*smart*BASIC

BL600 Extensions

User Manual

Release 1.5.70.0-r5

global solutions: local support™

Embedded Wireless Solutions Support Center: http://ews-support.lairdtech.com
Americas: +1-800-492-2320 Option 2
Europe: +44-1628-858-940
Asia: +852-2923-0610
www.lairdtech.com/bluetooth

Laird Technologies
Saturn House,
Mercury Park,
Wooburn Green,
Bucks HP10 0HH,
UK.

Tel: +44 (0) 1628 858 940
Fax: +44 (0) 1628 528 382

REVISION HISTORY

Version	Revisions Date	Change History	Approved By
1.0	1 Feb 2013	Initial Release	Jonathan Kaye
1.1.50.0r3	3 Apr 2013	Production Release	Jonathan Kaye
1.1.51.0	15 Apr 2013	Incorporate review comments for JG	Jonathan Kaye
1.1.51.5	24 Apr 2013	Engineering release	Jonathan Kaye
1.1.53.10	8 May 2013	Engineering release with custom service capability	Jonathan Kaye
1.1.53.20	12 Jun 2013	Engineering release with Virtual Serial Service capability	Jonathan Kaye
1.2.54.0	29 Jun 2013	Production Release	Jonathan Kaye
1.2.55.3	26 Jul 2013	Engineering release with PWM & FREQUENCY output	Jonathan Kaye
1.2.55.5	8 Aug 2013	Engineering release with VSP/Uart Bridging	Jonathan Kaye
1.2.55.8	12 Aug 2013	Engineering release with AT+CFG command	Jonathan Kaye
1.2.55.12	29 Aug 2013	Engineering release with sysinfo$()	Jonathan Kaye
1.3.57.0	12 Sep 2013	Engineering release with UartCloseEx	Jonathan Kaye
1.4.59.0	19 Dec 2013	Engineering release v1.4.59.0	Jonathan Kaye
1.5.62.0	4 Jan 2014	Production release v1.5.62.0 (Softdevice 6.0.0)	Jonathan Kaye
1.5.65.0	24 Feb 2014	Engineering release v1.5.65.0 (Softdevice 6.0.0)	Jonathan Kaye
1.5.66.0	28 Mar 2014	Production release v1.5.66.0 (Softdevice 6.0.0)	Jonathan Kaye
1.5.70.0	27 Apr 2014	Production releae v1.5.70.0 (Softdevice 6.0.0)	Jonathan Kaye
1.5.70.0-r1	28 Apr 2014	Added the VSP flowchart	Jonathan Kaye
1.5.70.0-r2	1 May 2014	Split manuals into Core and BL600 Extension	Jonathan Kaye
1.5.70.0-r4	27 Aug 2014	Sync information with Core Manual	Jonathan Kaye
1.5.70.0-r5	2 Dec 2014	Added UartOpen specifics for this module	Jonathan Kaye

CONTENTS

1. INTRODUCTION

Documentation Overview

This BL600 Extension Functionality user guide provides detailed information on BL600-specific *smart*BASIC extensions which provide a high level managed interface to the underlying Bluetooth stack in order to manage the following:

- GATT table – Services, characteristics, descriptors, advert reports
- Gatt server/client operation
- Advertisments and connections
- BLE security and bonding
- Attribute encoding and decoding
- Laird custom VSP service
- Power management
- Wireless status
- Events related to the above

What Does a BLE Module Contain?

Our *smart*BASIC-based BLE modules are designed to provide a complete wireless processing solution. Each module contains:

- A highly integrated radio with an integrated antenna (external antenna options are also available)
- BLE Physical and Link layer
- Higher level stack
- Multiple GPIO and ADC
- Wired communication interfaces like UART, I2C, and SPI
- A *smart*BASIC run-time engine
- Program accessible flash memory which contains a robust flash file system exposing a conventional file system and a database for storing user configuration data
- Voltage regulators and brown-out detectors

For simple end devices, these modules can completely replace an embedded processing system.

The following block diagram (Figure 1) illustrates the structure of the BLE *smart*BASIC module from a hardware perspective on the left and a firmware/software perspective on the right.

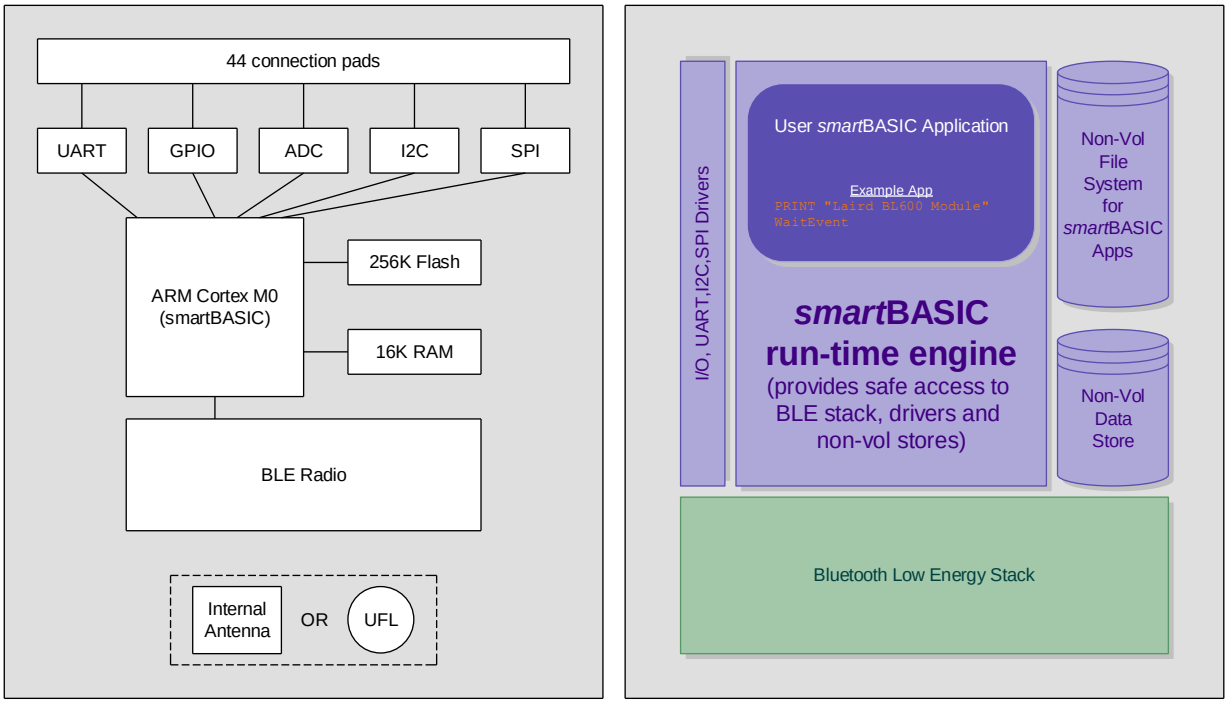


*Figure 1: BLE smart*BASIC *module block diagram*

2. Interactive Mode Commands

Interactive mode commands allow a host processor or terminal emulator to interrogate and control the operation of a *smart*BASIC-based module. Many of these emulate the functionality of AT commands. Others add extra functionality for controlling the filing system and compilation process.

Syntax Unlike commands for AT modems, a space character must be inserted between AT, the command, and subsequent parameters. This allows the *smart* BASIC tokeniser to efficiently distinguish between AT commands and other tokens or variables starting with the letters ***at***.

```
'Example:

AT I 3
```

The response to every Interactive mode command has the following form:

<linefeed character> response text <carriage return>

This format simplifies the parsing within the host processor. The response may be one or multiple lines. Where more than one line is returned, the last line has one of the following formats:

<lf>00<cr> for a successful outcome, or

<lf>01<tab> hex number <tab> optional verbose explanation <cr> for failure.

Note: In the case of the 01 response, the *<tab>optional_verbose_explanation* will be missing in resource constrained platforms like the BL600 modules. The *verbose explanation* is a constant string and since there are over 1000 error codes, these verbose strings can occupy more than ten kilobytes of flash memory.

The hex number in the response is the error result code consisting of two digits which can be used to help investigate the problem causing the failure. Rather than provide a list of all the error codes in this manual, you can use UWTerminal to obtain a verbose description of an error when it is not provided on a platform.

To get the verbose description, click on the BASIC tab (in UWTerminal) and, if the error value is hhhh, enter the command **ER 0xhhhh** and note the 0x prefix to *hhhh*. This is illustrated in Figure 2.

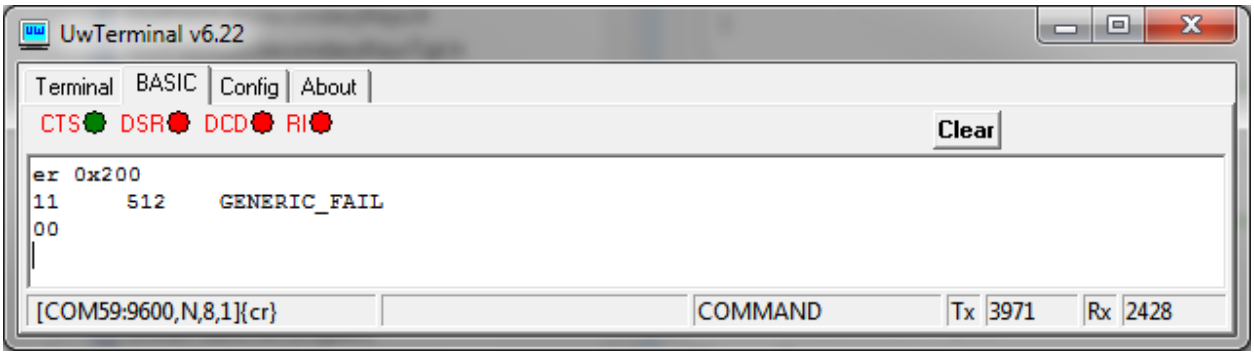


Figure 2: Optional verbose explanation

You can also obtain a verbose description of an error by highlighting the error value, right-clicking, and selecting **Lookup Selected ErrorCode** in the Terminal window.

If you get the text *UNKNOWN RESULT CODE 0xHHHH*, please contact Laird for the latest version of UWterminal.

The following are BL600-specific AT commands.

AT+CFG

COMMAND

AT+CFG is used to set a non-volatile configuration key. Configuration keys are are comparable to S registers in modems. Their values are kept over a power cycle but are deleted if the AT&F* command is used to clear the file system.

If a required configuration key isn't listed below, use the functions NvRecordSet() and NvRecordGet() to set and get these keys respectively.

The **num value** syntax is used to set a new value and the **num ?** syntax is used to query the current value. When the value is read, the syntax of the response is:

```
27    0xhhhhhhhh (dddd)
```

…where 0xhhhhhhhh is an eight hexdigit number which is 0 padded at the left and dddd is the decimal signed value.

AT+CFG num value or AT+CFG num ?

Returns	If the config key is successfully updated or read, the response is **\n00\r**.
Arguments	
num	Integer Constant The ID of the required configuration key. All of the configuration keys are stored as an array of 16 bit words.
value	Integer_constant This is the new value for the configuration key and the syntax allows decimal, octal, hexadecimal, or binary values.

This is an Interactive mode command and **MUST** be terminated by a carriage return for it to be processed.

The following Configuration Key IDs are defined:

ID	Definition	
40	Maximum size of locals simple variables	
41	Maximum size of locals complex variables	
42	Maximum depth of nested user defined functions and subroutines	
43	The size of stack for storing user functions simple variables	
44	The size of stack for storing user functions complex variables	
45	The size of the message argument queue length	
100	Enable/Disable Virtual Serial Port Service when in interactive mode. Valid values are:	
	0x0000	Disable
	0x0001	Enable
	0x80nn	Enable ONLY if signal pin *nn* on module is HIGH
	0xC0nn	Enable ONLY if signal pin *nn* on module is LOW
	0x81nn	Enable ONLY if signal pin *nn* on module is HIGH and auto-bridged to UART when connected
	0xC1nn	Enable ONLY if signal pin *nn* on module is LOW and auto-bridged to UART when connected
	ELSE	Disable
101	Virtual Serial Port Service to use INDICATE or NOTIFY to send data to client.	
	0	Prefer Notify
	Else	Prefer Indicate
	This is a preference and the actual value is forced by the property of the TX characteristic of the service.	
102	This is the advert interval in milliseconds when advertising for connections in interactive mode and AT parse mode. **Valid values:** 20 to 10240 milliseconds	
103	This is the advert timeout in milliseconds when advertising for connections in interactive mode and AT parse mode. **Valid values:** 1 to 16383 seconds	

ID	Definition
104	In the virtual serial port service manager, data transfer is managed. When sending data using NOTIFIES, the underlying stack uses transmission buffers of which there are a finite number. This specifies the number of transmissons to leave unused when sending a lot of data. This also allows other services to send notifies without having to wait for them. The total number of transmission buffers can be determined by calling SYSINFO(2014) or in interactive mode submitting the command ATi 2014
105	When in interactive mode and connected for virtual serial port services, this is the minimum connection interval in milliseconds to be negotiated with the master. **Valid values:** 0 to 4000 ms **Note:** If a value of less than 8 is specified, then the minimum value of 7.5 is selected.
106	When in interactive mode and connected for virtual serial port services, this is the maximum connection interval in milliseconds to be negotiated with the master. **Valid values:** 0 to 4000 ms **Note:** If a value of less than the minimum specified in 105, then it is forced to the value in 105 plus 2 ms.
107	When in interactive mode and connected for virtual serial port services, this is the connection supervision timeout in milliseconds to be negotiated with the master. **Valid range:** 0 to 32000 **Note:** If the value is less than the value in 106, then a value double the one specified in 106 is used.
108	When in interactive mode and connected for virtual serial port services, this is the slave latency to be negotiated with the master. **Note:** An adjusted value is used if this value times the value in 106 is greater than the supervision timeout in 107.
109	When in interactive mode and connected for virtual serial port services, this is the Tx power used for adverts and connections. **Note:** A low value is set to ensure that in production, if *smart*BASIC applications are downloaded over the air, the limited range allows many stations to be used to program devices.
110	If Virtual Serial Port Service is enabled in interactive mode (see 100), then this specifies the size of the transmit ring buffer in the managed layer sitting above the service characteristic fifo register. **Value range:** 32 to 256
111	If Virtual Serial Port Service is enabled in interactive mode (see 100), then this specifies the size of the receive ring buffer in the managed layer sitting above the service characteristic fifo register. **Value range:** 32 to 256
112	If set to 1, then the service UUID for the virtual serial port is as per Nordic's implementation and any other value is per Laird's modified service. See more details of the service definition here.
113	This is the advert interval in milliseconds when advertising for connections in interactive mode and UART bridge mode. **Valid values:** 20 to 10240 milliseconds
114	This is the advert timeout in milliseconds when advertising for connections in interactive mode and UART bridge mode. **Valid values:** 0 to 16383 seconds. 0 disables the timer (makes it continuous)

ID	Definition
115	This is used to specify the UART baudrate when Virtual Serial Mode Service is active and UART bridge mode is enabled. **Valid values:** 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, 76800, 115200, 230400, 250000, 460800, 921600, 1000000. **Note:** If an invalid value is entered, then the default value of 9600 is used.
116	In VSP/UART bridge mode, this value specifies the latency in milliseconds for data arriving via the UART and transfering to VSP and then onward on-air. This mechanism ensures that the underlying bridging algorithm waits for up to this amount of time before deciding that no more data is going to arrive to fill a BLE packet and so flushes the data onwards. **Note:** Given that the largest packet size takes 20 bytes, if more than 20 bytes arrive then the latency timer is overridden and the data is immediately sent.

Interactive Command: YES

AT+CFG is a core command.

Note: These values revert to factory default values if the flash file system is deleted using the AT & F * interactive command.

AT + BTD *

COMMAND

Deletes the bonded device database from the flash.

AT + BTD*	
Returns	\n00\r
Arguments	None
Interactive Command	YES

This is an interactive mode command and **MUST** be terminated by a carriage return for it to be processed.

Note: The module self-reboots so that the bonding manager context is also reset.

```
'Examples:

AT+BTD*
```

AT+BTD* is an extension command.

AT + MAC "12 hex digit mac address"

COMMAND

This is a command that is successful one time as it writes an IEEE MAC address to non-volatile memory. This address is then used instead of the random static MAC address that comes preprogrammed in the module.

Notes: If the module has an invalid licence, then this address is not visible.
If the address 000000000000 is written then it is treated as invalid and prevents a new address from being entered.

AT + MAC "12 hex digits"

Returns	\n00\r or \n01 192A\r Where the error code 192A is NVO_NVWORM_EXISTS. This means that an IEEE MAC address already exists; this can be read using the command AT I 24
Arguments	A string delimited by "" which shall be a valid 12 hex digit MAC address that is written to non-volatile memory.
Interactive Command	YES

This is an interactive mode command and **MUST** be terminated by a carriage return for it to be processed.

Note: The module self-reboots if the write is successful. Subsequent invocations of this command generate an error.

```
'Examples:

AT+MAC "008098010203"
```

AT+MAC is an extension command

AT + BLX

COMMAND

This command is used to stop all radio activity (adverts or connections) when in interactive mode. It is particularly useful when the virtual serial port is enabled while in interactive mode.

AT + BLX

Command

Returns	\n00\r
Arguments	None
Interactive Command	YES

This is an Interactive Mode command and MUST be terminated by a carriage return for it to be processed.

Note: The module self-reboots so that the bonding manager context is also reset.

```
'Examples:

AT+BLX
```

AT+BLX is an extension command.

3. Core Language Built-in Routines

Core language built-in routines are present in every implementation of *smart*BASIC. These routines provide the basic programming functionality. They are augmented with target-specific routines for different platforms which are described in the extention manual for each target platform.

All the core functionality is described in the document *smartBASIC Core Functionality*.

However some functions have small behaviour differences and they are listed below.

Information Routines

SYSINFO

FUNCTION

Returns an informational integer value depending on the value of varId argument.

SYSINFO(varId)

Returns	INTEGER. Value of information corresponding to integer ID requested.
Exceptions	▪ Local Stack Frame Underflow ▪ Local Stack Frame Overflow
Arguments:	
varId	*byVal* varId *AS INTEGER* An integer ID which is used to determine which information is to be returned as described below.

0	ID of device. Each platform type has a unique identifier. BL600 module value – 0x42460600
3	Version number of module Firmware. For example W.X.Y.Z is returned as a 32 bit value made up as follows: (W<<26) + (X<<20) + (Y<<6) + (Z) where Y is the build number and Z is the 'sub-build' number
33	BASIC core version number.
601	Flash File System: Data Segment: Total Space
602	Flash File System: Data Segment: Free Space
603	Flash File System: Data Segment: Deleted Space
611	Flash File System: FAT Segment: Total Space
612	Flash File System: FAT Segment: Free Space

	613	Flash File System: FAT Segment: Deleted Space
	631	NvRecord Memory Store Segment: Total Space
	632	NvRecord Memory Store Segment: Free Space
	633	NvRecord Memory Store Segment: Deleted Space
	1000	BASIC compiler HASH value as a 32 bit decimal value
	1001	How RAND() generates values: 0 for PRNG and 1 for hardware assist
	1002	Minimum baudrate
	1003	Maximum baudrate
	1004	Maximum STRING size
	1005	1 for run-time only implementation 3 for compiler included
	2000	Reason for reset: 8 – Self-Reset due to Flash Erase 9 – ATZ 10 – Self-Reset due to *smart* BASIC app invoking function RESET()
	2002	Timer resolution in microseconds
	2003	Number of timers available in a *smart* BASIC Application
	2004	Tick timer resolution in microseconds
Interactive Command	NO	

```
//Example :: SysInfo.sb (See in Firmware Zip file)
PRINT "\nSysInfo 1000   = ";SYSINFO(1000)    // BASIC compiler HASH value
PRINT "\nSysInfo 2003   = ";SYSINFO(2003)    // Number of timers
PRINT "\nSysInfo 0x8010 = ";SYSINFO(0x8010)  // Code memory page size from FICR
```

Expected Output (For BL600):

```
SysInfo 1000   = 1315489536
SysInfo 2003   = 8
SysInfo 0x8010 = 1024
```

SYSINFO is a core language function.

SYSINFO$

FUNCTION

Returns an informational string value depending on the value of **varId** argument.

SYSINFO$(varId)

Returns	STRING. Value of information corresponding to integer ID requested.
Exceptions	▪ Local Stack Frame Underflow ▪ Local Stack Frame Overflow
Arguments:	
varId	*byVal* varId *AS INTEGER* An integer ID which is used to determine which information is to be returned as described below.

4	The Bluetooth address of the module. It is seven bytes long. First byte is 00 for IEEE public address and 01 for random public address. Next six bytes are the address.
14	A random public address unique to this module. May be the same value as in 4 above unless AT+MAC was used to set an IEEE mac address. It is seven bytes long. First byte is 00 for IEEE public address and 01 for random public address. Next six bytes are the address.

Interactive Command	NO

```
//Example :: SysInfo$.sb (See in Firmware Zip file)
PRINT "\nSysInfo$(4)    = ";SYSINFO$(4)    // address of module
PRINT "\nSysInfo$(14)   = ";SYSINFO$(14)   // public random address
PRINT "\nSysInfo$(0)    = ";SYSINFO$(0)
```

Expected Output:

```
SysInfo$(4)    = \01\FA\84\D7H\D9\03
SysInfo$(14)   = \01\FA\84\D7H\D9\03
SysInfo$(0)    =
```

SYSINFO$ is a core language function.

UART (Universal Asynchronous Receive Transmit)

UartOpen

FUNCTION

This function is used to open the main default uart peripheral using the parameters specified.

See core manual for further details.

UARTOPEN (baudrate,txbuflen,rxbuflen,stOptions)

stOptions

byVal stOptions AS STRING
This string (can be a constant) MUST be exactly 5 characters long where each character is used to specify further comms parameters as follows.

Character Offset:

0	DTE/DCE role request: ▪ **T** – DTE ▪ **C** – DCE
1	Parity: ▪ **N** – None ▪ **O** – Odd (Not Available) ▪ **E** – Even (Not Available)
2	Databits: 8
3	Stopbits: 1
4	Flow Control: ▪ **N** – None ▪ **H** – CTS/RTS hardware ▪ **X** – Xon/Xof (Not Available)

UartCloseEx

Note: On the BL600 (firmware versions older than 1.3.57.3), the following bug exists:

If the RX and TX buffers are not empty, an internal pointer is still set to NULL. This results in unpredictable behavior.

Workaround: Use UartInfo(6) to check if the buffers are empty and then call UartCloseRx(1) as per the example below.

Workaround for FW **1.3.57.0 and earlier (BL600):**

```
 //Example :: UartCloseExWA.sb (

See in Firmware Zip file)
 DIM rc1
 DIM rc2

 UartClose()
 rc1 = UartOpen(9600,0,0,"CN81H")        //open as DTE at 300 baudrate, odd parity
                                         //8 databits, 1 stopbits, cts/rts flow control
 PRINT "Laird"
```

```
//---Workaround for bug for firmware versions older than 1.3.57.3
IF UartInfo(6)!=0 THEN
    PRINT "\nData in at least one buffer. Uart Port not closed"
ELSE
    rc2=UartCloseEx(1)
    rc1 = UartOpen(9600,0,0,"CN81H")    //open as DTE at 300 baudrate, odd parity
    PRINT "\nThe Uart Port was closed"
ENDIF
```

For FW 1.3.57.3 and newer (BL600):

```
//Example :: UartCloseEx.sb (See in Firmware Zip file)
DIM rc1
DIM rc2

UartClose()
rc1 = UartOpen(9600,0,0,"CN81H")        //open as DTE at 300 baudrate, odd parity
                                        //8 databits, 1 stopbits, cts/rts flow
control
PRINT "Laird"

IF UartCloseEx(1)!=0 THEN
    PRINT "\nData in at least one buffer. Uart Port not closed"
ELSE
    rc1 = UartOpen(9600,0,0,"CN81H")    //open as DTE at 300 baudrate, odd parity
    PRINT "\nUart Port was closed"
ENDIF
```

Expected Output:

```
Laird
Data in at least one buffer. Uart Port not closed
```

UARTCLOSEEX is a core function.

UartSetRTS

The BL600 module does not offer the capability to control the RTS pin because the underlying hardware does not allow it.

UartBREAK

The BL600 module does not offer the capability to send a BREAK signal.

If this feature is required, then the best way to expedite it is to put UART_TX and an I/O pin configured as an output through an AND gate.

For normal operation, the general purpose output pin is set to logic high which means the output of the AND gate follows the state of the UART_TX pin.

When a BREAK is to be sent, the general purpose pin is set to logic high which means the output of the AND gate are low and remain low regardless of the state of the UART_TX pin

I2C – Two Wire Interface (TWI)

The BL600 can only be configured as an I2C master with the additional constraint that it be the only master on the bus and only 7 bit slave addressing is supported.

Note: On the BL600 (firmware releases older than 1.2.54.4), there is an issue where some I2C slaves are not able to drive the ACK down to a low enough voltage level for the module to recognise it as an ACK. This is a result of a bug in the BL600's I2C driver which results in the SDA line not being released by the module. This has been corrected in release 1.2.54.4 and the firmware is available as a UART download on request. You should upgrade the firmware if you have an I2C slave not responding to the correct slave address.

SPI Interface

Note: The BL600 module can only be configured as a SPI master.

4. CORE EXTENSIONS BUILT-IN ROUTINES

Miscellaneous Routines

This section describes all miscellaneous functions and subroutines.

RESET

SUBROUTINE

This routine is used to force a reset of the module.

RESET (nType)

Exceptions	▪ Local Stack Frame Underflow ▪ Local Stack Frame Overflow
Arguments:	
nType	byVal *nType* AS INTEGER. This is for future use. Set to 0.
Interactive Command	NO

```
//Example :: RESET.sb (See in BL600CodeSnippets.zip)
RESET(0)    //force a reset of the module
```

Expected Output:

Like when you reset the module using the interactive command 'ATZ', the CTS indicator will momenterally change from green to red, then back to green.

RESET is a core subroutine.

ERASEFILESYSTEM

FUNCTION

This function is used to erase the flash file system which contains the application that invoked this function, ***if and only if***, the SIO7 input pin is held high.

Given that SIO7 is high, after erasing the file system, the module resets and reboots into command mode with the virtual serial port service enabled; the module advertises for a few seconds. See the virtual serial port service section for more details.

This facility allows the current $autorun$ application to be replaced with a new one.

WARNING
If this function is called from within $autorun$, and the SIO7 input is high, then it will get erased and a fresh download of the application is required which can be facilitated over the air.

ERASEFILESYSTEM (nArg)

Returns	INTEGER Indicates success of command:

	0	Successful erasure. The module reboots.
	<>0	Failure.
Exceptions	▪ Local Stack Frame Underflow ▪ Local Stack Frame Overflow	
Arguments:		
nArg	*byVal nArg AS INTEGER* This is for future use and MUST always be set to 1. Any other value will result in a failure.	

```
//Example :: EraseFileSystem.sb (See in BL600CodeSnippets.zip)
DIM rc
rc = EraseFileSystem(1234)
IF rc!=0 THEN
  PRINT "\nFailed to erase file system because incorrect parameter"
ENDIF
//Input SIO7 is low
rc = EraseFileSystem(1)
IF rc!=0 THEN
  PRINT "\nFailed to erase file system because SIO7 is low"
ENDIF
```

Expected Output:

```
Failed to erase file system because incorrect parameter
Failed to erase file system because SIO7 is low
00
```

ERASEFILESYSTEM is an extension function.

Input/Output Interface Routines

I/O and interface commands allow access to the physical interface pins and ports of the *smart*BASIC modules. Most of these commands are applicable to the range of modules. However, some are dependent on the actual I/O availability of each module.

GPIO Events

EVGPIOCHANn	Here, n is from 0 to N where N is platform dependent and an event is generated when a preconfigured digital input transition occurs. The number of digital inputs that can auto-generate is hardware dependent. For the BL600 module, N can be 0,1,2 or 3. Use GpioBindEvent() to generate these events. See example for GpioBindEvent()
EVDETECTCHANn	Here, n is from 0 to N where N is platform dependent and an event is generated when a preconfigured digital input transition occurs. The number of digital inputs that can auto-generate is hardware dependent. For the BL600 module, N can only be 0. Use GpioAssignEvent() to generate these events. See example for GpioAssignEvent()

GpioSetFunc

FUNCTION

This routine sets the function of the GPIO pin identified by the nSigNum argument.

The module datasheet contains a pinout table which denotes SIO (Special I/O) pins. The number designated for that special I/O pin corresponds to the nSigNum argument.

GPIOSETFUNC (*nSigNum, nFunction, nSubFunc)*

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments	
nSigNum	*byVal nSigNum AS INTEGER* The signal number as stated in the pinout table of the module.
nFunction	*byVal nFunction AS INTEGER* Specifies the configuration of the GPIO pin as follows: 1 – DIGITAL_IN 2 – DIGITAL_OUT 3 – ANALOG_IN 4 – ANALOG_REF (not currently available on the BL600 module) 5 – ANALOG_OUT (not currently available on the BL600 module)
nSubFunc	*byVal nSubFunc INTEGER* Configures the pin as follows: **If nFunction == DIGITAL_IN** Bits 0..3 0x01 – Pull down resistor (weak) 0x02 – Pull up resistor (weak) 0x03 – Pull down resistor (strong) 0x04 – Pull up resistor (strong) Else – No pull resistors Bits 4, 5 0x10 – When in deep sleep mode, awake when this pin is LOW 0x20 – When in deep sleep mode, awake when this pin is HIGH Else – No effect in deep sleep mode Bits 8..31 Must be 0s **If nFuncType == DIGITAL_OUT** Values: 0 – Initial output to LOW 1 – Initial output to HIGH 2 – Output is PWM (Pulse Width Modulated Output). See function GpioConfigPW() for more configuration. The duty cycle is set using function GpioWrite().

	3 — Output is FREQUENCY. The frequency is set using function GpioWrite() where 0 switches off the output; any value in range 1..4000000 generates an output signal with 50% duty cycle with that frequency. Bits 4..6 (output drive capacity) 0 — 0 – Standard 1 – Standard 1 — 0 – High 1 – Standard 2 — 0 – Standard 1 – High 3 — 0 – High 1 – High 4 — 0 – Disconnect 1 – Standard 5 — 0 – Disconnect 1 – High 6 — 0 – Standard 1 – Disconnect 7 — 0 – High 1 – Disconnect **If nFuncType == ANALOG_IN** 0 — Use the system default: 10-bit ADC, 2/3 scaling 0x13 — 10-bit ADC, 1/3 scaling 0x11 — 10-bit ADC, unity scaling
Interactive Command	NO

Note: The internal reference voltage is 1.2V with +/- 1.5% accuracy.

WARNING: This subfunc value is 'global' and once changed will apply to all ADC inputs.

```
//Example :: GpioSetFunc.sb (See in Firmware Zip file)
PRINT GpioSetFunc(3,1,2)     //Digital In Gpio pin 3, weak pull up resistor
PRINT GpioSetFunc(4,3,0)     //Analog In Gpio pin 4, default settings
PRINT GpioSetFunc(5,1,0x12)  //internal pull up on gpio5 and wake from deep sleep
                            //when there is transition from high to low
```

Expected Output:

```
000
```

GPIOSETFUNC is a Module function.

GpioConfigPwm

FUNCTION

This routine configures the PWM (Pulse Width Modulation) of all output pins when they are set as a PWM output using GpioSetFunc() function described above.

Note: This is a 'sticky' configuration; calling it affects all PWM outputs already configured. It is advised that this is called once at the beginning of your application and not changed again within the application unless all PWM outputs are deconfigured and then re-enabled after this function is called.

The PWM output is generated using 32-bit hardware timers. The timers are clocked by a 1 MHz clock source.

A PWM signal has a frequency and a duty cycle property; the frequency is set using this function and is defined by the nMaxResolution parameter. For a given nMaxResolution value, given that the timer is clocked using a 1 MHz source, the frequency of the generated signal is 1000000 divided by nMaxResolution. Hence if nMinFreqHz is more than the 1000000/nMaxResolution, this function will fail with a non-zero value.

The nMaxResolution can also be viewed as defining the resolution of the PWN output in the sense that the duty cycle can be varied from 0 to nMaxResolution. The duty cycle of the PWM signal is modified using the GpioWrite() command

For example, a period of 1000 generates an output frequency of 1KHz, a period of 500, and a frequency of 2Khz etc.

On exit, the function returns with the actual frequency in the nMinFreqHz parameter.

GPIOCONFIGPWM (*nMinFreqHz, nMaxResolution)*

Returns	INTEGER, a result code. **Most typical value:** 0x0000 (indicates a successful operation)
Arguments	
nMinFreqHz	*byRef nMinFreqHz AS INTEGER* On entry this variable contains the minimum frequency desired for the PWM output. On exit, if successful, it contains the actual frequency of the PWM output.
nMaxResolution	*byVal nMaxResolution INTEGER.* This specifies the duty cycle resolution and the value to set to get a 100% duty cycle.
Interactive Command	No

```
// Example :: GpioConfigPWM() (See in Firmware Zip file)
DIM rc
DIM nFreqHz, nMaxRes
// we want a minimum frequency of 500Hz so that we can use a 100Hz low pass filter to
// create an analogue output which has a 100Hz bandwidth
nFreqHz = 500
// we want a resolution of 1:1000 in the generated analogue output
nMaxValUs = 1000

PRINT GpioConfigPWM(nFreqHz,nMaxRes)

PRINT "\nThe actual frequency of the PWM output is ";nFreqHz;"\n"
// now configure SIO2 pin as a PWM output
PRINT GpioSetFunc(2,2,2)   //3rd parameter is subfunc == PWM output
// Set PWM output to 0%
GpioWrite(2,0)
// Set PWM output to 50%
GpioWrite(2,(nMaxRes/2))
// Set PWM output to 100%
```

```
 GpioWrite(2,nMaxRes)         // any value >= nMaxRes will give a 100% duty cycle
 // Set PWM output to 33.333%
GpioWrite(2,(nMaxRes/3))
```

Expected Output:

```
0
The actual frequency of the PWM output is 1000
0
```

GPIOCONFIGPWM is a Module function.

GpioRead

FUNCTION

This routine reads the value from a SIO (special purpose I/O) pin.

The module datasheet will contain a pinout table which will mention SIO (Special I/O) pins and the number designated for that special I/O pin corresponds to the nSigNum argument.

GPIOREAD (*nSigNum)*

Returns	INTEGER, the value from the signal. If the signal number is invalid, it returns the value 0. For digital pins, the value is 0 or 1. For ADC pins it is a value in the range of 0 to M where M is the maximum based on the bit resolution of the analogue to digital converter.
Arguments	
nSigNum	byVal nSigNum INTEGER The signal number as stated in the pinout table of the module.
Interactive Command	No

```
//Example :: GpioRead.sb (See in Firmware Zip file)
DIM signal
signal = GpioRead(3)
PRINT signal
```

Expected Output:

```
1
```

GPIOREAD is a Module function.

GpioWrite

SUBROUTINE

This routine writes a new value to the GPIO pin. If the pin number is invalid, nothing happens.

If the GPIO pin has been configured as a PWM output then the nNewValue specifies a value in the range 0 to N where N is the maximum PWM value that generates a 100% duty cycle output (a constant high signal) and N is a value that is configured using the function GpioConfigPWM().

If the GPIO pin has been configured as a FREQUENCY output then the nNewValue specifies the desired frequency in Hertz in the range 0 to 4000000. Setting a value of 0 makes the output a constant low value. Setting a value greater than 4000000 clips the output to a 4 MHz signal.

