

## BL600 and BL620 smartBASIC Application Walkthrough

### INTRODUCTION

This walkthrough is designed for those new to Bluetooth Low Energy (BLE) and *smartBASIC* on the Laird BL6xx modules. We begin by explaining some fundamental BLE principles that need to be understood before you can begin working with *smartBASIC* on the modules. We then break down a pair of *smartBASIC* programs that demonstrate the BLE principles explained in the first section of the document. The *smartBASIC* programs make use of the BL6xx development board buttons and LEDs. Pressing a button on one development board illuminates the corresponding LED on the other board and vice versa. A straightforward task but only once you have understood how BLE organizes data.

### BLUETOOTH LOW ENERGY BASICS

This section covers some key aspects of BLE that must be understood before we look at the *smartBASIC* code.

---

**Note:** This BLE overview is written with Bluetooth 4.0 in mind as used on the BL600/BL620.

---

The Bluetooth 4.0 core specification introduced Bluetooth Low Energy (BLE) also known as Bluetooth Smart. Bluetooth 4.0 covers both classic and low energy Bluetooth, so BLE can be thought of as a subset of the Bluetooth 4.0 specification. Although BLE shares some aspects of classic Bluetooth, it works in a very different way and should not be thought of as simply a lower power version of classic Bluetooth. Rather than streaming data in the way classic Bluetooth does, BLE focuses on exposing state information in a simple and efficient way (Figure 1).



**Figure 1: BLE communication**

### GAP – Advertising and Connections

#### Adverts

The Generic Access Profile (GAP) allows BLE devices to broadcast, discover, and connect with each other. Many aspects of BLE (including GAP) are very asymmetrical. A peripheral role device advertises its presence and a central role device scans for adverts from peripheral devices.

Everything in BLE starts with an advert from a peripheral device (the only devices that can advertise). Likewise, a central role device is the only device that can scan for adverts. An advert may be broadcasting information, it may be inviting a connection, or it may be doing both.

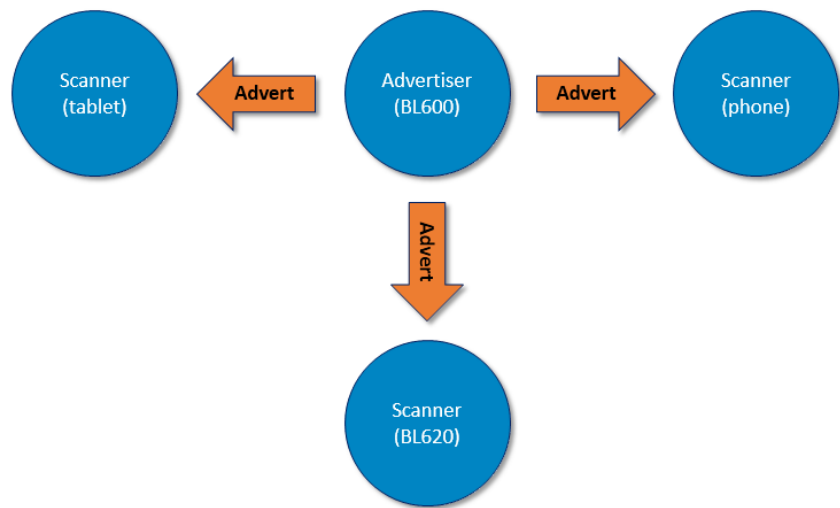


Figure 2: Advert (BL600)

Adverts are sent periodically and the longer the advertising interval (the time between adverts), the less power is consumed. The shorter the advertising interval, the more responsive the application feels but at the cost of higher power consumption (Figure 3).

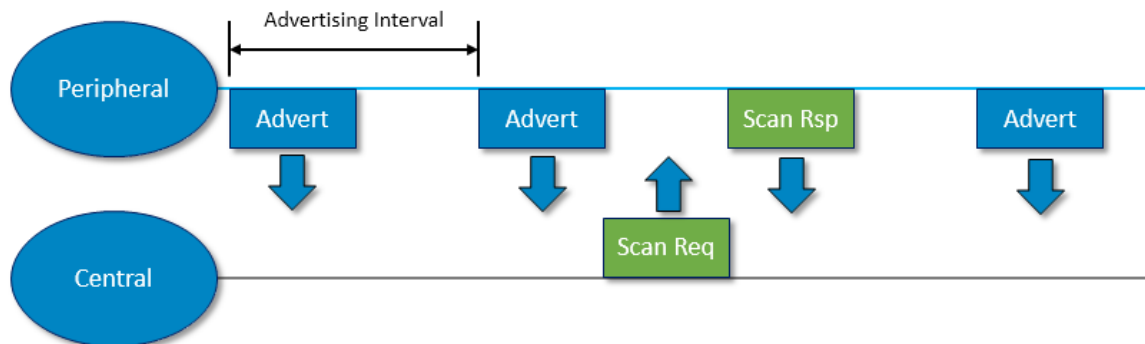


Figure 3: Advertising intervals

As well as inviting connections, adverts can indicate that more advert data is available if requested by the central device sending the scan request. Upon receiving a scan request, an advertiser sends a second packet of advertising data without the need of a connection.

Adverts are made up of one or more fields of data; each data field is identified by a data type value as listed on the GAP page of the BT SIG (Special Interest Group) website: <https://www.bluetooth.org/en-us/specification/assigned-numbers/generic-access-profile>.

For example, if you want to include the name of your device in the advert, you may use data type **0x09 «Complete Local Name»**.

## BL600 and BL620 smartBASIC Application Walkthrough

You can find a description along with the format for each datatype in the *Supplement to Bluetooth Core Specification* which can be found here:

[https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc\\_id=302735](https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc_id=302735)

We recommend that you become familiar with these two resources and their locations on the Bluetooth.org website as they are key to understanding and creating adverts.

The application designer can choose which data types to include in the advert (see the list located on the GAP page of the BT SIG website accessible from the earlier link). Adverts are limited to 31 bytes in size but, but using a scan response, it is possible to make use of two advertising packets.

A typical raw advert payload may appear as shown in the following. Notice how the total length in bytes exceeds the advert packet limit of 31 bytes. In this case, the complete local name is available by way of the scan response.

**Example** – Raw advert data captured with a sniffer with the different data types differentiated by color:

```
Ox02010611077C16A55EBA11CB920C497FB802199A561B094C616972642042746E204C45442044656
D6F202D20424C363030
```

Table 1 displays the advert data breakdown.

**Table 1: Advert data breakdown**

Length	Type	Value	Notes
0x02	0x01	0x06	Flags
0x11	0x07	0X7C16A55EBA11CB920C497FB802199A56	Complete list of 128 bit service UUIDs (what data to expect to be available when connected)
0x1B	0x09	0X4C616972642042746E204C45442044656 D6F202D20424C363030	Complete local name, in this case <i>Laird Btn LED Demo - BL600</i>

As previously stated, an advert can be just a broadcast, it can invite connections, or it can be/do both. The different types of adverts are listed in Table 2. An iBeacon is a good example of a broadcast-only application where the advert contains data to help a smartphone application provide location context information to the user.

**Table 2: Advert types**

Advert Type	Scannable	Connectable	Description
ADV_IND	YES	YES	Connectable and undirected
ADV_DIRECT_IND	NO	YES	Only the specified device may connect
ADV_NONCONN_IND	NO	NO	Broadcast
ADV_SCAN_IND	YES	NO	Scannable broadcast

## BL600 Single Mode Peripheral Device

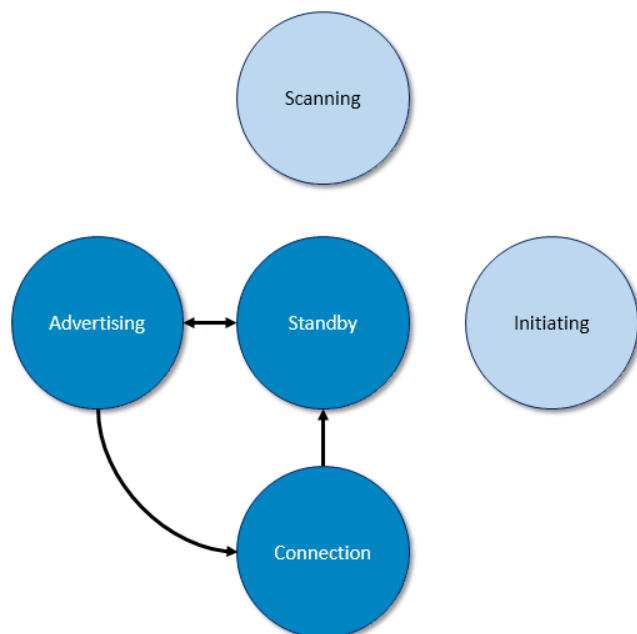
BLE is very asymmetrical, peripheral devices advertise their presence to scanning central devices. The Laird BL600 BLE module is a single mode device (BLE only) that supports only the peripheral device role. So it will be the BL600 that broadcasts adverts so that a central device can discover and connect to it.

### BL620 Single Mode Central Device

The Laird BL620 BLE module is a single mode device that supports only the central role. Therefore it is the BL620 that will search for and initiate a connection with an advertising peripheral device.

