of comments at the start (**Listing 4**).

The Dialog class appears when scanning for peripherals. Then, from the list, the application connects to our BL600 e-BoB. Watch out: the application only works with peripherals that have the UART service.

The buttons: We find the *setOnClickListener* for our application's two buttons: *btnScan* and *btnSend*. Here's what happens for *btnSend*: the text to be sent is read in the text box; if it is other than null, it is copied into the receive text area then sent using the *startDataTransfer* function (**Listing 5**).

(Tip) Launching the connection automatically: To establish an automatic connection, let's remember how we do it manually: we start the scan using the *btnScan* button, then we select our peripheral in the *Dialog* box. So for our automatic connection, we leave the main program to execute the code for the *setOnClickListener* function, then, instead of displaying the list of peripherals in *Dialog*, we start the connection to the module entered in the code *name*.

(Tip) Receiving and sending data: To enter the data to be sent and display the data received, we have two text boxes: *valueVspInputEt* and *valueVspOutTv*. There's nothing to stop your program sending the data via some other action. For example, you can send 0 or 1 depending on the position of a button *Switch*. You can handle the characters received in your program as we have done in the Thermometer application described in the January/February 2015 issue [6]. There, we performed a complex calculation that couldn't be done by the BL600 before displaying the result in large characters.

The DAC in Bluetooth

You now know how to create a Bluetooth application for the BL600 e-Bob. All you have to do is take as your base the program *upass.vsp.sb* from the firmware bundle downloadable from the manufacturer's website [3], rename it to *$autorun$.LTC1655L.uart.sb* [4], then create the handler *HandlerLoop*

Listing 3

```
import com.lairdtech.bt.BluetoothAdapterWrapperCallback;
import com.lairdtech.bt.BluetoothAdapterWrapper;
import com.lairdtech.bt.ble.BleDeviceBase;
import com.lairdtech.misc.DebugWrapper;
```

Listing 4

```
/*
 * *************************************
 * SerialManagerUiCallback
 * *************************************
 */
/*
 *************************************
 * Bluetooth adapter callbacks
 * ************************************
 */
```

Listing 5

```
mBtnSend.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        switch(v.getId())
        {
            case R.id.btnSend:
            {
                String data = mValueVspInputEt.getText().toString();
                if(data != null){
                    mBtnSend.setEnabled(false);
                    mValueVspInputEt.setEnabled(false);
                    if(mValueVspOutTv.getText().length() <= 0){
                        mValueVspOutTv.append(">");
                    } else{
                        mValueVspOutTv.append("\n\n>");
                    }

                    mSerialManager.startDataTransfer(data + "\r");

                    InputMethodManager inputManager = (InputMethodManager)
                            getSystemService(Context.INPUT_METHOD_SERVICE);

                    inputManager.hideSoftInputFromWindow(getCurrentFocus().getWindowToken(),
                            InputMethodManager.HIDE_NOT_ALWAYS);

                    if(isPrefClearTextAfterSending == true){
                        mValueVspInputEt.setText("");
                    } else{
                        // do not clear the text from the editText
                    }
                }
                break;
            }
        }
    }
});
```

(renamed to *MyHandlerLoop*) in order to recover the data received in Bluetooth (**Listing 6**).
Here is the list of the various actions to be carried out by the *handler* in order to process the character string received via Bluetooth:

- receive via Bluetooth the text corresponding to the desired voltage;
- convert the received text into a value to be sent to the DAC;
- send the value to the SPI port;
- receive a value via the SPI port;
- convert the value received into a character string;
- send the string via Bluetooth.

In red: you've already seen this code. We store the characters received via Bluetooth in the variable *text$*. If the character string contains the return code 0x0D (end of string), we enter the *IF* condition.

In green: First of all, we recover the value of the voltage wanted at the DAC output. To do this, we convert the character string into a numeric variable using the *StrValDec* function, then we calculate the value for the DAC. Watch out: if the value received yields a result higher than 65,535 ($2^{16}-1$), we force the value to 65,535, a 16-bit value. In order to adhere to the format of the *SpiReadWrite* function, the value is converted into two characters using the *StrSetChr* function, thereby creating a character string *stWrite$*.