GPIOWRITE (*nSigNum, nNewValue)*

Arguments	
nSigNum	byVal nSigNum INTEGER. The signal number as stated in the pinout table of the module.
nNewValue	byVal nNewValue INTEGER. The value to be written to the port. If the pin is configured as digital then 0 will clear the pin and a non-zero value will set it. If the pin is configured as analogue, then the value is written to the pin. If the pin is configured as a PWM, then this value sets the duty cycle. If the pin is configured as a FREQUENCY, then this value sets the frequency.
Interactive Command	No

```
//Example :: GpioWrite.sb (See in Firmware Zip file)
DIM rc,dutycycle,freqHz,minFreq
//set sio pin 1 to an output and initialise it to high
PRINT GpioSetFunc(1,2,0);"\n"
//set sio pin 5 to PWM output
minFreq = 500
PRINT GpioConfigPWM(minFreq,1024);"\n"   //set max pwm value/resolution to 1:1024
PRINT GpioSetFunc(5,2,2);"\n"
PRINT GpioSetFunc(7,2,3);"\n\n"          //set sio pin 7 to Frequency output

GpioWrite(18,0)                          //set pin 1 to low
GpioWrite(18,1)                          //set pin 1 to high
//Set the PWM output to 25%
GpioWrite(5,256)   //256 = 1024/4
//Set the FREQ output to 4.236 Khz
GpioWrite(7,4236)

//Note you can generate a chirp output on sio 7 by starting a timer which expires
//every 100ms and then in the timer handler call GpioWrite(7,xx) and then
//increment xx by a certain value
```

Expected Output:

```
0000
```

GPIOWRITE is a Module function.

GpioBindEvent

FUNCTION

This routine binds an event to a level transition on a specified special I/O line configured as a digital input so that changes in the input line can invoke a handler in *smart* BASIC user code.

Note: In the BL600 module, using this function results in over 1 mA of continuous current consumption from the power supply. If power is important, use GpioAssignEvent() instead which uses other resources to expedite an event.

GPIOBINDEVENT (*nEventNum, nSigNum, nPolarity)*

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
nEventNum	**byVal nEventNum INTEGER** The GPIO event number (in the range of 0 - N) which will result in the event EVGPIOCHANn being thrown to the smart BASIC runtime engine.
nSigNum	**byVal nSigNum INTEGER** The signal number as stated in the pinout table of the module.
nPolarity	**byVal *nPolarity* INTEGER** States the transition as follows: **0** – Low to high transition **1** – High to low transition **2** – Either a low to high or high to low transition
Interactive Command	No

```
//Example :: GpioBindEvent.sb (See in Firmware Zip file)
FUNCTION Btn0Press()
    PRINT "\nHello"
ENDFUNC 0

PRINT GpioBindEvent(0,16,1)          //Bind event 0 to high low transition on sio16
(button0)
ONEVENT EVGPIOCHAN0 CALL Btn0Press  //When event 0 happens, call Btn0Press

PRINT "\nPress button 0"

WAITEVENT
```

Expected Output:

```
0
Press button 0
Hello
```

GPIOBINDEVENT is a Module function.

GpioUnbindEvent

FUNCTION

This routine unbinds the runtime engine event from a level transition bound using GpioBindEvent().

GPIOUNBINDEVENT (*nEventNum)*

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)

Arguments	
nEventNum	byVal nEventNum INTEGER. The GPIO event number (in the range of 0 - N) which is disabled so that it no longer generates run-time events in smart BASIC.
Interactive Command	No

```
//Example :: GpioUnbindEvent.sb (See in Firmware Zip file)
FUNCTION Btn0Press()
    PRINT "\nHello"
ENDFUNC 1

FUNCTION Tmr0TimedOut()
    PRINT "\nNothing happened"
ENDFUNC 0

PRINT GpioBindEvent(0,16,1);"\n"

ONEVENT EVGPIOCHAN0 CALL Btn0Press
ONEVENT EVTMR0      CALL Tmr0TimedOut

PRINT GpioUnbindEvent(0);"\n"
PRINT "\nPress button 0\n"
TimerStart(0,8000,0)

WAITEVENT
```

Expected Output:

```
0
0

Press button 0

Nothing happened
```

GPIOUNBINDEVENT is a Module function.

GpioAssignEvent

FUNCTION

This routine assigns an event to a level transition on a specified special I/O line configured as a digital input. Changes in the input line can invoke a handler in *smart* BASIC user code

Note: In the BL600, this function results in around 4uA of continuous current consumption from the power supply. It is impossible to assign a polarity value which detects either level transitions.

GPIOASSIGNEVENT (*nEventNum, nSigNum, nPolarity)*

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
nEventNum	byVal nEventNum INTEGER. The GPIO event number (in the range of 0 - N) which results in the event EVDETECTCHANn

	being thrown to the smart BASIC runtime engine. **Note:** A value of 0 is only valid for the BL600.
nSigNum	**byVal nSigNum INTEGER.** The signal number as stated in the pinout table of the module.
nPolarity	**byVal *nPolarity* INTEGER.** States the transition as follows: **0** – Low to high transition **1** – High to low transition **2** – Either a low to high or high to low transition **Note:** This is not available in the BL600 module.
Interactive Command	No

```
//Example :: GpioAssignEvent.sb (See in Firmware Zip file)
FUNCTION Btn0Press()
    PRINT "\nHello"
ENDFUNC 0

PRINT GpioAssignEvent(0,16,1)           //Assign event 0 to high low transition on
sio16 (button0)
ONEVENT EVDETECTCHAN0 CALL Btn0Press    //When event 0 is detected, call Btn0Press

PRINT "\nPress button 0"

WAITEVENT
```

Expected Output:

```
0
Press button 0
Hello
```

GPIOASSIGNEVENT is a Module function.

GpioUnAssignEvent

FUNCTION

This routine unassigns the runtime engine event from a level transition assigned using GpioAssignEvent().

GPIOUNASSIGNEVENT (*nEventNum)*

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
nEventNum	**byVal nEventNum INTEGER.** The GPIO event number (in the range of 0 - N) which is disabled so that it no longer generates run-time events in smart BASIC. **Note:** A value of 0 is only valid for the BL600.
Interactive Command	No

```
//Example :: GpioUnAssignEvent.sb (See in Firmware Zip file)
FUNCTION Btn0Press()
    PRINT "\nHello"
ENDFUNC 1

FUNCTION Tmr0TimedOut()
    PRINT "\nNothing happened"
ENDFUNC 0

PRINT GpioAssignEvent(0,16,1);"\n"

ONEVENT EVDETECTCHAN0 CALL Btn0Press
ONEVENT EVTMR0      CALL Tmr0TimedOut

PRINT GpioUnAssignEvent(0);"\n"
PRINT "\nPress button 0\n"
TimerStart(0,8000,0)
WAITEVENT
```

Expected Output:

```
0
0

Press button 0

Nothing happened
```

GPIOUNASSIGNEVENT is a Module function.

5. BLE EXTENSIONS BUILT-IN ROUTINES

MAC Address

To address privacy concerns there are four types of MAC addresses in a BLE device which can change as often as required. For example, an iPhone regularly changes its BLE MAC address and always exposes only its resolvable random address.

To manage this, the usual six octet MAC address is qualified on-air by a single bit which qualifies the MAC address as public or random. If public, the format is as defined by the IEEE organization. If random, it can be up to three types and this qualification is done using the upper two bits of the most significant byte of the random MAC address. The exact details and format of how the specification requires this to be managed are not relevant for the purpose of how BLE functionality is exposed in this module and only how various API functions in smartBASIC expect MAC addresses to be provided is detailed here.

Where a MAC address is expected as a parameter (or provided as a response) it is always a STRING variable. This variable is seven octets long where the first octet is the address type and the the other six octets comprises the usual MAC address in big endian format (so that most significant octet of the address is at offset 1), whether public or random.

The address type is:

0	Public
1	Random Static
2	Random Private Resolvable
3	Randam Private Non-resolvable
All other values are illegal	

For example:

To specify a public address which has the MAC potion as 112233445566, the STRING variable will contain seven octets (00112233445566) and a variable can be initialized using a constant string by escaping as follows:

DIM addr : addr="\00\11\22\33\44\55\66". Likewise a static random address will be 01C12233445566 (upper 2 bits of MAC portion == 11), a resolvable random address will be 02412233445566 (upper 2 bits of MAC portion ==01) and a non-resolvable address will be 03112233445566 (upper 2 bits of MAC portion ==00).

Note: The MAC address portion in smartBASIC is always in big endian format. If you sniff on-air packets, the same six packets appear little endian format, hence reverse order – and you will NOT see seven bytes, but a bit in the packet somewhere which specifies it to be public or random.

Events and Messages

EVBLE_ADV_TIMEOUT

This event is thrown when adverts that are started using BleAdvertStart() time out. Usage is as per the example below.

```
//Example :: EvBle_Adv_Timeout.sb (See in BL600CodeSnippets.zip)
DIM peerAddr$

//handler to service an advert timeout
FUNCTION HndlrBleAdvTimOut()
    PRINT "\nAdvert stopped via timeout"
    //DbgMsg( "\n   - could use SystemStateSet(0) to switch off" )

    //-------------------------------------------------------------
    //  Switch off the system - requires a power cycle to recover
    //-------------------------------------------------------------
    //  rc = SystemStateSet(0)
ENDFUNC 0

//start adverts
//rc = BleAdvertStart(0,"",100,5000,0)
IF BleAdvertStart(0,peerAddr$,100,2000,0)==0 THEN
    PRINT "\nAdvertisement Successful"
ELSE
    PRINT "\n\nAdvertisement not successful"
ENDIF

ONEVENT  EVBLE_ADV_TIMEOUT  CALL   HndlrBleAdvTimOut

WAITEVENT
```

Expected Output:

```
Advert Started
Advert stopped via timeout
```

EVBLEMSG

The BLE subsystem is capable of informing a *smart* BASIC application when a significant BLE-related event has occurred. It does this by throwing this message (as opposed to an EVENT, which is akin to an interrupt and has no context or queue associated with it). The message contains two parameters. The first parameter (subsequently called **msgID**) identifies what event was triggered; the second parameter (subsequently called **msgCtx**) conveys some context data associated with that event. The *smart* BASIC application must register a handler function which takes two integer arguments to be able to receive and process this message.

Note: The messaging subsystem, unlike the event subsystem, has a queue associated with it; unless the queue is full, all of the messages remain pending until they are handled. Only messages that have handlers associated with them get inserted into the queue. This is to prevent unhandled messages from filling that queue. The following table provides a list of triggers and associated context parameter.

MsgID	Description
0	A connection has been established and msgCtx is the connection handle.
1	A disconnection event and msgCtx identifies the handle.
2	Immediate Alert Service Alert. The 2nd parameter contains new alert level.
3	Link Loss Alert. The 2nd parameter contains new alert level.
4	A BLE Service Error. The 2nd parameter contains the error code.
5	Thermometer Client Characteristic Descriptor value has changed. Indication enable state and msgCtx contains the new value: 0 – Disabled 1 – Enabled
6	Thermometer measurement indication has been acknowledged.
7	Blood Pressure Client Characteristic Descriptor value has changed. Indication enable state and msgCtx contains the new value: 0 – Disabled 1 – Enabled
8	Blood Pressure measurement indication has been acknowledged.
9	Pairing in progress and display passkey supplied in msgCtx.
10	A new bond has been successfully created.
11	Pairing in progress and authentication key requested. Key type is msgCtx is.
12	Heart Rate Client Characteristic Descriptor value has changed. Notification enable state and msgCtx contains the new value: 0 – Disabled 1 – Enabled
14	Connection parameters update and msgCtx is the conn handle.
15	Connection parameters update fail and msgCtx is the conn handle.
16	Connected to a bonded master and msgCtx is the conn handle.
17	A new pairing has replaced old key for the connection handle specified.
18	The connection is now encrypted and msgCtx is the conn handle.
19	The supply voltage has dropped below that specified in the most recent call of SetPwrSupplyThreshMv(i) and msgCtx is the current voltage in milliVolts.
20	The connection is no longer encrypted and msgCtx is the conn handle
21	The device name characteristic in the GAP service of the local gatt table has been written by the remote gatt client.

Note: Message ID 13 is reserved for future use

An example of how these messages can be used is as follows:

```
//Example :: EvBleMsg.sb (See in BL600CodeSnippets.zip)
DIM addr$ : addr$=""
DIM rc

//============================================================================
// This handler is called when there is a BLE message
```

```
//===============================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId AS INTEGER, BYVAL nCtx AS INTEGER)
    SELECT nMsgId
        CASE 0
            PRINT "\nBle Connection ";nCtx
            rc = BleAuthenticate(nCtx)
        CASE 1
            PRINT "\nDisconnected ";nCtx;"\n"
        CASE 18
            PRINT "\nConnection ";nCtx;" is now encrypted"
        CASE 16
            PRINT "\nConnected to a bonded master"
        CASE 17
            PRINT "\nA new pairing has replaced the old key";
        CASE ELSE
            PRINT "\nUnknown Ble Msg"
    ENDSELECT
ENDFUNC 1

FUNCTION HndlrBlrAdvTimOut()
   PRINT "\nAdvert stopped via timeout"
   PRINT "\nExiting..."
ENDFUNC 0

FUNCTION Btn0Press()
    PRINT "\nExiting..."
ENDFUNC 0

PRINT GpioSetFunc(16,1,0x12)
PRINT GpioBindEvent(0,16,0)

ONEVENT  EVBLEMSG            CALL HndlrBleMsg
ONEVENT  EVBLE_ADV_TIMEOUT   CALL HndlrBlrAdvTimOut
ONEVENT  EVGPIOCHAN0         CALL Btn0Press

// start adverts
IF BleAdvertStart(0,addr$,100,10000,0)==0 THEN
    PRINT "\nAdverts Started"
    PRINT "\nPress button 0 to exit\n"
ELSE
    PRINT "\n\nAdvertisement not successful"
ENDIF

WAITEVENT
```

Expected Output (When connection made with the module):

```
Adverts Started
Press button 0 to exit

BLE Connection 3634
Connected to a bonded master
Connection 3634 is now encrypted
A new pairing has replaced the old key
Disconnected 3634

Exiting...
```

Expected Output (When no connection made):

```
Adverts Started
Press button 0 to exit

Advert stopped via timeout
Exiting...
```

EVDISCON

This event is thrown when there is a disconnection. It comes with two parameters:

- Parameter 1 – Connection handle
- Parameter 2 – Reason for the disconnection The reason, for example, can be 0x08 which signifies a link connection supervision timeout which is used in the Proximity Profile.

A full list of Bluetooth HCI result codes for the 'reason of disconnection' can be determined and are provided here.

```
//Example :: EvDiscon.sb (See in BL600CodeSnippets.zip)
DIM addr$ : addr$=""

FUNCTION HndlrBleMsg(BYVAL nMsgId AS INTEGER, BYVAL nCtx AS INTEGER)
    IF nMsgID==0 THEN
        PRINT "\nNew Connection ";nCtx
    ENDIF
ENDFUNC 1

FUNCTION Btn0Press()
    PRINT "\nExiting..."
ENDFUNC 0

FUNCTION HndlrDiscon(BYVAL hConn AS INTEGER, BYVAL nRsn AS INTEGER) AS INTEGER
    PRINT "\nConnection ";hConn;" Closed: 0x";nRsn
ENDFUNC 0

ONEVENT EVBLEMSG    CALL HndlrBleMsg
ONEVENT EVDISCON    CALL HndlrDiscon

// start adverts
IF BleAdvertStart(0,addr$,100,10000,0)==0 THEN
    PRINT "\nAdverts Started\n"
ELSE
    PRINT "\n\nAdvertisement not successful"
```

```
ENDIF

WAITEVENT
```

Expected Output:

```
Adverts Started

New Connection 2915
Connection 2915 Closed: 0x19
```

EVCHARVAL

This event is thrown when a characteristic has been written to by a remote GATT client. It comes with three parameters:

- 1 - The characteristic handle that was returned when the characteristic was registered using the function BleCharCommit()
- 2 – The Offset
- 3 – The Length of the data from the characteristic value

```
//Example :: EvCharVal.sb (See in BL600CodeSnippets.zip)
DIM hMyChar,rc,at$,conHndl

//===============================================================================
// Initialise and instantiate service, characteristic, start adverts
//===============================================================================
FUNCTION OnStartup()
    DIM rc, hSvc, attr$, adRpt$, addr$, scRpt$ : attr$="Hi"

    //commit service
    rc=BleSvcCommit(1,BleHandleUuid16(0x18EE),hSvc)
    rc=BleServiceNew(1, BleHandleUuid16(0x18EE), hSvc)
    //initialise char, write/read enabled, accept signed writes
    rc=BleCharNew(0x0A,BleHandleUuid16(1),BleAttrMetaData(1,1,20,0,rc),0,0)
    //commit char initialised above, with initial value "hi" to service 'hSvc'
    rc=BleCharCommit(hSvc,attr$,hMyChar)
    //commit changes to service
    rc=BleServiceCommit(hSvc)
    rc=BleScanRptInit(scRpt$)
    //Add 1 service handle to scan report
    //rc=BleAdvRptAddUuid16(scRpt$,hSvc,-1,-1,-1,-1,-1)
    //commit reports to GATT table - adRpt$ is empty
    rc=BleAdvRptsCommit(adRpt$,scRpt$)
    rc=BleAdvertStart(0,addr$,20,300000,0)
ENDFUNC rc

//===============================================================================
// Close connections so that we can run another app without problems
//===============================================================================
SUB CloseConnections()
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
ENDSUB

//===============================================================================
// Ble event handler
```

```
//=============================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n--- Disconnected from client"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n--- Connected to client"
    ENDIF
ENDFUNC 1

//=============================================================================
// New char value handler
//=============================================================================
FUNCTION HandlerCharVal(BYVAL charHandle, BYVAL offset, BYVAL len)
    DIM s$
    IF charHandle == hMyChar THEN
        PRINT "\n";len;" byte(s) have been written to char value attribute from
offset ";offset

        rc=BleCharValueRead(hMyChar,s$)
        PRINT "\nNew Char Value: ";s$
    ENDIF
    CloseConnections()
ENDFUNC 1

ONEVENT  EVCHARVAL CALL HandlerCharVal
ONEVENT  EVBLEMSG  CALL HndlrBleMsg

IF OnStartup()==0 THEN
    rc = BleCharValueRead(hMyChar,at$)
    PRINT "\nThe characteristic's value is ";at$
    PRINT "\nWrite a new value to the characteristic\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF

WAITEVENT

PRINT "\nExiting..."
```

Expected Output:

```
The characteristic's value is Hi
Write a new value to the characteristic

--- Connected to client
5 byte(s) have been written to char value attribute from offset 0
New Char Value: Hello

--- Disconnected from client
Exiting...
```

EVCHARHVC

This event is thrown when a value sent via an indication to a client gets acknowledged. It comes with one parameter – the characteristic handle that was returned when the characteristic was registered using the function BleCharCommit().

```
// Example :: EVCHARHVC charHandle

// See example that is provided for EVCHARCCCD
```

EVCHARCCCD

This event is thrown when the client writes to the CCCD descriptor of a characteristic. It comes with two parameters:

- 1 – The characteristic handle returned when the characteristic was registered with BleCharCommit()
- 2 – The 16-bit value in the updated CCCD attribute.

```
//Example :: EvCharCccd.sb (See in BL600CodeSnippets.zip)
DIM hMyChar,rc,at$,conHndl

//============================================================================
// Initialise and instantiate service, characteristic, start adverts
//============================================================================
FUNCTION OnStartup()
    DIM rc, hSvc, metaSuccess, at$, attr$, adRpt$, addr$, scRpt$
    attr$="Hi"
    DIM svcUuid : svcUuid=0x18EE
    DIM charUuid : charUuid = BleHandleUuid16(1)
    DIM charMet : charMet = BleAttrMetaData(0,0,20,1,metaSuccess)
    DIM hSvcUuid : hSvcUuid = BleHandleUuid16(svcUuid)
    DIM mdCccd : mdCccd = BleAttrMetadata(1,1,2,0,rc)    //CCCD metadata for char

    //Create service
    rc=BleServiceNew(1,hSvcUuid,hSvc)

    //initialise char, write/read enabled, accept signed writes, indicatable
    rc=BleCharNew(0x20,charUuid,charMet,mdCccd,0)

    //commit char initialised above, with initial value "hi" to service 'hMyChar'
    rc=BleCharCommit(hSvc,attr$,hMyChar)

    //commit service to GATT table
    rc=BleServiceCommit(hSvc)

    rc=BleAdvertStart(0,addr$,20,300000,0)
ENDFUNC rc

//============================================================================
// Close connections so that we can run another app without problems
//============================================================================
SUB CloseConnections()
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
    rc=GpioUnbindEvent(1)
ENDSUB

//============================================================================
// Ble event handler
//============================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n--- Disconnected from client"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
```

```
        PRINT "\n--- Connected to client"
    ENDIF
ENDFUNC 1

//================================================================================
// Indication acknowledgement from client handler
//================================================================================
FUNCTION HndlrCharHvc(BYVAL charHandle AS INTEGER) AS INTEGER
    IF charHandle == hMyChar THEN
        PRINT "\nGot confirmation of recent indication"
    ELSE
        PRINT "\nGot confirmation of some other indication: ";charHandle
    ENDIF
ENDFUNC 1

//================================================================================
// Called when data received via the UART
//================================================================================
FUNCTION HndlrUartRx() AS INTEGER
ENDFUNC 0

//================================================================================
// CCCD descriptor written handler
//================================================================================
FUNCTION HndlrCharCccd(BYVAL charHandle, BYVAL nVal) AS INTEGER
    DIM value$
    IF charHandle==hMyChar THEN
        IF nVal & 0x02 THEN
            PRINT "\nIndications have been enabled by client"
            value$="hello"
            IF BleCharValueIndicate(hMyChar,value$)!=0 THEN
                PRINT "\nFailed to indicate new value"
            ENDIF
        ELSE
            PRINT "\nIndications have been disabled by client"
        ENDIF
    ELSE
        PRINT "\nThis is for some other characteristic"
    ENDIF
ENDFUNC 1

ONEVENT  EVBLEMSG    CALL HndlrBleMsg
ONEVENT  EVCHARHVC   CALL HndlrCharHvc
ONEVENT  EVCHARCCCD  CALL HndlrCharCccd
ONEVENT  EVUARTRX    CALL HndlrUartRx

IF OnStartup()==0 THEN
    rc = BleCharValueRead(hMyChar,at$)
    PRINT "\nValue of the characteristic ";hMyChar;" is: ";at$
    PRINT "\nYou can write to the CCCD characteristic."
    PRINT "\nThe BL600 will then indicate a new characteristic value\n"
    PRINT "\n--- Press any key to exit"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF

WAITEVENT

CloseConnections()

PRINT "\nExiting..."
```

Expected Output:

```
Value of the characteristic 1346437121 is: Hi
You can write to the CCCD characteristic.
The BL600 will then indicate a new characteristic value

--- Press any key to exit
--- Connected to client
Indications have been enabled by client
Got confirmation of recent indication
Exiting...
```

EVCHARSCCD

This event is thrown when the client writes to the SCCD descriptor of a characteristic. It comes with two parameters:

- 1 – The characteristic handle that was returned when the characteristic was registered using the function BleCharCommit()
- 2 – The new 16-bit value in the updated SCCD attribute.

The SCCD is used to manage broadcasts of characteristic values.

```
//Example :: EvCharSccd.sb (See in BL600CodeSnippets.zip)
DIM hMyChar,rc,chVal$,conHndl

//============================================================================
// Initialise and instantiate service, characteristic, start adverts
//============================================================================
FUNCTION OnStartup()
    DIM rc, hSvc, attr$, adRpt$, addr$, scRpt$  ,rc2
    attr$="Hi"
    DIM charMet : charMet = BleAttrMetaData(1,1,20,1,rc)

    //Create service
    rc=BleServiceNew(1,BleHandleUuid16(0x18EE),hSvc)

    //initialise broadcast capable, readable, writeable
    rc=BleCharNew(0x0B,BleHandleUuid16(1),charMet,0,BleAttrMetadata(1,1,1,0,rc2))

    //commit char initialised above, with initial value "hi" to service 'hMyChar'
    rc=BleCharCommit(hSvc,attr$,hMyChar)

    //commit service to GATT table
    rc=BleServiceCommit(hSvc)

    rc=BleAdvertStart(0,addr$,20,300000,0)
ENDFUNC rc

//============================================================================
// Close connections so that we can run another app without problems
//============================================================================
SUB CloseConnections()
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
    rc=GpioUnbindEvent(1)
ENDSUB
```

```
//================================================================================
// Broadcast characterstic value
//================================================================================
FUNCTION PrepAdvReport()
    dim adRpt$, scRpt$, svcDta$

    //initialise new advert report
    rc=BleAdvRptinit(adRpt$, 2, 0, 0)

    //encode service UUID into service data string
    rc=BleEncode16(svcDta$, 0x18EE, 0)

    //append characteristic value
    svcDta$ = svcDta$ + chVal$

    //append service data to advert report
    rc=BleAdvRptAppendAD(adRpt$, 0x16, svcDta$)

    //commit new advert report, and empty scan report
    rc=BleAdvRptsCommit(adRpt$, scRpt$)
ENDFUNC rc

//================================================================================
// Reset advert report
//================================================================================
FUNCTION ResetAdvReport()
    dim adRpt$, scRpt$

    //initialise new advert report
    rc=BleAdvRptinit(adRpt$, 2, 0, 20)

    //commit new advert report, and empty scan report
    rc=BleAdvRptsCommit(adRpt$, scRpt$)
ENDFUNC rc

//================================================================================
// Ble event handler
//================================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n--- Disconnected from client"
        dim addr$
        rc=BleAdvertStart(0,addr$,20,300000,0)
        IF rc==0 THEN
            PRINT "\nYou should now see the new characteristic value in the
advertisment data"
        ENDIF
    ELSEIF nMsgID==0 THEN
        PRINT "\n--- Connected to client"
    ENDIF
ENDFUNC 1

//================================================================================
// Called when data arrives via UART
//================================================================================
FUNCTION HndlrUartRx()
ENDFUNC 0

//================================================================================
// CCCD descriptor written handler
//================================================================================
```

```
FUNCTION HndlrCharSccd(BYVAL charHandle, BYVAL nVal) AS INTEGER
    DIM value$
    IF charHandle==hMyChar THEN
        IF nVal & 0x01 THEN
            PRINT "\nBroadcasts have been enabled by client"
            IF PrepAdvReport()==0 THEN
                rc=BleDisconnect(conHndl)
                PRINT "\nDisconnecting..."
            ELSE
                PRINT "\nError Committing advert reports: ";integer.h'rc
            ENDIF
        ELSE
            PRINT "\nBroadcasts have been disabled by client"
            IF ResetAdvReport()==0 THEN
                PRINT "\nAdvert reports reset"
            ELSE
                PRINT "\nError Resetting advert reports: ";integer.h'rc
            ENDIF
        ENDIF
    ELSE
        PRINT "\nThis is for some other characteristic"
    ENDIF
ENDFUNC 1

//================================================================================
// New char value handler
//================================================================================
FUNCTION HndlrCharVal(BYVAL charHandle, BYVAL offset, BYVAL len)
    DIM s$
    IF charHandle == hMyChar THEN
        rc=BleCharValueRead(hMyChar,chVal$)
        PRINT "\nNew Char Value: ";chVal$
    ENDIF
ENDFUNC 1

//================================================================================
// Called after a disconnection
//================================================================================
FUNCTION HndlrDiscon(hConn, nRsn)
    dim addr$
    rc=BleAdvertStart(0,addr$,20,300000,0)
ENDFUNC 1

ONEVENT  EVBLEMSG    CALL HndlrBleMsg
ONEVENT  EVCHARSCCD  CALL HndlrCharSccd
ONEVENT  EVUARTRX    CALL HndlrUartRx
ONEVENT  EVCHARVAL   CALL HndlrCharVal
ONEVENT  EVDISCON    CALL HndlrDiscon

IF OnStartup()==0 THEN
    rc = BleCharValueRead(hMyChar,chVal$)
    PRINT "\nCharacteristic Value: ";chVal$
    PRINT "\nWrite a new value to the characteristic, then enable broadcasting.\nThe
module will then disconnect and broadcast the new characteristic value."
    PRINT "\n--- Press any key to exit\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF

WAITEVENT

CloseConnections()
```

```
PRINT "\nExiting..."
```

Expected Output:

```
Characteristic Value: Hi
Write a new value to the characteristic, then enable broadcasting.
The module will then disconnect and broadcast the new characteristic value.
--- Press any key to exit

--- Connected to client
New Char Value: hello
Broadcasts have been enabled by client
Disconnecting...

--- Disconnected from client
You should now see the new characteristic value in the advertisment data
Exiting...
```

EVCHARDESC

This event is thrown when the client writes to writable descriptor of a characteristic which is not a CCCD or SCCD as they are catered for with their own dedicated messages. It comes with two parameters:

- 1 – The characteristic handle that was returned when the characteristic was registered using the function BleCharCommit()
- 2 – An index into an opaque array of handles managed inside the characteristic handle.

Both parameters are supplied as-is as the first two parameters to the function BleCharDescRead().

```
//Example :: EvCharDesc.sb (See in BL600CodeSnippets.zip)
DIM hMyChar,rc,at$,conHndl, hOtherDescr

//===============================================================================
// Initialise and instantiate service, characteristic, start adverts
//===============================================================================
FUNCTION OnStartup$()
    DIM rc, hSvc, at$, adRpt$, addr$, scRpt$, hOtherDscr,attr$, attr2$, rc2
    attr$="Hi"
    DIM charMet : charMet = BleAttrMetaData(1,0,20,0,rc)

    //Commit svc with handle 'hSvcUuid'
    rc=BleServiceNew(1,BleHandleUuid16(0x18EE),hSvc)
    //initialise characteristic - readable
    rc=BleCharNew(0x02,BleHandleUuid16(1),charMet,0,0)

    //Add user descriptor - variable length
    attr$="my char desc"
    rc=BleCharDescUserDesc(attr$,BleAttrMetadata(1,1,20,1,rc2))
    AssertRC(rc2,20)
    AssertRC(rc,33)

    //commit char initialised above, with initial value "char value" to service
'hSvc'
    attr2$="char value"
    rc=BleCharCommit(hSvc,attr2$,hMyChar)
```

```
    //commit service to GATT table
    rc=BleServiceCommit(hSvc)

    rc=BleAdvertStart(0,addr$,20,300000,0)
ENDFUNC attr$

//================================================================================
// Close connections so that we can run another app without problems
//================================================================================
SUB CloseConnections()
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
    rc=GpioUnbindEvent(1)
ENDSUB

//================================================================================
// Ble event handler
//================================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n--- Disconnected from client"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n--- Connected to client"
    ENDIF
ENDFUNC 1

//================================================================================
// Called when data arrives via UART
//================================================================================
FUNCTION HndlrUartRx()
ENDFUNC 0

//================================================================================
// Client has written to writeable descriptor
//================================================================================
FUNCTION HndlrCharDesc(BYVAL hChar AS INTEGER, BYVAL hDesc AS INTEGER) AS INTEGER
    dim duid,a$,rc
    IF hChar == hMyChar THEN
        rc = BleCharDescRead(hChar,hDesc,0,20,duid,a$)
        IF rc ==0 THEN
            PRINT "\nNew value for desriptor ";hDesc;" with uuid ";integer.h'duid;"
is ";a$
        ELSE
            PRINT "\nCould not read the descriptor value"
        ENDIF
    ELSE
        PRINT "\nThis is for some other characteristic"
    ENDIF
ENDFUNC 1

ONEVENT  EVBLEMSG    CALL HndlrBleMsg
ONEVENT  EVCHARDESC  CALL HndlrCharDesc
ONEVENT  EVUARTRX    CALL HndlrUartRx

PRINT "\nOther Descriptor Value: ";OnStartup$()
PRINT "\nWrite a new value \n--- Press any key to exit\n"

WAITEVENT

CloseConnections()
```

```
PRINT "\nExiting..."
```

Expected Output:

```
Other Descriptor Value: my char desc
Write a new value
--- Press any key to exit

--- Connected to client
New value for desriptor 0 with uuid FE012901 is hello
```

EVVSPRX

This event is thrown when the Virtual Serial Port service is open and data has arrived from the peer.

EVVSPTXEMPTY

This event is thrown when the Virtual Serial Port service is open and the last block of data in the transmit buffer is sent via a notify or indicate. See VSP (Virtual Serial Port) Events

EVNOTIFYBUF

When in a connection and attribute data is sent to the GATT Client using a notify procedure (for example using the function BleCharValueNotify( ) or when a Write_with_no_response is sent by the Gatt Client to a remote server, they are stored in temporary buffers in the underlying stack. There is finite number of these temporary buffers and, if they are exhausted, the notify function or the write_with_no_resp command fails with a result code of 0x6803 (BLE_NO_TX_BUFFERS). Once the attribute data is transmitted over the air, given there are no acknowledges for Notify messages, the buffer is freed to be reused.

This event is thrown when at least one buffer has been freed and so the *smart*BASIC application can handle this event to retrigger the data pump for sending data using notifies or writes_with_no_resp commands.

Note: When sending data using Indications, this event is not thrown because those messages have to be confirmed by the client which results in a EVCHARHVC message to the *smart*BASIC application. Likewise, writes which are acknowledged also do not consume these buffers.

```
//Example :: EvNotifyBuf.sb (See in BL600CodeSnippets.zip)
DIM hMyChar,rc,at$,conHndl,ntfyEnabled

//============================================================================
// Initialise and instantiate service, characteristic, start adverts
//============================================================================
FUNCTION OnStartup()
    DIM rc, hSvc, at$, attr$, adRpt$, addr$, scRpt$
    attr$="Hi"
    DIM mdCccd : mdCccd = BleAttrMetadata(1,1,2,0,rc)    //CCCD metadata for char

    //Commit svc with handle 'hSvc'
    rc=BleServiceNew(1, BleHandleUuid16(0x18EE), hSvc)
    rc=BleSvcCommit(1,BleHandleUuid16(0x18EE),hSvc)
    //initialise char, write/read enabled, accept signed writes, notifiable
    rc=BleCharNew(0x12,BleHandleUuid16(1),BleAttrMetaData(1,0,20,0,rc),mdCccd,0)
    //commit char initialised above, with initial value "hi" to service 'hMyChar'
    rc=BleCharCommit(hSvc,attr$,hMyChar)
    //commit changes to service
```

```
    rc=BleServiceCommit(hSvc)
    rc=BleScanRptInit(scRpt$)
    //Add 1 service handle to scan report
    rc=BleAdvRptAddUuid16(scRpt$,hSvc,-1,-1,-1,-1,-1)
    //commit reports to GATT table - adRpt$ is empty
    rc=BleAdvRptsCommit(adRpt$,scRpt$)
    rc=BleAdvertStart(0,addr$,50,0,0)
ENDFUNC rc

//============================================================================
// Close connections so that we can run another app without problems
//============================================================================
SUB CloseConnections()
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
ENDSUB

SUB SendData()
    DIM tx$, count
    IF ntfyEnabled then
        PRINT "\n--- Notifying"
        DO
            tx$="SomeData"
            rc=BleCharValueNotify(hMyChar,tx$)
            count=count+1
        UNTIL rc!=0
        PRINT "\n--- Buffer full"
        PRINT "\nNotified ";count;" times"
    ENDIF
ENDSUB

//============================================================================
// Ble event handler
//============================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    conHndl=nCtx
    IF nMsgID==0 THEN
        PRINT "\n--- Connected to client"
    ELSEIF nMsgID THEN
        PRINT "\n--- Disconnected from client"
        EXITFUNC 0
    ENDIF
ENDFUNC 1

//============================================================================
// Tx Buffer free handler
//============================================================================
FUNCTION HndlrNtfyBuf()
    SendData()
ENDFUNC 0

//============================================================================
// CCCD descriptor written handler
//============================================================================
FUNCTION HndlrCharCccd(BYVAL charHandle, BYVAL nVal) AS INTEGER
    DIM value$,tx$
    IF charHandle==hMyChar THEN
        IF nVal THEN
            PRINT " : Notifications have been enabled by client"
            ntfyEnabled=1
            tx$="Hello"
            rc=BleCharValueNotify(hMyChar,tx$)
```

```
        ELSE
            PRINT "\nNotifications have been disabled by client"
            ntfyEnabled=0
        ENDIF
    ELSE
        PRINT "\nThis is for some other characteristic"
    ENDIF
ENDFUNC 1

ONEVENT  EVNOTIFYBUF CALL HndlrNtfyBuf
ONEVENT  EVBLEMSG    CALL HndlrBleMsg
ONEVENT  EVCHARCCCD  CALL HndlrCharCccd

IF OnStartup()==0 THEN
    rc = BleCharValueRead(hMyChar,at$)
    PRINT "\nYou can connect and write to the CCCD characteristic."
    PRINT "\nThe BL600 will then send you data until buffer is full\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF

WAITEVENT

CloseConnections()
PRINT "\nExiting..."
```

Expected Output:

```
You can connect and write to the CCCD characteristic.
The BL600 will then send you data until buffer is full

--- Connected to client
Notifications have been disabled by client : Notifications have been
enabled by client
--- Notifying
--- Buffer full
Notified 1818505336 times
Exiting...
```

Miscellaneous Functions

This section describes all BLE-related functions that are not related to advertising, connection, security manager, or GATT.