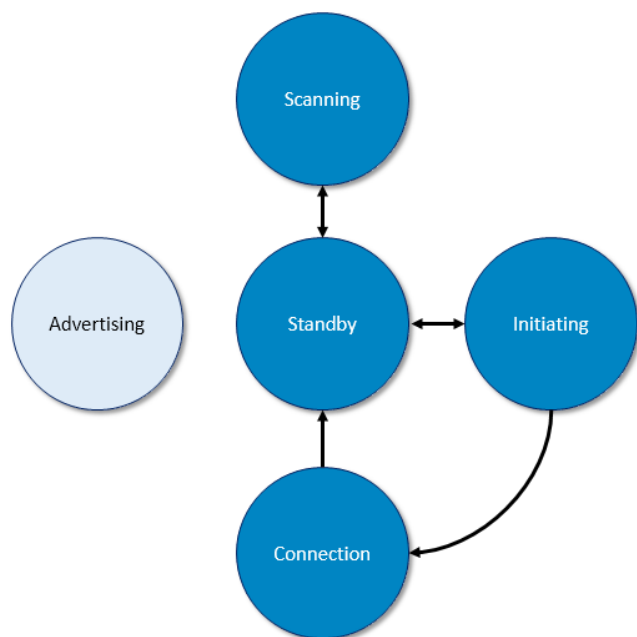
#### CONNECTIONS

As mentioned previously, a BLE application might involve only adverts (broadcasts) or it might invite other devices to connect with an advert. Only peripheral devices advertise, inviting connections from scanning central devices. The functional states between which a BLE device moves differ between peripheral and central device (see Figure 4 and Figure 5). The darker circles indicate the possible functional states when performing each role.



**Figure 4: Peripheral role device**

The central device initiates the connection after discovering an advert from a peripheral by way of a scan.



**Figure 5: Central role device**

### Connection Parameters

One of the fundamental things to understand is that during a BLE connection the radios are switched off for as much of the time as possible, the more you can keep the radios switched off the lower the power consumption will be. Although it should be noted that this is at odds with throughput, but remember BLE is about exposing state information not streaming large amounts of data.

Connections are made up of connection events, each connection event involves an exchange of packets between the master (central in a connection) and the slave (peripheral) in a connection. The time between each connection event is known as the connection interval (Figure 6).

Slave latency is the number of connection events that the slave can ignore but still remain in a connection. This allows for low latency provided by frequent connection events but allow low power operation for the peripheral device by only actively taking part in a connection event when it needs to.

There is also a connection timeout, which is the time between two packets before a connection is considered lost.

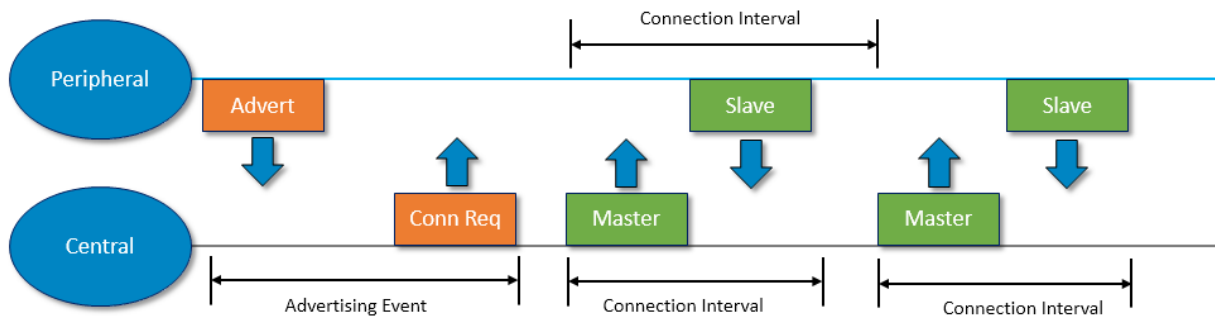


Figure 6: Connection events

### Connected Topology

A BLE master (Figure 7) may be in a connection with multiple peripheral/slave devices but a slave device can only be in a connection with a single master. Exactly how many concurrent connections depends on the central/slave resources. A smartphone or tablet is likely to be able support more connections than an embedded module and a BLE equipped personal computer more again.

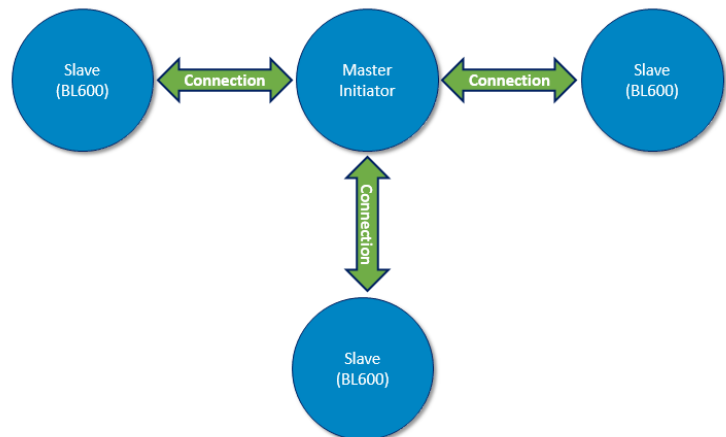


Figure 7: BLE master

### GATT - SERVICES AND CHARACTERISTICS

BLE uses GATT (General Attribute Profile) and GATT communicates using the ATT (Attribute Protocol) where data is transferred as attributes. Attributes are just pieces of data at the end of the day. There are different types of attributes such as characteristics, values and descriptors each identified with a 16 bit UUID listed on the Bluetooth.org website at:

<https://www.bluetooth.org/en-us/specification/assigned-numbers>. 128 bit UUIDs can be used by developers for proprietary services and characteristics so that data not adopted by the Bluetooth SIG can be sent using BLE.

#### Characteristics

A characteristic can be thought of as a pot of data, which contains a value with descriptors providing additional information about the characteristic. GATT is essentially giving structure and hierarchy to attributes. Characteristics can contain zero or more descriptors.

#### Services

Services can contain zero or more characteristics, Services group characteristics together into logical groupings.

#### Profiles

BLE profiles differ from classic BT profiles in that they don't define a protocol, instead they describe how specific device types go about discovering, connecting with and sharing data with each other.

#### GATT Server/Client

The attributes that make up characteristics and services are grouped together on a GATT server where a server has data and a client wants data. When a client wants the data it reads from the GATT server but it can also write data to the GATT server.

For the scope of this document the BL600 will be the GATT server and the BL620 will be the GATT client. It should be noted that a GATT server can reside on either the peripheral role, the central role device or even both at the same time.

#### Handles

An attribute handle is a 16 bit identifier that is used to locate a particular attribute in a GATT server. Handles are discovered by the client by way of a discovery process. A handle will not change between transactions and between bonded devices will not change between connections.