Listing 6

```
OnEvent  EVVSPRX           call MyHandlerLoop //EVVSPRX is thrown when VSP is open and data has arrived
OnEvent  EVUARTRX          call MyHandlerLoop //EVUARTRX  = data has arrived at the UART interface
OnEvent  EVVSPTXEMPTY     call MyHandlerLoop
OnEvent  EVUARTTXEMPTY    call MyHandlerLoop
```

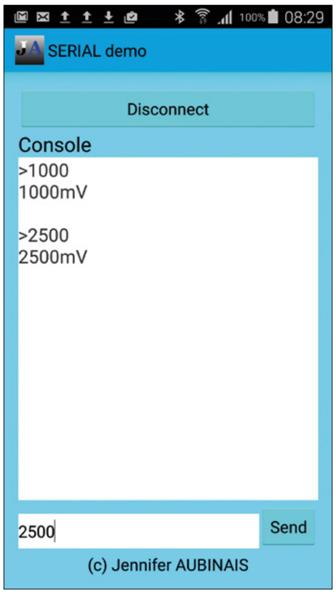


Figure 7. Using our Android application to control the output voltage of the LTC1655L DAC.

In blue: We shan't be coming back to this part of using the SPI port. The function *SpiReadWrite* is executed twice in order to recover the DAC value written in the first *SpiReadWrite*. The value written cannot be read from the DAC output port, so it will be necessary to write again so as to read the previously-loaded value. This is an oddity of the SPI protocol rather than of the device itself.

In orange: Now the value read from the DAC is a character string. We need to extract the two numerical values (high value and low value) via the *StrGetChr* function. Then we calculate the voltage value in millivolts using an inverse calculation. The conversion of the numerical value into a character string is achieved using the SPRINT function. To end, we transmit the voltage value to the Bluetooth via the *BleVspWrite* function (**Figure 7**).

(150272)

Web Links

[1] Previously in the series of articles published on the BL600:

1. BL600-eBoB (1): Bluetooth Low Energy communication module: www.elektormagazine.com/140270
2. BL600-eBoB (2): Editing, compiling, and transferring a program using the BLE module www.elektormagazine.com/150014
3. BL600-eBoB (3): smartBASIC programming for the Bluetooth Low Energy module www.elektormagazine.com/150129
4. BL600-eBoB (4): The I²C port and the temperature sensor: www.elektormagazine.com/150130

[2] www.linear.com/product/LTC1655

[3] https://laird-ews-support.desk.com/?b_id=1945

[4] Elektor July & August 2015, www.elektormagazine.com/150272

[5] https://developer.android.com/sdk/index.html

[6] Wireless thermometer: Elektor January & February 2015, www.elektormagazine.com/140190

[7] e-BoB BL600 www.elektor.de/bl600-e-bob-140270-91
e-BoB FT232 www.elektor.de/ft232r-usb-serial-bridge-bob-110553-91

Listing 7

```
function MyHandlerLoop()
  DIM n, rc, tempo$, tx$, text$
  DIM value, valtemp, pos, return$
  DIM handle
  DIM stWrite$, stRead$
  // Wait return from received data
  tx$ = "0D"
  return$ = StrDehexize$(tx$)
  tempo$ = ""
  n = BleVSpRead(tempo$,20)
  text$ = text$ + tempo$
  pos = STRPOS(text$,return$,0)
  IF ( pos >= 0 ) THEN
    //------------------
    // convert Voltage value for LTC1655
    //------------------
    value = StrValDec(text$)
    value = value * 65535
    value = value / 2500
    IF value > 65535 THEN : value = 65535 : ENDIF
    // Init string for SPI write
    stWrite$ = "00"
    valtemp = value / 256
    value = value - (valtemp * 256)
    // write chr value
    rc = StrSetChr(stWrite$,valtemp,0)
    rc = StrSetChr(stWrite$,value,1)
    //------------------
    // SPI
    //------------------
    rc=SpiOpen(0,1000000,0,handle)
    // CS (pin select) at low
    GpioWrite(2,0)
    stRead$=""
    rc = SpiReadWrite(stWrite$, stRead$)
    stRead$=""
    rc = SpiReadWrite(stWrite$, stRead$)
    // CS (pin select) at high
    GpioWrite(2,1)
    // close the SPI port twice
    SpiClose(handle)
    SpiClose(handle)
    //------------------
    // convert SPI value to Voltage
    //------------------
    // read chr value
    valtemp = StrGetChr(stRead$,0)
    value = StrGetChr(stRead$,1)
    value = (valtemp * 256) + value
    value = value * 2500
    value = value / 65535
    // convert value to string
    SPRINT #tx$,value
    tx$ = "\n" + tx$ + "mV"
    // send to Bluetooth
    n = BleVSpWrite(tx$)
    text$ = ""
  ENDIF
ENDFUNC 1
```