BleTxPowerSet

FUNCTION

This function sets the power of all packets that are transmitted subsequently.

The actual value is determined by scanning through the value list (4, 0, -4, -8, -12, -16, -20, -30 , -55) so that the highest value that is less than the desired value is selected. If the desired value is higher than -55, -55 is set.

For example, setting 1000 results in +4, -3 results in -4, -100 results in -55.

At any time SYSINFO(2008) returns the actual transmit power setting. When in command mode, use the command AT I 2008.

BLETXPOWERSET(nTxPower)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
nTxPower	**byVal nTxPower AS INTEGER** Specifies the new transmit power in dBm units to be used for all subsequent Tx packets. The actual value is determined by scanning through the value list (4, 0, -4, -8, -12, -16, -20, -30 , -55) so that the highest value that is less than the desired value is selected. If the desired value is higher than -55, -55 is set.
Interactive Command	No

```
//Example :: BleTxPowerSet.sb (See in BL600CodeSnippets.zip)
DIM rc,dp

dp=1000 : rc = BleTxPowerSet(dp)
PRINT "\nrc = ";rc
PRINT "\nTx power : desired= ";dp," actual= "; SysInfo(2008)
dp=8 : rc = BleTxPowerSet(dp)
PRINT "\nTx power : desired= ";dp," "," actual= "; SysInfo(2008)
dp=2 : rc = BleTxPowerSet(dp)
PRINT "\nTx power : desired= ";dp," "," actual= "; SysInfo(2008)
dp=-10 : rc = BleTxPowerSet(dp)
PRINT "\nTx power : desired= ";dp," "," actual= "; SysInfo(2008)
dp=-25 : rc = BleTxPowerSet(dp)
PRINT "\nTx power : desired= ";dp," "," actual= "; SysInfo(2008)
dp=-45 : rc = BleTxPowerSet(dp)
PRINT "\nTx power : desired= ";dp," "," actual= "; SysInfo(2008)
dp=-1000 : rc = BleTxPowerSet(dp)
PRINT "\nTx power : desired= ";dp," actual= "; SysInfo(2008)
```

Expected Output:

```
rc = 0
Tx power : desired= 1000        actual= 4
Tx power : desired= 8           actual= 4
Tx power : desired= 2           actual= 0
Tx power : desired= -10         actual= -12
Tx power : desired= -25         actual= -30
Tx power : desired= -45         actual= -55
Tx power : desired= -1000       actual= -55
```

BLETXPOWERSET is an extension function.

BleTxPwrWhilePairing

FUNCTION

This function sets the transmit power of all packets that are transmitted while a pairing is in progress. This mode of pairing is referred to as Whsiper Mode Pairing. The actual value is clipped to the transmit power for normal operation which is set using BleTxPowerSet() function.

The actual value is determined by scanning through the value list (4, 0, -4, -8, -12, -16, -20, -30 , -55) so that the highest value that is less than the desired value is selected. If the desired value is higher than -55, -55 is set.

For example, setting 1000 results in +4, -3 results in -4, -100 results in -55.

At any time, SYSINFO(2018) returns the actual transmit power setting. Or when in command mode, use the command AT I 2018.

BLETXPWRWHILEPAIRING(nTxPower)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
nTxPower	**byVal nTxPower AS INTEGER** Specifies the new transmit power in dBm units to be used for all subsequent Tx packets. The actual value is determined by scanning through the value list (4, 0, -4, -8, -12, -16, -20, -30 , -55) so that the highest value that is less than the desired value is selected. If the desired value is higher than -55, -55 is set.
Interactive Command	No

```
//Example :: BleTxPwrWhilePairing.sb (See in BL600CodeSnippets.zip)
DIM rc,dp

dp=1000 : rc = BleTxPwrWhilePairing(dp)
PRINT "\nrc = ";rc
PRINT "\nTx power while pairing: desired= ";dp," actual= "; SysInfo(2018)
dp=8 : rc = BleTxPwrWhilePairing(dp)
PRINT "\nTx power while pairing: desired= ";dp," "," actual= "; SysInfo(2018)
dp=2 : rc = BleTxPwrWhilePairing(dp)
PRINT "\nTx power while pairing: desired= ";dp," "," actual= "; SysInfo(2018)
dp=-10 : rc = BleTxPwrWhilePairing(dp)
PRINT "\nTx power while pairing: desired= ";dp," actual= "; SysInfo(2018)
dp=-25 : rc = BleTxPwrWhilePairing(dp)
PRINT "\nTx power while pairing: desired= ";dp," actual= "; SysInfo(2018)
dp=-45 : rc = BleTxPwrWhilePairing(dp)
PRINT "\nTx power while pairing: desired= ";dp," actual= "; SysInfo(2018)
dp=-1000 : rc = BleTxPwrWhilePairing(dp)
PRINT "\nTx power while pairing: desired= ";dp," actual= "; SysInfo(2018)
```

Expected Output:

```
Tx power while pairing: desired= 8          actual= 4
Tx power while pairing: desired= 2          actual= 0
Tx power while pairing: desired= -10        actual= -12
Tx power while pairing: desired= -25        actual= -30
Tx power while pairing: desired= -45        actual= -55
Tx power while pairing: desired= -1000      actual= -55
```

BLETXPOWERSET is an extension function.

BleConfigDcDc

SUBROUTINE

This routine is used to configure the DC to DC converter to one of 3 states:- OFF, ON or AUTOMATIC.

Note: Until a future revision when the chipset vendor has fixed a hardware issue at the silicon level this function will not function as stated and any *nNewState* value will be interpreted as OFF

BLECONFIGDCDC(nNewState)

Returns	None	
Arguments		
nNewState	byVal *nNewState* AS INTEGER. Configure the internal DC to DC converter as follows:	
	0	Off
	2	Auto
	All other values	On
Interactive Command	No	

```
BleConfigDcDc(2)                    //Set for automatic operation
```

BLECONFIGDCDC is an extension function.

Advertising Functions

This section describes all advertising-related routines.

An advertisement consists of the following:

- A packet of information with a header identifying it as one of four types
- An optional payload that consists of multiple advertising records, referred to as AD in the rest of this manual.

Each AD record consists of up to three fields:

- Field 1 – One octet in length and contains the number of octets that follow it that belong to that record.
- Field 2 – One octet and is a tag value which identifies the type of payload that starts at the next octet. Hence the payload data is 'length – 1'.
- A special NULL AD record consists of only one field, that is, the length field, when it contains just the 00 value.

The specification also allows custom AD records to be created using the 'Manufacturer Specific Data' AD record.

Note: Refer to the Supplement to the Bluetooth Core Specification, Version 1, Part A which has the latest list of all AD records. You will need to register as at least an Adopter, which is free, to gain access to this information.

BleAdvertStart

FUNCTION

This function causes a BLE advertisement event as per the Bluetooth Specification. An advertisement event consists of an advertising packet in each of the three advertising channels.

The type of advertisement packet is determined by the nAdvType argument and the data in the packet is initialised, created, and submitted by the **BLEADVRPTINIT**, **BLEADVRPTADDxxx,** and **BLEADVRPTCOMMIT** functions respectively.

If the Advert packet type (nAdvType) is specified as 1 (ADV_DIRECT_IND) then the peerAddr$ string must not be empty and should be a valid address. When advertising with this packet type, the timeout is automatically set to 1280 ms.

When filter policy is enabled, the whitelist consisting of all bonded masters is submitted to the underlying stack so that only those bonded masters result in scan and connection requests being serviced.

Note: nAdvTimeout is rounded up to the nearest 1000 msec.

BLEADVERTSTART (nAdvType,peerAddr$,nAdvInterval, nAdvTimeout, nFilterPolicy)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
If a 0x6A01 resultcode is received, it implies whitelist has been enabled but the Flags AD in the advertising report is set for limited and/or general discoverability. The solution is to resubmit a new advert report which is made up so that the nFlags argument to BleAdvRptInit() function is 0. The BT 4.0 spec disallows discoverability when a whitelist is enabled during advertisement see Volume 3, Sections 9.2.3.2 and 9.2.4.2.	
Arguments	
nAdvType	byVal *nAdvType* AS INTEGER. Specifies the advertisement type as follows:

0	ADV_IND – Invites connection requests
1	ADV_DIRECT_IND – Invites connection from addressed device
2	ADV_SCAN_IND – Invites scan requests for more advert data
3	ADV_NONCONN_IND – Does not accept connections and/or active scans

peerAddr$	byRef peerAddr$ AS STRING It can be an empty string that is omitted if the advertisement type is not ADV_DIRECT_IND. This is only required when nAdvType == 1. When not empty, a valid address string is exactly seven octets long (such as \00\11\22\33\44\55\66), where the first octet is the address type and the rest of the 6 octets is the usual MAC address in big endian format (so that most significant octet of the address is at offset 1), whether public or random. 0 Public 1 Random Static 2 Random Private Resolvable 3 Random Private Non-resolvable All other values are illegal.
nAdvInterval	byVal nAdvInterval AS INTEGER. The interval between two advertisement events (in milliseconds). An advertisement event consists of a total of three packets being transmitted in the three advertising channels. **Interval range:** Between 20 and 10240 milliseconds.
nAdvTimeout	byVal nAdvTimeout AS INTEGER. The time after which the module stops advertising (in milliseconds). **Value range:** Between 0 and 16383000 milliseconds (rounded up to the nearest one seconds or 1000 ms). A value of 0 means disable the timeout, but note that if limited advert modes was specified in BleAdvRptInit() then the timeout is capped to 180000 ms as per the Bluetooth Specification. When the advert type specified is ADV_DIRECT_IND , the timeout is automatically set to 1280 ms as per the Bluetooth Specification. **Warning:** To save power, do not set this to (e.g.) 100 ms.
nFilterPolicy	byVal *nFilterPolicy* AS INTEGER. Specifies the filter policy for the whitelist consisting of all bonded masters as follows: 0 Disable whitelist 1 Filter scan request; allow connection request from any 2 Filter connection request; allow scan request from any 3 Filter scan request and connection request If the filter policy is not 0, the whitelist is enabled and filled with all the addresses of all the devices in the trusted device database.
Interactive Command	No

```
//Example :: BleAdvertStart.sb (See in BL600CodeSnippets.zip)
DIM addr$ : addr$=""

FUNCTION HndlrBlrAdvTimOut()
   PRINT "\nAdvert stopped via timeout"
   PRINT "\nExiting..."
ENDFUNC 0

//The advertising interval is set to 25 milliseconds. The module will stop
//advertising after 60000 ms (1 minute)
IF BleAdvertStart(0,addr$,25,60000,0)==0 THEN
    PRINT "\nAdverts Started"
```

```
    PRINT "\nIf you search for bluetooth devices on your device, you should see
'Laird BL600'"
ELSE
    PRINT "\n\nAdvertisement not successful"
ENDIF


ONEVENT  EVBLE_ADV_TIMEOUT  CALL HndlrBlrAdvTimOut

WAITEVENT
```

Expected Output:

```
Adverts Started

If you search for bluetooth devices on your device, you should see 'Laird
BL600'

Advert stopped via timeout
Exiting...
```

BLEADVERTSTART is an extension function.

BleAdvertStop

FUNCTION

This function causes the BLE module to stop advertising.

BLEADVERTSTOP ()

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	None
Interactive Command	No

```
//Example :: BleAdvertStop.sb (See in BL600CodeSnippets.zip)
DIM addr$ : addr$=""
DIM rc

FUNCTION HndlrBlrAdvTimOut()
   PRINT "\nAdvert stopped via timeout"
   PRINT "\nExiting..."
ENDFUNC 0

FUNCTION Btn0Press()
    IF BleAdvertStop()==0 THEN
        PRINT "\nAdvertising Stopped"
    ELSE
        PRINT "\n\nAdvertising failed to stop"
    ENDIF

    PRINT "\nExiting..."
ENDFUNC 0
```

```
IF BleAdvertStart(0,addr$,25,60000,0)==0 THEN
    PRINT "\nAdverts Started. Press button 0 to stop.\n"
ELSE
    PRINT "\n\nAdvertisement not successful"
ENDIF

rc = GpioSetFunc(16,1,2)
rc = GpioBindEvent(0,16,1)

ONEVENT  EVBLE_ADV_TIMEOUT  CALL HndlrBlrAdvTimOut
ONEVENT  EVGPIOCHAN0        CALL Btn0Press

WAITEVENT
```

Expected Output:

```
Adverts Started. Press button 0 to stop.

Advertising Stopped
Exiting...
```

BLEADVERTSTOP is an extension function.

BleAdvRptInit

FUNCTION

This function is used to create and initialise an advert report with a minimal set of ADs (advertising records) and store it the string specified. It is not advertised until BLEADVRPTSCOMMIT is called.

This report is for use with advertisement packets.

BLEADVRPTINIT(advRpt$, nFlagsAD, nAdvAppearance, nMaxDevName)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
advRpt$	byRef ***advRpt$*** AS STRING. This will contain an advertisement report.
nFlagsAD	byVal ***nFlagsAD*** AS INTEGER. Specifies the flags AD bits where bit 0 is set for limited discoverability and bit 1 is set for general discoverability. Bit 2 will be forced to 1 and bits 3 & 4 will be forced to 0. Bits 3 to 7 are reserved for future use by the BT SIG and must be set to 0. **Note:** If a whitelist is enabled in the BleAdvertStart() function then both Limited and General Discoverability flags MUST be 0 as per the BT 4.0 specification (Volume 3, Sections 9.2.3.2 and 9.2.4.2)
nAdvAppearance	byVal ***nAdvAppearance*** AS INTEGER. Determines whether the appearance advert should be added or omitted as follows: 0 — Omit appearance advert 1 — Add appearance advert as specified in the GAP service which is supplied via the BleGapSvcInit() function.
nMaxDevName	byVal ***nMaxDevName*** AS INTEGER. The n leftmost characters of the device name specified in the GAP service. If this value is set to 0, then the device name is not included.

Interactive Command	No

```
//Example :: BleAdvRptInit.sb (See in BL600CodeSnippets.zip)
DIM advRpt$ : advRpt$=""
DIM discovMode : discovMode=0
DIM advAppearance : advAppearance = 1
DIM maxDevName : maxDevName = 10

IF BleAdvRptInit(advRpt$, discovMode, advAppearance, maxDevName)==0 THEN
    PRINT "\nAdvert report initialised"
ENDIF
```

Expected Output:

```
Advert report initialised
```

BLEADVRPTINIT is an extension function.

BleScanRptInit

FUNCTION

This function is used to create and initialise a scan report which is sent in a SCAN_RSP message. It is not used until BLEADVRPTSCOMMIT is called.

This report is for use with SCAN_RESPONSE packets.

BLESCANRPTINIT(scanRpt)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
scanRpt	byRef scanRpt ASSTRING This contains a scan report.
Interactive Command	No

```
//Example :: BleScanRptInit.sb (See in BL600CodeSnippets.zip)
DIM scnRpt$ : scnRpt$=""

IF BleScanRptInit(scnRpt$)==0 THEN
    PRINT "\nScan report initialised"
ENDIF
```

Expected Output:

```
Scan report initialised
```

BLESCANRPTINIT is an extension function.

BleAdvRptGetSpace

FUNCTION

This function returns the free space in the advert advRpt$

BLEADVRPTGETSPACE(advRpt)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
advRpt$	byRef *advRpt$ AS* STRING. This contains an advert/scan report.
Interactive Command	No

```
dim rc, s$, dn$
rc=BleScanRptInit(s$)
dn$ = BleGetDeviceName$()

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

print "\nFree space in scan report: "; BleAdvRptGetSpace(s$); " bytes"
```

Expected Output:

```
Free space in scan report: 18 bytes
```

BLESCANRPTINIT is an extension function.

BleAdvRptAddUuid16

FUNCTION

This function is used to add a 16 bit UUID service list AD (Advertising record) to the advert report. This consists of all of the 16 bit service UUIDs that the device supports as a server.

BLEADVRPTADDUUID16 (advRpt, nUuid1, nUuid2, nUuid3, nUuid4, nUuid5, nUuid6)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
AdvRpt	byRef *AdvRpt AS* STRING. The advert report onto which the 16 bit uuids AD record is added.
Uuid1	*byVal uuid1 AS INTEGER* UUID in the range 0 to FFFF; if the value is outside that range it is ignored. Set the value to -1 to have it ignored and then all further UUID arguments will also be ignored.
Uuid2	*byVal uuid2 AS INTEGER* UUID in the range 0 to FFFF; if the value is outside that range it is ignored. Set the value to -1 to have it ignored and then all further UUID arguments will also be ignored.
Uuid3	*byVal uuid3 AS INTEGER* UUID in the range 0 to FFFF; if the value is outside that range it is ignored.

	Set the value to -1 to have it ignored and then all further UUID arguments will also be ignored.
Uuid4	*byVal uuid4 AS INTEGER* UUID in the range 0 to FFFF; if the value is outside that range it is ignored. Set the value to -1 to have it ignored and then all further UUID arguments will also be ignored.
Uuid5	*byVal uuid5 AS INTEGER* UUID in the range 0 to FFFF; if the value is outside that range it is ignored. Set the value to -1 to have it ignored and then all further UUID arguments will also be ignored.
Uuid6	*byVal uuid6 AS INTEGER* UUID in the range 0 to FFFF; if the value is outside that range it is ignored. Set the value to -1 to have it ignored.
Interactive Command	No

```
//Example :: BleAdvAddUuid16.sb (See in BL600CodeSnippets.zip)
DIM advRpt$, rc
DIM discovMode : discovMode=0
DIM advAppearance : advAppearance = 1
DIM maxDevName : maxDevName = 10

rc = BleAdvRptInit(advRpt$, discovMode, advAppearance, maxDevName)

//BatteryService =  0x180F
//DeviceInfoService = 0x180A

IF BleAdvRptAddUuid16(advRpt$,0x180F,0x180A, -1, -1, -1, -1)==0 THEN
    PRINT "\nUUID Service List AD added"
ENDIF

//Only the battery and device information services are included in the advert report
```

Expected Output:

```
UUID Service List AD added
```

BLEADVRPTADDUUID16 is an extension function.

BleAdvRptAddUuid128

FUNCTION

This function is used to add a 128 bit UUID service list AD (Advertising record) to the advert report specified. Given that an advert can have a maximum of only 31 bytes, it is not possible to have a full UUID list unless there is only one to advertise.

BLEADVRPTADDUUID128 (advRpt, nUuidHandle)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
advRpt	byRef *AdvRpt AS* STRING.

	The advert report into which the 128 bit uuid AD record is to be added.
nUuidHandle	*byVal nUuidHandle AS INTEGER* This is handle to a 128 bit uuid which was obtained using the function BleHandleUuid128() or some other function which returns one, such as BleVSpOpen()
Interactive Command	No

```
//Example :: BleAdvAddUuid128.sb (See in BL600CodeSnippets.zip)
DIM tx$,scRpt$,adRpt$,addr$, hndl
scRpt$=""
PRINT BleScanRptInit(scRpt$)

//Open the VSP
PRINT BleVSpOpen(128,128,0,hndl)

//Advertise the VSPservice in a scan report
PRINT BleAdvRptAddUuid128(scRpt$,hndl)
adRpt$=""
PRINT BleAdvRptsCommit(adRpt$,scRpt$)
addr$=""  //because we are not doing a DIRECT advert
PRINT BleAdvertStart(0,addr$,20,30000,0)
```

Expected Output:

```
00000
```

BLEADVRPTADDUUID128 is an extension function.

BleAdvRptAppendAD

FUNCTION

This function adds an arbitrary AD (Advertising record) field to the advert report. An AD element consists of a LEN:TAG:DATA construct where TAG can be any value from 0 to 255 and DATA is a sequence of octets.

BLEADVRPTAPPENDAD (advRpt, nTag, stData$)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
AdvRpt	byRef *AdvRpt AS* STRING. The advert report onto which the AD record is to be appended.
nTag	TAG field for the record. **Valid range:** 0 to FF
stData$	*byRef stData$ AS STRING* This is an octet string which can be 0 bytes long. The maximum length is governed by the space available in AdvRpt (a maximum of 31 bytes long).
Interactive Command	No

```
//Example :: BleAdvRptAppendAD.sb (See in BL600CodeSnippets.zip)
DIM scnRpt$,ad$
ad$="\01\02\03\04"
```

```
PRINT BleScanRptInit(scnRpt$)

IF BleAdvRptAppendAD(scnRpt$,0x31,ad$)==0 THEN //6 bytes will be used up in the
report
    PRINT "\nAD with data '";ad$;"' was appended to the advert report"
ENDIF
```

Expected Output:

```
0
AD with data '\01\02\03\04' was appended to the advert report
```

BLEADVRPTAPPENDAD is an extension function.

BleGetADbyIndex

FUNCTION

This function is used to extract a copy of the nth (zero based) advertising data (AD) element from a string which is assumed to contain the data portion of an advert report, incoming or outgoing.

Note: If the last AD element is malformed, it will be treated as not existing. For example, it will be malformed if the length byte for that AD element suggests that more data bytes are required than actually exist in the report string.

BLEGETADBYINDEX (nIndex, rptData$, nADtag, ADval$)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
nIndex	byVAL *nIndex AS* INTEGER This is a zero based index of the AD element that will be copied into the output data parameter ADval$.
rptData$	byREF *rptData$ AS* STRING. This parameter is a string that contains concatenated AD elements which will have been either constructed for an outgoing advert or will have been received in a scan (depends on module variant)
nADTag	*byREF nADTag AS INTEGER* When the nth index is found, the single byte tag value for that AD element is returned in this parameter.
ADval$	*byREF ADval$ AS STRING* When the nth index is found, the data excluding single byte the tag value for that AD element is returned in this parameter.
Interactive Command	No

```
//Example :: BleAdvGetADbyIndex.sb (See in BL600CodeSnippets.zip)
DIM rc,  ad1$, ad2$, fullAD$, nADTag, ADval$

'//AD with length = 6 bytes, tag = 0xDD
ad1$="\06\DD\11\22\33\44\55"
```

```
'//AD with length = 7 bytes, tag = 0xDA
ad2$="\07\EE\AA\BB\CC\DD\EE\FF"

fullAD$ = ad1$ + ad2$
PRINT "\n\n"; Strhexize$(fullAD$);"\n"

rc=BleGetADbyIndex(0, fullAD$ , nADTag, ADval$ )
IF rc==0 THEN
    PRINT "\nFirst AD element with tag 0x"; INTEGER.H'nADTag ;" is
";StrHexize$(ADval$)
ELSE
    PRINT "\nError reading AD: " ;INTEGER.H'rc
ENDIF

rc=BleGetADbyIndex(1, fullAD$ , nADTag, ADval$)
IF rc==0 THEN
    PRINT "\nSecond AD element with tag 0x"; INTEGER.H'nADTag ;" is
";StrHexize$(ADval$)
ELSE
    PRINT "\nError reading AD: "; INTEGER.H'rc
ENDIF

'//Will fail because there are only 2 AD elements
rc=BleGetADbyIndex(2, fullAD$ , nADTag, ADval$)
IF rc==0 THEN
    PRINT "\nThird AD element with tag 0x"; INTEGER.H'nADTag ;" is
";StrHexize$(ADval$)
ELSE
    PRINT "\nError reading AD: "; INTEGER.H'rc
ENDIF
```

Expected Output:

```
06DD112233445507EEAABBCCDDEEFF

First AD element with tag 0x000000DD is 1122334455
Second AD element with tag 0x000000EE is AABBCCDDEEFF
Error reading AD: 00006060
```

BLEGETADBYINDEX is an extension function.

BleGetADbyTag

FUNCTION

This function is used to extract a copy of the first advertising data (AD) element that has the tag byte specified from a string which is assumed to contain the data portion of an advert report, incoming or outgoing. If multiple instances of that AD tag type are suspected, then use the function BleGetADbyIndex to extract.

Note: If the last AD element is malformed then it will be treated as not existing. For example, it will be malformed if the length byte for that AD element suggests that more data bytes are required than actually exist in the report string.

BLEGETADBYTAG (rptData$, nADtag, ADval$)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
rptData$	**byREF** ***rptData$ AS*** **STRING**. This parameter is a string that contains concatenated AD elements which will have been either constructed for an outgoing advert or will have been received in a scan (depends on module variant)
nADTag	***byVAL nADTag AS INTEGER*** This parameter specifies the single byte tag value for the AD element that is to returned in the ADval$ parameter. Only the first instance can be catered for. If multiple instances are suspected then use BleAdvADbyIndex() to extract it.
ADval$	***byREF ADval$ AS STRING*** When the nth index is found, the data excluding single byte the tag value for that AT element is returned in this parameter.
Interactive Command	No

```
//Example :: BleAdvGetADbyIndex.sb (See in BL600CodeSnippets.zip)
DIM rc,  ad1$, ad2$, fullAD$, nADTag, ADval$

'//AD with length = 6 bytes, tag = 0xDD
ad1$="\06\DD\11\22\33\44\55"

'//AD with length = 7 bytes, tag = 0xDA
ad2$="\07\EE\AA\BB\CC\DD\EE\FF"

fullAD$ = ad1$ + ad2$
PRINT "\n\n"; Strhexize$(fullAD$);"\n"

nADTag = 0xDD
rc=BleGetADbyTag(fullAD$ , nADTag, ADval$ )
IF rc==0 THEN
    PRINT "\nAD element with tag 0x"; INTEGER.H'nADTag ;" is ";StrHexize$(ADval$)
ELSE
    PRINT "\nError reading AD: " ;INTEGER.H'rc
ENDIF

nADTag = 0xEE
rc=BleGetADbyTag(fullAD$ , nADTag, ADval$)
IF rc==0 THEN
    PRINT "\nAD element with tag 0x"; INTEGER.H'nADTag ;" is ";StrHexize$(ADval$)
ELSE
    PRINT "\nError reading AD: "; INTEGER.H'rc
ENDIF

nADTAG = 0xFF
'//Will fail because no AD exists in 'fullAD$' with the tag 'FF'
rc=BleGetADbyTag(fullAD$ , nADTag, ADval$)
IF rc==0 THEN
    PRINT "\nAD element with tag 0x"; INTEGER.H'nADTag ;" is ";StrHexize$(ADval$)
ELSE
    PRINT "\nError reading AD: "; INTEGER.H'rc
ENDIF
```

Expected Output:

```
06DD112233445507EEAABBCCDDEEFF

AD element with tag 0x000000DD is 1122334455
AD element with tag 0x000000EE is AABBCCDDEEFF
Error reading AD: 00006060
```

BLEGETADBYTAG is an extension function.

BleAdvRptsCommit

FUNCTION

This function is used to commit one or both advert reports. If the string is empty then that report type is not updated. If both strings are empty, this call will have no effect.

The advertisements will not happen until they are started using BleAdvertStart() function.

BLEADVRPTSCOMMIT(advRpt, scanRpt)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
advRpt	byRef *advRpt* AS STRING. The most recent advert report.
scanRpt	byRef *scanRpt* AS STRING. The most recent scan report.
Note:	If any one of the two strings is not valid then the call will be aborted without updating the other report even if this other report is valid.
Interactive Command	No

```
//Example :: BleAdvRptsCommit.sb (See in BL600CodeSnippets.zip)
DIM advRpt$ : advRpt$=""
DIM scRpt$ : scRpt$=""
DIM discovMode : discovMode = 0
DIM advApprnce : advApprnce = 1
DIM maxDevName : maxDevName = 10

PRINT BleAdvRptInit(advRpt$, discovMode, advApprnce, maxDevName)
PRINT BleAdvRptAddUuid16(advRpt$, 0x180F,0x180A, -1, -1, -1, -1)
PRINT BleAdvRptsCommit(advRpt$, scRpt$)

// Only the advert report will be updated.
```

Expected Output:

```
000
```

BLEADVRPTSCOMMIT is an extension function.

Connection Functions

This section describes all the connection manager related routines.

The Bluetooth specification stipulates that a peripheral cannot initiate a connection, but can perform disconnections. Only Central Role devices are allowed to connect when an appropriate advertising packet is received from a peripheral.

Events and Messages

See also Events and Messages for BLE-related messages that are thrown to the application when there is a connection or disconnection. The relevant message IDs are (0), (1), (14), (15), (16), (17), (18) and (20):

MsgId	Description
0	There is a connection and the context parameter contains the connection handle.
1	There is a disconnection and the context parameter contains the connection handle.
14	New connection parameters for connection associated with connection handle.
15	Request for new connection parameters failed for connection handle supplied.
16	The connection is to a bonded master
17	The bonding has been updated with a new long term key
18	The connection is encrypted
20	The connection is no longer encrypted

BleDisconnect

FUNCTION

This function causes an existing connection identified by a handle to be disconnected from the peer.

When the disconnection is complete a EVBLEMSG message with msgId = 1 and context containing the handle will be thrown to the *smart* BASIC runtime engine.

BLEDISCONNECT (nConnHandle)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
nConnHandle	byVal *nConnHandle* AS INTEGER. Specifies the handle of the connection that must be disconnected.
Interactive Command	No

```
//Example :: BleDisconnect.sb (See in BL600CodeSnippets.zip)
DIM addr$ : addr$=""
DIM rc

FUNCTION HndlrBleMsg(BYVAL nMsgId AS INTEGER, BYVAL nCtx AS INTEGER)
    SELECT nMsgId
        CASE 0
            PRINT "\nNew Connection ";nCtx
            rc = BleAuthenticate(nCtx)
            PRINT BleDisconnect(nCtx)
        CASE 1
            PRINT "\nDisconnected ";nCtx;"\n"
            EXITFUNC 0
    ENDSELECT
ENDFUNC 1

ONEVENT  EVBLEMSG           CALL HndlrBleMsg
```

```
IF BleAdvertStart(0,addr$,100,30000,0)==0 THEN
    PRINT "\nAdverts Started\n"
ELSE
    PRINT "\n\nAdvertisement not successful"
ENDIF

WAITEVENT
```

Expected Output:

```
Adverts Started

New Connection 35800
Disconnected 3580
```

BLEDISCONNECT is an extension function.

BleSetCurConnParms

FUNCTION

This function triggers an existing connection identified by a handle to have new connection parameters. For example: interval, slave latency, and link supervision timeout.

When the request is complete, a EVBLEMSG message with msgId = 14 and context containing the handle is thrown to the *smart*BASIC runtime engine if it was successful. If the request to change the connection parameters fails, an EVBLEMSG message with msgid = 15 is thrown to the *smart*BASIC runtime engine.

BLESETCURCONNPARMS (nConnHandle, nMinIntUs, nMaxIntUs, nSuprToutUs, nSlaveLatency)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
nConnHandle	byVal *nConnHandle* AS INTEGER. Specifies the handle of the connection that must have the connection parameters changed.
nMinIntUs	byVal *nMinIntUs* AS INTEGER. The minimum acceptable connection interval in microseconds.
nMaxIntUs	byVal *nMaxIntUs* AS INTEGER. The maximum acceptable connection interval in microseconds.
nSuprToutUs	byVal *nSuprToutUs* AS INTEGER. The link supervision timeout for the connection in microseconds. It should be greater than the slave latency times the actual granted connection interval.
nSlaveLatency	byVal *nSlaveLatency* AS INTEGER. The number of connection interval polls that the peripheral may ignore. This times the connection interval shall not be greater than the link supervision timeout.

Note: Slave latency is a mechanism that reduces power usage in a peripheral device and maintains short latency. Generally a slave reduces power usage by setting the largest connection interval possible. This means the latency is equivalent to that connection interval. To mitigate this, the peripheral can greatly reduce the connection interval and then have a non-zero slave latency.

For example, a keyboard could set the connection interval to 1000 msec and slave latency to 0. In this case, key presses are reported to the central device once per second, a poor user experience.

Instead, the connection interval can be set to e.g. 50 msec and slave latency to 19. If there are no key presses, the power use is the same as before because ((19+1) * 50) equals 1000. When a key is pressed, the peripheral knows that the central device will poll within 50 msec, so it can send that keypress with a latency of 50 msec. A connection interval of 50 and slave latency of 19 means the slave is allowed to NOT acknowledge a poll for up to 19 poll messages from the central device.

Interactive Command	No

```
//Example :: BleSetCurConnParms.sb (See in BL600CodeSnippets.zip)
DIM rc
DIM addr$ : addr$=""

FUNCTION HandlerBleMsg(BYVAL nMsgId AS INTEGER, BYVAL nCtx AS INTEGER) AS INTEGER
    DIM intrvl,sprvTo,sLat

    SELECT nMsgId
        CASE 0 //BLE_EVBLEMSGID_CONNECT
            PRINT "\n --- New Connection : ","",nCtx
            rc=BleGetCurconnParms(nCtx,intrvl,sprvto,slat)
            IF rc==0 THEN
                PRINT "\nConn Interval","","",intrvl
                PRINT "\nConn Supervision Timeout",sprvto
                PRINT "\nConn Slave Latency","",slat
                PRINT "\n\nRequest new parameters"
                //request connection interval in range 50ms to 75ms and link
                //supervision timeout of 4seconds with a slave latency of 19
                rc =  BleSetCurconnParms(nCtx, 50000,75000,4000000,19)
            ENDIF
        CASE 1 //BLE_EVBLEMSGID_DISCONNECT
            PRINT "\n --- Disconnected : ",nCtx
            EXITFUNC 0
        CASE 14 //BLE_EVBLEMSGID_CONN_PARMS_UPDATE
            rc=BleGetCurconnParms(nCtx,intrvl,sprvto,slat)
            IF rc==0 THEN
                PRINT "\n\nConn Interval",intrvl
                PRINT "\nConn Supervision Timeout",sprvto
                PRINT "\nConn Slave Latency",slat
            ENDIF
        CASE 15 //BLE_EVBLEMSGID_CONN_PARMS_UPDATE_FAIL
            PRINT "\n ??? Conn Parm Negotiation FAILED"
        CASE ELSE
            PRINT "\nBle Msg",nMsgId
    ENDSELECT
ENDFUNC 1

ONEVENT  EVBLEMSG    CALL HandlerBleMsg

IF BleAdvertStart(0,addr$,25,60000,0)==0 THEN
    PRINT "\nAdverts Started\n"
    PRINT "\nMake a connection to the BL600"
ELSE
    PRINT "\n\nAdvertisement not successful"
ENDIF

WAITEVENT
```

Expected Output (Unsuccessful Negotiation):

```
Adverts Started

Make a connection to the BL600
 --- New Connection : 1352
Conn Interval              7500
Conn Supervision Timeout   7000000
Conn Slave Latency         0

Request new parameters
 ??? Conn Parm Negotiation FAILED
 --- Disconnected :   1352
```

Expected Output (Successful Negotiation):

```
Adverts Started

Make a connection to the BL600
 --- New Connection : 134
Conn Interval              30000
Conn Supervision Timeout   720000
Conn Slave Latency         0

Request new parameters

New conn Interval                75000
New conn Supervision Timeout     4000000
New conn Slave Latency           19
--- Disconnected :    134
```

Note: First set of parameters will differ depending on your central device.