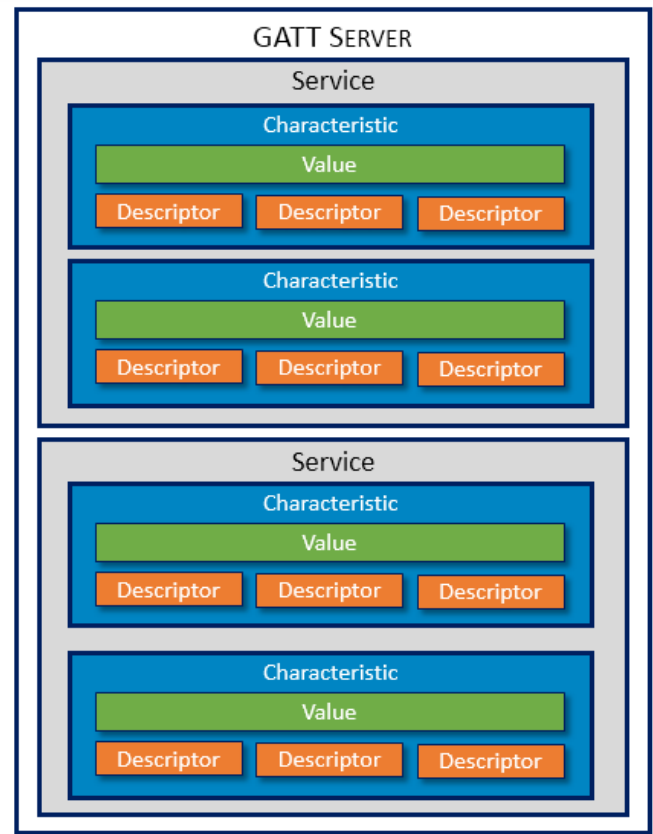


Figure 8: GATT services and characteristics

## BL600 and BL620 *smartBASIC* Application Walkthrough

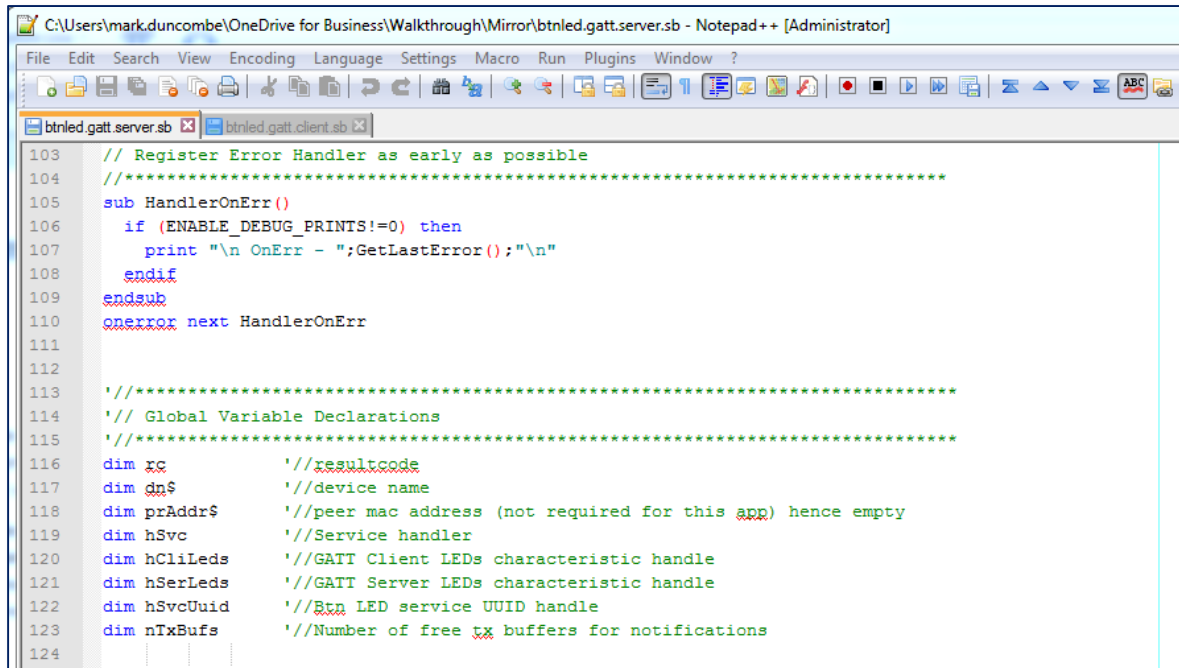
It should be noted that some client devices may cache handles so that a discovery operation need not be carried out during each connection. Instead the client maintains a list of previously discovered handles locally. BLE provides a method for the server to communicate to the client that a service and its handles have changed. This is achieved by way of the service changed characteristic that is part of the mandatory GATT service.

Problems can occur if a BL600 module has been loaded with a *smartBASIC* that program creates a set of services different from the services it had previously used when bonding with a client. Therefore it is recommended to either use a **at&f\*** to clear the non-volatile memory before loading a new *smartBASIC* program or use **at+btd\*** to clear any bonding information. This should be done on both the central and client devices to ensure you are starting from a known state.

## SOFTWARE TOOLS

### Text Editor

*smartBASIC* source can be written/edited in your preferred text editor; however we provide color syntax files for Textpad and Notepad++ (Figure 9) to make the source code more attractive and readable. More information on how to setup color syntax can be found on the BL600 support page. [https://laird-ews-support.desk.com/?b\\_id=1945](https://laird-ews-support.desk.com/?b_id=1945)



```
103 // Register Error Handler as early as possible
104 //*****
105 sub HandlerOnErr()
106     if (ENABLE_DEBUG_PRINTS!=0) then
107         print "\n OnErr - ";GetLastError();"\n"
108     endif
109 endsub
110 onerror next HandlerOnErr
111
112
113 //*****
114 // Global Variable Declarations
115 //*****
116 dim rc           '//resultCode
117 dim dn$          '//device name
118 dim prAddr$      '//peer mac address (not required for this app) hence empty
119 dim hSvc         '//Service handler
120 dim hCliLeds     '//GATT Client LEDs characteristic handle
121 dim hSerLeds     '//GATT Server LEDs characteristic handle
122 dim hSvcUuid     '//Bt LED service UUID handle
123 dim nTxBufs     '//Number of free tx buffers for notifications
124
```

Figure 9: Notepad++ text editor

### UwTerminal

Laird provides a terminal emulator called UwTerminal for use with our Bluetooth modules. Any terminal emulator such as Terraterm or RealTerm can also be used be UwTerminal has a number of very useful additions for example being able to compile and load *smartBASIC* programs onto the development board.

Further information on using UwTerminal can be found in the BL600 section of the Laird module support center.

[https://laird-ews-support.desk.com/?b\\_id=1945](https://laird-ews-support.desk.com/?b_id=1945)



## BL600 and BL620 *smartBASIC* Application Walkthrough

### BLE BUTTON/LED DEMO *SMARTBASIC*

This walkthrough uses the following *smartBASIC* sample programs available from our Github site accessible from: <https://github.com/LairdCP/BL600-Applications>

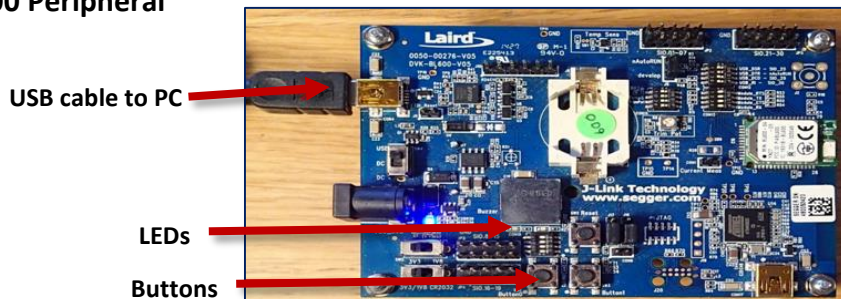
- btnled.gatt.server.sb (BL600)
- btnled.gatt.client.sb (BL620)

Because they are designed to be run on the BL600/620 development boards, two development boards are required – one loaded with BL600 firmware and the other loaded with BL620 firmware.

Instructions for loading firmware onto a development board and the latest firmware are available from our support site: [https://laird-ews-support.desk.com/?b\\_id=1909](https://laird-ews-support.desk.com/?b_id=1909)

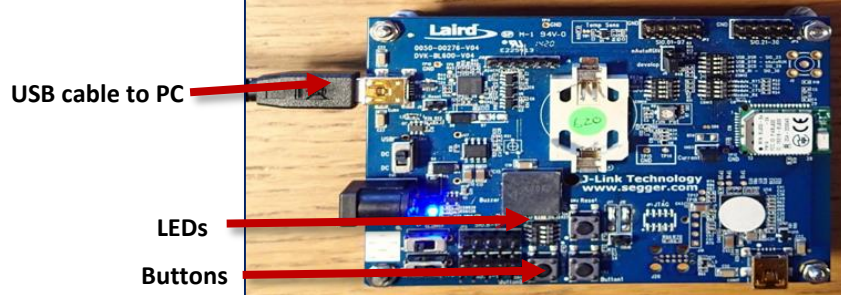
It is also possible to replace the BL620 and the client *smartBASIC* program with an Android BT4.0 smartphone running Nordics Mater Control Panel application: [https://play.google.com/store/apps/details?id=no.nordicsemi.android.mcp&hl=en\\_GB](https://play.google.com/store/apps/details?id=no.nordicsemi.android.mcp&hl=en_GB). Nordic master control panel is an Android BLE application that takes on the central role, allowing you to inspect and interact with GAP and GATT data.

#### BL600 Peripheral



btnled.gatt.server.sb

#### BL620 Central



btnled.gatt.client.sb

Figure 10: BL600 Peripheral and BL620 Central



## BL600 and BL620 *smart*BASIC Application Walkthrough

Ensure that switches 3 and 4 on Conn1 5 are to the left to enable the LEDs on the development boards as [Figure 11](#).

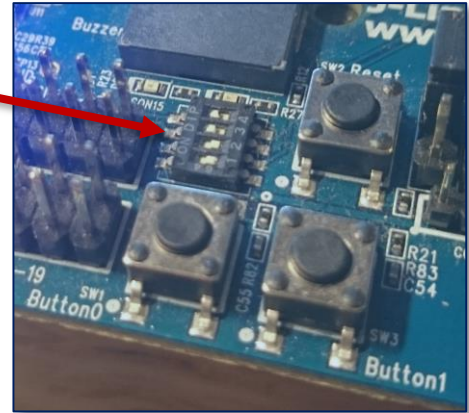


Figure 11: Switch setup

## Running the Demo

### **at&f\***

It's good practice to start from a known state by clearing the non-volatile memory from both of the development boards using **at&f\***.

### **ati 0**

It's also good practice to confirm you have communications with the development board; using the **ati 0** command confirms to which device you are connected.

## Compiling

UwTerminalX also allows for *smart*BASIC text source files (.sb) to be compiled and loaded into a module by right-clicking in its main window and choosing **Xcompile + Load**. For this demo, we want to XCompile and load the following.

- btnled.gatt.server.sb (BL600)
- btnled.gatt.client.sb (BL620)

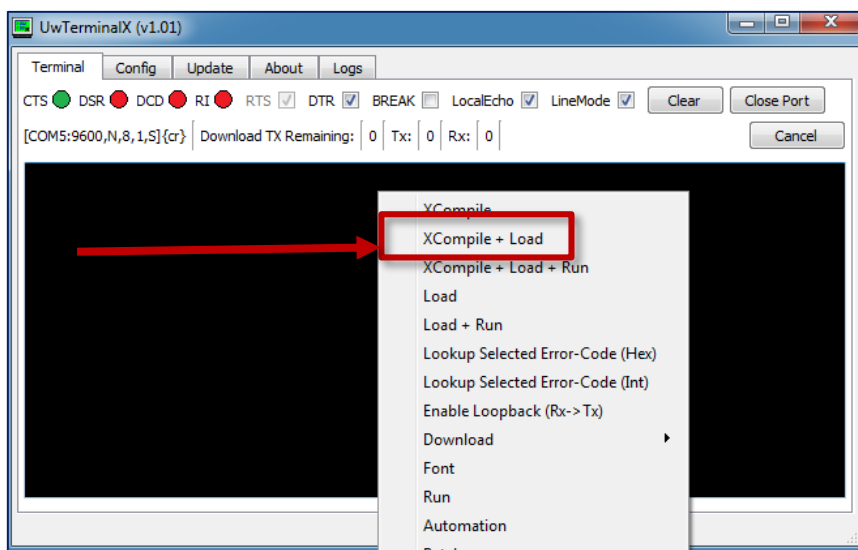


Figure 12: XCompile and Load - UwTerminalX

## BL600 and BL620 *smart*BASIC Application Walkthrough

### at+dir

Once the programs have been compiled and loaded onto the deployment boards, its good practice to confirm they have been loaded by using **at+dir** to check the file system.

### Btnled

To run the programs on each of the development boards, simply type **btnled** followed by a return, first on the BL600 (server) which then begins advertising (Figure 13).

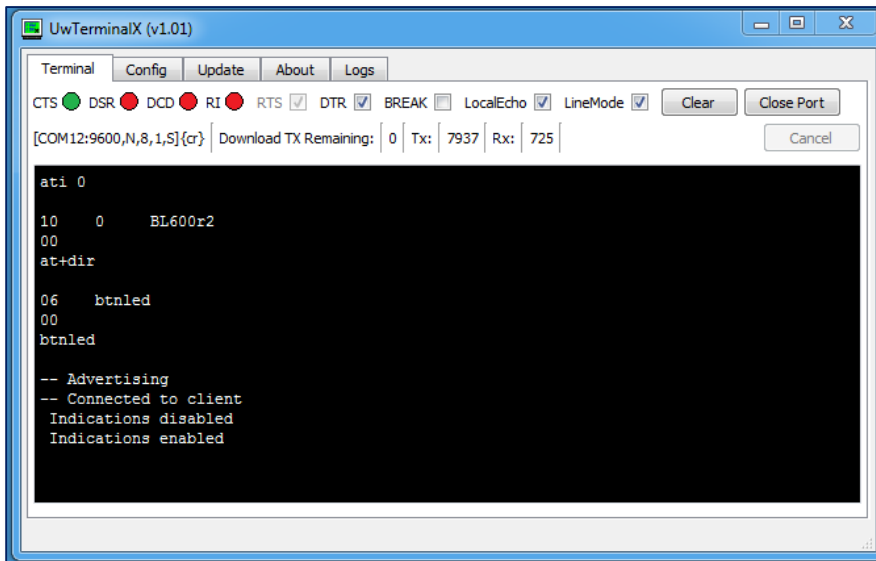


Figure 13: BL600 advertising

Once the BL600 is advertising, you can run **btnled** on the BL620 (client) at which point it scans for the adverts from the BL600 and connects to it.

Once connected, pressing a button on one of the development boards should result in the corresponding LED lighting on the other (Figure 14).

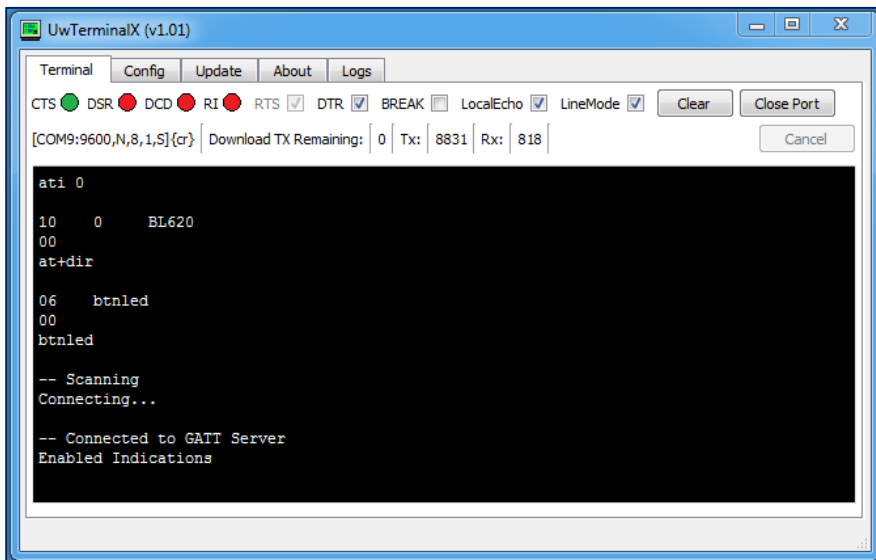


Figure 14: BL620 scanning

### Overview

The aim of this demonstration is to allow the buttons on a BL600 development board, when pressed, to illuminate the LEDs on a BL620 development board and vice versa.

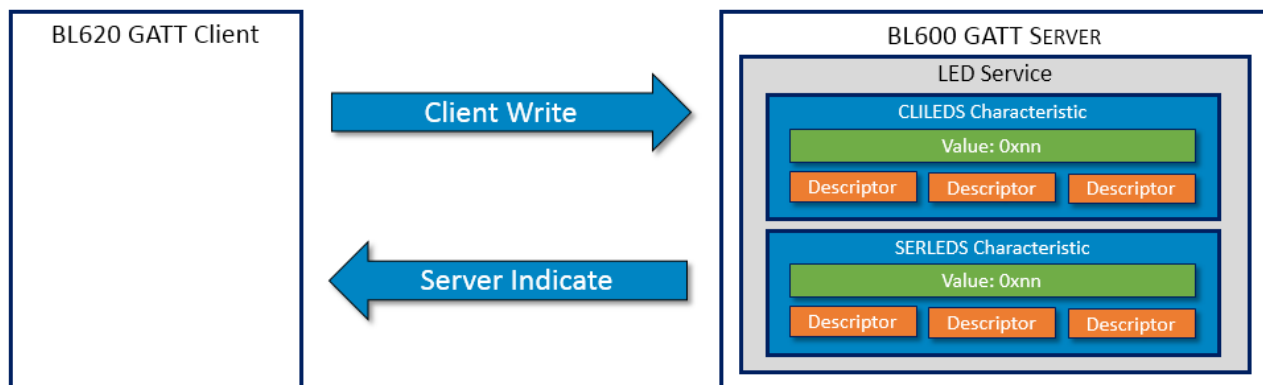
To do this we must create a custom service on the GATT server (BL600) with two characteristics. One characteristic (SERLEDS\_CHAR) controls the BL600 development board LEDs and the other (CLILEDS\_CHAR) controls the BL620 development board LEDs. As both characteristics reside on the GATT server, one is written to by the client and the other is read by the client.

A custom service requires a 128 bit UUID, as opposed to a 16 bit UUID, for SIG adopted services. Each characteristic also needs a UUID but, to save memory, we use a randomly-generated 128 bit UUID as our Laird base UUID and then use 16 bit offsets of that base UUID for the service and characteristics.

**Table 3: Custom service**

	Name	UUID
	Laird Base UUID	569A----B87F490C92CB11BA5EA5167C
<b>Service</b>	LED_SVC	569A <b>1902</b> B87F490C92CB11BA5EA5167C
<b>Characteristic</b>	CLILEDS_CHAR	569A <b>2030</b> B87F490C92CB11BA5EA5167C
<b>Characteristic</b>	SERLEDS_CHAR	569A <b>2031</b> B87F490C92CB11BA5EA5167C

There are also mandatory services which the BL600 creates automatically. For the purposes of this demo we are going to concentrate on custom LED\_SVC service. These services and characteristics can be explored using Nordics master control panel application for Android and are not covered further in this document.



**Figure 15: Custom service**

#### CLILED and SERLEDCharacteristic Values:

- 0x00 → Both LEDs are off
- 0x01 → LED 0 is on, LED 1 is off
- 0x02 → LED 1 is on, LED 0 is off
- 0x03 → Both LEDs are on

## BL600 and BL620 *smart*BASIC Application Walkthrough

### Basic Program Flow

The following flow chart (Figure 16) shows the *smart*BASIC startup routine. This runs when the program starts but does not show variable and constant definitions; these are described later in this walkthrough document.

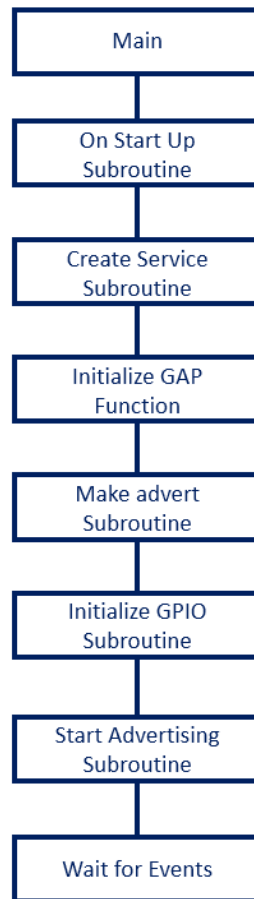


Figure 16: Basic program flow chart

### BL600 *smart*BASIC Code Sections

In this section, we break down the *smart*BASIC programs block-by-block.