BLESETCURCONNPARMS is an extension function.

BleGetCurConnParms

FUNCTION

This function gets the current connection parameters for the connection identified by the connection handle. Given there are three connection parameters, the function takes three variables by reference so that the function can return the values in those variables.

BLEGETCURCONNPARMS (nConnHandle, nIntervalUs, nSuprToutUs, nSlaveLatency)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
nConnHandle	byVal *nConnHandle* AS INTEGER. Specifies the handle of the connection that needs to have the connection parameters changed
nIntervalUs	byRef *nIntervalUs* AS INTEGER. The current connection interval in microseconds
nSuprToutUs	byRef *nSuprToutUs* AS INTEGER. The current link supervision timeout in microseconds for the connection.
nSlaveLatency	byRef *nSlaveLatency* AS INTEGER. This is the current number of connection interval polls that the peripheral may ignore. This value multiplied by the connection interval will not be greater than the link supervision timeout.
	Note: See Note on Slave Latency.
Interactive Command	No

See previous example

BLEGETCURCONNPARMS is an extension function.

Security Manager Functions

This section describes routines which manage all aspects of BLE security such as saving, retrieving, and deleting link keys and creation of those keys using pairing and bonding procedures.

Events & Messages

The following security manager messages are thrown to the run-time engine using the EVBLEMSG message with msgIDs as follows:

MsgId	Description
9	Pairing in progress and display Passkey supplied in msgCtx.
10	A new bond has been successfully created
11	Pairing in progress and authentication key requested. Type of key is in msgCtx. msgCtx is 1 for passkey_type which will be a number in the range 0 to 999999 and 2 for OOB key which is a 16 byte key.

To submit a passkey, use the function BLESECMNGRPASSKEY.

BleSecMngrPasskey

FUNCTION

This function submits a passkey to the underlying stack during a pairing procedure when prompted by the EVBLEMSG with msgId set to 11. See Events & Messages.

BLESECMNGRPASSKEY(connHandle, nPassKey)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
connHandle	byVal connHandle AS INTEGER. This is the connection handle as received via the EVBLEMSG event with msgId set to 0.
nPassKey	byVal *nPassKey* AS INTEGER. This is the passkey to submit to the stack. Submit a value outside the range 0 to 999999 to reject the pairing.
Interactive Command	No

```
//Example :: BleSecMngrPasskey.sb (See in BL600CodeSnippets.zip)

DIM rc, connHandle
DIM addr$ : addr$=""

FUNCTION HandlerBleMsg(BYVAL nMsgId AS INTEGER, BYVAL nCtx AS INTEGER) AS INTEGER
    SELECT nMsgId
        CASE 0
            connHandle = nCtx
            PRINT "\n--- Ble Connection, ",nCtx
        CASE 1
            PRINT "\n--- Disconnected ";nCtx;"\n"
            EXITFUNC 0
        CASE 11
            PRINT "\n +++ Auth Key Request, type=";nCtx
            rc=BleSecMngrPassKey(connHandle,123456)
            IF rc==0 THEN //key is 123456
                PRINT "\nPasskey 123456 was used"
            ELSE
                PRINT "\nResult Code 0x";integer.h'rc
            ENDIF
        CASE ELSE
    ENDSELECT
ENDFUNC 1

ONEVENT  EVBLEMSG  CALL HandlerBleMsg

rc=BleSecMngrIoCap(4)  //Set i/o capability - Keyboard Only (authenticated pairing)
IF BleAdvertStart(0,addr$,25,0,0)==0 THEN
    PRINT "\nAdverts Started\n"
    PRINT "\nMake a connection to the BL600"
ELSE
    PRINT "\n\nAdvertisement not successful"
ENDIF

WAITEVENT
```

Expected Output:

```
Adverts Started

Make a connection to the BL600
--- Ble Connection,        1655
 +++ Auth Key Request, type=1
Passkey 123456 was used
--- Disconnected 1655
```

BLESECMNGRPASSKEY is an extension function.

BleSecMngrKeySizes

FUNCTION

This function sets minimum and maximum long term encryption key size requirements for subsequent pairings.

If this function is not called, default values are 7 and 16 respectively. To ship your end product to a country with an export restriction, reduce nMaxKeySize to an appropriate value and ensure it is not modifiable.

BLESECMNGRKEYSIZES(nMinKeysize, nMaxKeysize)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
nMinKeysiz	byVal *nMinKeysiz* AS INTEGER. The minimum key size. The range of this value is from 7 to 16.
nMaxKeysize	byVal *nMaxKeysize* AS INTEGER. The maximum key size. The range of this value is from nMinKeysize to 16.
Interactive Command	No

```
//Example :: BleSecMngrKeySizes.sb (See in BL600CodeSnippets.zip)
PRINT BleSecMngrKeySizes(8,15)
```

Expected Output:

```
0
```

BLESECMNGRKEYSIZES is an extension function.

BleSecMngrIoCap

FUNCTION

This function sets the user I/O capability for subsequent pairings and is used to determine if the pairing is authenticated or not. This is related to Simple Secure Pairing as described in the following whitepapers:

https://www.bluetooth.org/docman/handlers/DownloadDoc.ashx?doc_id=86174

https://www.bluetooth.org/docman/handlers/DownloadDoc.ashx?doc_id=86173

In addition, the *Security Manager Specification* in the core 4.0 specification Part H provides a full description.

You must be registered with the Bluetooth SIG (www.bluetooth.org) to get access to all these documents.

An authenticated pairing is deemed to be one with less than 1 in a million probability that the pairing was compromised by a MITM (Man in the middle) security attack.

The valid user I/O capabilities are as described below.

BLESECMNGRIOCAP (nIoCap)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
nIoCap	byVal *nIoCap AS* INTEGER. The user I/O capability for all subsequent pairings. 0 None also known as 'Just Works' (unauthenticated pairing) 1 Display with Yes/No input capability (authenticated pairing) 2 Keyboard Only (authenticated pairing) 3 Display Only (authenticated pairing – if other end has input cap) 4 Keyboard only (authenticated pairing)
Interactive Command	No

```
//Example :: BleSecMngrIoCap.sb (See in BL600CodeSnippets.zip)
PRINT BleSecMngrIoCap(1)
```

Expected Output:

```
0
```

BLESECMNGRIOCAP is an extension function.

BleSecMngrBondReq

FUNCTION

This function is used to enable or disable bonding when pairing.

> **Note**: This function will be deprecated in future releases. It is recommended to invoke this function, with the parameter set to 0, before calling BleAuthenticate().

BLESECMNGRBONDREQ (nBondReq)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
nBondReq	byVal *nBondReq AS* INTEGER. 0 Disable 1 Enable
Interactive Command	No

```
//Example :: BleSecMngrBondReq.sb (See in BL600CodeSnippets.zip)
IF BleSecMngrBondReq(0)==0 THEN
    PRINT "\nBonding disabled"
ENDIF
```

Expected Output:

```
Bonding disabled
```

BLESECMNGRBONDREQ is an extension function.

BleAuthenticate

FUNCTION

This routine is used to induce the device to authenticate the peer. This will be deprecated in future firmware.

BLEAUTHENTICATE (nConnCtx)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
nConnCtx	byVal *nConnCtx* AS INTEGER. This is the context value provided in the EVBLEMSG(0) message which informed the stack that a connection had been established.
Interactive Command	No

See example for BleDisconnect:

Change "`rc = BleAuthenticate(nCtx)`" to "`PRINT BleAuthenticate(nCtx)`"

BLEAUTHENTICATE is an extension function.

GATT Server Functions

This section describes all functions related to creating and managing services that collectively define a GATT table from a GATT server role perspective. These functions allow the developer to create any Service that has been described and adopted by the Bluetooth SIG or any custom Service that implements some custom unique functionality, within resource constraints such as the limited RAM and FLASH memory that is exist in the module.

A GATT table is a collection of adopted or custom Services which in turn are a collection of adopted or custom Characteristics. Although keep in mind that by definition an adopted service cannot contain custom characteristics but the reverse is possible where a custom service can include both adopted and custom characteristics.

Descriptions of Services and Characteristics are available in the Bluetooth Specification v4.0 or newer and like most specifications are concise and difficult to understand. What follows is an attempt to familiarise the reader with those concepts using the perspective of the smartBASIC programming environment.

To help understand the terms Service and Characteristic better, think of a Characteristic as a container (or a pot) of data where the pot comes with space to store the data and a set of properties that are officially called 'Descriptors' in the BT spec. In the 'pot' analogy, think of Descriptor as colour of the pot, whether it has a lid, whether the lid has a lock or whether it has a handle or a spout etc. For a full list of these Descriptors online see http://developer.bluetooth.org/gatt/descriptors/Pages/DescriptorsHomePage.aspx . These descriptors are assigned 16 bit UUIDs (value 0x29xx) and are referenced in some of the smartBASIC API functions if you decide to add those to your characteristic definition.

To wrap up the loose analogy, think of Service as just a carrier bag to hold a group of related Characterisics together where the printing on the carrier bag is a UUID. You will find that from a smartBASIC developer's

perspective, a set of characteristics is what you will need to manage and the concept of Service is only required at GATT table creation time.

A GATT table can have many Services each containing one or more Characteristics. The differentiation between Services and Characteristics is expedited using an identification number called a UUID (Universally Unique Identifier) which is a 128 bit (16 byte) number. Adopted Services or Characteristics have a 16 bit (2 byte) shorthand identifier (which is just an offset plus a base 128 bit UUID defined and reserved by the Bluetooth SIG) and custom Service or Characteristics **shall** have the full 128 bit UUID. The logic behind this is that when you come across a 16 bit UUID, it implies that a specification will have been published by the Bluetooth SIG whereas using a 128 bit UUID does NOT require any central authority to maintain a register of those UUIDs or specifications describing them.

The lack of requirement for a central register is important to understand, in the sense that if a custom service or characteristic needs to be created, the developer can use any publicly available UUID (sometimes also known as GUID) generation utility.

These utilities use entropy from the real world to generate a 128 bit random number that has an extremely low probability to be the same as that generated by someone else at the same time or in the past or future.

As an example, at the time of writing this document, the following website http://www.guidgenerator.com/online-guid-generator.aspx offers an immediate UUID generation service, although it uses the term GUID. From the GUID Generator website:

> *How unique is a GUID?*
>
> *128-bits is big enough and the generation algorithm is unique enough that if 1,000,000,000 GUIDs per second were generated for 1 year the probability of a duplicate would be only 50%. Or if every human on Earth generated 600,000,000 GUIDs there would only be a 50% probability of a duplicate.*

This extremely low probability of generating the same UUID is why there is no need for a central register maintained by the Bluetooth SIG for custom UUIDs.

Please note that Laird does not warrant or guarantee that the UUID generated by this website or any other utility is unique. It is left to the judgement of the developer whether to use it or not.

Note: If the developer does intend to create custom Services and/or Characteristics then it is recommended that a single UUID is generated and be used from then on as a 128 bit (16 byte) company/developer unique base along with a 16 bit (2 byte) offset, in the same manner as the Bluetooth SIG.

This will then allow up to 65536 custom services and characteristics to be created, with the added advantage that it will be easier to maintain a list of 16 bit integers.

The main reason for avoiding more than one long UUID is to keep RAM usage down given that 16 bytes of RAM is used to store a long UUID. *Smart* BASIC functions have been provided to manage these custom 2 byte UUIDs along with their 16 byte base UUIDs.

In this document when a Service or Characteristic is described as adopted, it implies that the Bluetooth SIG has published a specification which defines that Service or Characteristic and there is a requirement that any device claiming to support them SHALL have approval to prove that the functionality has been tested and verified to behave as per that specification.

Currently there is no requirement for custom Service and/or Characteristics to have any approval. By definition, interoperability is restricted to just the provider and implementer.

A Service is an abstraction of some collectivised functionality which, if broken down further into smaller components, would cease to provide the intended behaviour. A couple of examples in the BLE domain that

have been adopted by the Bluetooth SIG are Blood Pressure Service and Heart Rate Service. Each have sub-components that map to Characteristics.

Blood Pressure is defined by a collection of data entities like for example Systolic Pressure, Diastolic Pressure, Pulse Rate and many more. Likewise a Heart Rate service also has a collection which includes entities such as the Pulse Rate and Body Sensor Location.

A list of all the adopted Services is at:http://developer.bluetooth.org/gatt/services/Pages/ServicesHome.aspx. Laird recommends that if you decide to create a custom Service then it is defined and described in a similar fashion, so that your goal should be to get the Bluetooth SIG to adopt it for everyone to use in an interoperable manner.

These Services are also assigned 16 bit UUIDs (value 0x18xx) and are referenced in some of the *smart*BASIC API functions described in this section.

Services, as described above, are a collection of one or more Characteristics. A list of all adopted characteristics is found at http://developer.bluetooth.org/gatt/characteristics/Pages/CharacteristicsHome.aspx. You should note that these descriptors are also assigned 16 bit UUIDs (value 0x2Axx) and are referenced in some of the API functions described in this section. Custom Characteristics will have 128 bit (16 byte) UUIDs and API functions are provided to handle those too.

Note: If you intend to create a custom Service or Characteristic, and adopt the recommendation, stated above, of a single long 16 byte base UUID, so that the service can be identified using a 2 byte UUID, then allocate a 16 bit value which is not going to coincide with any adopted values to minimise confusion. Selecting a similar value is possible and legal given that the base UUID is different. The recommendation is just for ease of maintenance.

Finally, having prepared a background to Services and Characteristics, the rest of this introduction will focus on the specifics of how to create and manage a GATT table from a perspective of the *smart*BASIC API functions in the module.

Recall that a Service has been described as a carrier bag that groups related characteristics together and a Characteristic is just a data container (pot). Therefore, a remote GATT Client, looking at the Server, which is presented in your GATT table, sees multiple carrier bags each containing one or more pots of data.

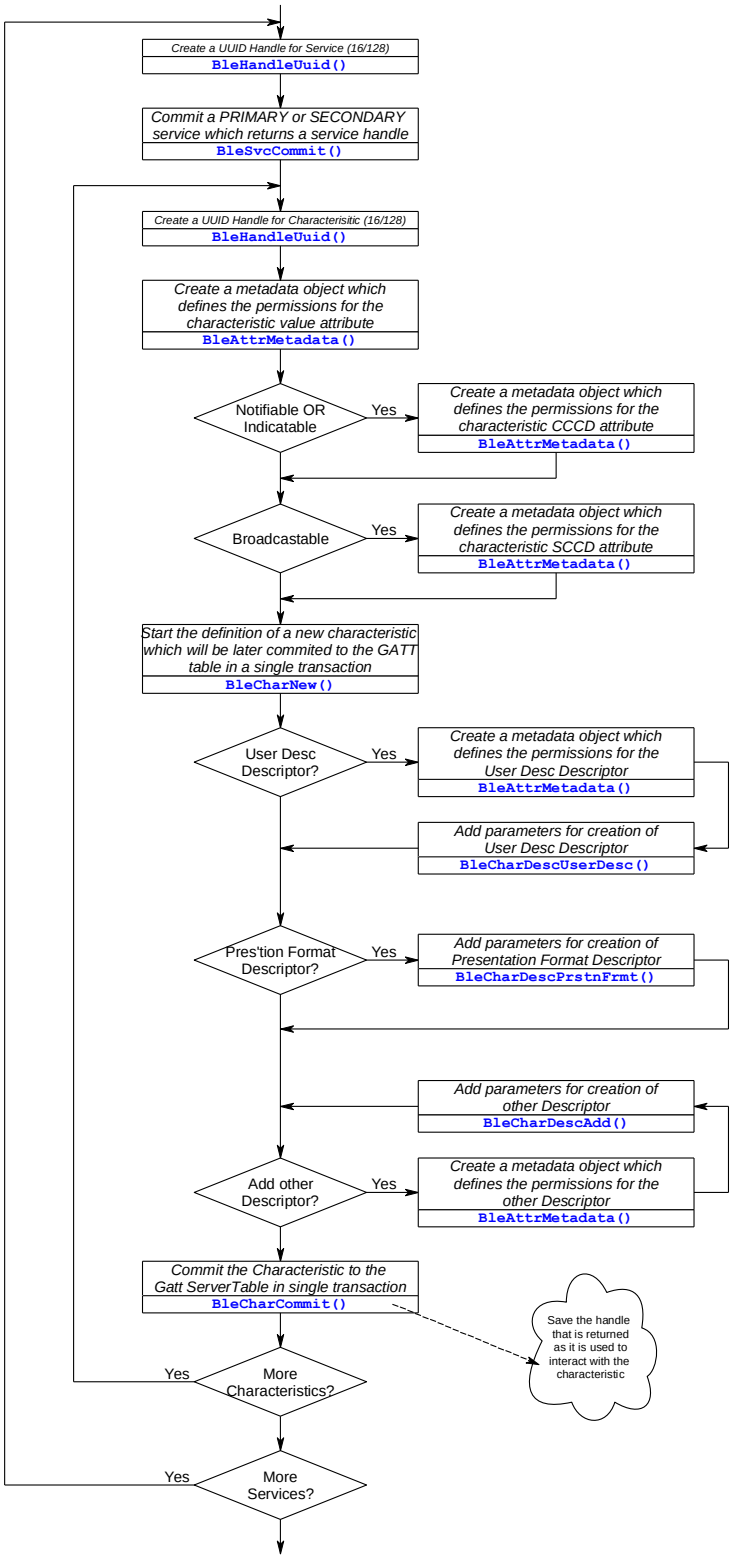
The GATT Client (remote end of the wireless connection) needs to see those carrier bags to determine the groupings and once it has identified the pots it will only need to keep a list of references to the pots it is interested in. Once that list is made at the client end, it can 'throw away the carrier bag'.

Similarly in the module, once the GATT table is created and after each Service is fully populated with one or more Characteristics there is no need to keep that 'carrier bag'. However, as each Characterstic is 'placed in the carrier bag' using the appropriate smartBASIC API function, a 'receipt' will be returned and is referred to as a char_handle. The developer will then need to keep those handles to be able to read and write and generally interact with that particular characteristic. The handle does not care whether the Characteristic is adopted or custom because from then on the firmware managing it behind the scenes in smartBASIC does not care.

Therefore from the smartBASIC app developer's **logical** perspective a GATT table looks nothing like the table that is presented in most BLE literature. Instead the GATT table is purely and simply just a collection of char_handles that reference the characteristics (data containers) which have been registered with the underlying GATT table in the BLE stack.

A particular char_handle is in turn used to make something happen to the referenced characteristic (data container) using a *smart*BASIC function and conversely if data is written into that characteristic (data container), by a remote GATT Client, then an event is thrown, in the form of a message, into the *smart*BASIC runtime engine which will get processed if and only if a handler function has been registered by the apps developer using the ONEVENT statement.

With this simple model in mind, an overview of how the *smart*BASIC functions are used to register Services and Characteristics is illustrated in the flowchart on the right and sample code follows.




```
//Example :: ServicesAndCharacteristics.sb (See in BL600CodeSnippets.zip)
```

```
//===============================================================================
//Register two Services in the GATT Table. Service 1 with 2 Characteristics and
//Service 2 with 1 characteristic. This implies a total of 3 characteristics to
//manage.
//The characteristic 2 in Service 1 will not be readable or writable but only
//indicatable
//The characteristic 1 in Service 2 will not be readable or writable but only
//notifyable
//===============================================================================

DIM rc      //result code
DIM hSvc    //service handle
DIM mdAttr
DIM mdCccd
DIM mdSccd
DIM chProp
DIM attr$

DIM hChar11  // handles for characteristic 1 of Service 1
DIM hChar21  // handles for characteristic 2 of Service 1
DIM hChar12  // handles for characteristic 1 of Service 2

DIM hUuidS1   // handles for uuid of Service 1
DIM hUuidS2   // handles for uuid of Service 2
DIM hUuidC11  // handles for uuid of characteristic 1 in Service 1
DIM hUuidC12  // handles for uuid of characteristic 2 in Service 1
DIM hUuidC21  // handles for uuid of characteristic 1 in Service 2

//---Register Service 1
hUuidS1 = BleHandleUuid16(0x180D)
rc = BleServiceNew(BLE_SERVICE_PRIMARY, hUuidS1, hSvc)

//---Register Characteristic 1 in Service 1
mdAttr = BleAttrMetadata(BLE_ATTR_ACCESS_OPEN,BLE_ATTR_ACCESS_OPEN,10,0,rc)
mdCccd = BLE_CHAR_METADATA_ATTR_NOT_PRESENT
mdSccd = BLE_CHAR_METADATA_ATTR_NOT_PRESENT
chProp = BLE_CHAR_PROPERTIES_READ + BLE_CHAR_PROPERTIES_WRITE
hUuidC11 = BleHandleUuid16(0x2A37)
rc = BleCharNew(chProp, hUuidC11,mdAttr,mdCccd,mdSccd)
rc = BleCharCommit(shHrs,hrs$,hChar11)

//---Register Characteristic 2 in Service 1
mdAttr = BleAttrMetadata(BLE_ATTR_ACCESS_OPEN,BLE_ATTR_ACCESS_OPEN,10,0,rc)
mdCccd = BleAttrMetadata(BLE_ATTR_ACCESS_OPEN,BLE_ATTR_ACCESS_OPEN,2,0,rc)
mdSccd = BLE_CHAR_METADATA_ATTR_NOT_PRESENT
chProp = BLE_CHAR_PROPERTIES_INDICATE
hUuidC12 = BleHandleUuid16(0x2A39)
rc = BleCharNew(chProp, hUuidC12,mdAttr,mdCccd,mdSccd)
attr$="\00\00"
rc = BleCharCommit(hSvc,attr$,hChar21)
rc = BleServiceCommit(hSvc)

//---Register Service 2  (can now reuse the service handle)
hUuidS2 = BleHandleUuid16(0x1856)
rc = BleServiceNew(BLE_SERVICE_PRIMARY, hUuidS2, hSvc)

//---Register Characteristic 1 in Service 2
mdAttr = BleAttrMetadata(BLE_ATTR_ACCESS_NONE,BLE_ATTR_ACCESS_NONE,10,0,rc)
mdCccd = BleAttrMetadata(BLE_ATTR_ACCESS_OPEN,BLE_ATTR_ACCESS_OPEN,2,0,rc)
mdSccd = BLE_CHAR_METADATA_ATTR_NOT_PRESENT
chProp = BLE_CHAR_PROPERTIES_NOTIFY
```

```
hUuidC21 = BleHandleUuid16(0x2A54)
rc = BleCharNew(chProp, hUuidC21,mdAttr,mdCccd,mdSccd)
attr$="\00\00\00\00"
rc = BleCharCommit(hSvc,attr$,hChar12)
rc = BleServiceCommit(hSvc)
//===The 2 services are now visible in the gatt table
```

Writes into a characteristic from a remote client is detected and processed as follow:

```
//-------------------------------------------------------------------------
// To deal with writes from a gatt client into characteristic 1 of Service 1
// which has the handle hChar11
//-------------------------------------------------------------------------

// This handler is called when there is a EVCHARVAL message
FUNCTION HandlerCharVal(BYVAL hChar AS INTEGER) AS INTEGER
  DIM attr$
  IF hChar == hChar11 THEN
    rc = BleCharValueRead(hChar11,attr$)
    print "Svc1/Char1 has been writen with = ";attr$

  ENDIF
ENDFUNC 1

//enable characteristic value write handler
OnEvent  EVCHARVAL          call HandlerCharVal

WAITEVENT
```

Assuming there is a connection and notify has been enabled then a value notification is expedited as follows:

```
//-------------------------------------------------------------------------
// Notify a value for characteristic 1 in service 2
//-------------------------------------------------------------------------
attr$="somevalue"
rc = BleCharValueNotify(hChar12,attr$)
```

Assuming there is a connection and indicate has been enabled then a value indication is expedited as follows:

```
//-------------------------------------------------------------------------
// indicate  a value for characteristic 2 in service 1
//-------------------------------------------------------------------------

// This handler is called when there is a EVCHARHVC message
FUNCTION HandlerCharHvc(BYVAL hChar AS INTEGER) AS INTEGER
  IF hChar == hChar12 THEN
    PRINT "Svc1/Char2 indicate has been confirmed"
  ENDIF
ENDFUNC 1

//enable characteristic value indication confirm  handler
OnEvent  EVCHARHVC          CALL HandlerCharHvc

attr$="somevalue"
rc = BleCharValueIndicate(hChar12,attr$)
```

The rest of this section details all the *smart*BASIC functions that help create that framework.

Events and Messages

See also Events and Messages for the messages that are thrown to the application which are related to the generic characteristics API. The relevant messages are those that start with EVCHARxxx.

BleGapSvcInit

FUNCTION

This function updates the GAP service, which is mandatory for all approved devices to expose, with the information provided. If it is not called before adverts are started, default values are exposed. Given this is a mandatory service, unlike other services which need to be registered, this one must only be initialised as the underlying BLE stack unconditionally registers it when starting up.

The GAP service contains five characteristics as listed at the following site:
http://developer.bluetooth.org/gatt/services/Pages/ServiceViewer.aspx?u=org.bluetooth.service.generic_access.xml

BLEGAPSVCINIT (deviceName, nameWritable, nAppearance, nMinConnInterval, nMaxConnInterval, nSupervisionTout, nSlaveLatency)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
deviceName	byRef *deviceName* AS STRING The name of the device (e.g. Laird_Thermometer) to store in the 'Device Name' characteristic of the GAP service.
Note:	When an advert report is created using BLEADVRPTINIT() this field is read from the service and an attempt is made to append it in the Device Name AD. If the name is too long, that function fails to initialise the advert report and a default name is transmitted. It is recommended that the device name submitted in this call be as short as possible.
nameWritable	byVal *nameWritable* AS INTEGER If non-zero, the peer device is allowed to write the device name. Some profiles allow this to be made optional.
nAppearance	byVal *nAppearance* AS INTEGER Field lists the external appearance of the device and updates the Appearance characteristic of the GAP service. Possible values: org.bluetooth.characteristic.gap.appearance.
nMinConnInterval	byVal *nMinConnInterval* AS INTEGER The preferred minimum connection interval, updates the 'Peripheral Preferred Connection Parameters' characteristic of the GAP service. Range is between 7500 and 4000000 microseconds (rounded to the nearest 1250 microseconds). This must be smaller than nMaxConnInterval.
nMaxConnInterval	byVal *nMaxConnInterval* AS INTEGER The preferred maximum connection interval, updates the 'Peripheral Preferred Connection Parameters' characteristic of the GAP service. Range is between 7500 and 4000000 microseconds (rounded to the nearest 1250 microseconds). This must be larger than nMinConnInterval.
nSupervisionTimeout	byVal *nSupervisionTimeout* AS INTEGER The preferred link supervision timeout and updates the 'Peripheral Preferred Connection Parameters' characteristic of the GAP service. Range is between 100000 to 32000000 microseconds (rounded to the nearest 10000 microseconds).
nSlaveLatency	byVal *nSlaveLatency* AS INTEGER The preferred slave latency is the number of communication intervals that a slave may ignore without losing the connection and updates the 'Peripheral Preferred Connection Parameters' characteristic of the GAP service. This value must be smaller than (nSupervisionTimeout/ nMaxConnInterval) -1. i.e. nSlaveLatency <

	(nSupervisionTimeout / nMaxConnInterval) -1
Interactive Command	No

```
//Example :: BleGapSvcInit.sb (See in BL600CodeSnippets.zip)

DIM rc,dvcNme$,nmeWrtble,apprnce,MinConnInt,MaxConnInt,ConnSupTO,sL,s$

dvcNme$= "Laird_TS"
nmeWrtble = 0             //Device name will not be writable by peer
apprnce = 768             //The device will appear as a Generic Thermometer
MinConnInt = 500000       //Minimum acceptable connection interval is 0.5 seconds
MaxConnInt = 1000000      //Maximum acceptable connection interval is 1 second
ConnSupTO = 4000000       //Connection supervisory timeout is 4 seconds
sL = 0                    //Slave latency--number of conn events that can be missed

rc=BleGapSvcInit(dvcNme$,nmeWrtble,apprnce,MinConnInt,MaxConnInt,ConnSupTO,sL)

IF !rc THEN
    PRINT "\nSuccess"
ELSE
    PRINT "\nFailed 0x"; INTEGER.H'rc      //Print result code as 4 hex digits
ENDIF
```

Expected Output:

```
Success
```

BLEGAPSVCINIT is an extension function.

BleGetDeviceName$

FUNCTION

This function reads the device name characteristic value from the local gatt table. This value is the same as that supplied in BleGapSvcInit() if the 'nameWritable' parameter was 0, otherwise it can be different.

EVBLEMSG event is thrown with 'msgid' == 21 when the GATT client writes a new value and is the best time to call this function.

BLEGETDEVICENAME$ ()

Returns	STRING, the current device name in the local GATT table. It is the same as that supplied in BleGapSvcInit() if the 'nameWritable' parameter was 0, otherwise it can be different. EVBLEMSG event is thrown with 'msgid' == 21 when the GATT client writes a new value.
Arguments	None
Interactive Command	No

```
//Example :: BleGetDeviceName$.sb (See in BL600CodeSnippets.zip)

DIM rc,dvcNme$,nmeWrtble,apprnce,MinConnInt,MaxConnInt,ConnSupTO,sL

PRINT "\n --- DevName : "; BleGetDeviceName$()

// Changing device name manually
dvcNme$= "My BL600"
nmeWrtble = 0
apprnce = 768
MinConnInt = 500000
MaxConnInt = 1000000
ConnSupTO = 4000000
sL = 0

rc = BleGapSvcInit(dvcNme$,nmeWrtble,apprnce,MinConnInt,MaxConnInt,ConnSupTO,sL)
PRINT "\n --- New DevName : "; BleGetDeviceName$()
```

Expected Output:

```
--- DevName : LAIRD BL600
--- New DevName : My BL600
```

BLEGETDEVICENAME$ is an extension function.

BleSvcRegDevInfo

FUNCTION

This function is used to register the Device Information service with the GATT server. The 'Device Information' service contains nine characteristics as listed at the following website: http://developer.bluetooth.org/gatt/services/Pages/ServiceViewer.aspx?u=org.bluetooth.service.device_information.xml

The firmware revision string will always be set to "BL600:vW.X.Y.Z" where W,X,Y,Z are as per the revision information which is returned to the command AT I 4.

BLESVCREGDEVINFO (manfName$, modelNum$, serialNum$, hwRev$,
swRev$, sysId$, regDataList$, pnpId$)

FUNCTION

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
manfName$	byVal *manfName$* AS STRING The device manufacturer. Can be set empty to omit submission.
modelNum$	byVal *modelNum$* AS STRING The device model number. Can be set empty to omit submission.
serialNum$	byVal *serialNum$* AS STRING The device serial number. Can be set empty to omit submission.
hwRev$	byVal *hwRev$* AS STRING The device hardware revision string. Can be set empty to omit submission.
swRev$	byVal *swRev$* AS STRING The device software revision string. Can be set empty to omit submission.
sysId$	byVal *sysId$* AS STRING The device system ID as defined in the specifications. Can be set empty to omit submission. Otherwise it shall be a string exactly 8 octets long, where: Byte 0..4 := Manufacturer Identifier Byte 5..7 := Organisationally Unique Identifier For the special case of the string being exactly 1 character long and containing "@", the system ID is created from the MAC address if (and only if) an IEEE public address is set. If the address is the random static variety, this characteristic is omitted.
regDataList$	byVal *regDataList$* AS STRING The device's regulatory certification data list as defined in the specification. It can be set as an empty string to omit submission.
pnpId$	byVal *pnpId$* AS STRING The device's plug and play ID as defined in the specification. Can be set empty to omit submission. Otherwise, it shall be exactly 7 octets long, where: Byte 0 := Vendor Id Source Byte 1,2 := Vendor Id (Byte 1 is LSB) Byte 3,4 := Product Id (Byte 3 is LSB) Byte 5,6 := Product Version (Byte 5 is LSB)
Interactive Command	No

```
//Example :: BleSvcRegDevInfo.sb (See in BL600CodeSnippets.zip)
```

```
DIM rc,manfNme$,mdlNum$,srlNum$,hwRev$,swRev$,sysId$,regDtaLst$,pnpId$

manfNme$ = "Laird Technologies"
mdlNum$ = "BL600"
srlNum$ = ""                    //empty to omit submission
hwRev$ = "1.0"
swRev$ = "1.0"
sysId$ = ""                     //empty to omit submission
regDtaLst$ = ""                 //empty to omit submission
pnpId$ = ""                    //empty to omit submission

rc=BleSvcRegDevInfo(manfNme$,mdlNum$,srlNum$,hwRev$,swRev$,sysId$,regDtaLst$,pnpId$)

IF !rc THEN
    PRINT "\nSuccess"
ELSE
    PRINT "\nFailed 0x"; INTEGER.H'rc
ENDIF
```

Expected Output:

```
Success
```

BLESVCREGDEVINFO is an extension function.

BleHandleUuid16

FUNCTION

This function takes an integer in the range 0 to 65535 and converts it into a 32 bit integer handle that associates the integer as an offset into the Bluetooth SIG 128 bit (16byte) base UUID which is used for all adopted services, characteristics and descriptors.

If the input value is not in the valid range then an invalid handle (0) is returned

The returned handle shall be treated by the developer as an opaque entity and no further logic shall be based on the bit content, apart from all 0's which represents an invalid UUID handle.

BLEHANDLEUUID16 (nUuid16)

Returns	INTEGER, a nonzero handle shorthand for the UUID. Zero is an invalid UUID handle.
Arguments	
nUuid16	byVal *nUuid16* AS INTEGER nUuid16 is first bitwise ANDed with 0xFFFF and the result will be treated as an offset into the Bluetooth SIG 128 bit base UUID.
Interactive Command	No

```
//Example :: BleHandleUuid16.sb (See in BL600CodeSnippets.zip)
DIM uuid
DIM hUuidHRS

uuid = 0x180D  //this is UUID for Heart Rate Service
hUuidHRS = BleHandleUuid16(uuid)
IF hUuidHRS == 0 THEN
    PRINT "\nFailed to create a handle"
ELSE
    PRINT "Handle for HRS Uuid is "; integer.h' hUuidHRS;"(";hUuidHRS;")"
ENDIF
```

Expected Output:

```
Handle for HRS Uuid is FE01180D (-33482739)
```

BLEHANDLEUUID16 is an extension function.

BleHandleUuid128

FUNCTION

This function takes a 16 byte string and converts it into a 32 bit integer handle. The handle consists of a 16 bit (2 byte) offset into a new 128 bit base UUID.

The base UUID is basically created by taking the 16 byte input string and setting bytes 12 and 13 to zero after extracting those bytes and storing them in the handle object. The handle also contains an index into an array of these 16 byte base UUIDs which are managed opaquely in the underlying stack.