#### Definitions

**#define** is used to define constants which are values that do not change while a *smart*BASIC program is running. For example, with the following, any occurrence of **DEVICENAME** in the source will be replaced with the string **Laird Btn LED Demo - BL600**.

```
#define DEVICENAME "Laird Btn LED Demo - BL600"
```

This allows for the device name to be changed once rather than multiple times throughout the source code. We also define the Laird base UUID here – this is a 128 bit randomly-generated number that Laird uses for any custom services and characteristics. To make efficient use of memory, each service and characteristic takes that base UUID and uses a 16 bit offset. This way we avoid having multiple 128 bit identifiers that would consume memory.

## BL600 and BL620 *smart*BASIC Application Walkthrough

### *Global Variable Declarations*

Global variables are those available from any part of the program (as opposed to local variables only used within a function); they are only accessible after they have been declared and therefore should be declared early in the source code.

### *Initialize Global Variables*

Unlike constants, variables may change while a program runs. Because of this, it may be necessary to set an initial value when the program starts. For example, the following takes the DEVICENAME constant, defined in the definitions section, and places it into the string variable called **dn\$**.

```
dn$=DEVICENAME
```

### *Function and Subroutine Definitions*

#### **Debugging**

Throughout this sample application you will see the use of result codes when calling a function.

For example:

```
rc=gpiofunc(GPIO_BTN0,1,2)    //button 0
digital input with weak pull up resistor

AssertRC(rc,188)
```

Functions always return a value which indicates success or failure and knowing why a function call fails can help when debugging.

**rc** is a variable having been declared previously using the Dim statement. The value returned by the function is placed into **rc** allowing it to be passed to the AssertRC subroutine.

If the function completes successfully, then 0 is returned; if a non-0 value is returned and debugging is enabled then the result code and the line are printed to the UART. The second argument 188 is an arbitrary number created by the programmer so that, should this number be printed to the UART, the programmer can search the source code and find which function threw the error.

```
Sub AssertRC(rc,ln)
    if rc!=0 then
        if (ENABLE_DEBUG_PRINTS!=0) then
            print "\nFail :";integer.h' rc;" at tag ";ln
        endif
    endif
EndSub
```

The result code can then be looked up by highlighting it in a UwTerminal window as in the following example (Figure 17).

## BL600 and BL620 smartBASIC Application Walkthrough

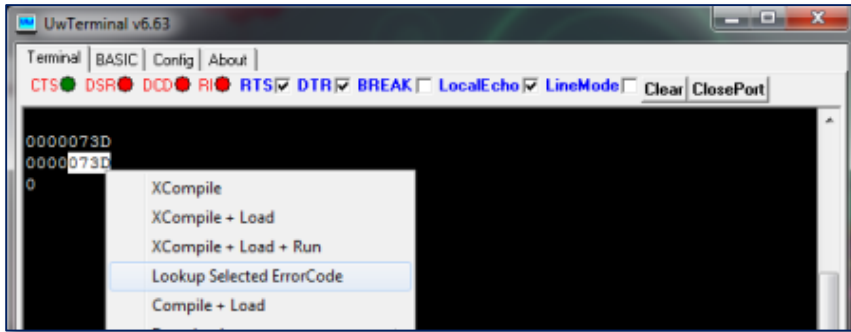


Figure 17: Highlighted result code

Expected Output:

```
//smartBASIC Error Code: 073D -> "RUN_INV_CIRCBUF_HANDLE"
```

### Delay Loop

This subroutine is used to create a delay loop that can be called from other parts of the program. In this case it is used after a button has been pressed and the button GPIO handler called.

```
Sub Delay(ms)
    dim i : i = GetTickCount()
    while GetTickCount(i) < ms
    endwhile
EndSub
```

When the subroutine is called from another part of the program, value is passed to it using the following command:

```
Delay(150)
```

The value of 150 is now used in place of ms effectively substituting ms for the value 150.

Tickcount is a free running timer, so this subroutine captures the current value of Tickcount, putting the value into the local *i* variable. GetTickCount then compares that value against ms (150); when it is no longer less than 150 milliseconds, the subroutine ends.

### Initialize Adverts

Before we can begin advertising we must create the advert report. This is done in three steps:

1. **Initialize** – Initialize the basic advert structure using the BleAdvRptinit function. This function does not start advertising; it only creates the basic format.

We then do the same for the scan report using BleScanRptInit, don't forget advert data can be sent in two packets, the main advert packet and an optional, additional scan response packet.

The advert we are creating will be known as advRpt\$ and the scan report will be known as scnRpt\$.

At this point if we commit and start the advert it would contain the following data which just contains the flags datatype and nothing else.

Len	Type	Value	Notes
2	0x01	0x06	Flags

2. **Append** – To add data to the advert we must append data to it. We are going to be adding the local name to the advert and the 128 bit UUID to the scan report. Don't forget – space is limited in advertising packets so you may need to put some of your data into the main advert and the rest of the data in the scan report. You're using the advert packet to identify the application which allows the central device to decide if it's interested in getting the extra data in the scan report.

The function BLEADVPTADDUUID128 (advRpt, nUuidHandle) is used to add the 128 bit UUID to the advert packet. The UUID was previously set in the Initialize Service section above.

After committing and starting the advert, it contains the following information:

Len	Type	Value	Notes
2	0x01	0x06	Flags
17	0x07	7C16A55EBA11CB920C497FB802199A56	128 bit UUID

Now we add the scan report data using the function BLEADVPTAPPENDAD (advRpt, nTag, stData\$).

The advRpt argument is the name of the advert or scan report to which we want to append; in this case, it's the scan report named **ScnRpt\$**.

The nTag argument is the datatype we wish to add to the scan report.

The stData\$ is the value for the AD type, in our case the value is held within the dn\$ variable. Or advert data will now look as follows, note that this is the combined advert and scan report. Data types where previously covered in a previous section.

Len	Type	Value	Notes
2	0x01	0x06	Flags
17	0x07	0x7C16A55EBA11CB920C497FB802199A56	128 bit UUID
27	0x09	0x4C616972642042746E204C45442044656D6F202D20424C363030	Local name

```
Sub MakeAdvRpts ()  
    dim advRpt$, scnRpt$  
  
    '//Initialise the advert report
```



```
rc = BleAdvRptInit(advRpt$, 2, 0, 0)

'//initialise scan report
rc=BleScanRptInit(scnRpt$)

'//Add local name to scan report
rc=BleAdvRptAppendAD(scnRpt$, 0x09, dn$)

'//Add led svc uuid to advert report
rc=BleAdvRptAddUuid128(advRpt$, hSvcUuid)

// print "\n";advRpt$
// print "\n";StrHexize$(advRpt$)

'//Commit the reports to stack
rc = BleAdvRptsCommit(advRpt$, scnRpt$)

EndSub
```

3. **Commit** – The last stage in creating an advert is the commit which is handled by the BleAdvRptsCommit function. This function takes the advert and scan report and creates the advert. The actual advertisements will not happen until the BLEAdvertStart function is called.

### Start Advertising

Once an advert has been initialized, appended, and committed, the advert module starts advertising once the BLEADVERTSTART function has been called. This function has the following functions:

- nAdvType – Defined in the # define section of the source; in our case this value is 0 (ADV\_IND), a scannable and connectable advert.
- peerAddr\$ – Only used with directed adverts so is not relevant to this example.
- nAdvInterval – The time between advertising events.
- nAdvTimeout – The time after which the module stops advertising.
- nFilterPolicy – Whitelisting and filter policies are not in the scope of this document.

```
Sub StartAdvertising()
    rc = BleAdvertStart(ADVERT_TYPE, prAddr$, ADV_INTERVAL_MS, ADV_TIMEOUT_MS, 0)
    print "\n-- Advertising"
EndSub
```

### Initialize GPIO

Because we use LEDs and buttons on both development kits, we must configure the GPIO correctly with the GpioSetFunc function. The following are the applicable parameters:

GPIOSETFUNC (nSigNum, nFunction, nSubFunc)

## BL600 and BL620 smartBASIC Application Walkthrough

```
rc=gpioSetFunc(GPIO_BTN0,1,2)    //button 0 digital input with weak pull up
resistor
rc=gpioSetFunc(GPIO_BTN1,1,2)    //button 1 digital input with weak pull up
resistor
rc=GpioSetFunc(GPIO_LED0,2,0)    //sets sio18 (LED0) as a digital out
rc=GpioSetFunc(GPIO_LED1,2,0)    //sets sio19 (LED1) as a digital out
GpioWrite(GPIO_LED0,0)
```

The SigNum refers to the GPIO pin in question and the Function and SubFunctions are shown in Table 4. Please refer to the BL600 smartBASIC guide for full GPIO configuration options.

**Table 4: Functions and subfunctions**

nFunction	nSubFunc	Function Description	Sub function Description	Notes
1	1	Digital in	Pull down weak	
<b>1</b>	<b>2</b>	<b>Digital in</b>	<b>Pull up weak</b>	<b>Demo button configuration</b>
1	3	Digital in	Pull down strong	
1	4	Digital in	Pull up strong	
<b>2</b>	<b>0</b>	<b>Digital out</b>	<b>Initial output low</b>	<b>Demo LED configuration</b>
2	1	Digital out	Initial output high	
2	2	Digital out	PWM output	
2	3	Digital out	Frequency output	

### On Start-up

This subroutine performs tasks we require when the program first runs. Optionally, we first print some information to the UART showing some basic information about the operation of this demo. We then call the following subroutines (described elsewhere in this document):

- CreateSvc
- MakeAdvRpts
- InitGpios
- StartAdvertising

As well as initializing the GAP service with the following:

```
rc=BleGapSvcInit(dn$,DEVICENAME_WRITABLE,APPEARANCE,MIN_CONN_INTERVAL,MAX_CONN_INT
ERVAL,CONN_SUP_TIMEOUT,SLAVE_LATENCY)
```

The GAP service is a mandatory GATT service that every device must include in its attributes. It is freely readable during a connection and contains the following:

- Device name – A user-readable friendly name
- Appearance Characteristic – Typically used to provide a generic identification icon to connected devices, often seen in a smartphone's Bluetooth settings page.
- Preferred Connection Parameters – These are the connection parameters preferred by the peripheral device but note that the central device is under no obligation to use them.

## BL600 and BL620 *smart*BASIC Application Walkthrough

### Handler Definitions

Smartbasic is an event driven language and therefore its structure revolves around events and their handlers. When an event occurs, it calls the appropriate handler. The handler is one or more functions that define what should be done when that event occurs.

#### New Characteristic Value (HndlrCharVal)

This handler is triggered when a characteristic value has changed. In our case, this happens when the client writes a new value into SERLEDS char which is then used to determine which of the development boards LEDs is to be switched on. The event passes the characteristic handle to the function from which the value is read and then applies it to the GPIOs.

```
Function HndlrCharVal(charHndl, offset, len)
    dim i,s$

    //Get characteristic value
    rc=BleCharValueRead(charHndl, s$)

    //write values to LEDs
    GpioWrite(GPIO_LED0, StrGetChr(s$,0))
    GpioWrite(GPIO_LED1, StrGetChr(s$,1))
EndFunc 1
```

#### BLE Event (HndlrBleMsg)

This handler follows a BLE event coming up from the stack communicating if a connection or disconnection has occurred and prints it to the UART. Two parameters are passed, nMsgID and connHandle. Various messages can be passed up from the stack and, in this case, to see if the msgid is 1 (which indicates a disconnection) or 0 (which indicates a connection has occurred). The connection handle is not used in this case.

```
Function HndlrBleMsg(ByVal nMsgId, ByVal connHndl)
    if nMsgID==1 then
        print "\n\n-- Disconnected from client\n"
    elseif nMsgID==0 then
        print "\n-- Connected to client"
    endif
EndFunc 1
```

#### Button Transition (HndlrGpio)

We have already enabled a GpioBind to look for a level transition on a GPIO which in turn triggers a GPIO event. That GPIO event then calls this function which takes the value from the GPIO and passes it to BleCharIndicate which in turn sends an indication to the client to allow the client LED's to be set/cleared as required.

```
Function HndlrGpio()
    OnEvent EvGpioChan0 disable
    OnEvent EvGpioChan1 disable
```

## BL600 and BL620 smartBASIC Application Walkthrough

```

Delay(150)

dim val$
rc=StrSetChr(val$, !GpioRead(GPIO_BTN1), 1)
AssertRC(rc,1203)
rc=StrSetChr(val$, !GpioRead(GPIO_BTN0), 0)
AssertRC(rc,1203)

// print "\n";val$

if nTxBufs > 0 then
    //GATT Server indicates the value to the GATT client
    //In other words, "Turn on your LED"
    rc=BleCharValueIndicate(hCliLeds,val$)
    AssertRC(rc,1203)
    nTxBufs = nTxBufs-1
endif

OnEvent EvGpioChan0 call HndlrGpio
OnEvent EvGpioChan1 call HndlrGpio
EndFunc 1

```

The following diagram (Figure 18) shows the process starting at the button press on the server to the LED action on the client.

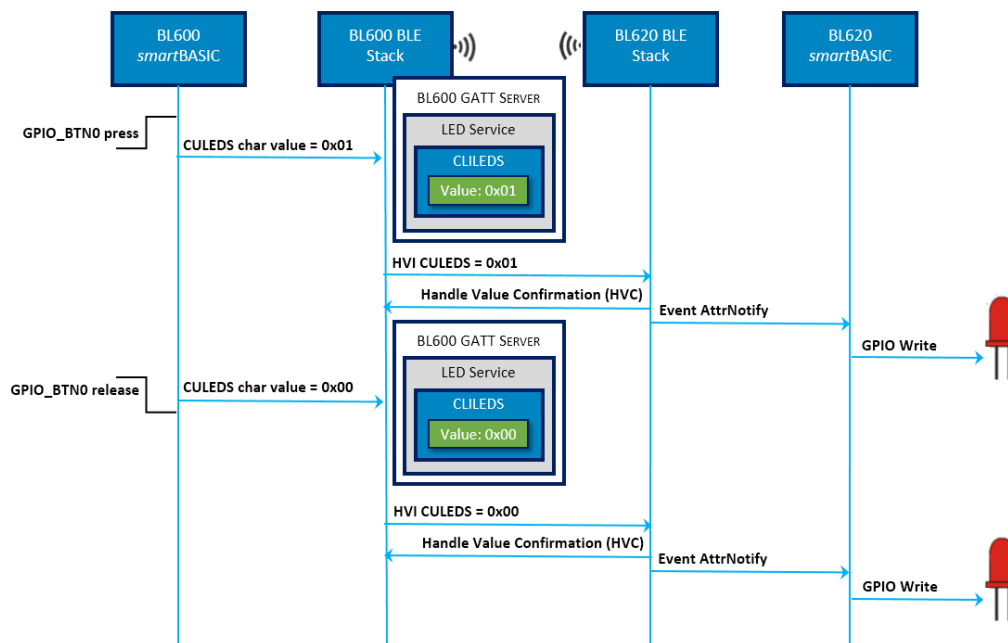


Figure 18: Process - button press on the server to the LED action on the client

## BL600 and BL620 *smart*BASIC Application Walkthrough

### Write CCCD (HndlrCccd)

CCCD stands for Client Characteristic Configuration Descriptor. A descriptor is extra information about a characteristic and its value. In our case, the descriptor is used to enable/disable automatic server-initiated updates of the characteristic value. Once enabled, whenever the CLILEDS characteristic value is changed on the server as a result of a button press on the BL600 development board, the server automatically sends that value by way of an unsolicited indication to the client, without the client having to request that value.

The CCCD can have the following values:

- 0x00 disabled
- 0x01 Notifications enabled
- 0x02 Indications enabled

The difference between notifications and indications is that indications make use of an acknowledgement from the client in the form of a confirmation whereas notifications do not.

```
Function HndlrCccd(charHndl, val)
    if charHndl == hCliLeds then
        // print "\n";val
        if val==2 then
            rc=GpioBindEvent(0,GPIO_BTN0,2)           //binds a gpio transition
            high or low on button 0 (SIO16) to event 0
            AssertRC(rc,309)
            rc=GpioBindEvent(1,GPIO_BTN1,2)           //binds a gpio transition
            high or low on button 1 (SIO17) to event 1
            AssertRC(rc,311)
            print "\n Indications enabled"
            nTxBufs = nTxBufs + 1
        else
            rc=GpioUnbindEvent(0)
            rc=GpioUnbindEvent(1)
            print "\n Indications disabled"
        endif
    endif
EndFunc 1
```

If the value (val) passed to the function equals 2 (indications enabled), then the function enables level changes on the GPIO to trigger an event using GpioBindEvent.

### Disconnection (HndlrDiscon)

When a disconnection event is thrown, this handler restarts the BL600 advertising so that it's ready to accept a new connection. The two parameters hConn and rsn passed are the connection handle and the reason for the disconnection, but they are not used in this example.

```
Function HndlrDiscon(hConn, rsn)
```

```
    StartAdvertising()  
EndFunc 1
```

### Indicate Ack (HndlrCharHvc)

HVC stands for handle value confirmation.

This handler is called from an event EvCharHvc triggered by an acknowledgement to an indication being received by the radio. The single parameter passed to the function from the event is the related handle characteristic. In our case, it is not important as we are only using a single indicating characteristic.

nTxBufs is used in the GPIO handler so that an indication is only sent when txBufs is > 0. Each time we send an indication we deduct one from the TxBuf and each time we get a confirmation we add 1.

```
Function HndlrCharHvc(hChar)  
    nTxBufs = nTxBufs + 1  
EndFunc 1
```

### Main

This part of the program is key; it's here that the program starts running when the onstartup() subroutine is called, which in turn triggers the subroutines described above.