The returned handle shall be treated by the developer as an opaque entity and no further logic shall be based on the bit content. However, note that a string of zeroes represents an invalid UUID handle.

Please ensure that you use a 16 byte UUID that has been generated using a random number generator with sufficient entropy to minimise duplication, as stated in an earlier section and that the first byte of the array is the most significant byte of the UUID.

BLEHANDLEUUID128 (stUuid$)

Returns	INTEGER, A handle representing the shorthand UUID. If zero, which is an invalid UUID handle, there is either no spare RAM memory to save the 16 byte base or more than 253 custom base UUIDs have been registered.
Arguments	
stUuid$	byRef *stUuid$* AS STRING Any 16 byte string that was generated using a UUID generation utility that has enough entro to ensure that it is random. The first byte of the string is the MSB of the UUID – that is, big endian format.
Interactive Command	No

```
//Example :: BleHandleUuid128.sb (See in BL600CodeSnippets.zip)
DIM uuid$ : hUuidCustom

//create a custom uuid for my ble widget
uuid$ = "ced9d91366924a1287d56f2764762b2a"
uuid$ = StrDehexize$(uuid$)
hUuidCustom = BleHandleUuid128(uuid$)
IF hUuidCustom == 0 THEN
    PRINT "\nFailed to create a handle"
ELSE
    PRINT "Handle for custom Uuid is ";integer.h' hUuidCustom; "(";hUuidCustom;")"
ENDIF
// hUuidCustom now references an object which points to
// a base uuid = ced9d91366924a1287d56f2747622b2a  (note 0's in byte position 2/3)
// and an offset = 0xd913
```

Expected Output:

```
Handle for custom Uuid is FC03D913 (-66856685)
```

BLEHANDLEUUID128 is an extension function.

BleHandleUuidSibling

FUNCTION

This function takes an integer in the range 0 to 65535 along with a UUID handle which had been previously created using BleHandleUuid16() or BleHandleUuid128() to create a new UUID handle. This handle references the same 128 base UUID as the one referenced by the UUID handle supplied as the input parameter.

The returned handle shall be treated by the developer as an opaque entity and no further logic shall be based on the bit content, apart from all 0's which represents an invalid UUID handle.

BLEHANDLEUUIDSIBLING (nUuidHandle, nUuid16)

Returns	INTEGER, a handle representing the shorthand UUID and can be zero which is an invalid UUID handle, if nUuidHandle is an invalid handle in the first place.
Arguments	
nUuidHandle	byVal *nUuidHandle* AS INTEGER A handle that was previously created using either BleHandleUui16() or BleHandleUuid128().
nUuid16	byVal *nUuid16* AS INTEGER A UUID value in the range 0 t0 65535 which will be treated as an offset into the 128 bit base UUID referenced by nUuidHandle.
Interactive Command	No

```
//Example :: BleHandleUuidSibling.sb (See in BL600CodeSnippets.zip)
DIM uuid$ ,hUuid1, hUuid2    //hUuid2 will have the same base uuid as hUuid1

//create a custom uuid for my ble widget
uuid$ = "ced9d91366924a1287d56f2764762b2a"
uuid$ = StrDehexize$(uuid$)
hUuid1 = BleHandleUuid128(uuid$)
IF hUuid1 == 0 THEN
    PRINT "\nFailed to create a handle"
ELSE
    PRINT "Handle for custom Uuid is ";integer.h' hUuid1;"(";hUuid1;")"
ENDIF
// hUuid1 now references an object which points to
// a base uuid = ced9000066924a1287d56f2747622b2a  (note 0's in byte position 2/3)
// and an offset = 0xd913

hUuid2 = BleHandleUuidSibling(hUuid1,0x1234)
IF hUuid2 == 0 THEN
    PRINT "\nFailed to create a handle"
ELSE
    PRINT "\nHandle for custom sibling Uuid is ";integer.h';hUuid2;"(";hUuid2;")"
ENDIF
// hUuid2 now references an object which also points to
// the base uuid = ced9000066924a1287d56f2700004762 (note 0's in byte position 2/3)
// and has the offset = 0x1234
```

Expected Output:

```
Handle for custom Uuid is FC03D913 (-66856685)
Handle for custom sibling Uuid is FC031234 (-66907596)
```

BLEHANDLEUUIDSIBLING is an extension function.

BleSvcCommit

This function is now deprecated, use BleServiceNew() & BleServiceCommt() instead.

BleServiceNew

FUNCTION

As explained in an earlier section, a Service in the context of a GATT table is a collection of related Characteristics. This function is used to inform the underlying GATT table manager that one or more related characteristics are going to be created and installed in the GATT table and that until the next call of this function they shall be associated with the service handle that it provides upon return of this call.

Under the hood, this call results in a single attribute being installed in the GATT table with a type signifying a PRIMARY or a SECONDARY service. The value for this attribute shall be the UUID that will identify this service and in turn have been precreated using one of the functions; BleHandleUuid16(), BleHandleUuid128() or BleHandleUuidSibling().

Note: When a GATT Client queries a GATT Server for services over a BLE connection, it will only get a list of PRIMARY services. SECONDARY services are a mechanism for multiple PRIMARY services to reference single instances of shared Characteristics that are collected in a SECONDARY service. This referencing is expedited within the definition of a service using the concept of 'INCLUDED SERVICE' which itself is just an attribute that is grouped with the PRIMARY service definition. An 'Included Service' is expedited using the function BleSvcAddIncludeSvc() which is described immediately after this function.

This function now replaces BleSvcCommit() and marks the beginning of a service definition in the gatt server table. When the last descriptor of the last characteristic has been registered the service definition should be terminated by calling BleServiceCommit().

BLESERVICENEW (nSvcType, nUuidHandle, hService)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
nSvcType	byVal *nSvcType* AS INTEGER This will be 0 for a SECONDARY service and 1 for a PRIMARY service and all other values are reserved for future use and will result in this function failing with an appropriate result code.
nUuidHandle	byVal *nUuidHandle* AS INTEGER This is a handle to a 16 bit or 128 bit UUID that identifies the type of Service function provided by all the Characteristics collected under it. It will have been pre-created using one of the three functions: BleHandleUuid16(), BleHandleUuid128() or BleHandleUuidSibling()
hService	byRef hService AS INTEGER If the Service attribute is created in the GATT table then this will contain a composite handle which references the actual attribute handle. This is then subsequently used when adding Characteristics to the GATT table. If the function fails to install the Service attribute for any reason this variable will contain 0 and the returned result code will be non-zero.
Interactive Command	No

```
//Example :: BleServiceNew.sb (See in BL600CodeSnippets.zip)
```

```
#DEFINE BLE_SERVICE_SECONDARY                           0
#DEFINE BLE_SERVICE_PRIMARY                             1

//---------------------------------------------------------------------------
//Create a Health Thermometer PRIMARY service attribute which has a uuid of 0x1809
//---------------------------------------------------------------------------
DIM hHtsSvc    //composite handle for hts primary service
DIM hUuidHT : hUuidHT = BleHandleUuid16(0x1809)     //HT Svc UUID Handle

IF BleServiceNew(BLE_SERVICE_PRIMARY,hUuidHT,hHtsSvc)==0 THEN
    PRINT "\nHealth Thermometer Service attribute written to GATT table"
    PRINT "\nUUID Handle value: ";hUuidHT
    PRINT "\nService Attribute Handle value: ";hHtsSvc
ELSE
    PRINT "\nService Commit Failed"
ENDIF


//---------------------------------------------------------------------------
//Create a Battery PRIMARY service attribute which has a uuid of 0x180F
//---------------------------------------------------------------------------
DIM hBatSvc    //composite handle for battery primary service
              //or we could have reused nHtsSvc
DIM hUuidBatt : hUuidBatt = BleHandleUuid16(0x180F)     //Batt Svc UUID Handle

IF BleServiceNew(BLE_SERVICE_PRIMARY,hUuidBatt,hBatSvc)==0 THEN
    PRINT "\n\nBattery Service attribute written to GATT table"
    PRINT "\nUUID Handle value: ";hUuidBatt
    PRINT "\nService Attribute Handle value: ";hBatSvc
ELSE
    PRINT "\nService Commit Failed"
ENDIF
```

Expected Output:

```
Health Thermometer Service attribute written to GATT table
UUID Handle value: -33482743
Service Attribute Handle value: 16

Battery Service attribute written to GATT table
UUID Handle value: -33482737
Service Attribute Handle value: 17
```

BLESERVICENEW is an extension function.

BleServiceCommit

This function in the BL600 is a dummy function and does not do anything. However, for portability to other Laird 4.0 compatible modules, always invoke this function after the last descriptor of the last characteristic of a service has been committed to the gatt server.

BLESERVICECOMMIT (hService)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
hService	byVal hService AS INTEGER This handle will have been returned from BleServiceNew().

BleSvcAddIncludeSvc

FUNCTION

Note: This function is currently not available for use on this module

This function is used to add a reference to a service within another service. This will usually, but not necessarily, be a SECONDARY service which is virtually identical to a PRIMARY service from the GATT Server perspective and the only difference is that when a GATT client queries a device for all services it does not get any mention of SECONDARY services.

When a GATT client encounters an INCLUDED SERVICE object when querying a particular service it shall perform a sub-procedure to get handles to all the characteristics that are part of that INCLUDED service.

This mechanism is provided to allow for a single set of Characteristics to be shared by multiple primary services. This is most relevant if a Characteristic is defined so that it can have only one instance in a GATT table but needs to be offered in multiple PRIMARY services. Hence a typical implementation, where a characteristic is part of many PRIMARY services, installs that Characteristic in a SECONDARY service (see BleSvcCommit()) and then uses the function defined in this section to add it to all the PRIMARY services that want to have that characteristic as part of their group.

It is possible to include a service which is also a PRIMARY or SECONDARY service, which in turn can include further PRIMARY or SECONDARY services. The only restriction to nested includes is that there cannot be recursion.

Further note that if a service has INCLUDED services, then they shall be installed in the GATT table immediately after a Service is created using BleSvcCommit() and before BleCharCommit(). The BT 4.0 specification mandates that any 'included service' attribute be present before any characteristic attributes within a particular service group declaration.

BleSvcAddIncludeSvc (hService)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
hService	byVal hService AS INTEGER This argument will contain a handle that was previously created using the function BleSvcCommit().
Interactive Command	No

```
//Example :: BleSvcAddIncludeSvc.sb (See in BL600CodeSnippets.zip)
#define BLE_SERVICE_SECONDARY                           0
#define BLE_SERVICE_PRIMARY                             1

//--------------------------------------------------------------------------
//Create a Battery SECONDARY service attribure which has a uuid of 0x180F
//--------------------------------------------------------------------------
dim hBatSvc    //composite handle for batteru primary service
dim rc              //or we could have reused nHtsSvc
dim metaSuccess
DIM charMet : charMet = BleAttrMetaData(1,1,10,1,metaSuccess)
DIM s$ : s$ = "Hello"       //initial value of char in Battery Service
DIM hBatChar

rc = BleServiceNew(BLE_SERVICE_SECONDARY, BleHandleUuid16(0x180F), hBatSvc)
rc = BleCharNew(3,BleHandleUuid16(0x2A1C),charMet,0,0)
rc = BleCharCommit(hBatSvc, s$ ,hBatChar)
```

```
rc = BleServiceCommit(hBatSvc)

//------------------------------------------------------------------------
//Create a Health Thermometer PRIMARY service attribure which has a uuid of 0x1809
//------------------------------------------------------------------------
DIM hHtsSvc    //composite handle for hts primary service

rc = BleServiceNew(BLE_SERVICE_PRIMARY, BleHandleUuid16(0x1809), hHtsSvc)
rc = BleServiceCommit(hHtsSvc)

//Have to add includes before any characteristics are committed
PRINT INTEGER.h'BleSvcAddIncludeSvc(hBatSvc)
```

BleSvcAddIncludeSvc is an extension function.

BleAttrMetadata

FUNCTION

A GATT Table is an array of attributes which are grouped into Characteristics which in turn are further grouped into Services. Each attribute consists of a data value which can be anything from 1 to 512 bytes long according to the specification and properties such as read and write permissions, authentication and security properties. When Services and Characteristics are added to a GATT server table, multiple attributes with appropriate data and properties get added.

This function allows a 32 bit integer to be created, which is an opaque object, which defines those properties and is then submitted along with other information to add the attribute to the GATT table.

When adding a Service attribute (not the whole service, in this present context), the properties are defined in the BT specification so that it is open for reads without any security requirements but cannot be written and always has the same data content structure. This implies that a metadata object does NOT need to be created.

However, when adding Characteristics, which consists of a minimum of 2 attributes, one similar in function as the aforementioned Service attribute and the other the actual data container, then properties for the **value attribute** must be specified. Here, 'properties' refers to properties for the attribute, not properties for the Characteristic container as a whole. These also exist and must be specified, but that is done in a different manner as explained later.

For example, the value attribute must be specified for read / write permission and whether it needs security and authentication to be accessed.

If the Characteristic is capable of notification and indication, the client implicitly must be able to enable or disable that. This is done through a Characteristic Descriptor which is also another attribute. The attribute will also need to have a metadata supplied when the Characteristic is created and registered in the GATT table. This attribute, if it exists, is called a Client Characteristic Configuration Descriptor or CCCD for short. A CCCD always has 2 bytes of data and currently only 2 bits are used as on/off settings for notification and indication.

A Characteristic can also optionally be capable of broadcasting its value data in advertisements. For the GATT client to be able to control this, there is yet another type of Characteristic Descriptor which also needs a metadata object to be supplied when the Characteristic is created and registered in the GATT table. This attribute, if it exists, is called a Server Characteristic Configuration Descriptor or SCCD for short. A SCCD always has 2 bytes of data and currently only 1 bit is used as on/off settings for broadcasts.

Finally if the Characteristic has other Descriptors to qualify its behaviour, a separate API function is also supplied to add that to the GATT table and when setting up a metadata object will also need to be supplied.

In a nutshell, think of a metadata object as a note to define how an attribute will behave and the GATT table manager will need that before it is added. Some attributes have those 'notes' specified by the BT specification and so the GATT table manager will not need to be provided with any, but the rest require it.

This function helps write that metadata.

BLEATTRMETADATA (nReadRights, nWriteRights, nMaxDataLen, flsVariableLen, resCode)

Returns	INTEGER, a 32 bit opaque data object to be used in subsequent calls when adding Characteristics to a GATT table.
Arguments	
nReadRights	byVal *nReadRights* AS INTEGER This specifies the read rights and shall have one of the following values: 0 No Access 1 Open 2 Encrypted with No Man-In-The-Middle (MITM) Protection 3 Encrypted with Man-In-The-Middle (MITM) Protection 4 Signed with No Man-In-The-Middle (MITM) Protection (not available) 5 Signed with Man-In-The-Middle (MITM) Protection (not available)
	Note: In early releases of the firmware, 4 and 5 are not available.
nWriteRights	byVal *nWriteRights* AS INTEGER This specifies the write rights and shall have one of the following values: 0 No Access 1 Open 2 Encrypted with No Man-In-The-Middle (MITM) Protection 3 Encrypted with Man-In-The-Middle (MITM) Protection 4 Signed with No Man-In-The-Middle (MITM) Protection (not available) 5 Signed with Man-In-The-Middle (MITM) Protection (not available)
	Note: In early releases of the firmware, 4 and 5 are not available.
nMaxDataLen	byVal *nMaxDataLen* AS INTEGER This specifies the maximum data length of the VALUE attribute. Range is from 1 to 512 bytes according to the BT specification; the stack implemented in the module may limit it for early versions. At the time of writing the limit is **20 bytes**.
flsVariableLen	byVal *flsVariableLen* AS INTEGER Set this to non-zero only if you want the attribute to automatically shorten it's length according to the number of bytes written by the client. For example, if the initial length is 2 and the client writes only 1 byte, then if this is 0, then only the first byte gets updated and the rest remain unchanged. If this parameter is set to 1, then when a single byte is written the attribute will shorten it's length to accommodate. If the client tries to write more bytes than the initial maximum length, then the client will get an error response.
resCode	byRef resCode AS INTEGER This variable will be updated with result code which will be 0 if a metadata object was successfully returned by this call. Any other value implies a metadata object did not get created.
Interactive Command	No

```
//Example :: BleAttrMetadata.sb (See in BL600CodeSnippets.zip)

DIM mdVal    //metadata for value attribute of Characteristic
DIM mdCccd   //metadata for CCCD attribute of Characteristic
DIM mdSccd   //metadata for SCCD attribute of Characteristic
```

```
DIM rc

//++++
// Create the metadata for the value attribute in the characteristic
// and Heart Rate attribute has variable length
//++++

//There is always a Value attribute in a characteristic
mdVal=BleAttrMetadata(17,0,20,0,rc)
//There is a CCCD and SCCD in this characteristic
mdCccd=BleAttrMetadata(1,2,2,0,rc)
mdSccd=BleAttrMetadata(0,0,2,0,rc)

//Create the Characteristic object
IF BleCharNew(3,BleHandleUuid16(0x2A1C),mdVal,mdCccd,mdSccd)==0 THEN
    PRINT "\nSuccess"
ELSE
    PRINT "\nFailed"
ENDIF
```

Expected Output:

```
Success
```

BLEATTRMETADATA is an extension function.

BleCharNew

FUNCTION

When a Characteristic is to be added to a GATT table, multiple attribute 'objects' must be precreated. After they are all created successfully, they are committed to the GATT table in a single atomic transaction.

This function is the first function that SHALL be called to start the process of creating those multiple attribute 'objects'. It is used to select the Characteristic properties (which are distinct and different from attribute properties), the UUID to be allocated for it and then up to three metadata objects for the value attribute, and CCCD/SCCD Descriptors respectively.

BLECHARNEW (nCharProps,nUuidHandle,mdVal,mdCccd,mdSccd)

<table>
<tr><td>Returns</td><td>INTEGER, a result code.
Typical value: 0x0000 (indicates a successful operation)</td></tr>
<tr><td colspan="2">Arguments</td></tr>
<tr><td>nCharProps</td><td>byVal nCharProps AS INTEGER
This variable contains a bit mask to specify the following high level properties for the Characteristic that will get added to the GATT table:
<table>
<tr><th>Bit</th><th>Description</th></tr>
<tr><td>0</td><td>Broadcast capable (Sccd Descriptor has to be present)</td></tr>
<tr><td>1</td><td>Can be read by the client</td></tr>
<tr><td>2</td><td>Can be written by the client without response</td></tr>
<tr><td>3</td><td>Can be written</td></tr>
<tr><td>4</td><td>Can be Notifiable (Cccd Descriptor has to be present)</td></tr>
<tr><td>5</td><td>Can be Indicatable (Cccd Descriptor has to be present)</td></tr>
<tr><td>6</td><td>Can accept signed writes</td></tr>
<tr><td>7</td><td>Reliable writes</td></tr>
</table></td></tr>
</table>

nUuidHandle	byVal *nUuidHandle* AS INTEGER This specifies the UUID that will be allocated to the Characteristic, either 16 or 128 bits. This variable is a handle, pre-created using one of the following functions: BleHandleUuid16() , BleHandleUuid128() , BleHandleUuidSibling().
mdVal	byVal *mdVal* AS INTEGER This is the mandatory metadata that is used to define the properties of the Value attribute that will be created in the Characteristic and will have been pre-created using the help of the function BleAttrMetadata().
mdCccd	byVal *mdCccd* AS INTEGER This is an optional metadata that is used to define the properties of the CCCD Descriptor attribute that will be created in the Characteristic and will have been pre-created using the help of the function BleAttrMetadata() or set to 0 if CCCD is not to be created. If nCharProps specifies that the Characteristic is notifiable or indicatable and this value contains 0, this function will abort with an appropriate result code.
mdSccd	byVal *mdSccd* AS INTEGER This is an optional metadata that is used to define the properties of the SCCD Descriptor attribute that will be created in the Characteristic and will have been pre-created using the help of the function BleAttrMetadata() or set to 0 if SCCD is not to be created. If nCharProps specifies that the Characteristic is broadcastable and this value contains 0, this function will abort with an appropriate resultcode.
Interactive Command	No

```
// Example :: BleCharNew.sb (See in BL600CodeSnippets.zip)
DIM rc
DIM charUuid : charUuid = BleHandleUuid16(2)        //Characteristic's UUID
DIM mdVal : mdVal = BleAttrMetadata(1,0,20,0,rc)    //Metadata for value attribute
DIM mdCccd : mdCccd = BleAttrMetadata(1,1,2,0,rc)   //Metadata for CCCD attribute of
Characteristic

//================================================================================
// Create a new char:
// --- Indicatable, not Broadcastable (so mdCccd is included, but not mdSccd)
// --- Can be read, not written (shown in mdVal as well)
//================================================================================
IF BleCharNew(0x22,charUuid,mdVal,mdCccd,0)==0 THEN
    PRINT "\nNew Characteristic created"
ELSE
    PRINT "\nFailed"
ENDIF
```

Expected Output:

```
New Characteristic created
```

BLECHARNEW is an extension function.

BleCharDescUserDesc

FUNCTION

This function adds an optional User Description Descriptor to a Characteristic and can *only* be called after BleCharNew() has started the process of describing a new Characteristic.

The BT 4.0 specification describes the User Description Descriptor as ".. a UTF-8 string of variable size that is a textual description of the characteristic value." It further stipulates that this attribute is optionally writable and so a metadata argument exists to configure it to be so. The metadata automatically updates the "Writable Auxilliaries" properties flag for the Characteristic. This is why that flag bit is NOT specified for the nCharProps argument to the BleCharNew() function.

BLECHARDESCUSERDESC(userDesc$, mdUser)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
userDesc$	byRef *userDesc$* AS STRING The user description string to initiliase the Descriptor with. If the length of the string exceeds the maximum length of an attribute then this function will abort with an error result code.
mdUser	byVal *mdUser* AS INTEGER This is a mandatory metadata that defines the properties of the User Description Descriptor attribute created in the Characteristic and will have been pre-created using the help of BleAttrMetadata(). If the write rights are set to 1 or greater, the attribute will be marked as writable and the client will be able to provide a user description that overwrites the one provided in this call.
Interactive Command	No

```
//Example :: BleCharDescUserDesc.sb (See in BL600CodeSnippets.zip)
DIM rc, metaSuccess,usrDesc$ : usrDesc$="A description"
DIM charUuid : charUuid = BleHandleUuid16(1)
DIM charMet : charMet = BleAttrMetaData(1,1,20,0,metaSuccess)
DIM mdUsrDsc : mdUsrDsc = BleAttrMetaData(1,1,20,0,metaSuccess)
DIM mdSccd : mdSccd = BleAttrMetadata(1,1,2,0,rc)    //CCCD metadata for char

//initialise char, write/read enabled, accept signed writes, indicatable
rc=BleCharNew(0x4B,charUuid,charMet,0,mdSccd)
rc=BleCharDescUserDesc(usrDesc$,mdUsrDsc)

IF rc==0 THEN
    PRINT "\nChar created and User Description '";usrDesc$;"' added"
ELSE
    PRINT "\nFailed"
ENDIF
```

Expected Output:

```
Char created and User Description 'A description' added
```

BLECHARDESCUSERDESC is an extension function.

BleCharDescPrstnFrmt

FUNCTION

This function adds an optional Presentation Format Descriptor to a Characteristic and can *only* be called after BleCharNew() has started the process of describing a new Characteristic. It adds the descriptor to the gatt table with open read permission and no write access, which means a metadata parameter is not required.

The BT 4.0 specification states that one or more presentation format descriptors can occur in a Characteristic and that, if more than one, then an Aggregate Format description is also included.

The book "Bluetooth Low Energy: The Developer's Handbook" by Robin Heydon, says the following on the subject of the Presentation Format Descriptor:

> *"One of the goals for the Generic Attribute Profile was to enable generic clients. A generic client is defined as a device that can read the values of a characteristic and display them to the user without understanding what they mean…*
> *The most important aspect that denotes if a characteristic can be used by a generic client is the Characteristic Presentation Format descriptor. If this exists, it's possible for the generic client to display its value, and it is safe to read this value."*

BLECHARDESCPRSTNFRMT (nFormat,nExponent,nUnit,nNameSpace,nNSdesc)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
nFormat	byVal *nFormat* AS INTEGER Valid range 0 to 255. The format specifies how the data in the Value attribute is structured. A list of valid values for this argument is found at http://developer.bluetooth.org/gatt/Pages/FormatTypes.aspx and the enumeration is described in the BT 4.0 spec, section 3.3.3.5.2. At the time of writing, the enumeration list is as follows:
nExponent	byVal *nExponent* AS INTEGER **Valid range**: -128 to 127 This value is used with integer data types given by the enumeration in nFormat to further qualify the value so that the actual value is: ***actual value = Characteristic Value * 10 to the power of nExponent.***
nUnit	byVal *nUnit* AS INTEGER **Valid range**: 0 to 65535. This value is a 16 bit UUID used as an enumeration to specify the units which are listed in the Assigned Numbers document published by the Bluetooth SIG, found at: http://developer.bluetooth.org/gatt/units/Pages/default.aspx
nNameSpace	byVal *nNameSpace* AS INTEGER **Valid range**: 0 to 255.

0x00	RFU	0x01	boolean
0x02	2bit	0x03	nibble
0x04	uint8	0x05	uint12
0x06	uint16	0x07	uint24
0x08	uint32	0x09	uint48
0x0A	uint64	0x0B	uint128
0x0C	sint8	0x0D	sint12
0x0E	sint16	0x0F	sint24
0x10	sint32	0x11	sint48
0x12	sint64	0x13	sint128
0x14	float32	0x15	float64
0x16	SFLOAT	0x17	FLOAT
0x18	duint16	0x19	utf8s
0x1A	utf16s	0x1B	struct
0x1C-0xFF	RFU		

	The value identifies the organization, defined in the Assigned Numbers document published by the Bluetooth SIG, found at: https://developer.bluetooth.org/gatt/Pages/GattNamespaceDescriptors.aspx
nNSdesc	byVal *nNSdesc* AS INTEGER Valid range: 0 to 65535. This value is a description of the organisation specified by nNameSpace.
Interactive Command	No

```
//Example :: BleCharDescPrstnFrmt.sb (See in BL600CodeSnippets.zip)

DIM rc, metaSuccess,usrDesc$ : usrDesc$="A description"
DIM charUuid : charUuid = BleHandleUuid16(1)
DIM charMet : charMet = BleAttrMetaData(1,1,20,0,metaSuccess)
DIM mdUsrDsc : mdUsrDsc = BleAttrMetaData(1,1,20,0,metaSuccess)
DIM mdSccd : mdSccd = BleAttrMetadata(1,1,2,0,rc)    //CCCD metadata for char

//initialise char, write/read enabled, accept signed writes, indicatable
rc=BleCharNew(0x4B,charUuid,charMet,0,mdSccd)
rc=BleCharDescUserDesc(usrDesc$,mdUsrDsc)

IF rc==0 THEN
    PRINT "\nChar created and User Description '";usrDesc$;"' added"
ELSE
    PRINT "\nFailed"
ENDIF

// ~ ~ ~
// other optional descriptors
// ~ ~ ~

// 16 bit signed integer = 0x0E
// exponent = 2
// unit = 0x271A ( amount concentration (mole per cubic metre) )
// namespace = 0x01 == Bluetooth SIG
// description = 0x0000 == unknown
IF BleCharDescPrstnFrmt(0x0E,2,0x271A,0x01,0x0000)==0 THEN
    PRINT "\nPresentation Format Descriptor added"
ELSE
    PRINT "\nPresentation Format Descriptor not added"
ENDIF
```

Expected Output:

```
Char created and User Description 'A description' added
Presentation Format Descriptor added
```

BLECHARDESCPRSTNFRMT is an extension function.

BleCharDescAdd

Note: This function has a bug for firmware versions prior to 1.4.X.Y

FUNCTION

This function is used to add any Characteristic Descriptor as long as its UUID is not in the range 0x2900 to 0x2904 inclusive as they are treated specially using dedicated API functions. For example, 0x2904 is the Presentation Format Descriptor and it is catered for by the API function BleCharDescPrstnFrmt().

Since this function allows existing / future defined Descriptors to be added that may or may not have write access or require security requirements, a metadata object must be supplied allowing that to be configured.

BLECHARDESCADD (nUuid16, attr$, mdDesc)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
nUuid16	byVal *nUuid16* AS INTEGER **Value range:** 0x2905 to 0x2999 **Note:** This is the actual UUID value, NOT the handle.. The highest value at the time of writing is 0x2908, defined for the Report Reference Descriptor. See http://developer.bluetooth.org/gatt/descriptors/Pages/DescriptorsHomePage.aspx for a of Descriptors defined and adopted by the Bluetooth SIG.
attr$	byRef *attr$* AS STRING This is the data that will be saved in the Descriptor's attribute
mdDesc	byVal *n* AS INTEGER This is mandatory metadata that is used to define the properties of the Descriptor attribute that will be created in the Characteristic and will have been pre-created using the help of the function BleAttrMetadata(). If the write rights are set to 1 or greater, then the attribute is marked as writable and so the client will be able to modify the attribute value.
Interactive Command	No

```
//Example :: BleCharDescAdd.sb (See in BL600CodeSnippets.zip)

DIM rc, metaSuccess,usrDesc$ : usrDesc$="A description"
DIM charUuid : charUuid = BleHandleUuid16(1)
DIM charMet : charMet = BleAttrMetaData(1,1,20,0,metaSuccess)
DIM mdUsrDsc : mdUsrDsc = charMet
DIM mdSccd : mdSccd = charMet

//initialise char, write/read enabled, accept signed writes, indicatable
rc=BleCharNew(0x4B,charUuid,charMet,0,mdSccd)
rc=BleCharDescUserDesc(usrDesc$,mdUsrDsc)
rc=BleCharDescPrstnFrmt(0x0E,2,0x271A,0x01,0x0000)

// ~ ~ ~
// other descriptors
// ~ ~ ~

//++++
//Add the other Descriptor 0x29XX  -- first one
//++++
DIM mdChrDsc : mdChrDsc = BleAttrMetadata(1,0,20,0,metaSuccess)
DIM attr$ : attr$="some_value1"
rc=BleCharDescAdd(0x2905,attr$,mdChrDsc)

//++++
//Add the other Descriptor 0x29XX  -- second one
```

```
//++++
attr$="some_value2"
rc=rc+BleCharDescAdd(0x2906,attr$,mdChrDsc)

//++++
//Add the other Descriptor 0x29XX  -- last one
//++++
attr$="some_value3"
rc=rc+BleCharDescAdd(0x2907,attr$,mdChrDsc)

IF rc==0 THEN
    PRINT "\nOther descriptors added successfully"
ELSE
    PRINT "\nFailed"
ENDIF
```

Expected Output:

```
Other descriptors added successfully
```

BLECHARDESCADD is an extension function.

BleCharCommit

FUNCTION

This function commits a Characteristic which was prepared by calling BleCharNew() and optionally BleCharDescUserDesc(), BleCharDescPrstnFrmt(), or BleCharDescAdd().

It is an instruction to the GATT table manager that all relevant attributes that make up the Characteristic should appear in the GATT table in a single atomic transaction. If it successfully created, a single composite Characteristic handle is returned which should not be confused with GATT table attribute handles. If the Characteristic was not accepted then this function returns a non-zero result code which conveys the reason; and the handle argument that is returned has a special invalid handle of 0.

The characteristic handle that is returned refers to an internal opaque object that is a linked list of all the attribute handles in the Characteristic. This implies that there is a minimum of one (for the characteristic value attribute) and more as appropriate. For example, if the Characteristic's property specified is notifiable, then a single CCCD attribute also exists.

Note: In the GATT table, when a Characteristic is registered there are actually a minimum of two attribute handles, one for the Characteristic Declaration and the other for the Value. However there is no need for the *smart*BASIC apps developer to ever access it, so it is not exposed. Access is not required because the Characteristic was created by the application developer and so shall already know its content – which never changes once created.

BLECHARCOMMIT (hService,attr$,charHandle)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
hService	byVal hService AS INTEGER This is the handle of the service that this Characteristic shall belong to, which in turn was created using the function BleSvcCommit().

attr$	byRef *attr$* AS STRING This string contains the initial value of the Value attribute in the Characteristic. The content of this string is copied into the GATT table and so the variable can be reused after this function returns.
charHandle	byRef *charHandle* AS INTEGER The composite handle for the newly created Characteristic is returned in this argument. It is zero if the function fails with a non-zero result code. This handle is then used as an argument in subsequent function calls to perform read/write actions, so it is must be placed in a global smartBASIC variable. When a significant event occurs as a result of action by a remote client, an event message is sent to the application which can be serviced using a handler. That message contains a handle field corresponding to this composite characteristic handle. Standard procedure is to 'select' on that value to determine which Characteristic the message is intended for. See event messages: EVCHARHVC, EVCHARVAL, EVCHARCCCD, EVCHARSCCD, EVCHARDESC.
Interactive Command	No

```
// Example :: BleCharCommit.sb (See in BL600CodeSnippets.zip)

#DEFINE BLE_SERVICE_SECONDARY                           0
#DEFINE BLE_SERVICE_PRIMARY                             1

DIM rc
DIM attr$,usrDesc$ : usrDesc$="A description"
DIM hHtsSvc    //composite handle for hts primary service
DIM mdCharVal : mdCharVal = BleAttrMetaData(1,1,20,0,rc)
DIM mdCccd : mdCccd = BleAttrMetadata(1,1,2,0,rc)
DIM mdUsrDsc : mdUsrDsc = BleAttrMetaData(1,1,20,0,rc)
DIM hHtsMeas    //composite handle for htsMeas characteristic

//-------------------------------------------------------------------------
//Create a Health Thermometer PRIMARY service attribute which has a uuid of 0x1809
//-------------------------------------------------------------------------
rc=BleServiceNew(BLE_SERVICE_PRIMARY, BleHandleUuid16(0x1809), hHtsSvc)

//-------------------------------------------------------------------------
//Create the Measurement Characteristic object, add user description descriptor
//-------------------------------------------------------------------------
rc=BleCharNew(0x2A,BleHandleUuid16(0x2A1C),mdCharVal,mdCccd,0)
rc=BleCharDescUserDesc(usrDesc$,mdUsrDsc)

//-------------------------------------------------------------------------
//Commit the characteristics with some initial data
//-------------------------------------------------------------------------
attr$="hello\00worl\64"
IF BleCharCommit(hHtsSvc,attr$,hHtsMeas)==0 THEN
    PRINT "\nCharacteristic Commited"
ELSE
    PRINT "\nFailed"
ENDIF
rc=BleServiceCommit(hHtsSvc)

//the characteristic will now be visible in the GATT table
//and is refrenced by 'hHtsMeas' for subsequent calls
```

Expected Output:

```
Characteristic Commited
```

BLECHARCOMMIT is an extension function.

BleCharValueRead

FUNCTION

This function reads the current content of a characteristic identified by a composite handle that was previously returned by the function BleCharCommit().

In most cases a read will be performed when a GATT client writes to a characteristic value attribute. The write event is presented asynchronously to the *smart*BASIC application in the form of EVCHARVAL event and so this function will most often be accessed from the handler that services that event.