As *smart*BASIC is an event-driven language, it's here we use the OnEvent functions to drive the program when events occur.

```
OnStartup()  
  
OnEvent EvCharVal      call HndlrCharVal  
OnEvent EvBleMsg       call HndlrBleMsg  
OnEvent EvCharCccd     call HndlrCccd  
OnEvent EvGpioChan0    call HndlrGpio  
OnEvent EvGpioChan1    call HndlrGpio  
OnEvent EvDiscon       call HndlrDiscon  
OnEvent EvCharHvc      call HndlrCharHvc
```

### Wait event

The final statement in the program is Waitevent, which never returns but waits for events to happen and calls the appropriate handler. Ideally, a *smart*BASIC program should be at the waitevent statement as much as possible. The following diagram shows the basic flow when each event is triggered. See [Figure 19](#).

## BL600 and BL620 *smart*BASIC Application Walkthrough

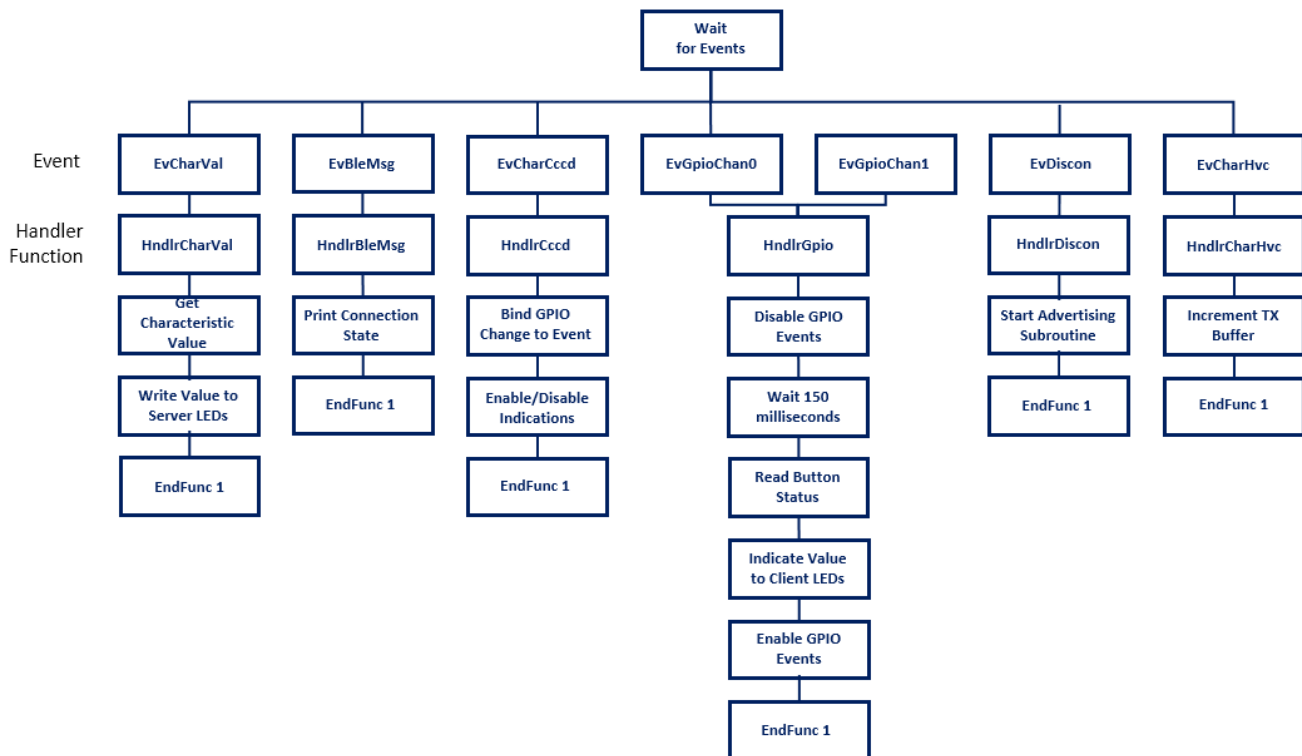


Figure 19: Wait event

## BL620 *SMART*BASIC

Because some sections of the BL620 client code are identical or very similar to the BL600 sever code, we have only included, in this section, those code sections that differ in functionality.

### Function and Subroutine Definitions

#### Start Scanning

This function starts the BL620 scanning for adverts from nearby peripheral devices. If an advert is found then the event EVBLE\_ADV\_REPORT is thrown.

```
Function StartScanning()
    rc = BleScanStart(0,0)
    AssertRC(rc,197)
    print "\n-- Scanning"
EndFunc 1
```

#### Start-up

This subroutine runs when the program starts running. It first calls another subroutine that sets up the GPIO (described elsewhere in this document). BleGattcOpen is used to initialize the GATT client functionality and allocate memory for the appropriate buffers. For the purposes of this demo, we'll use the default settings. More information can be found in the BL620 *smart*BASIC extensions manual available from the BL620 Laird website product page.



```
Sub OnStartup()  
    InitGpios()  
    rc=BleGattcOpen(0,0)  
    // rc=BleScanConfig(2,1)  
    rc=StartScanning()  
EndSub
```

### Handler Definitions

#### *Attribute Notify (HndlrAttrNotify)*

The BL600 GATT server sends its button status by way of an unsolicited indication packet whenever the button state changes. Each time we get an indication from the BL600 GATT server, the event EvNotifyBuf calls this handler which then uses BleGattcNotifyRead to take the indication value (dta\$) and write it to the GPIO to light the LED. The function also returns the connection handle, attribute handle and number of discarded indications. For this simple demo we are assuming that only one indication is received for the client LEDs and therefore we are not concerned with the handles but other application may involve multiple indicating/notifying characteristic values where the handles become important to identify the originating characteristics. StrGetChr simply takes the character from the dta\$ at the position indicated where the character at position 0 is LED 0 and the character at position 1 is LED 1.

```
Function HndlrAttrNotify()  
    // print "\nAttrNotify"  
    dim dta$, nDisc, hAttr, dta  
    '///Read the data  
    rc=BleGattcNotifyRead(hConn, hAttr, dta$, nDisc)  
    AssertRC(rc,239)  
  
    if nDisc == 0 then  
        //write values to LEDs  
        GpioWrite(GPIO_LED0,StrGetChr(dta$,0))  
        GpioWrite(GPIO_LED1,StrGetChr(dta$,1))  
    else  
        print "\n:: ";nDisc;" notifications discarded"  
    endif  
EndFunc 1
```

#### *Attribute Write (HndlrAttrWrite)*

To enable indications from the server we must enable them by writing to the CCCD of the characteristic. If successful then the event EVATTRWRITE is thrown, which in turn calls this handler. Although the connection handle and attribute handle are passed to the handler, since this demo only uses a single indication, we do not need to concern ourselves with these handles; we only need to know whether the action is successful or not

## BL600 and BL620 smartBASIC Application Walkthrough

using the status parameter. ATTR\_WRT\_SUCCESS is a constant, defined in the definitions part of the program as 0. So if the status does not equal 0 then there is a problem.

```
Function HndlrAttrWrite (nCtx, hAttr, status)
    if status != ATTR_WRT_SUCCESS then
        print "\n:: ATT Error 0x"; integer.h'status
    else
        print "\nEnabled Indications"
    endif
EndFunc 1
```

### Notify Buffer (HndlrNotifyBuf)

This handler is called by EVNOTIFYBUF which is thrown when there is at least one free buffer in the stack when writing server LEDs characteristic value on the server.

```
Function HndlrNotifyBuf ()
    txBufs = txBufs + 1
EndFunc 1
```

As long as txBufs is greater than 0, we can write to the value. Conversely, each time we write the value to the characteristic in HndlrGpio, we deduct 1 from txBufs.

### Handler Advert Report (HndlrAdvRpt)

When an advert is seen, this handler is called by EVBLE\_ADV\_REPORT. BleScanGetAdvReport then takes each queued advert and extracts the data payload as advDta\$. We then use the StrPos function to see if the advert contains the UUID of our LED service. If it does, we stop scanning for further adverts and make a connection using BleConnect to the device with the LED service UUID. We know which device to connect to because the address of the device with BleScanAdvertReport was passed.

```
Function HndlrAdvRpt ()
    dim pAddr$, advDta$, nDisc, nRssi, adVal$, tag, found    //found is set to 1
    when laird VSP UUID is found in advert data
    dim uuid$ : uuid$ = LED_SVC_UUID
    do
        rc=BleScanGetAdvReport (pAddr$, advDta$, nDisc, nRssi)

        if StrPos (advDta$,uuid$, 0)>0 then
            rc=BleScanStop ()
            AssertRC (rc,89)
            rc=BleConnect (pAddr$, 15000, MIN_CONN_INTERVAL,
MAX_CONN_INTERVAL, CONN_SUP_TIMEOUT)
            print "\nConnecting...\n"
            AssertRC (rc,92)
            break
        end if
    loop
```

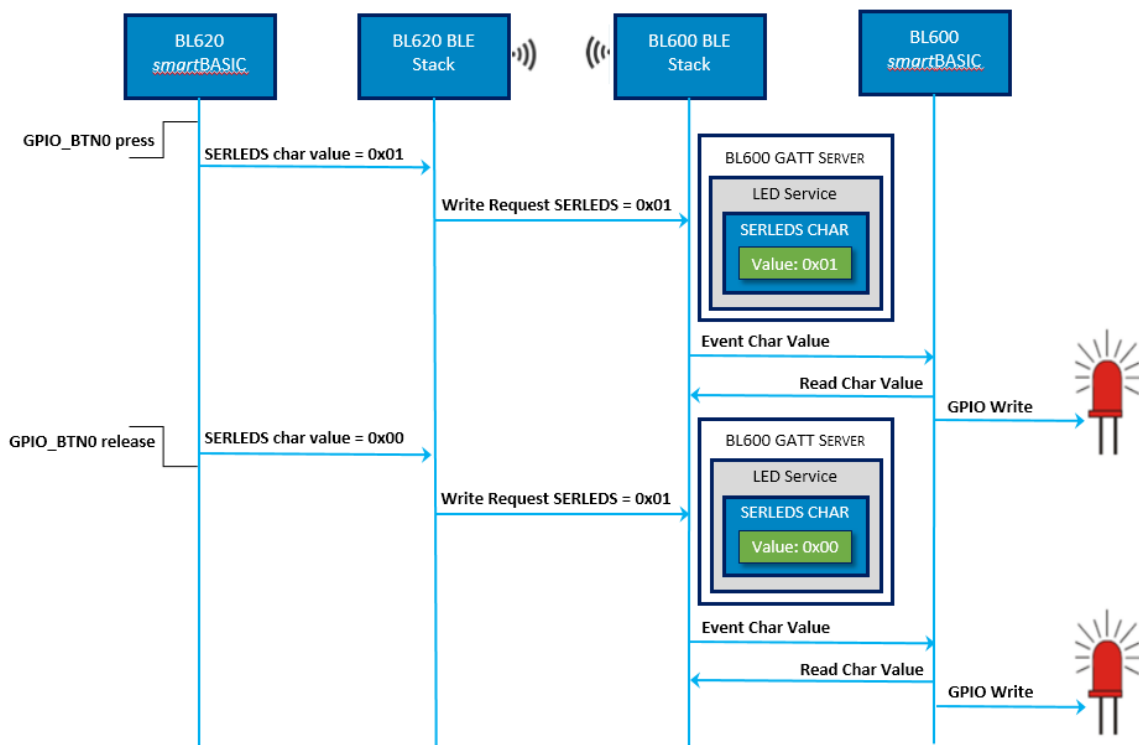
## BL600 and BL620 *smartBASIC* Application Walkthrough

```
endif
until rc!=0
EndFunc 1
```

The `BleConnect` function is used to make a connection to an advertising peripheral such as the BL600. This includes the connection parameters that control the connection interval, slave latency, and timeout. We can see these values are defined in the `#definitions` part of the code in both the BL600 server code and the BL620 client code, but it's important to understand the differences. The BL600 as the peripheral can suggest suitable connection parameters but ultimately it's the BL620 as the central device that actually sets the parameters to be used. This is because a central role device may be servicing multiple peripheral devices and may need to share its resources amongst those peripherals.

### *Handler GPIO (HndlrGpio)*

As soon as a connection is established, the GPIO event is active, calling the GPIO handler whenever a button level changes. This causes the sequence of actions shown in the diagram below to occur and ultimately set/unset the LED on the BL600 server development board.



**Figure 20: Client BL620 Right**

```
Function HndlrGpio()
    OnEvent EvGpioChan0 disable
    OnEvent EvGpioChan1 disable

    Delay(150)

    dim val$
```

## BL600 and BL620 *smart*BASIC Application Walkthrough

```
rc=StrSetChr(val$, !GpioRead(GPIO_BTN1), 1)
AssertRC(rc,1203)
rc=StrSetChr(val$, !GpioRead(GPIO_BTN0), 0)
AssertRC(rc,1203)

if txBufs>0 then
    //GATT Client requests to write to the GATT server's characteristic
    //In other words, "Can I turn on your LED?"
    rc=BleGattcWriteCmd(hConn, SERLEDS_CHAR_HANDLE, val$)
    AssertRC(rc,1203)
    txBufs = txBufs-1
    // print "\n";val$
endif

OnEvent EvGpioChan0 call HndlrGpio
OnEvent EvGpioChan1 call HndlrGpio

EndFunc 1
```

### BLE Event (*HndlrBleMsg*)

Once we have connected to the GATT server on the BL600, we then must enable indications to ensure that button presses on the BL600 are sent to the BL620. To enable indication, we must write to the CCCD of the SERLEDS characteristic. To do this, we must know the handle of the CCCD which comes from the #define in the definitions section, 15 in this case. We can define this as a constant because the handle should never change as we control both the server and the client. However, if connecting to an unknown device, use the discovery process to discover services, UUIDS, and handles.

```
Function HndlrBleMsg(ByVal nMsgId, ByVal connHndl)
    if nMsgID==1 then
        print "\n\n-- Disconnected from client\n"
    elseif nMsgID==0 then
        print "\n\n-- Connected to GATT Server"
        hConn = connHndl

        //write 1 to CCCD to enable notifications
        dim notf$ : notf$ = "\02\00"
        rc=BleGattcWrite(hConn, SERLEDS_CHAR_CCCD_HANDLE, notf$)
        AssertRC(rc,277)

        OnEvent EvGpioChan0 call HndlrGpio
        OnEvent EvGpioChan1 call HndlrGpio
```

## BL600 and BL620 *smart*BASIC Application Walkthrough

```
endif  
EndFunc 1
```

### Main

This is the code that runs on start, with the OnStartup subroutine being called and the events registered. Each time an event occurs the appropriate handler is called.

```
OnStartup()  
  
OnEvent EvAttrWrite call HndlrAttrWrite  
OnEvent EvNotifyBuf call HndlrNotifyBuf  
onevent EVATTRNOTIFY call HndlrAttrNotify  
onevent EVBLE_SCAN_TIMEOUT call StartScanning  
onevent EVBLE_ADV_REPORT call HndlrAdvRpt  
OnEvent EvBleMsg call HndlrBleMsg  
OnEvent EvDiscon call HndlrDiscon
```

## GLOSSARY

Term	Definition
128-bit UUID	Used to identify proprietary services and characteristics defined by a developer
16-bit UUID	A shortened UUID for BT SIG-adopted services and characteristics
Advert	The method by which a peripheral device shares data outside of a connection and invites connections
Advert Data Type	Adverts data is made up of one or more fields, each defined by an advert data type
Advert Report	A string variable in <i>smart</i> BASIC that contains advert data types
Advertiser	A peripheral device sharing data outside of a connection or inviting a connection
Advertising Interval	The time in milliseconds between adverts from a peripheral
Attribute	The smallest piece of addressable data defined in a GATT server
Base UUID	A randomly generated 128-bit number used as a base from which 16-bit UUIDs can be derived
CCCD	Client Characteristic Configuration Descriptor – Used to enable/disable server-initiated characteristic values
Central	A BLE device that scans for adverts from peripheral role devices or initiates connections
Characteristic	Pots of data made up of two or more attributes
Client	GATT client – Clients want a server's data
Connectable	A peripheral role device is connectable if advertising with a connectable advert
Connection Interval	The time in milliseconds during a connection when data packets are exchanged
CSS	Core Specification Supplement <a href="https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc_id=302735">https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc_id=302735</a>

## BL600 and BL620 *smartBASIC* Application Walkthrough

Descriptor	Additional metadata for a characteristic and its value
Dual Mode	Supporting both classic and low energy Bluetooth
Event	A trigger in <i>smartBASIC</i>
Event Handler	The <i>smartBASIC</i> code that runs after being triggered by an event
GATT	Generic Attribute Profile – Defines how data is exchanged within a connection
GAP	Generic Access Profile – Defines how devices broadcast data and invite connections
HVC	Handle Value Confirmation (see CCCD)
HVI	Handle Value Indication (see CCCD)
HVN	Handle Value Notification (see CCCD)
Indication	Unsolicited messages from a GATT server that invoke confirmations from a client
Master	A central role device within a connection
MCP	Nordic Master Control Panel Android application <a href="https://play.google.com/store/apps/details?id=no.nordicsemi.android.mcp&amp;hl=en_GB">https://play.google.com/store/apps/details?id=no.nordicsemi.android.mcp&amp;hl=en_GB</a>
Notification	Unsolicited messages from a GATT server that do not invoke confirmations from the client
Peripheral	A BLE device that broadcasts data or invites connections
Profile	A set of instructions on how devices should share application data
Scan Report	An optional additional piece of advertising data represented as a string in <i>smartBASIC</i>
Scannable	A BLE advertiser with extra data available without needing a connection
Scanner	A BLE device that can receive peripheral broadcasts and connection invitations
Server	A GATT server that holds data
Service	A group of characteristics organised logically
Sibling UUID	A 16-bit UUID derived from a 128-bit base UUID
SIG	Bluetooth Special Interest Group
Single Mode	A device that is BLE only
Slave	A peripheral device within a connection
Slave Latency	The number of connection events a peripheral role device can ignore while still remaining in a connection
UwTerminal	A terminal program developed by Laird specifically for use with its Bluetooth module

## REVISION HISTORY

Version	Date	Notes	Approved By
1.0	01 Dec 2015	Initial Version	Mark Duncombe