BLECHARVALUEREAD (charHandle,attr$)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
charHandle	byVal *charHandle* AS INTEGER This is the handle to the characteristic whose value must be read which was returned when BleCharCommit() was called.
attr$	byRef attr$ AS STRING This string variable contains the new value from the characteristic.
Interactive Command	No

```
//Example :: BleCharValueRead.sb (See in BL600CodeSnippets.zip)
DIM hMyChar,rc, conHndl

//============================================================================
// Initialise and instantiate service, characteristic,
//============================================================================
FUNCTION OnStartup()
    DIM rc, hSvc, scRpt$, adRpt$, addr$, attr$ : attr$="Hi"

    //commit service
    rc=BleServiceNew(1, BleHandleUuid16(0x18EE), hSvc)
    //initialise char, write/read enabled, accept signed writes
    rc=BleCharNew(0x0A,BleHandleUuid16(1),BleAttrMetaData(1,1,20,0,rc),0,0)
    //commit char initialised above, with initial value "hi" to service 'hSvc'
    rc=BleCharCommit(hSvc,attr$,hMyChar)
    //commit changes to service
    rc=BleServiceCommit(hSvc)
    //initialise scan report
    rc=BleScanRptInit(scRpt$)
    //Add 1 service handle to scan report
    rc=BleAdvRptAddUuid16(scRpt$,hSvc,-1,-1,-1,-1,-1)
    //commit reports to GATT table - adRpt$ is empty
    rc=BleAdvRptsCommit(adRpt$,scRpt$)
    rc=BleAdvertStart(0,addr$,150,0,0)
ENDFUNC rc
```

```
//================================================================================
// New char value handler
//================================================================================
FUNCTION HndlrChar(BYVAL chrHndl, BYVAL offset, BYVAL len)
    dim s$
    IF chrHndl == hMyChar THEN
        PRINT "\n";len;" byte(s) have been written to char value attribute from
offset ";offset

        rc=BleCharValueRead(hMyChar,s$)
        PRINT "\nNew Char Value: ";s$
    ENDIF
    rc=BleAdvertStop()
    rc=BleDisconnect(conHndl)
ENDFUNC 0

//================================================================================
// Get the connnection handle
//================================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtn)
    conHndl=nCtn
ENDFUNC 1

IF OnStartup()==0 THEN
    DIM at$ : rc = BleCharValueRead(hMyChar,at$)
    PRINT "\nCharacteristic value attribute: ";at$;"\nConnect to BL600 and send a new
value\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF

ONEVENT EVCHARVAL  CALL HndlrChar
ONEVENT EVBLEMSG  CALL HndlrBleMsg

WAITEVENT

PRINT "\nExiting..."
```

Expected Output:

```
Characteristic value attribute: Hi
Connect to BL600 and send a new value

New characteristic value: Laird
Exiting...
```

BLECHARVALUEREAD is an extension function.

BleCharValueWrite

Note: For firmware versions prior to 1.4.X.Y, the module must be in a connection for this function to work.

FUNCTION

This function writes new data into the VALUE attribute of a Characteristic, which is in turn identified by a composite handle returned by the function BleCharCommit().

BLECHARVALUEWRITE (charHandle,attr$)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
charHandle	byVal *charHandle* AS INTEGER This is the handle to the characteristic whose value must be updated which was return when BleCharCommit() was called.
attr$	byRef attr$ AS STRING String variable, contains new value to write to the characteristic.
Interactive Command	No

```
//Example :: BleCharValueWrite.sb (See in BL600CodeSnippets.zip)
DIM hMyChar,rc

//============================================================================
// Initialise and instantiate service, characteristic,
//============================================================================
FUNCTION OnStartup()
    DIM rc, hSvc, attr$ : attr$="Hi"

    //commit service
    rc = BleServiceNew(1, BleHandleUuid16(0x18EE), hSvc)
    //initialise char, write/read enabled, accept signed writes
    rc=BleCharNew(0x4A,BleHandleUuid16(1),BleAttrMetaData(1,1,20,0,rc),0,0)
    //commit char initialised above, with initial value "hi" to service 'hSvc'
    rc=BleCharCommit(hSvc,attr$,hMyChar)
    //commit changes to service
    rc = BleServiceCommit(hSvc)
ENDFUNC rc

//============================================================================
// Uart Rx handler - write input to characteristic
//============================================================================
FUNCTION HndlrUartRx()
    TimerStart(0,10,0)
ENDFUNC 1

//============================================================================
// Timer0 timeout handler
//============================================================================
FUNCTION HndlrTmr0()
    DIM t$ : rc=UartRead(t$)
    rc = BleCharValueWrite(hMyChar,t$)
    IF rc==0 THEN
        PRINT "\nNew characteristic value: ";t$
    ELSE
        PRINT "\nFailed to write new characteristic value ";integer.h'rc;"\n"
    ENDIF
ENDFUNC 0

IF OnStartup()==0 THEN
    DIM at$ : rc = BleCharValueRead(hMyChar,at$)
    PRINT "\nCharacteristic value attribute: ";at$;"\nType a new value\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF
```

```
ONEVENT EVUARTRX    CALL HndlrUartRx
ONEVENT EVTMR0     CALL HndlrTmr0

WAITEVENT

PRINT "\nExiting..."
```

Expected Output:

```
Characteristic value attribute: Hi
Send a new value
Laird

New characteristic value: Laird
Exiting...
```

BLECHARVALUEWRITE is an extension function.

BleCharValueNotify

FUNCTION

If there is BLE connection, this function writes new data into the VALUE attribute of a Characteristic so that it can be sent as a notification to the GATT client. The characteristic is identified by a composite handle that was returned by the function BleCharCommit().

A notification does not result in an acknowledgement from the client.

BLECHARVALUENOTIFY (charHandle,attr$)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
charHandle	byVal *charHandle* AS INTEGER This is the handle to the characteristic whose value must be updated which was return when BleCharCommit() was called.
attr$	byRef attr$ AS STRING String variable containing new value to write to the characteristic and then send as a notification to the client. If there is no connection, this function fails with an appropriate result code.
Interactive Command	No

```
//Example :: BleCharValueNotify.sb (See in BL600CodeSnippets.zip)
DIM hMyChar,rc,at$,conHndl

//============================================================================
// Initialise and instantiate service, characteristic, start adverts
//============================================================================
FUNCTION OnStartup()
    DIM rc, hSvc, at$, attr$, adRpt$, addr$, scRpt$
    attr$="Hi"
    DIM mdCccd : mdCccd = BleAttrMetadata(1,1,2,0,rc)    //CCCD metadata for char
```

```
    //Commit svc with handle 'hSvcUuid'
    rc=BleServiceNew(1, BleHandleUuid16(0x18EE), hSvc)
    //initialise char, write/read enabled, accept signed writes, notifiable
    rc=BleCharNew(0x12,BleHandleUuid16(1),BleAttrMetaData(1,0,20,0,rc),mdCccd,0)
    //commit char initialised above, with initial value "hi" to service 'hMyChar'
    rc=BleCharCommit(hSvc,attr$,hMyChar)
    //commit changes to service
    rc=BleServiceCommit(hSvc)
    rc=BleScanRptInit(scRpt$)
    //Add 1 service handle to scan report
    rc=BleAdvRptAddUuid16(scRpt$,hSvc,-1,-1,-1,-1,-1)
    //commit reports to GATT table - adRpt$ is empty
    rc=BleAdvRptsCommit(adRpt$,scRpt$)
    rc=BleAdvertStart(0,addr$,50,0,0)
ENDFUNC rc

//============================================================================
// Close connections so that we can run another app without problems
//============================================================================
SUB CloseConnections()
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
ENDSUB

//============================================================================
// Ble event handler
//============================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n--- Disconnected from client"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n--- Connected to client"
    ENDIF
ENDFUNC 1

//============================================================================
// CCCD descriptor written handler
//============================================================================
FUNCTION HndlrCharCccd(BYVAL charHandle, BYVAL nVal) AS INTEGER
    DIM value$
    IF charHandle==hMyChar THEN
        PRINT "\nCCCD Val: ";nVal
        IF nVal THEN
            PRINT " : Notifications have been enabled by client"
            value$="hello"
            IF BleCharValueNotify(hMyChar,value$)!=0 THEN
                PRINT "\nFailed to notify new value :";INTEGER.H'rc
            ELSE
                PRINT "\nSuccessful notification of new value"
                EXITFUNC 0
            ENDIF
        ELSE
            PRINT " : Notifications have been disabled by client"
        ENDIF
    ELSE
        PRINT "\nThis is for some other characteristic"
    ENDIF
ENDFUNC 1

ONEVENT  EVBLEMSG    CALL HndlrBleMsg
```

```
ONEVENT  EVCHARCCCD  CALL HndlrCharCccd

IF OnStartup()==0 THEN
    rc = BleCharValueRead(hMyChar,at$)
    PRINT "\nCharacteristic Value: ";at$
    PRINT "\nYou can connect and write to the CCCD characteristic."
    PRINT "\nThe BL600 will then notify your device of a new characteristic value\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF

WAITEVENT

CloseConnections()
PRINT "\nExiting..."
```

Expected Output:

```
Characteristic Value: Hi
You can connect and write to the CCCD characteristic.
The BL600 will then notify your device of a new characteristic value

--- Connected to client
CCCD Val: 0 : Notifications have been disabled by client
CCCD Val: 1 : Notifications have been enabled by client
Successful notification of new value
Exiting...
```

BLECHARVALUENOTIFY is an extension function.

BleCharValueIndicate

FUNCTION

If there is BLE connection this function is used to write new data into the VALUE attribute of a Characteristic so that it can be sent as an indication to the GATT client. The characteristic is identified by a composite handle returned by the function BleCharCommit().

An indication results in an acknowledgement from the client and that will be presented to the *smart*BASIC application as the EVCHARHVC event.

BLECHARVALUEINDICATE (charHandle,attr$)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
charHandle	byVal *charHandle* AS INTEGER This is the handle to the characteristic whose value must be updated which was return when BleCharCommit() was called.
attr$	byRef attr$ AS STRING String variable containing new value to write to the characteristic and then to send as a notification to the client. If there is no connection, this function fails with an appropriate result code.
Interactive Command	No

```
//Example :: BleCharValueIndicate.sb (See in BL600CodeSnippets.zip)
DIM hMyChar,rc,at$,conHndl
```

```
//================================================================================
// Initialise and instantiate service, characteristic, start adverts
//================================================================================
FUNCTION OnStartup()
    DIM rc, hSvc, at$, attr$, adRpt$, addr$, scRpt$
    attr$="Hi"
    DIM mdCccd : mdCccd = BleAttrMetadata(1,1,2,0,rc)   //CCCD metadata for char

    //Commit svc with handle 'hSvcUuid'
    rc=BleServiceNew(1, BleHandleUuid16(0x18EE), hSvc)
    //initialise char, write/read enabled, accept signed writes, notifiable
    rc=BleCharNew(0x22,BleHandleUuid16(1),BleAttrMetaData(1,0,20,0,rc),mdCccd,0)
    //commit char initialised above, with initial value "hi" to service 'hMyChar'
    rc=BleCharCommit(hSvc,attr$,hMyChar)
    //commit changes to service
    rc=BleServiceCommit(hSvc)
    rc=BleScanRptInit(scRpt$)
    //Add 1 service handle to scan report
    rc=BleAdvRptAddUuid16(scRpt$,hSvc,-1,-1,-1,-1,-1)
    //commit reports to GATT table - adRpt$ is empty
    rc=BleAdvRptsCommit(adRpt$,scRpt$)
    rc=BleAdvertStart(0,addr$,50,0,0)
ENDFUNC rc

//================================================================================
// Ble event handler
//================================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n--- Disconnected from client"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n--- Connected to client"
    ENDIF
ENDFUNC 1

//================================================================================
// CCCD descriptor written handler
//================================================================================
FUNCTION HndlrCharCccd(BYVAL charHandle, BYVAL nVal)
    DIM value$
    IF charHandle==hMyChar THEN
        PRINT "\nCCCD Val: ";nVal
        IF nVal THEN
            PRINT " : Indications have been enabled by client"
            value$="hello"
            rc=BleCharValueIndicate(hMyChar,value$)
            IF rc!=0 THEN
                PRINT "\nFailed to indicate new value :";INTEGER.H'rc
            ELSE
                PRINT "\nSuccessful indication of new value"
                EXITFUNC 1
            ENDIF
        ELSE
            PRINT " : Indications have been disabled by client"
        ENDIF
    ELSE
        PRINT "\nThis is for some other characteristic"
    ENDIF
ENDFUNC 1
```

```
//===============================================================================
// Indication Acknowledgement Handler
//===============================================================================
FUNCTION HndlrChrHvc(BYVAL charHandle)
    IF charHandle == hMyChar THEN
        PRINT "\n\nGot confirmation of recent indication"
    ELSE
        PRINT "\n\nGot confirmation of some other indication: ";charHandle
    ENDIF
ENDFUNC 0

ONEVENT  EVBLEMSG    CALL HndlrBleMsg
ONEVENT  EVCHARCCCD  CALL HndlrCharCccd
ONEVENT  EVCHARHVC   CALL HndlrChrHvc

IF OnStartup()==0 THEN
    rc = BleCharValueRead(hMyChar,at$)
    PRINT "\nCharacteristic Value: ";at$
    PRINT "\nYou can connect and write to the CCCD characteristic."
    PRINT "\nThe BL600 will then indicate a new characteristic value\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF

WAITEVENT

rc=BleDisconnect(conHndl)
rc=BleAdvertStop()
PRINT "\nExiting..."
```

Expected Output:

```
Characteristic Value: Hi
You can connect and write to the CCCD characteristic.
The BL600 will then indicate a new characteristic value

--- Connected to client
CCCD Val: 0 : Indications have been disabled by client
CCCD Val: 2 : Indications have been enabled by client
Successful indication of new value

Got confirmation of recent indication
Exiting...
```

BLECHARVALUEINDICATE is an extension function.

BleCharDescRead

FUNCTION

This function reads the current content of a writable Characteristic Descriptor identified by the two parameters supplied in the EVCHARDESC event message after a Gatt Client writes to it.

In most cases a local read is performed when a GATT client writes to a characteristic descriptor attribute. The write event is presented asynchronously to the *smart*BASIC application in the form of an EVCHARDESC event and so this function is most often accessed from the handler that services that event.

BLECHARDESCREAD (charHandle,nDescHandle,nOffset,nLength,nDescUuidHandle,attr$))

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
charHandle	byVal *charHandle* AS INTEGER This is the handle to the characteristic whose descriptor must be read which was returned when BleCharCommit() was called and will have been supplied in the EVCHARDESC event message.
nDescHandle	byVal *nDescHandle* AS INTEGER This is an index into an opaque array of descriptor handles inside the charHandle and will have been supplied as the second parameter in the EVCHARDESC event message.
nOffset	byVal *nOffset* AS INTEGER This is the offset into the descriptor attribute from which the data shoud be read and copied into attr$.
nLength	byVal *nLength* AS INTEGER This is the number of bytes to read from the descriptor attribute from offset nOffset and copied into attr$.
nDescUuidHandle	byRef *nDescUuidHandle* AS INTEGER On exit this will be updated with the uuid handle of the descriptor that got updated.
attr$	byRef attr$ AS STRING On exit this string variable contains the new value from the characteristic descriptor.
Interactive Command	No

```
//Example :: BleCharDescRead.sb (See in BL600CodeSnippets.zip)

DIM rc,conHndl,hMyChar

//---------------------------------------------------------------------------
//Create some PRIMARY service attribure which has a uuid of 0x18FF
//---------------------------------------------------------------------------
SUB OnStartup()
    DIM hSvc,attr$,scRpt$,adRpt$,addr$
    rc=BleSvcCommit(1,BleHandleUuid16(0x18FF),hSvc)
    // Add one or more characteristics
    rc=BleCharNew(0x0a,BleHandleUuid16(0x2AFF),BleAttrMetadata(1,1,20,1,rc),0,0)

    //Add a user description
    DIM s$ : s$="You can change this"
    rc=BleCharDescAdd(0x2999,s$,BleAttrMetadata(1,1,20,1,rc))

    //commit characteristic
    attr$="\00"  //no initial alert
    rc = BleCharCommit(hSvc,attr$,hMyChar)
    rc=BleScanRptInit(scRpt$)
    //Add 1 char handle to scan report
    rc=BleAdvRptAddUuid16(scRpt$,hMyChar,-1,-1,-1,-1,-1)
    //commit reports to GATT table - adRpt$ is empty
    rc=BleAdvRptsCommit(adRpt$,scRpt$)
    rc=BleAdvertStart(0,addr$,200,0,0)
ENDSUB
```

```
//=============================================================================
// Close connections so that we can run another app without problems
//=============================================================================
SUB CloseConnections()
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
ENDSUB

//=============================================================================
// Ble event handler - Just to get the connection handle
//=============================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    conHndl=nCtx
ENDFUNC 1

//=============================================================================
// Handler to service writes to descriptors by a gatt client
//=============================================================================
FUNCTION HandlerCharDesc(BYVAL hChar AS INTEGER, BYVAL hDesc AS INTEGER)
    DIM instnc,nUuid,a$, offset,duid

    IF hChar == hMyChar THEN
        rc = BleCharDescRead(hChar,hDesc,0,20,duid,a$)
        IF rc==0 THEN
            PRINT "\nRead 20 bytes from index ";offset;" in new char value."
            PRINT "\n  ::New Descriptor Data:  ";StrHexize$(a$);
            PRINT "\n  ::Length=";StrLen(a$)
            PRINT "\n  ::Descriptor UUID   ";integer.h' duid
            EXITFUNC 0
        ELSE
            PRINT "\nCould not access the uuid"
        ENDIF
    ELSE
        PRINT "\nThis is for some other characteristic"
    ENDIF
ENDFUNC 1



//install a handler for writes to characteristic values
ONEVENT  EVCHARDESC   CALL HandlerCharDesc
ONEVENT  EVBLEMSG  CALL HndlrBleMsg

OnStartup()
PRINT "\nWrite to the User Descriptor with UUID 0x2999"

//wait for events and messages
WAITEVENT

CloseConnections()
PRINT "\nExiting..."
```

Expected Output:

```
Write to the User Descriptor with UUID 0x2999
Read 20 bytes from index 0 in new char value.
  ::New Descriptor Data:  4C61697264
  ::Length=5
  ::Descriptor UUID   FE012999
Exiting...
```

BLECHARDESCREAD is an extension function.

GATT Client Functions

This section describes all functions related to GATT Client capability which enables interaction with GATT servers at the other end of the BLE connection. The Bluetooth Specification 4.0 and newer allows for a device to be a GATT server and/or GATT Client simultaneously and the fact that a peripheral mode device accepts a connection and in all use cases has a GATT server table does not preclude it from interacting with a GATT table in the central role device which is connected to it.

These GATT Client functions allow the developer to discover services, characteristics and descriptors, read and write to characteristics and descriptors and handle either notifications or indications.

To interact with a remote GATT server it is important to have a good understanding of how it is constructed and the best way is to see it as a table consisting of many rows and 3 visible columns (handle, type, value) and at least one more column which is not visible but the content will affect access to the data column.

16 bit Handle	Type (16 or 128 bit)	Value (1 to 512 bytes)	Permissions

These rows are grouped into collections called services and characteristics. The grouping is achieved by creating a row with Type = 0x2800 or 0x2801 for services (primary and secondary respectively) and 0x2803 for characteristics.

Basically, a table should be scanned from top to bottom and the specification stipulates that the 16 bit handle field SHALL contain values in the range 1 to 65535 and SHALL be in ascending order and gaps are allowed.

When scanning, if a row is encountered with the value 0x2800 or 0x2801 in the 'Type' column then it SHALL be understood as the start of a primary or secondary service which in turn SHALL contain at least one charactestic or one 'included service' which have Type=0x2803 and 0x2802 respectively.

When a row with Type = 0x2803, a characteristic, is encountered, then the next row shall contain the value for that characteristic and then after that there may be 0 or more descriptors.

This means each characteristic shall consist of at least 2 rows in the table, and if descriptors exist for that characteristic then a single row per descriptor.

Handle	Type	Value	Comments
0x0001	0x2800	UUID of the Service	Primary Service 1 Start
0x0002	0x2803	Properties, Value Handle, Value UUID1	Characteristic 1 Start
0x0003	Value UUID1	Value : 1 to 512 bytes	Actual data
0x0004	0x2803	Properties, Value Handle, Value UUID2	Characteristic 2 Start
0x0005	Value UUID2	Value : 1 to 512 bytes	Actual data
0x0006	0x2902	Value	Descriptor 1(CCCD)
0x0007	0x2903	Value	Descriptor 2 (SCCD)
0x0008	0x2800	UUID of the Service	Primary Service 2 Start
0x0009	0x2803	Properties, Value Handle, Value UUID3	Characteristic 1 Start
0x000A	Value UUID3	Value : 1 to 512 bytes	Actual data
0x000B	0x2800	UUID of the Service	Primary Service 3 Start
0x000C	0x2803	Properties, Value Handle, Value UUID3	Characteristic 3 Start
0x000D	Value UUID3	Value : 1 to 512 bytes	Actual data
0x000E	0x2902	Value	Descriptor 1(CCCD)

Handle	Type	Value	Comments
0x000F	0x2903	Value	Descriptor 2 (SCCD)
0x0010	0x2904	Value (presentation format data)	Descriptor 3
0x00111	0x2906	Value (valid range)	Descriptor 4 (Range)

A colour highlighted example of a GATT Server table is shown above which shows there are 3 services (at handles 0x0001,0x0008 and 0x000B) because there are 3 rows where the Type = 0x2803 and all rows up to the next instance of a row with Type=0x2800 or 2801 belong to that service.

In each group of rows for a service, you can see one or more characteristics where Type=0x2803. For example the service beginning at handle 0x0008 has one characteristic which contains 2 rows identified by handles 0x0009 and 0x000A and the actual value for the characteristic starting at 0x0009 is in the row identified by 0x000A.

Likewise, each characteristic starts with a row with Type=0x2803 and all rows following it up to a row with type = 0x2800/2801/2803 are considered belonging to that characteristic. For example see characteristic at row with handle = 0x0004 which has the mandatory value row and then 2 descriptors.

The Bluetooth specification allows for multiple instances of the same service or characteristics or descriptors and they are differentiated by the unique handle. Hence when a handle is known there is no ambiguity.

Each GATT Server table allocates the handle numbers, the only stipulation being that they be in ascending order (gaps are allowed). This is important to understand because two devices containing the same services and characteristic and in EXACTLY the same order may NOT allocate the same handle values, especially if one device increments handles by 1 and another with some other arbitrary random value. The specification DOES however stipulate that once the handle values are allocated they be fixed for all subsequent connections, unless the device exposes a GATT Service which allows for indications to the client that the handle order has changed and thus force it to flush it's cache and rescan the GATT table.

When a connection is first established, there is no prior knowledge as to which services exist and of their handles, so the GATT protocol which is used to interact with GATT servers provides procedures that allow for the GATT table to be scanned so that the client can ascertain which services are offered. This section describes smartBASIC functions which encapsulate and manage those procedures to enable a smartBASIC application to map the table.

These helper functions have been written to help gather the handles of all the rows which contain the value type for appropriate characteristics as those are the ones that will be read or written to. The smartBASIC internal engine also maintains data objects so that it is possible to interact with descriptors associated with the characteristic.

In a nutshell, the table scanning process will reveal characteristic handles (as handles of handles) and these are then used in other GATT client related smartBASIC functions to interact with the table to for example read/write or accept and process incoming notifications and indications.

This encapsulated approach is to ensure that the least amount of RAM resource is required to implement a GATT Client and given that these procedures operate at speeds many orders of magnitude slower compared to the speed of the cpu and energy consumption is to be kept as low as possible, the response to a command will be delivered asynchnornously as an event for which a handler will have to be specified in the user smartBASIC application.

The rest of this chapter describes all the GATT Client commands, responses and events in detail along with example code demonstrating usage and expected output.

Events and Messages

The nature of GATT Client operation consists of multiple queries and acting on the responses. Due to the connection intervals being vastly slower than the speed of the cpu, responses can arrive many 10s of

milliseconds after the precudure was triggered, which are delivered to an app using an event or message. Since these event/messages are tightly coupled with the appropriate commands, all but one will be described when the command that triggers them is described.

The event EVGATTCTOUT is applicable for all Gatt Client related functions which result in transactions over the air. The Bluetooth specification states that if an operation is initiated and is not completed within 30 seconds then the connection shall be dropped as no further Gatt Client transaction can be initiated.

EVGATTCTOUT event message

This event message **WILL** be thrown if a Gatt Client transaction takes longer than 30 seconds. It contains 1 INTEGER parameter:

Connection Handle

```
//Example :: EVGATTCTOUT.sb (See in BL600CodeSnippets.zip)
//

DIM rc,conHndl

//============================================================================
// Initialise and instantiate service, characteristic, start adverts
//============================================================================
FUNCTION OnStartup()
    DIM rc, adRpt$, addr$, scRpt$
    rc=BleAdvRptInit(adRpt$, 2, 0, 10)
    IF rc==0 THEN : rc=BleScanRptInit(scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvRptsCommit(adRpt$,scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvertStart(0,addr$,50,0,0) : ENDIF
    //open the gatt client with default notify/indicate ring buffer size
    IF rc==0 THEN : rc = BleGattcOpen(0,0) : ENDIF
ENDFUNC rc


//============================================================================
// Ble event handler
//============================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n- Disconnected"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n- Connected"
    ENDIF
ENDFUNC 1

'//============================================================================
'//============================================================================
FUNCTION HandlerGattcTout(cHndl) AS INTEGER
    PRINT "\nEVGATTCTOUT connHandle=";cHndl
ENDFUNC 1

//============================================================================
// Main() equivalent
//============================================================================
ONEVENT  EVBLEMSG           CALL HndlrBleMsg
OnEvent  EVGATTCTOUT        call HandlerGattcTout

rc = OnStartup()

WAITEVENT
```

Expected Output:

```
. . .
. . .
EVGATTCTOUT connHandle=123
. . .
. . .
```

BleGattcOpen

FUNCTION

This function is used to initialise the GATT Client functionality for immediate use so that appropriate buffers for caching GATT responses are created in the heap memory. About 300 bytes of RAM is required by the GATT Client manager and given that a majority of BL600 use cases will not utilise it, the sacrifice of 300 bytes, which is nearly 15% of the available memory, is not worth the permament allocation of memory.

There are various buffers that need to be created that are needed for scanning a remote GATT table which are of fixed size. There is however, one buffer which can be configured by the smartBASIC apps developer and that is the ring buffer that is used to store incoming notifiable and indicatable characteristics. At the time of writing this user manual the default minimum size is 64 unless a bigger one is desired and in that case the input parameter to this function specifies that size. A maximum of 2048 bytes is allowed, but that can result in unreliable operation as the smartBASIC runtime engine will be starved of memory very quickly.

Use SYSINFO(2019) to obtain the actual default size and SYSINFO(2020) to obtain the maximum allowed. The same information can be obtained in interactive mode using the commands AT I 2019 and 2020 respectively.

Note that when the ring buffer for the notifiable and indicatable characteristics is full, then any new messages will get discarded and depending on the flags parameter the indicates will or will not get confirmed.

This function is safe to call when the gatt client manager is already open, however, in that case the parameters are ignored and existing values are retained and any existing gattc client operations are not interrupted.

It is recommended that this function NOT be called when in a connection.

BLEGATTCOPEN (nNotifyBufLen, nFlags)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
nNotifyBufLen	byVal *nNotifyBufLen* AS INTEGER This is the size of the ring buffer used for incoming notifiable and indicatable characterstic data. Set to 0 to use the default size.
nFlags	byVal *nFlags* AS INTEGER **Bit 0:** Set to 1 to disable automatic indication confirmations if buffer is full then the Handle Value Confirmation will only be sent when BleGattcNotifyRead() is called to read the ring buffer. **Bit 1..31:** Reserved for future use and must be set to 0s
Interactive Command	No

```
//Example :: BleGattcOpen.sb (See in BL600CodeSnippets.zip)
```

```
DIM rc
//open the gatt client with default notify/indicate ring buffer size
rc = BleGattcOpen(0,0)
IF rc == 0 THEN
    PRINT "\nGatt Client is now open"
ENDIF
//open the client with default notify/indicate ring buffer size - again
rc = BleGattcOpen(128,1)
IF rc == 0 THEN
    PRINT "\nGatt Client is still open, because already open"
ENDIF
```

Expected Output:

```
Gatt Client is now open
Gatt Client is still open, because already open
```

BLEGATTCOPEN is an extension function.

BleGattcClose

SUBROUTINE

This function is used to close the GATT client manager and is safe to call if it is already closed.

It is recommended that this function NOT be called when in a connection.

BLEGATTCCLOSE ()

Arguments	None
Interactive Command	No

```
//Example :: BleGattcClose.sb (See in BL600CodeSnippets.zip)

DIM rc
//open the gatt client with default notify/indicate ring buffer size
rc = BleGattcOpen(0,0)
IF rc == 0 THEN
    PRINT "\nGatt Client is now open"
ENDIF
BleGattcClose()
PRINT "\nGatt Client is now closed"
BleGattcClose()
PRINT "\nGatt Client is closed - was safe to call when already closed"
```

Expected Output:

```
Gatt Client is now open
Gatt Client is now closed
Gatt Client is closed - was safe to call when already closed
```

BLEGATTCCLOSE is an extension subroutine.

BleDiscServiceFirst / BleDiscServiceNext

FUNCTIONS

This pair of functions is used to scan the remote Gatt Server for all primary services with the help of the EVDISCPRIMSVC message event and when called a handler for the event message **must** be registered as the discovered primary service information is passed back in that message.

A generic or uuid based scan can be initiated. The former will scan for all primary services and the latter will scan for a primary service with a particular uuid, the handle of which must be supplied and is generated by using either BleHandleUuid16() or BleHandleUuid128().

While the scan is in progress and waiting for the next piece of data from a Gatt server the module will enter low power state as the WAITEVENT statement is used as normal to wait for events and messages.

Depending on the size of the remote GATT server table and the connection interval, the scan of all primary may take many 100s of milliseconds, and while this is in progress it is safe to do other non Gatt related operations like for example servicing sensors and displays or any of the onboard peripherals.

EVDISCPRIMSVC event message

This event message **WILL** be thrown if either BleDiscServiceFirst() or BleDiscServiceNext() returns a success. The message contains 4 INTEGER parameters:-

Connection Handle
Service Uuid Handle
Start Handle of the service in the Gatt Table
End Handle for the service.

If no more services were discovered because the end of the table was reached, then all parameters will contain 0 apart from the Connection Handle.

BLEDISCSERVICEFIRST (connHandle,startAttrHandle,uuidHandle)

A typical pseudo code for discovering primary services involves first calling BleDiscServiceFirst(), then waiting for the EVDISCPRIMSVC event message and depending on the information returned in that message calling BleDiscServiceNext(), which in turn will result in another EVDISCPRIMSVC event message and typically is as follows:-

```
Register a handler for the EVDISCPRIMSVC event message

On EVDISCPRIMSVC event message
    If Start/End Handle == 0 then scan is complete
    Else Process information then
         call BleDiscServiceNext()
         if BleDiscServiceNext() not OK then scan complete

Call BleDiscServiceFirst()
If BleDiscServiceFirst() ok then Wait for EVDISCPRIMSVC
```

Returns	INTEGER, a result code. **Typical value:** 0x0000 Indicates a successful operation and means an EVDISCPRIMSVC event message WILL be thrown by the smartBASIC runtime engine containing the results. A non-zero return value implies an EVDISCPRIMSVC message will NOT be thrown.
Arguments	
nConnHandle	byVal *nConnHandle* AS INTEGER This is the connection handle as returned in the on-connect event for the connection on which the remote Gatt Server can be accessed. This will have been returned in the EVBLEMSG event message with msgId == 0 and msgCtx will have been the connectior handle.
startAttrHandle	byVal *startAttrHandle* AS INTEGER This is the attribute handle from where the scan for primary services will be started and you can typically set it to 0 to ensure that the entire remote Gatt Server is scanned.
uuidHandle	byVal *uuidHandle* AS INTEGER Set this to 0 if you want to scan for any service, otherwise this value will have been generated either by BleHandleUuid16() or BleHandleUuid128() or BleHandleUuidSibling().
Interactive Command	No

BLEDISCSERVICENEXT (connHandle)

Calling this assumes that BleDiscServiceFirst() has been called at least once to set up the internal primary services scanning state machine.

Returns	INTEGER, a result code. **Typical value**: 0x0000 Indicates a successful operation and it means an EVDISCPRIMSVC event message WILL be thrown by the smartBASIC runtime engine containing the results. A non-zero return value implies an EVDISCPRIMSVC message will NOT be thrown.
Arguments	
nConnHandle	byVal *nConnHandle* AS INTEGER This is the connection handle as returned in the on-connect event for the connection on whic the remote Gatt Server can be accessed. This will have been returned in the EVBLEMSG event message with msgId == 0 and msgCtx will have been the connection handle.
Interactive Command	No

```
//Example :: BleDiscServiceFirst.Next.sb (See in BL600CodeSnippets.zip)
//
//Remote server has 5 prim services with 16 bit uuid and 3 with 128 bit uuids
//  3 of the 16 bit uuid are the same value 0xDEAD and
//  2 of the 128 bit uuids are also the same 112233445566778899AABBCCDDEEFF
//
// Server created using BleGattcTblDiscPrimSvc.sub invoked in _OpenMcp.scr
// using Nordic Usb Dongle PC10000

DIM rc,at$,conHndl,uHndl,uuid$

//============================================================================
// Initialise and instantiate service, characteristic, start adverts
//============================================================================
FUNCTION OnStartup()
    DIM rc, adRpt$, addr$, scRpt$
    rc=BleAdvRptInit(adRpt$, 2, 0, 10)
    IF rc==0 THEN : rc=BleScanRptInit(scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvRptsCommit(adRpt$,scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvertStart(0,addr$,50,0,0) : ENDIF
    //open the gatt client with default notify/indicate ring buffer size
    IF rc==0 THEN : rc = BleGattcOpen(0,0) : ENDIF
ENDFUNC rc

//============================================================================
// Close connections so that we can run another app without problems
//============================================================================
SUB CloseConnections()
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
ENDSUB

//============================================================================
// Ble event handler
//============================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    DIM uu$
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n- Disconnected"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
```

```
        PRINT "\n- Connected, so scan remote Gatt Table for ALL services"
        rc = BleDiscServiceFirst(conHndl,0,0)
        IF rc==0 THEN
            //HandlerPrimSvc() will exit with 0 when operation is complete
            WAITEVENT

            PRINT "\nScan for service with uuid = 0xDEAD"
            uHndl = BleHandleUuid16(0xDEAD)
            rc = BleDiscServiceFirst(conHndl,0,uHndl)
            IF rc==0 THEN
                //HandlerPrimSvc() will exit with 0 when operation is complete
                WAITEVENT

                uu$ = "112233445566778899AABBCCDDEEFF00"
                PRINT "\nScan for service with custom uuid ";uu$
                uu$ = StrDehexize$(uu$)
                uHndl = BleHandleUuid128(uu$)
                rc = BleDiscServiceFirst(conHndl,0,uHndl)
                IF rc==0 THEN
                    //HandlerPrimSvc() will exit with 0 when operation is complete
                    WAITEVENT
                ENDIF
            ENDIF
        ENDIF
        CloseConnections()
    ENDIF
ENDFUNC 1

//================================================================================
// EVDISCPRIMSVC event handler
//================================================================================
FUNCTION HandlerPrimSvc(cHndl,svcUuid,sHndl,eHndl) AS INTEGER
    PRINT "\nEVDISCPRIMSVC :"
    PRINT " cHndl=";cHndl
    PRINT " svcUuid=";integer.h' svcUuid
    PRINT " sHndl=";sHndl
    PRINT " eHndl=";eHndl
    IF sHndl == 0 THEN
        PRINT "\nScan complete"

        EXITFUNC 0
    ELSE
        rc = BleDiscServiceNext(cHndl)
        IF rc != 0 THEN
            PRINT "\nScan abort"

            EXITFUNC 0
        ENDIF
    ENDIF
endfunc 1

//================================================================================
// Main() equivalent
//================================================================================
ONEVENT  EVBLEMSG           CALL HndlrBleMsg
OnEvent  EVDISCPRIMSVC      call HandlerPrimSvc

//Register base uuids with the underlying stack, otherwise the services with the
//128bit uuid's will be delivered with a uuid handle == FF000000 == UNKNOWN
uuid$ = "112233445566778899AABBCCDDEEFF00"
uuid$ = StrDehexize$(uuid$)
uHndl = BleHandleUuid128(uuid$)
```

```
uuid$ = "1122DEAD5566778899AABBCCDDBEEF00"
uuid$ = StrDehexize$(uuid$)
uHndl = BleHandleUuid128(uuid$)

IF OnStartup()==0 THEN
    PRINT "\nAdvertising, and Gatt Client is open\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF

WAITEVENT
PRINT "\nExiting..."
```

Expected Output:

```
Advertising, and Gatt Client is open


- Connected, so scan remote Gatt Table for ALL services
EVDISCPRIMSVC : cHndl=2804 svcUuid=FE01FE01 sHndl=1 eHndl=3
EVDISCPRIMSVC : cHndl=2804 svcUuid=FC033344 sHndl=4 eHndl=6
EVDISCPRIMSVC : cHndl=2804 svcUuid=FE01DEAD sHndl=7 eHndl=9
EVDISCPRIMSVC : cHndl=2804 svcUuid=FB04BEEF sHndl=10 eHndl=12
EVDISCPRIMSVC : cHndl=2804 svcUuid=FC033344 sHndl=13 eHndl=15
EVDISCPRIMSVC : cHndl=2804 svcUuid=FE01DEAD sHndl=16 eHndl=18
EVDISCPRIMSVC : cHndl=2804 svcUuid=FE01FE03 sHndl=19 eHndl=21
EVDISCPRIMSVC : cHndl=2804 svcUuid=FE01DEAD sHndl=22 eHndl=24
EVDISCPRIMSVC : cHndl=2804 svcUuid=00000000 sHndl=0 eHndl=0
Scan complete
Scan for service with uuid = 0xDEAD
EVDISCPRIMSVC : cHndl=2804 svcUuid=FE01DEAD sHndl=7 eHndl=9
EVDISCPRIMSVC : cHndl=2804 svcUuid=FE01DEAD sHndl=16 eHndl=18
EVDISCPRIMSVC : cHndl=2804 svcUuid=FE01DEAD sHndl=22 eHndl=65535
```

BLEDISCSERVICEFIRST and BLEDISCSERVICENEXT are both extension functions.

BleDiscCharFirst / BleDiscCharNext

FUNCTIONS

These pair of functions are used to scan the remote Gatt Server for characteristics in a service with the help of the EVDISCCHAR message event and when called a handler for the event message must be registered as the discovered characteristics information is passed back in that message

A generic or UUID based scan can be initiated. The former scans for all characteristics and the latter scans for a characteristic with a particular UUID, the handle of which must be supplied and is generated by using either BleHandleUuid16() or BleHandleUuid128().

If instead it is known that a Gatt table has a specific service and a specific characteristic, then a more efficient method for locating details of that characteristic is to use the function BleGattcFindChar() which is described later.

While the scan is in progress and waiting for the next piece of data from a Gatt server the module will enter low power state as the WAITEVENT statement is used as normal to wait for events and messages.

Depending on the size of the remote GATT server table and the connection interval, the scan of all characteristics may take many 100s of milliseconds, and while this is in progress it is safe to do other non Gatt related operations like for example servicing sensors and displays or any of the onboard peripherals.

Note: It is not currently possible to scan for characteristics in included services. This will be a future enhancement.

EVDISCCHAR event message

This event message is thrown if either BleDiscCharFirst() or BleDiscCharNext() returns a success. The message contains 5 INTEGER parameters:

Connection Handle
Characteristic Uuid Handle
Characteristic Properties
Handle for the Value Attribute of the Characteristic
Included Service Uuid Handle

If no more characteristics were discovered because the end of the table was reached, then all parameters will contain 0 apart from the Connection Handle.

'Characteristic Uuid Handle' contains the uuid of the characteristic and supplied as a handle.

'Characteristic Properties' contains the properties of the characteristic and is a bit mask as follows:

Bit 0 : Set if BROADCAST is enabled
Bit 1 : Set if READ is enabled
Bit 2 : Set if WRITE_WITHOUT_RESPONSE is enabled
Bit 3 : Set if WRITE is enabled
Bit 4 : Set if NOTIFY is enabled
Bit 5 : Set if INDICATE is enabled
Bit 6 : Set if AUTHENTICATED_SIGNED_WRITE is enabled
Bit 7 : Set if RELIABLE_WRITE is enabled
Bit 15 : Set if the characteristic has extended properties

'Handle for the Value Attribute of the Characteristic' is the handle for the value attribute and is the value to store to keep track of important characteristics in a gatt server for later read/write operations.

'Included Service Uuid Handle' is for future use and will always be 0.

BLEDISCCHARFIRST (connHandle, charUuidHandle, startAttrHandle,endAttrHandle)

A typical pseudo code for discovering characteristic involves first calling BleDiscCharFirst() with information obtained from a primary services scan and then waiting for the EVDISCCHAR event message and depending on the information returned in that message calling BleDiscCharNext() which in turn will result in another EVDISCCHAR event message and typically is as follows:-

```
Register a handler for the EVDISCCHAR event message
```

```
On EVDISCCHAR event message
    If Char Value Handle == 0 then scan is complete
    Else Process information then
         call BleDiscCharNext()
         if BleDiscCharNext() not OK then scan complete

Call BleDiscCharFirst( --information from EVDISCPRIMSVC )
If BleDiscCharFirst() ok then Wait for EVDISCCHAR
```

Returns	INTEGER, a result code. Typical value: 0x0000 Indicates a successful operation and means an EVDISCCHAR event message WILL be thrown by the smartBASIC runtime engine containing the results. A non-zero return value implies an EVDISCCHAR message will NOT be thrown.
Arguments	
nConnHandle	byVal *nConnHandle* AS INTEGER This is the connection handle as returned in the on-connect event for the connection on which the remote Gatt Server can be accessed. This will have been returned in the EVBLEMSG event message with msgId == 0 and msgCtx will have been the connection handle.
charUuidHandle	byVal *charUuidHandle* AS INTEGER Set this to 0 if you want to scan for any characteristic in the service, otherwise this value will have been generated either by BleHandleUuid16() or BleHandleUuid128() or BleHandleUuidSibling().
startAttrHandle	byVal *startAttrHandle* AS INTEGER This is the attribute handle from where the scan for characteristic will be started and will have been acquired by doing a primary services scan, which returns the start and end handles of services.
endAttrHandle	byVal *endAttrHandle* AS INTEGER This is the end attribute handle for the scan and will have been acquired by doing a primary services scan, which returns the start and end handles of services.
Interactive Command	No

BLEDISCCHARNEXT (connHandle)

Calling this assumes that BleDiscCharFirst() has been called at least once to set up the internal characteristics scanning state machine. It scans for the next characteristic.

Returns	INTEGER, a result code. Typical value: 0x0000 Indicates a successful operation and means an EVDISCCHAR event message WILL be thrown by the smartBASIC runtime engine containing the results. A non-zero return value implies an EVDISCCHAR message will NOT be thrown.
Arguments	
nConnHandle	byVal *nConnHandle* AS INTEGER This is the connection handle as returned in the on-connect event for the connection on whic the remote Gatt Server can be accessed. This will have been returned in the EVBLEMSG event message with msgId == 0 and msgCtx will have been the connection handle.

Interactive Command	No

```
//Example :: BleDiscCharFirst.Next.sb (See in BL600CodeSnippets.zip)
//
//Remote server has 1 prim service with 16 bit uuid and 8 characteristics where
// 5 uuids are 16 bit and 3 are 128 bit
//  3 of the 16 bit uuid are the same value 0xDEAD and
//  2 of the 128 bit uuids are also the same 112233445566778899AABBCCDDEEFF
//
// Server created using BleGattcTblDiscChar.sub invoked in _OpenMcp.scr
// using Nordic Usb Dongle PC10000

DIM rc,at$,conHndl,uHndl,uuid$,sAttr,eAttr


//=============================================================================
// Initialise and instantiate service, characteristic, start adverts
//=============================================================================
FUNCTION OnStartup()
    DIM rc, adRpt$, addr$, scRpt$
    rc=BleAdvRptInit(adRpt$, 2, 0, 10)
    IF rc==0 THEN : rc=BleScanRptInit(scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvRptsCommit(adRpt$,scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvertStart(0,addr$,50,0,0) : ENDIF
    //open the gatt client with default notify/indicate ring buffer size
    IF rc==0 THEN : rc = BleGattcOpen(0,0) : ENDIF
ENDFUNC rc


//=============================================================================
// Close connections so that we can run another app without problems
//=============================================================================
SUB CloseConnections()
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
ENDSUB

//=============================================================================
// Ble event handler
//=============================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    DIM uu$
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n- Disconnected"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n- Connected, so scan remote Gatt Table for first service"
        PRINT "\n- and a characeristic scan will be initiated in the event"
        rc = BleDiscServiceFirst(conHndl,0,0)
        IF rc==0 THEN
            //wait for start and end handles for first primary service
            WAITEVENT
            PRINT "\n\nScan for characteristic with uuid = 0xDEAD"
            uHndl = BleHandleUuid16(0xDEAD)
            rc = BleDiscCharFirst(conHndl,uHndl,sAttr,eAttr)
            IF rc == 0 THEN
                //HandlerCharDisc() will exit with 0 when operation is complete
                WAITEVENT
                uu$ = "112233445566778899AABBCCDDEEFF00"
```

```
                PRINT "\n\nScan for service with custom uuid ";uu$
                uu$ = StrDehexize$(uu$)
                uHndl = BleHandleUuid128(uu$)
                rc = BleDiscCharFirst(conHndl,uHndl,sAttr,eAttr)
                IF rc==0 THEN
                    //HandlerCharDisc() will exit with 0 when operation is complete
                    WAITEVENT
                ENDIF
            ENDIF
        ENDIF
        CloseConnections()
    ENDIF
ENDFUNC 1

//===============================================================================
// EVDISCPRIMSVC event handler
//===============================================================================
FUNCTION HandlerPrimSvc(cHndl,svcUuid,sHndl,eHndl) AS INTEGER
    PRINT "\nEVDISCPRIMSVC :"
    PRINT " cHndl=";cHndl
    PRINT " svcUuid=";integer.h' svcUuid
    PRINT " sHndl=";sHndl
    PRINT " eHndl=";eHndl
    IF sHndl == 0 THEN
        PRINT "\nPrimary Service Scan complete"
        EXITFUNC 0
    ELSE
        PRINT "\nGot first primary service so scan for ALL characteristics"
        sAttr = sHndl
        eAttr = eHndl
        rc = BleDiscCharFirst(conHndl,0,sAttr,eAttr)
        IF rc != 0 THEN
            PRINT "\nScan characteristics failed"
            EXITFUNC 0
        ENDIF
    ENDIF
endfunc 1

'//===============================================================================
// EVDISCCHAR event handler
'//===============================================================================
function HandlerCharDisc(cHndl,cUuid,cProp,hVal,isUuid) as integer
    print "\nEVDISCCHAR :"
    print " cHndl=";cHndl
    print " chUuid=";integer.h' cUuid
    print " Props=";cProp
    print " valHndl=";hVal
    print " ISvcUuid=";isUuid
    IF hVal == 0 THEN
        PRINT "\nCharacteristic Scan complete"
        EXITFUNC 0
    ELSE
        rc = BleDiscCharNext(conHndl)
        IF rc != 0 THEN
            PRINT "\nCharacteristics scan abort"
            EXITFUNC 0
        ENDIF
    ENDIF
endfunc 1


//===============================================================================
```

```
// Main() equivalent
//===============================================================================
ONEVENT  EVBLEMSG           CALL HndlrBleMsg
OnEvent  EVDISCPRIMSVC      call HandlerPrimSvc
OnEvent  EVDISCCHAR         call HandlerCharDisc

//Register base uuids with the underlying stack, otherwise the services with the
//128bit uuid's will be delivered with a uuid handle == FF000000 == UNKNOWN
uuid$ = "112233445566778899AABBCCDDEEFF00"
uuid$ = StrDehexize$(uuid$)
uHndl = BleHandleUuid128(uuid$)
uuid$ = "1122DEAD5566778899AABBCCDDBEEF00"
uuid$ = StrDehexize$(uuid$)
uHndl = BleHandleUuid128(uuid$)

IF OnStartup()==0 THEN
    PRINT "\nAdvertising, and Gatt Client is open\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF

WAITEVENT
PRINT "\nExiting..."
```

Expected Output:

```
Advertising, and Gatt Client is open

- Connected, so scan remote Gatt Table for first service
- and a characeristic scan will be initiated in the event
EVDISCPRIMSVC : cHndl=3549 svcUuid=FE01FE02 sHndl=1 eHndl=17
Got first primary service so scan for ALL characteristics
EVDISCCHAR : cHndl=3549 chUuid=FE01FC21 Props=2 valHndl=3 ISvcUuid=0
EVDISCCHAR : cHndl=3549 chUuid=FC033344 Props=2 valHndl=5 ISvcUuid=0
EVDISCCHAR : cHndl=3549 chUuid=FE01DEAD Props=2 valHndl=7 ISvcUuid=0
EVDISCCHAR : cHndl=3549 chUuid=FB04BEEF Props=2 valHndl=9 ISvcUuid=0
EVDISCCHAR : cHndl=3549 chUuid=FC033344 Props=2 valHndl=11 ISvcUuid=0
EVDISCCHAR : cHndl=3549 chUuid=FE01FC23 Props=2 valHndl=13 ISvcUuid=0
EVDISCCHAR : cHndl=3549 chUuid=FE01DEAD Props=2 valHndl=15 ISvcUuid=0
EVDISCCHAR : cHndl=3549 chUuid=FE01DEAD Props=2 valHndl=17 ISvcUuid=0
EVDISCCHAR : cHndl=3549 chUuid=00000000 Props=0 valHndl=0 ISvcUuid=0
Characteristic Scan complete

Scan for characteristic with uuid = 0xDEAD
EVDISCCHAR : cHndl=3549 chUuid=FE01DEAD Props=2 valHndl=7 ISvcUuid=0
EVDISCCHAR : cHndl=3549 chUuid=FE01DEAD Props=2 valHndl=15 ISvcUuid=0
EVDISCCHAR : cHndl=3549 chUuid=FE01DEAD Props=2 valHndl=17 ISvcUuid=0
EVDISCCHAR : cHndl=3549 chUuid=00000000 Props=0 valHndl=0 ISvcUuid=0
Characteristic Scan complete

Scan for service with custom uuid 112233445566778899AABBCCDDEEFF00
EVDISCCHAR : cHndl=3549 chUuid=FC033344 Props=2 valHndl=5 ISvcUuid=0
EVDISCCHAR : cHndl=3549 chUuid=FC033344 Props=2 valHndl=11 ISvcUuid=0
EVDISCCHAR : cHndl=3549 chUuid=00000000 Props=0 valHndl=0 ISvcUuid=0
Characteristic Scan complete
```

BLEDISCCHARFIRST and BLEDISCCHARNEXT are both extension functions.

BleDiscDescFirst / BleDiscDescNext

FUNCTIONS

These pair of functions are used to scan the remote Gatt Server for descriptors in a characteristic with the help of the EVDISCDESC message event and when called a handler for the event message **must** be registered as the discovered descriptor information is passed back in that

A generic or uuid based scan can be initiated. The former will scan for all descriptors and the latter will scan for a descriptor with a particular uuid, the handle of which must be supplied and is generated by using either BleHandleUuid16() or BleHandleUuid128().

If instead it is known that a gatt table has a specific service, characteristic and a specific descriptor, then a more efficient method for locating details of that characteristic is to use the function BleGattcFindDesc() which is described later.

While the scan is in progress and waiting for the next piece of data from a Gatt server the module will enter low power state as the WAITEVENT statement is used as normal to wait for events and messages.

Depending on the size of the remote GATT server table and the connection interval, the scan of all descriptors may take many 100s of milliseconds, and while this is in progress it is safe to do other non Gatt related operations like for example servicing sensors and displays or any of the onboard peripherals.

EVDISCDESC event message

This event message **WILL** be thrown if either BleDissDescFirst() or BleDiscDescNext() returns a success. The message contains 3 INTEGER parameters:-

Connection Handle
Descriptor Uuid Handle
Handle for the Descriptor in the remote Gatt Table

If no more descriptors were discovered because the end of the table was reached, then all parameters will contain 0 apart from the Connection Handle.

'**Descriptor Uuid Handle**' contains the uuid of the descriptor and supplied as a handle.

'**Handle for the Descriptor in the remote Gatt Table**' is the handle for the descriptor, and also is the value to store to keep track of important characteristics in a gatt server for later read/write operations.

BLEDISCDESCFIRST (connHandle, descUuidHandle, charValHandle)

A typical pseudo code for discovering descriptors involves first calling BleDiscDescFirst() with information obtained from a characteristics scan and then waiting for the EVDISCDESC event message and depending on the information returned in that message calling BleDiscDescNext() which in turn will result in another EVDISCDESC event message and typically is as follows:-

```
Register a handler for the EVDISCDESC event message

On EVDISCDESC event message
    If Descriptor Handle == 0 then scan is complete
    Else Process information then
         call BleDiscDescNext()
         if BleDiscDescNext() not OK then scan complete

Call BleDiscDescFirst( --information from EVDISCCHAR )
If BleDiscDescFirst() ok then Wait for EVDISCDESC
```

Returns	INTEGER, a result code.

	Typical value: 0x0000 Indicates a successful operation and means an EVDISCDESC event message WILL be thrown by the smartBASIC runtime engine containing the results. A non-zero return value implies an EVDISCDESC message will NOT be thrown.
Arguments	
connHandle	byVal *nConnHandle* AS INTEGER This is the connection handle as returned in the on-connect event for the connection on which the remote Gatt Server can be accessed. This will have been returned in the EVBLEMSG event message with msgId == 0 and msgCtx will have been the connection handle.
descUuidHandle	byVal *descUuidHandle* AS INTEGER Set this to 0 if you want to scan for any descriptor in the characteristic, otherwise this value will have been generated either by BleHandleUuid16() or BleHandleUuid128() or BleHandleUuidSibling().
charValHandle	byVal *charValHandle* AS INTEGER This is the value attribute handle of the characteristic on which the descriptor scan is to be performed. It will have been acquired from an EVDISCCHAR event
Interactive Command	No

BLEDISCDESCNEXT (connHandle)

Calling this assumes that BleDiscCharFirst() has been called at least once to set up the internal characteristics scanning state machine, and that BleDiscDescFirst() has been called at least once to start the descriptor discovery process.

Returns	INTEGER, a result code. **Typical value:** 0x0000 Indicates a successful operation and means an EVDISCDESC event message WILL be thrown by the smartBASIC runtime engine containing the results. A non-zero return value implies an EVDISCDESC message will NOT be thrown.
Arguments	
connHandle	byVal *nConnHandle* AS INTEGER This is the connection handle as returned in the on-connect event for the connection o which the remote Gatt Server can be accessed. This will have been returned in the EVBLEMSG event message with msgId == 0 and msgCtx will have been the connectior handle.
Interactive Command	No

```
//Example :: BleDiscDescFirst.Next.sb (See in BL600CodeSnippets.zip)
//
//Remote server has 1 prim service with 16 bit uuid and 1 characteristics
// which contains 8 descriptors, that are ...
// 5 uuids are 16 bit and 3 are 128 bit
//  3 of the 16 bit uuid are the same value 0xDEAD and
//  2 of the 128 bit uuids are also the same 112233445566778899AABBCCDDEEFF
//
// Server created using BleGattcTblDiscDesc.sub invoked in _OpenMcp.scr
// using Nordic Usb Dongle PC10000

DIM rc,at$,conHndl,uHndl,uuid$,sAttr,eAttr,cValAttr
```

```
//================================================================================
// Initialise and instantiate service, characteristic, start adverts
//================================================================================
FUNCTION OnStartup()
    DIM rc, adRpt$, addr$, scRpt$
    rc=BleAdvRptInit(adRpt$, 2, 0, 10)
    IF rc==0 THEN : rc=BleScanRptInit(scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvRptsCommit(adRpt$,scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvertStart(0,addr$,50,0,0) : ENDIF
    //open the gatt client with default notify/indicate ring buffer size
    IF rc==0 THEN : rc = BleGattcOpen(0,0) : ENDIF
ENDFUNC rc
//================================================================================
// Close connections so that we can run another app without problems
//================================================================================
SUB CloseConnections()
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
ENDSUB

//================================================================================
// Ble event handler
//================================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    DIM uu$
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n- Disconnected"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n- Connected, so scan remote Gatt Table for first service"
        PRINT "\n- and a characeristic scan will be initiated in the event"
        rc = BleDiscServiceFirst(conHndl,0,0)
        IF rc==0 THEN
            //wait for start and end handles for first primary service
            WAITEVENT
            PRINT "\n\nScan for descritors with uuid = 0xDEAD"
            uHndl = BleHandleUuid16(0xDEAD)
            rc = BleDiscDescFirst(conHndl,uHndl,cValAttr)
            IF rc == 0 THEN
                //HandlerDescDisc() will exit with 0 when operation is complete
                WAITEVENT
                uu$ = "112233445566778899AABBCCDDEEFF00"
                PRINT "\n\nScan for service with custom uuid ";uu$
                uu$ = StrDehexize$(uu$)
                uHndl = BleHandleUuid128(uu$)
                rc = BleDiscDescFirst(conHndl,uHndl,cValAttr)
                IF rc==0 THEN
                    //HandlerDescDisc() will exit with 0 when operation is complete
                    WAITEVENT
                ENDIF
            ENDIF
        ENDIF
        CloseConnections()
    ENDIF
ENDFUNC 1

//================================================================================
// EVDISCPRIMSVC event handler
//================================================================================
FUNCTION HandlerPrimSvc(cHndl,svcUuid,sHndl,eHndl) AS INTEGER
```

```
    PRINT "\nEVDISCPRIMSVC :"
    PRINT " cHndl=";cHndl
    PRINT " svcUuid=";integer.h' svcUuid
    PRINT " sHndl=";sHndl
    PRINT " eHndl=";eHndl
    IF sHndl == 0 THEN
        PRINT "\nPrimary Service Scan complete"
        EXITFUNC 0
    ELSE
        PRINT "\nGot first primary service so scan for ALL characteristics"
        sAttr = sHndl
        eAttr = eHndl
        rc = BleDiscCharFirst(conHndl,0,sAttr,eAttr)
        IF rc != 0 THEN
            PRINT "\nScan characteristics failed"
            EXITFUNC 0
        ENDIF
    ENDIF
endfunc 1

'//============================================================================
// EVDISCCHAR event handler
'//============================================================================
function HandlerCharDisc(cHndl,cUuid,cProp,hVal,isUuid) as integer
    print "\nEVDISCCHAR :"
    print " cHndl=";cHndl
    print " chUuid=";integer.h' cUuid
    print " Props=";cProp
    print " valHndl=";hVal
    print " ISvcUuid=";isUuid
    IF hVal == 0 THEN
        PRINT "\nCharacteristic Scan complete"
        EXITFUNC 0
    ELSE
        PRINT "\nGot first characteristic service at handle ";hVal
        PRINT "\nScan for ALL Descs"
        cValAttr = hVal
        rc = BleDiscDescFirst(conHndl,0,cValAttr)
        IF rc != 0 THEN
            PRINT "\nScan descriptors failed"
            EXITFUNC 0
        ENDIF
    ENDIF
endfunc 1

'//============================================================================
// EVDISCDESC event handler
'//============================================================================
function HandlerDescDisc(cHndl,cUuid,hndl) as integer
    print "\nEVDISCDESC"
    print " cHndl=";cHndl
    print " dscUuid=";integer.h' cUuid
    print " dscHndl=";hndl
    IF hndl == 0 THEN
        PRINT "\nDescriptor Scan complete"
        EXITFUNC 0
    ELSE
        rc = BleDiscDescNext(cHndl)
        IF rc != 0 THEN
            PRINT "\nDescriptor scan abort"
            EXITFUNC 0
        ENDIF
```

```
    ENDIF
endfunc 1

//===============================================================================
// Main() equivalent
//===============================================================================
ONEVENT  EVBLEMSG           CALL HndlrBleMsg
OnEvent  EVDISCPRIMSVC      call HandlerPrimSvc
OnEvent  EVDISCCHAR         call HandlerCharDisc
OnEvent  EVDISCDESC         call HandlerDescDisc

//Register base uuids with the underlying stack, otherwise the services with the
//128bit uuid's will be delivered with a uuid handle == FF000000 == UNKNOWN
uuid$ = "112233445566778899AABBCCDDEEFF00"
uuid$ = StrDehexize$(uuid$)
uHndl = BleHandleUuid128(uuid$)
uuid$ = "1122DEAD5566778899AABBCCDDBEEF00"
uuid$ = StrDehexize$(uuid$)
uHndl = BleHandleUuid128(uuid$)

IF OnStartup()==0 THEN
    PRINT "\nAdvertising, and Gatt Client is open\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF

WAITEVENT
PRINT "\nExiting..."
```

Expected Output:

```
Advertising, and Gatt Client is open

- Connected, so scan remote Gatt Table for first service
- and a characeristic scan will be initiated in the event
EVDISCPRIMSVC : cHndl=3790 svcUuid=FE01FE02 sHndl=1 eHndl=11
Got first primary service so scan for ALL characteristics
EVDISCCHAR : cHndl=3790 chUuid=FE01FC21 Props=2 valHndl=3 ISvcUuid=0
Got first characteristic service at handle 3
Scan for ALL Descs
EVDISCDESC cHndl=3790 dscUuid=FE01FD21 dscHndl=4
EVDISCDESC cHndl=3790 dscUuid=FC033344 dscHndl=5
EVDISCDESC cHndl=3790 dscUuid=FE01DEAD dscHndl=6
EVDISCDESC cHndl=3790 dscUuid=FB04BEEF dscHndl=7
EVDISCDESC cHndl=3790 dscUuid=FC033344 dscHndl=8
EVDISCDESC cHndl=3790 dscUuid=FE01FD23 dscHndl=9
EVDISCDESC cHndl=3790 dscUuid=FE01DEAD dscHndl=10
EVDISCDESC cHndl=3790 dscUuid=FE01DEAD dscHndl=11
EVDISCDESC cHndl=3790 dscUuid=00000000 dscHndl=0
Descriptor Scan complete

Scan for descritors with uuid = 0xDEAD
EVDISCDESC cHndl=3790 dscUuid=FE01DEAD dscHndl=6
EVDISCDESC cHndl=3790 dscUuid=FE01DEAD dscHndl=10
EVDISCDESC cHndl=3790 dscUuid=FE01DEAD dscHndl=11
EVDISCDESC cHndl=3790 dscUuid=00000000 dscHndl=0
Descriptor Scan complete

Scan for service with custom uuid 112233445566778899AABBCCDDEEFF00
EVDISCDESC cHndl=3790 dscUuid=FC033344 dscHndl=5
EVDISCDESC cHndl=3790 dscUuid=FC033344 dscHndl=8
EVDISCDESC cHndl=3790 dscUuid=00000000 dscHndl=0
Descriptor Scan complete

- Disconnected
Exiting...
```

BLEDISCDESCFIRST and BLEDISCDESCNEXT are both extension functions.

BleGattcFindChar

FUNCTION

This function facilitates a quick and efficient way of locating the details of a characteristic if the uuid is known along with the uuid of the service containing it and the results will be delived in a EVFINDCHAR event message. If the Gatt server table has multiple instances of the same service/characteristic combination then this function will work because in addition to the uuid handles to be searched for, it also accepts instance parameters which are indexed from 0, which means the 4th instance of a characteristic with the same uuid in the 3rd instance of a service with the same uuid will be located with index values 3 and 2 respectively.

Given that the results are returned in an event message, a handler **must** be registered for the EVFINDCHAR event.

Depending on the size of the remote GATT server table and the connection interval, the search of the characteristic may take many 100s of milliseconds, and while this is in progress it is safe to do other non Gatt related operations like for example servicing sensors and displays or any of the onboard peripherals.

Note: It is not currently possible to scan for characteristics in included services. This will be a future enhancement.

EVFINDCHAR event message

This event message **WILL** be thrown if BleGattcFindChar() returns a success. The message contains 4 INTEGER parameters:-

Connection Handle
Characteristic Properties
Handle for the Value Attribute of the Characteristic
Included Service Uuid Handle

If the specified instance of the service/characteristic is not present in the remote Gatt Server Table then all parameters will contain 0 apart from the Connection Handle.

'**Characteristic Properties**' contains the properties of the characteristic and is a bit mask as follows:

Bit 0	Set if BROADCAST is enabled
Bit 1	Set if READ is enabled
Bit 2	Set if WRITE_WITHOUT_RESPONSE is enabled
Bit 3	Set if WRITE is enabled
Bit 4	Set if NOTIFY is enabled
Bit 5	Set if INDICATE is enabled
Bit 6	Set if AUTHENTICATED_SIGNED_WRITE is enabled
Bit 7	Set if RELIABLE_WRITE is enabled
Bit 15	Set if the characteristic has extended properties

'**Handle for the Value Attribute of the Characteristic**' is the handle for the value attribute and is the value to store to keep track of important characteristics in a gatt server for later read/write operations.

'**Included Service Uuid Handle**' is for future use and will always be 0.

BLEGATTCFINDCHAR (connHandle, svcUuidHndl, svcIndex,charUuidHndl, charIndex)

A typical pseudo code for finding a characteristic involves calling BleGattcFindChar() which in turn will result in the EVFINDCHAR event message and typically is as follows:-

```
Register a handler for the EVFINDCHAR event message

On EVFINDCHAR event message
    If Char Value Handle == 0 then
        Characteristic not found
    Else
        Characteristic has been found

Call BleGattcFindChar()
If BleGattcFindChar () ok then Wait for EVFINDCHAR
```

Returns	INTEGER, a result code. Typical value: 0x0000 Indicates a successful operation and it means an EVFINDCHAR event message WILL be thrown by the smartBASIC runtime engine containing the results. A non-zero return value implies an EVFINDCHAR message will NOT be thrown.
Arguments	
connHandle	byVal *nConnHandle* AS INTEGER This is the connection handle as returned in the on-connect event for the connection on wh the remote Gatt Server can be accessed. This will have been returned in the EVBLEMSG ever message with msgId == 0 and msgCtx will have been the connection handle.
svcUuidHndl	byVal *svcUuidHndl* AS INTEGER Set this to the service uuid handle which will have been generated either by BleHandleUuid16() or BleHandleUuid128() or BleHandleUuidSibling().
svcIndex	byVal *svcIndex* AS INTEGER This is the instance of the service to look for with the uuid handle svcUuidHndl, where 0 is the first instance, 1 is the second etc
charUuidHndl	byVal *charUuidHndl* AS INTEGER Set this to the characteristic uuid handle which will have been generated either by BleHandleUuid16() or BleHandleUuid128() or BleHandleUuidSibling().
charIndex	byVal *charIndex* AS INTEGER This is the instance of the characteristic to look for with the uuid handle charUuidHndl, where 0 is the first instance, 1 is the second etc
Interactive Command	No

```
//Example :: BleGattcFindChar.sb (See in BL600CodeSnippets.zip)
//
//Remote server has 5 prim services with 16 bit uuid and 3 with 128 bit uuids
//  3 of the 16 bit uuid are the same value 0xDEAD and
//  2 of the 128 bit uuids are also the same 112233445566778899AABBCCDDEEFF
//
// Server created using BleGattcTblFindChar.sub invoked in _OpenMcp.scr
// using Nordic Usb Dongle PC10000

DIM rc,at$,conHndl,uHndl,uuid$,sIdx,cIdx


//============================================================================
// Initialise and instantiate service, characteristic, start adverts
```

```
//===============================================================================
FUNCTION OnStartup()
    DIM rc, adRpt$, addr$, scRpt$
    rc=BleAdvRptInit(adRpt$, 2, 0, 10)
    IF rc==0 THEN : rc=BleScanRptInit(scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvRptsCommit(adRpt$,scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvertStart(0,addr$,50,0,0) : ENDIF
    //open the gatt client with default notify/indicate ring buffer size
    IF rc==0 THEN : rc = BleGattcOpen(0,0) : ENDIF
ENDFUNC rc


//===============================================================================
// Close connections so that we can run another app without problems
//===============================================================================
SUB CloseConnections()
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
ENDSUB


//===============================================================================
// Ble event handler
//===============================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    DIM uu$,uHndS,uHndC
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n- Disconnected"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n- Connected, so scan remote Gatt Table for an instance of char"
        uHndS = BleHandleUuid16(0xDEAD)
        uu$ = "11223344556677889AABBCCDDEEFF00"
        uu$ = StrDehexize$(uu$)
        uHndC = BleHandleUuid128(uu$)
        sIdx = 2
        cIdx = 1 //valHandle will be 32
        rc = BleGattcFindChar(conHndl,uHndS,sIdx,uHndC,cIdx)
        IF rc==0 THEN
            //BleDiscCharFirst() will exit with 0 when operation is complete
            WAITEVENT
        ENDIF
        sIdx = 1
        cIdx = 3 //does not exist
        rc = BleGattcFindChar(conHndl,uHndS,sIdx,uHndC,cIdx)
        IF rc==0 THEN
            //BleDiscCharFirst() will exit with 0 when operation is complete
            WAITEVENT
        ENDIF
        CloseConnections()
    ENDIF
ENDFUNC 1

'//===============================================================================
'//===============================================================================
function HandlerFindChar(cHndl,cProp,hVal,isUuid) as integer
    print "\nEVFINDCHAR "
    print " cHndl=";cHndl
    print " Props=";cProp
    print " valHndl=";hVal
    print " ISvcUuid=";isUuid
```

```
    IF hVal == 0 THEN
        PRINT "\nDid NOT find the characteristic"
    ELSE
        PRINT "\nFound the characteristic at handle ";hVal
        PRINT "\nSvc Idx=";sIdx;" Char Idx=";cIdx
    ENDIF
endfunc 0


//============================================================================
// Main() equivalent
//============================================================================
ONEVENT  EVBLEMSG           CALL HndlrBleMsg
OnEvent  EVFINDCHAR         call HandlerFindChar

//Register base uuids with the underlying stack, otherwise the services with the
//128bit uuid's will be delivered with a uuid handle == FF000000 == UNKNOWN
uuid$ = "112233445566778899AABBCCDDEEFF00"
uuid$ = StrDehexize$(uuid$)
uHndl = BleHandleUuid128(uuid$)
uuid$ = "1122DEAD5566778899AABBCCDDBEEF00"
uuid$ = StrDehexize$(uuid$)
uHndl = BleHandleUuid128(uuid$)

IF OnStartup()==0 THEN
    PRINT "\nAdvertising, and Gatt Client is open\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF

WAITEVENT
PRINT "\nExiting..."
```

Expected Output:

```
Advertising, and Gatt Client is open

- Connected, so scan remote Gatt Table for an instance of char
EVFINDCHAR  cHndl=866 Props=2 valHndl=32 ISvcUuid=0
Found the characteristic at handle 32
Svc Idx=2 Char Idx=1
EVFINDCHAR  cHndl=866 Props=0 valHndl=0 ISvcUuid=0
Did NOT find the characteristic

- Disconnected
Exiting...
```

BLEGATTCFINDCHAR is an extension function.

BleGattcFindDesc

FUNCTION

This function facilitates a quick and efficient way of locating the details of a descriptor if the UUID is known along with the UUID of the service and the UUID of the characteristic containing it and the results are delivered in a EVFINDDESC event message. If the Gatt server table has multiple instances of the same service/characteristic/descriptor combination then this function works because, in addition to the UUID handles to be searched for, it also accepts instance parameters which are indexed from 0, which means the 2nd instance of a descriptor in the 4th instance of a characteristic with the same UUID in the 3rd instance of a service with the same UUID is located with index values 1, 3, and 2 respectively.

Given that the results are returned in an event message, a handler **must** be registered for the EVFINDDESC event.

Depending on the size of the remote GATT server table and the connection interval, the search of the characteristic may take many 100s of milliseconds, and while this is in progress it is safe to do other non Gatt related operations like for example servicing sensors and displays or any of the onboard peripherals.

Note: It is not currently possible to scan for characteristics in included services. This will be a future enhancement.

EVFINDDESC event message

This event message **WILL** be thrown if BleGattcFindDesc()returned a success. The message contains 2 INTEGER parameters:-

Connection Handle
Handle of the Descriptor

If the specified instance of the service/characteristic/descriptor is not present in the remote Gatt Server Table then all parameters will contain 0 apart from the Connection Handle.

'**Handle of the Descriptor**' is the handle for the descriptor and is the value to store to keep track of important descriptors in a gatt server for later read/write operations – for example CCCD's to enable notifications and/or indications.

BLEGATTCFINDDESC (connHndl, svcUuHndl, svcIdx, charUuHndl, charIdx,descUuHndl, descIdx)

A typical pseudo code for finding a descrirptor involves calling BleGattcFindDesc() which in turn will result in the EVFINDDESC event message and typically is as follows:-

```
Register a handler for the EVFINDDESC event message

On EVFINDDESC event message
    If Descriptor Handle == 0 then
        Descriptor not found
    Else
        Descriptor has been found

Call BleGattcFindDesc()
If BleGattcFindDesc() ok then Wait for EVFINDDESC
```

Returns	INTEGER, a result code. Typical value: 0x0000 Indicates a successful operation and means an EVFINDDESC event message WILL be thrown by the smartBASIC runtime engine containing the results. A non-zero return value implies an EVFINDDESC message will NOT be thrown.
Arguments	

connHndl	byVal *connHndl* AS INTEGER This is the connection handle as returned in the on-connect event for the connection on which the remote Gatt Server can be accessed. This will have been returned in the EVBLEMSG event message with msgId == 0 and msgCtx will have been the connection handle.
svcUuHndl	byVal *svcUuHndl* AS INTEGER Set this to the service uuid handle which will have been generated either by BleHandleUuid16() or BleHandleUuid128() or BleHandleUuidSibling().
svcIdx	byVal *svcIdx* AS INTEGER This is the instance of the service to look for with the uuid handle svcUuidHndl, where 0 is the first instance, 1 is the second etc
charUuHndl	byVal *charUuHndl* AS INTEGER Set this to the characteristic uuid handle which will have been generated either by BleHandleUuid16() or BleHandleUuid128() or BleHandleUuidSibling().
charIdx	byVal *charIdx* AS INTEGER This is the instance of the characteristic to look for with the uuid handle charUuidHndl, where 0 is the first instance, 1 is the second etc
descUuHndl	byVal *descUuHndl* AS INTEGER Set this to the descriptor uuid handle which will have been generated either by BleHandleUuid16() or BleHandleUuid128() or BleHandleUuidSibling().
descIdx	byVal *descIdx* AS INTEGER This is the instance of the descriptor to look for with the uuid handle charUuidHndl, where 0 is the first instance, 1 is the second etc
Interactive Command	No

```
//Example :: BleGattcFindDesc.sb (See in BL600CodeSnippets.zip)
//
//Remote server has 5 prim services with 16 bit uuid and 3 with 128 bit uuids
//  3 of the 16 bit uuid are the same value 0xDEAD and
//  2 of the 128 bit uuids are also the same 112233445566778899AABBCCDDEEFF
//
// Server created using BleGattcTblFindDesc.sub invoked in _OpenMcp.scr
// using Nordic Usb Dongle PC10000

DIM rc,at$,conHndl,uHndl,uuid$,sIdx,cIdx,dIdx


//============================================================================
// Initialise and instantiate service, characteristic, start adverts
//============================================================================
FUNCTION OnStartup()
    DIM rc, adRpt$, addr$, scRpt$
    rc=BleAdvRptInit(adRpt$, 2, 0, 10)
    IF rc==0 THEN : rc=BleScanRptInit(scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvRptsCommit(adRpt$,scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvertStart(0,addr$,50,0,0) : ENDIF
    //open the gatt client with default notify/indicate ring buffer size
    IF rc==0 THEN : rc = BleGattcOpen(0,0) : ENDIF
ENDFUNC rc


//============================================================================
// Close connections so that we can run another app without problems
//============================================================================
SUB CloseConnections()
```

```
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
ENDSUB


//============================================================================
// Ble event handler
//============================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    DIM uu$,uHndS,uHndC,uHndD
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n- Disconnected"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n- Connected, so scan remote Gatt Table for ALL services"
        uHndS = BleHandleUuid16(0xDEAD)
        uu$ = "112233445566778899AABBCCDDEEFF00"
        uu$ = StrDehexize$(uu$)
        uHndC = BleHandleUuid128(uu$)
        uu$ = "1122C0DE5566778899AABBCCDDEEFF00"
        uu$ = StrDehexize$(uu$)
        uHndD = BleHandleUuid128(uu$)
        sIdx = 2
        cIdx = 1
        dIdx = 1 // handle will be 37
        rc = BleGattcFindDesc(conHndl,uHndS,sIdx,uHndC,cIdx,uHndD,dIdx)
        IF rc==0 THEN
            //BleDiscCharFirst() will exit with 0 when operation is complete
            WAITEVENT
        ENDIF
        sIdx = 1
        cIdx = 3
        dIdx = 4 //does not exist
        rc = BleGattcFindDesc(conHndl,uHndS,sIdx,uHndC,cIdx,uHndD,dIdx)
        IF rc==0 THEN
            //BleDiscCharFirst() will exit with 0 when operation is complete
            WAITEVENT
        ENDIF
        CloseConnections()
    ENDIF
ENDFUNC 1

'//============================================================================
'//============================================================================
function HandlerFindDesc(cHndl,hndl) as integer
    print "\nEVFINDDESC "
    print " cHndl=";cHndl
    print " dscHndl=";hndl
    IF hndl == 0 THEN
        PRINT "\nDid NOT find the descriptor"
    ELSE
        PRINT "\nFound the descriptor at handle ";hndl
        PRINT "\nSvc Idx=";sIdx;" Char Idx=";cIdx;" desc Idx=";dIdx
    ENDIF
endfunc 0


//============================================================================
// Main() equivalent
//============================================================================
ONEVENT  EVBLEMSG           CALL HndlrBleMsg
```

```
OnEvent  EVFINDDESC         call HandlerFindDesc

//Register base uuids with the underlying stack, otherwise the services with the
//128bit uuid's will be delivered with a uuid handle == FF000000 == UNKNOWN
uuid$ = "11223344556677889900AABBCCDDEEFF00"
uuid$ = StrDehexize$(uuid$)
uHndl = BleHandleUuid128(uuid$)
uuid$ = "1122DEAD5566778899AABBCCDDBEEF00"
uuid$ = StrDehexize$(uuid$)
uHndl = BleHandleUuid128(uuid$)

IF OnStartup()==0 THEN
    PRINT "\nAdvertising, and Gatt Client is open\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF

WAITEVENT
PRINT "\nExiting..."
```

Expected Output:

```
Advertising, and Gatt Client is open

- Connected, so scan remote Gatt Table for ALL services
EVFINDDESC  cHndl=1106 dscHndl=37
Found the descriptor at handle 37
Svc Idx=2 Char Idx=1 desc Idx=1
EVFINDDESC  cHndl=1106 dscHndl=0
Did NOT find the descriptor

- Disconnected
Exiting...
```

BLEGATTCFINDDESC is an extension function.

BleGattcRead / BleGattcReadData

FUNCTIONS

If the handle for an attribute is known then these functions are used to read the content of that attribute from a specified offset in the array of octets in that attribute value.

Given that the success or failure of this read operation is returned in an event message, a handler **must** be registered for the EVATTRREAD event.

Depending on the connection interval, the read of the attribute may take many 100s of milliseconds, and while this is in progress it is safe to do other non Gatt related operations like for example servicing sensors and displays or any of the onboard peripherals.

BleGattcRead is used to trigger the procedure and BleGattcReadData is used to read the data from the underlying cache when the EVATTRREAD event message is received with a success status.

EVATTRREAD event message

This event message **WILL** be thrown if BleGattcRead() returns a success. The message contains 3 INTEGER parameters:-

Connection Handle
Handle of the Attribute
Gatt status of the read operation.

'**Gatt status of the read operation**' is one of the following values, where 0 implies the read was successfully expedited and the data can be obtained by calling BlePubGattClientReadData().

```
0x0000  Success
0x0001  Unknown or not applicable status
0x0100  ATT Error: Invalid Error Code
0x0101  ATT Error: Invalid Attribute Handle
0x0102  ATT Error: Read not permitted
0x0103  ATT Error: Write not permitted
0x0104  ATT Error: Used in ATT as Invalid PDU
0x0105  ATT Error: Authenticated link required
0x0106  ATT Error: Used in ATT as Request Not Supported
0x0107  ATT Error: Offset specified was past the end of the attribute
0x0108  ATT Error: Used in ATT as Insufficient Authorisation
0x0109  ATT Error: Used in ATT as Prepare Queue Full
0x010A  ATT Error: Used in ATT as Attribute not found
0x010B  ATT Error: Attribute cannot be read or written using read/write blob requests
0x010C  ATT Error: Encryption key size used is insufficient
0x010D  ATT Error: Invalid value size
0x010E  ATT Error: Very unlikely error
0x010F  ATT Error: Encrypted link required
0x0110  ATT Error: Attribute type is not a supported grouping attribute
0x0111  ATT Error: Encrypted link required
0x0112  ATT Error: Reserved for Future Use range #1 begin
0x017F  ATT Error: Reserved for Future Use range #1 end
0x0180  ATT Error: Application range begin
0x019F  ATT Error: Application range end
0x01A0  ATT Error: Reserved for Future Use range #2 begin
0x01DF  ATT Error: Reserved for Future Use range #2 end
0x01E0  ATT Error: Reserved for Future Use range #3 begin
0x01FC  ATT Error: Reserved for Future Use range #3 end
0x01FD  ATT Common Profile and Service Error: Client Characteristic Configuration Descriptor
                                              (CCCD) improperly configured
0x01FE  ATT Common Profile and Service Error:Procedure Already in Progress
0x01FF  ATT Common Profile and Service Error: Out Of Range
```

BLEGATTCREAD (connHndl, attrHndl, offset)

A typical pseudo code for reading the content of an attribute calling BleGattcRead() which in turn will result in the EVATTRREAD event message and typically is as follows:-

```
Register a handler for the EVATTRREAD event message

On EVATTRREAD event message
    If Gatt_Status == 0 then
```

```
            BleGattcReadData()  //to actually get the data
        Else
            Attribute could not be read

    Call BleGattcRead()
    If BleGattcRead() ok then Wait for EVATTRREAD
```

Returns	INTEGER, a result code. Typical value: 0x0000 Indicates a successful operation and means an EVATTRREAD event message WILL be thrown by the smartBASIC runtime engine containing the results. A non-zero return value implies an EVATTRREAD message will NOT be thrown.
Arguments	
connHndl	byVal *connHndl* AS INTEGER This is the connection handle as returned in the on-connect event for the connection on whic the remote Gatt Server can be accessed. This will have been returned in the EVBLEMSG event message with msgId == 0 and msgCtx will have been the connection handle.
attrHndl	byVal *attrHndl* AS INTEGER Set this to the handle of the attribute to read and will be a value in the range 1 to 65535
offset	byVal *offset* AS INTEGER This is the offset from which the data in the attribute is to be read.
Interactive Command	No

BLEGATTCREADDATA (connHndl, attrHndl, offset, attrData$)

This function is used to collect the data from the underlying cache when the EVATTRREAD event message has a success gatt status code.

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
connHndl	byVal *connHndl* AS INTEGER This is the connection handle as returned in the on-connect event for the connection on whic the remote Gatt Server can be accessed. This will have been returned in the EVBLEMSG event message with msgId == 0 and msgCtx will have been the connection handle.
attrHndl	byRef *attrHndl* AS INTEGER The handle for the attribute that was read is returned in this variable. Will be the same as the one supplied in BleGattcRead, but supplied here so that the code can be stateless.
offset	byRef *offset* AS INTEGER The offset into the attribute data that was read is returned in this variable. Will be the same as the one supplied in BleGattcRead, but supplied here so that the code can be stateless.
attrData$	byRef *attrData$* AS STRING The attribute data which was read is supplied in this parameter.
Interactive Command	No

```
//Example :: BleGattcRead.sb (See in BL600CodeSnippets.zip)
//
```

```
//Remote server has 3 prim services with 16 bit uuid. First service has one
//characteristic whose value attribute is at handle 3 and has read/write props
//
// Server created using BleGattcTblRead.sub invoked in _OpenMcp.scr
// using Nordic Usb Dongle PC10000

DIM rc,at$,conHndl,uHndl,nOff,atHndl


//============================================================================
// Initialise and instantiate service, characteristic, start adverts
//============================================================================
FUNCTION OnStartup()
    DIM rc, adRpt$, addr$, scRpt$
    rc=BleAdvRptInit(adRpt$, 2, 0, 10)
    IF rc==0 THEN : rc=BleScanRptInit(scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvRptsCommit(adRpt$,scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvertStart(0,addr$,50,0,0) : ENDIF
    //open the gatt client with default notify/indicate ring buffer size
    IF rc==0 THEN : rc = BleGattcOpen(0,0) : ENDIF
ENDFUNC rc


//============================================================================
// Close connections so that we can run another app without problems
//============================================================================
SUB CloseConnections()
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
ENDSUB


//============================================================================
// Ble event handler
//============================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    DIM uHndA
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n- Disconnected"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n- Connected, so read attibute handle 3"
        atHndl = 3
        nOff = 0
        rc=BleGattcRead(conHndl,atHndl,nOff)
        IF rc==0 THEN
            WAITEVENT
        ENDIF
        PRINT "\nread attibute handle 300 which does not exist"
        atHndl = 300
        nOff = 0
        rc=BleGattcRead(conHndl,atHndl,nOff)
        IF rc==0 THEN
            WAITEVENT
        ENDIF
        CloseConnections()
    ENDIF
ENDFUNC 1

'//============================================================================
'//============================================================================
```

```
function HandlerAttrRead(cHndl,aHndl,nSts) as integer
    dim nOfst,nAhndl,at$
    print "\nEVATTRREAD "
    print " cHndl=";cHndl
    print " attrHndl=";aHndl
    print " status=";integer.h' nSts
    if nSts == 0 then
        print "\nAttribute read OK"
        rc = BleGattcReadData(cHndl,nAhndl,nOfst,at$)
        print "\nData    = ";StrHexize$(at$)
        print " Offset= ";nOfst
        print " Len=";strlen(at$)
        print "\nhandle = ";nAhndl
    else
        print "\nFailed to read attribute"
    endif
endfunc 0


//===============================================================================
// Main() equivalent
//===============================================================================
ONEVENT  EVBLEMSG           CALL HndlrBleMsg
OnEvent  EVATTRREAD         call HandlerAttrRead

IF OnStartup()==0 THEN
    PRINT "\nAdvertising, and Gatt Client is open\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF

WAITEVENT
PRINT "\nExiting..."
```

Expected Output:

```
Advertising, and Gatt Client is open

- Connected, so read attibute handle 3
EVATTRREAD  cHndl=2960 attrHndl=3 status=00000000
Attribute read OK
Data    = 00000000 Offset= 0 Len=4
handle = 3
read attibute handle 300 which does not exist
EVATTRREAD  cHndl=2960 attrHndl=300 status=00000101
Failed to read attribute

- Disconnected
Exiting...
```

BLEGATTCREAD and BLEGATTREADDATA are extension functions.

BleGattcWrite

FUNCTION

If the handle for an attribute is known then this function is used to write into an attribute starting at offset 0. The acknowledgement will be returned via a EVATTRWRITE event message.

Given that the success or failure of this write operation is returned in an event message, a handler **must** be registered for the EVATTRWRITE event.

Depending on the connection interval, the write to the attribute may take many 100s of milliseconds, and while this is in progress it is safe to do other non Gatt related operations like for example servicing sensors and displays or any of the onboard peripherals.

EVATTRWRITE event message

This event message **WILL** be thrown if BleGattcWrite() returns a success. The message contains 3 INTEGER parameters:-

Connection Handle
Handle of the Attribute
Gatt status of the write operation.

'**Gatt status of the write operation**' is one of the following values, where 0 implies the write was successfully expedited.

```
0x0000  Success
0x0001  Unknown or not applicable status
0x0100  ATT Error: Invalid Error Code
0x0101  ATT Error: Invalid Attribute Handle
0x0102  ATT Error: Read not permitted
0x0103  ATT Error: Write not permitted
0x0104  ATT Error: Used in ATT as Invalid PDU
0x0105  ATT Error: Authenticated link required
0x0106  ATT Error: Used in ATT as Request Not Supported
0x0107  ATT Error: Offset specified was past the end of the attribute
0x0108  ATT Error: Used in ATT as Insufficient Authorisation
0x0109  ATT Error: Used in ATT as Prepare Queue Full
0x010A  ATT Error: Used in ATT as Attribute not found
0x010B  ATT Error: Attribute cannot be read or written
                   using read/write blob requests
0x010C  ATT Error: Encryption key size used is insufficient
0x010D  ATT Error: Invalid value size
0x010E  ATT Error: Very unlikely error
0x010F  ATT Error: Encrypted link required
0x0110  ATT Error: Attribute type is not a supported grouping attribute
0x0111  ATT Error: Encrypted link required
0x0112  ATT Error: Reserved for Future Use range #1 begin
0x017F  ATT Error: Reserved for Future Use range #1 end
0x0180  ATT Error: Application range begin
0x019F  ATT Error: Application range end
0x01A0  ATT Error: Reserved for Future Use range #2 begin
0x01DF  ATT Error: Reserved for Future Use range #2 end
0x01E0  ATT Error: Reserved for Future Use range #3 begin
0x01FC  ATT Error: Reserved for Future Use range #3 end
0x01FD  ATT Common Profile and Service Error:
                   Client Characteristic Configuration Descriptor (CCCD)
                   improperly configured
0x01FE  ATT Common Profile and Service Error:
                   Procedure Already in Progress
```

```
0x01FF  ATT Common Profile and Service Error:
                  Out Of Range
```

BLEGATTCWRITE (connHndl, attrHndl, attrData$)

A typical pseudo code for writing to an attribute which will result in the EVATTRWRITE event message and typically is as follows:-

```
Register a handler for the EVATTRWRITE event message

On EVATTWRITE event message
    If Gatt_Status == 0 then
        Attribute was written successfully
    Else
        Attribute could not be written

Call BleGattcWrite()
If BleGattcWrite() ok then Wait for EVATTRWRITE
```

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
connHndl	byVal *connHndl* AS INTEGER This is the connection handle as returned in the on-connect event for the connection on which the remote Gatt Server can be accessed. This will have been returned in the EVBLEMSG event message with msgId == 0 and msgCtx will have been the connection handle.
attrHndl	byVal *attrHndl* AS INTEGER The handle for the attribute that is to be written to.
attrData$	byRef *attrData$* AS STRING The attribute data to write
Interactive Command	No

```
//Example :: BleGattcWrite.sb (See in BL600CodeSnippets.zip)
//
//Remote server has 3 prim services with 16 bit uuid. First service has one
//characteristic whose value attribute is at handle 3 and has read/write props
//
// Server created using BleGattcTblWrite.sub invoked in _OpenMcp.scr
// using Nordic Usb Dongle PC10000

DIM rc,at$,conHndl,uHndl,atHndl


//============================================================================
// Initialise and instantiate service, characteristic, start adverts
//============================================================================
FUNCTION OnStartup()
    DIM rc, adRpt$, addr$, scRpt$
    rc=BleAdvRptInit(adRpt$, 2, 0, 10)
    IF rc==0 THEN : rc=BleScanRptInit(scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvRptsCommit(adRpt$,scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvertStart(0,addr$,50,0,0) : ENDIF
    //open the gatt client with default notify/indicate ring buffer size
    IF rc==0 THEN : rc = BleGattcOpen(0,0) : ENDIF
```

```
ENDFUNC rc


//===========================================================================
// Close connections so that we can run another app without problems
//===========================================================================
SUB CloseConnections()
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
ENDSUB


//===========================================================================
// Ble event handler
//===========================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    DIM uHndA
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n- Disconnected"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n- Connected, so write to attibute handle 3"
        atHndl = 3
        at$="\01\02\03\04"
        rc=BleGattcWrite(conHndl,atHndl,at$)
        IF rc==0 THEN
            WAITEVENT
        ENDIF
        PRINT "\nwrite to attibute handle 300 which does not exist"
        atHndl = 300
        rc=BleGattcWrite(conHndl,atHndl,at$)
        IF rc==0 THEN
            WAITEVENT
        ENDIF
        CloseConnections()
    ENDIF
ENDFUNC 1

'//===========================================================================
'//===========================================================================
function HandlerAttrWrite(cHndl,aHndl,nSts) as integer
    dim nOfst,nAhndl,at$
    print "\nEVATTRWRITE "
    print " cHndl=";cHndl
    print " attrHndl=";aHndl
    print " status=";integer.h' nSts
    if nSts == 0 then
        print "\nAttribute write OK"
    else
        print "\nFailed to write attribute"
    endif
endfunc 0


//===========================================================================
// Main() equivalent
//===========================================================================
ONEVENT  EVBLEMSG           CALL HndlrBleMsg
OnEvent  EVATTRWRITE        call HandlerAttrWrite

IF OnStartup()==0 THEN
```

```
    PRINT "\nAdvertising, and Gatt Client is open\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF

WAITEVENT
PRINT "\nExiting..."
```

Expected Output:

```
Advertising, and Gatt Client is open

- Connected, so read attibute handle 3
EVATTRWRITE  cHndl=2687 attrHndl=3 status=00000000
Attribute write OK
Write to attibute handle 300 which does not exist
EVATTRWRITE  cHndl=2687 attrHndl=300 status=00000101
Failed to write attribute

- Disconnected
Exiting...
```

BLEGATTCWRITE is an extension function.

BleGattcWriteCmd

FUNCTION

If the handle for an attribute is known then this function is used to write into an attribute at offset 0when no acknowledgment response is expected. The signal that the command has actually been transmitted and that the remote link layer has acknowledged is by the EVNOTIFYBUF event.

Note that the acknowledgement received for the BleGattcWrite() command is from the higher level GATT layer, not to be confused with the link layer ack in this case.

> *All packets are acknowledged at link layer level. If a packet fails to get through then that condition will manifest as a connection drop due to the link supervision timeout.*

Given that the transmission and link layer ack of this write operation is indicated in an event message, a handler **must** be registered for the EVNOTIBUF event.

Depending on the connection interval, the write to the attribute may take many 100s of milliseconds, and while this is in progress it is safe to do other non Gatt related operations like for example servicing sensors and displays or any of the onboard peripherals.

EVNOTIFYBUF event

This event message **WILL** be thrown if BleGattcWriteCmd() returned a success. The message contains no parameters.

BLEGATTCWRITECMD (connHndl, attrHndl, attrData$)

A typical pseudo code for writing to an attribute which will result in the EVNOTIFYBUF event is as follows:-

```
Register a handler for the EVNOTIFYBUF event message

On EVNOTIFYBUF event message
```

```
        Can now send another write command

    Call BleGattcWriteCmd()
    If BleGattcWrite() ok then Wait for EVNOTIFYBUF
```

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
connHndl	byVal *connHndl* AS INTEGER This is the connection handle as returned in the on-connect event for the connection on whic the remote Gatt Server can be accessed. This will have been returned in the EVBLEMSG event message with msgId == 0 and msgCtx will have been the connection handle.
attrHndl	byVal *attrHndl* AS INTEGER The handle for the attribute that is to be written to.
attrData$	byRef *attrData$* AS STRING The attribute data to write.
Interactive Command	No

```
//Example :: BleGattcWriteCmd.sb (See in BL600CodeSnippets.zip)
//
//Remote server has 3 prim services with 16 bit uuid. First service has one
//characteristic whose value attribute is at handle 3 and has read/write props
//
// Server created using BleGattcTblWriteCmd.sub invoked in _OpenMcp.scr
// using Nordic Usb Dongle PC10000

DIM rc,at$,conHndl,uHndl,atHndl


//============================================================================
// Initialise and instantiate service, characteristic, start adverts
//============================================================================
FUNCTION OnStartup()
    DIM rc, adRpt$, addr$, scRpt$
    rc=BleAdvRptInit(adRpt$, 2, 0, 10)
    IF rc==0 THEN : rc=BleScanRptInit(scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvRptsCommit(adRpt$,scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvertStart(0,addr$,50,0,0) : ENDIF
    //open the gatt client with default notify/indicate ring buffer size
    IF rc==0 THEN : rc = BleGattcOpen(0,0) : ENDIF
ENDFUNC rc


//============================================================================
// Close connections so that we can run another app without problems
//============================================================================
SUB CloseConnections()
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
ENDSUB


//============================================================================
// Ble event handler
//============================================================================
```

```
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    DIM uHndA
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n- Disconnected"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n- Connected, so write to attribute handle 3"
        atHndl = 3
        at$="\01\02\03\04"
        rc=BleGattcWriteCmd(conHndl,atHndl,at$)
        IF rc==0 THEN
            WAITEVENT
        ENDIF
        PRINT "\n- write again to attribute handle 3"
        atHndl = 3
        at$="\05\06\07\08"
        rc=BleGattcWriteCmd(conHndl,atHndl,at$)
        IF rc==0 THEN
            WAITEVENT
        ENDIF
        PRINT "\n- write again to attribute handle 3"
        atHndl = 3
        at$="\09\0A\0B\0C"
        rc=BleGattcWriteCmd(conHndl,atHndl,at$)
        IF rc==0 THEN
            WAITEVENT
        ENDIF
        PRINT "\nwrite to attribute handle 300 which does not exist"
        atHndl = 300
        rc=BleGattcWriteCmd(conHndl,atHndl,at$)
        IF rc==0 THEN
            PRINT "\nEven when the attribute does not exist an event will occur"
            WAITEVENT
        ENDIF
        CloseConnections()
    ENDIF
ENDFUNC 1

'//============================================================================
'//============================================================================
function HandlerNotifyBuf() as integer
  print "\nEVNOTIFYBUF Event"
endfunc 0  '//need to progress the WAITEVENT

//============================================================================
// Main() equivalent
//============================================================================
ONEVENT  EVBLEMSG           CALL HndlrBleMsg
OnEvent  EVNOTIFYBUF        call HandlerNotifyBuf

IF OnStartup()==0 THEN
    PRINT "\nAdvertising, and Gatt Client is open\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF

WAITEVENT
PRINT "\nExiting..."
```

Expected Output:

```
Advertising, and Gatt Client is open

- Connected, so write to attribute handle 3
EVNOTIFYBUF Event
- write again to attribute handle 3
EVNOTIFYBUF Event
- write again to attribute handle 3
EVNOTIFYBUF Event
write to attribute handle 300 which does not exist
Even when the attribute does not exist an event will occur
EVNOTIFYBUF Event

- Disconnected
Exiting...
```

BLEGATTCWRITECMD is an extension function.

BleGattcNotifyRead

FUNCTION

A Gatt Server has the ability to notify or indicate the value attribute of a characteristic when enabled via the Client Characeristic Configuration Descriptor (CCCD). This means data will arrive from a Gatt Server at any time and so has to be managed so that it can synchronised with the smartBASIC runtime engine.

Data arriving via a notification does not require Gatt acknowledgements, however indications require them. This Gatt Client manager saves data arriving via a notification in the same ring buffer for later extraction using the command BleGattcNotifyRead() and for indications an automatic gatt acknowledgement is sent when the data is saved in the ring buffer. This acknowledgment happens even if the data was discarded because the ring buffer was full. If however it is required that the data NOT be acknowledged when it is discarded on a full buffer then set the flags parameter in the BleGattcOpen() function where the Gatt Client manager is opened.

In the case when an ack is NOT sent on data discard, the Gatt Server will be throttled and so no further data will be notified or indicated by it until BleGattNotifyRead() is called to extract data from the ring buffer to create space and it will trigger a delayed acknowledgement.

When the Gatt Client manager is opened using BleGattcOpen() it is possible to specify the size of the ring buffer. If a value of 0 is supplied then a default size is created. SYSINFO(2019) in a smartBASIC application or the interactive mode command AT I 2019 will return the default size. Likewise SYSINFO(2020) or the command AT I 2020 will return the maximum size.

Data that arrives via notifications or indications get stored in the ring buffer and at the same time a EVATTRNOTIFY event is thrown to the smartBASIC runtime engine. This is an event, in the same way an incoming UART receive character generates an event, that is, no data payload is attached to the event.

EVATTRTOTIFY event message

This event **WILL** be thrown when an notification or an indication arrives from a gatt server . The event contains no parameters. Please note that if one notification/indication arrives or many, like in the case of UART events, the same event mask bit is asserted. The paradigm being that the smartBASIC application is informed that it needs to go and service the ring buffer using the function BleGattcNotifyRead.

BLEGATTCNOTIFYREAD (connHndl, attrHndl, attrData$, discardCount)

A typical pseudo code for handling and accessing notification/indication data is as follows:-

```
Register a handler for the EVATTRNOTIFY event message

On EVATTRNOTIRY event
    BleGattcNotifyRead()  //to actually get the data
    Process the data

Enable notifications and/or indications via CCCD descriptors
```

Returns	INTEGER, a result code. Typical value: 0x0000 (indicates a successful operation)
Arguments	
connHndl	byRef *connHndl* AS INTEGER On exit this will be the connection handle of the gatt server that sent the notification or indication.
attrHndl	byRef *attrHndl* AS INTEGER On exit this will be the handle of the characteristic value attribute in the notification or indication.
attrData$	byRef *attrData$* AS STRING On exit this will be the data of the characteristic value attribute in the notification or indication. It is always from offset 0 of the source attribute.
discardedCount	byRef *discardedCount* AS INTEGER On exit this should contain 0 and it signifies the total number of notifications or indications that got discared because the ring buffer in the gatt client manager was full. If non-zero values are encountered, it is recommended that the ring buffer size be increased by using BleGattcClose() when the gatt client was opened using BleGattcOpen().
Interactive Command	No

```
//Example :: BleGattcNotifyRead.sb (See in BL600CodeSnippets.zip)
//
// Server created using BleGattcTblNotifyRead.sub invoked in _OpenMcp.scr
// using Nordic Usb Dongle PC10000
//
// Charactersitic at handle 15 has notify   (16==cccd)
// Charactersitic at handle 18 has indicate (19==cccd)

DIM rc,at$,conHndl,uHndl,atHndl



//============================================================================
// Initialise and instantiate service, characteristic, start adverts
//============================================================================
FUNCTION OnStartup()
    DIM rc, adRpt$, addr$, scRpt$
    rc=BleAdvRptInit(adRpt$, 2, 0, 10)
    IF rc==0 THEN : rc=BleScanRptInit(scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvRptsCommit(adRpt$,scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvertStart(0,addr$,50,0,0) : ENDIF
    //open the gatt client with default notify/indicate ring buffer size
    IF rc==0 THEN : rc = BleGattcOpen(0,0) : ENDIF
ENDFUNC rc
```

```
//===============================================================================
// Close connections so that we can run another app without problems
//===============================================================================
SUB CloseConnections()
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
ENDSUB


//===============================================================================
// Ble event handler
//===============================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n- Disconnected"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n- Connected, so enable notification for char with cccd at 16"
        atHndl = 16
        at$="\01\00"
        rc=BleGattcWrite(conHndl,atHndl,at$)
        IF rc==0 THEN
            WAITEVENT
        ENDIF
        PRINT "\n- enable indication for char with cccd at 19"
        atHndl = 19
        at$="\02\00"
        rc=BleGattcWrite(conHndl,atHndl,at$)
        IF rc==0 THEN
            WAITEVENT
        ENDIF
    ENDIF
ENDFUNC 1

'//===============================================================================
'//===============================================================================
function HandlerAttrWrite(cHndl,aHndl,nSts) as integer
    dim nOfst,nAhndl,at$
    print "\nEVATTRWRITE "
    print " cHndl=";cHndl
    print " attrHndl=";aHndl
    print " status=";integer.h' nSts
    if nSts == 0 then
        print "\nAttribute write OK"
    else
        print "\nFailed to write attribute"
    endif
endfunc 0

'//===============================================================================
'//===============================================================================
function HandlerAttrNotify() as integer
    dim chndl,aHndl,att$,dscd
    print "\nEVATTRNOTIFY Event"
    rc=BleGattcNotifyRead(cHndl,aHndl,att$,dscd)
    print "\n  BleGattcNotifyRead()"
    if rc==0 then
        print " cHndl=";cHndl
        print " attrHndl=";aHndl
        print " data=";StrHexize$(att$)
```

```
        print " discarded=";dscd
    else
        print " failed with ";integer.h' rc
    endif
endfunc 1

//===============================================================================
// Main() equivalent
//===============================================================================
ONEVENT  EVBLEMSG           CALL HndlrBleMsg
OnEvent  EVATTRWRITE        call HandlerAttrWrite
OnEvent  EVATTRNOTIFY       call HandlerAttrNotify

IF OnStartup()==0 THEN
    PRINT "\nAdvertising, and Gatt Client is open\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF

WAITEVENT
PRINT "\nExiting..."
```

Expected Output:

```
Advertising, and Gatt Client is open

- Connected, so enable notification for char with cccd at 16
EVATTRWRITE  cHndl=877 attrHndl=16 status=00000000
Attribute write OK
- enable indication for char with cccd at 19
EVATTRWRITE  cHndl=877 attrHndl=19 status=00000000
Attribute write OK
EVATTRNOTIFY Event
  BleGattcNotifyRead() cHndl=877 attrHndl=15 data=BAADC0DE discarded=0
EVATTRNOTIFY Event
  BleGattcNotifyRead() cHndl=877 attrHndl=18 data=DEADBEEF discarded=0
EVATTRNOTIFY Event
  BleGattcNotifyRead() cHndl=877 attrHndl=15 data=BAADC0DE discarded=0
EVATTRNOTIFY Event
   BleGattcNotifyRead() cHndl=877 attrHndl=18 data=DEADBEEF discarded=0
```

BLEGATTCNOTIFYREAD is an extension function.

Attribute Encoding Functions

Data for Characteristics are stored in Value attributes, arrays of bytes. Multibyte Characteristic Descriptors content is stored similarly. Those bytes are manipulated in *smart* BASIC applications using STRING variables.

The Bluetooth specification stipulates that multibyte data entities are stored communicated in little endian format and so all data manipulation is done similarly. Little endian means that a multibyte data entity will be

stored so that lowest significant byte is position at the lowest memory address and likewise when transported, the lowest byte will get on the wire first.

This section describes all the encoding functions which allow those strings to be written to in smaller bytewise subfields in a more efficient manner compared to the generic STRXXXX functions that are made available in *smart* BASIC.

Note: CCCD and SCCD Descriptors are special cases; they have two bytes which are treated as 16 bit integers. This is reflected in smartBASIC applications so that INTEGER variables are used to manipulate those values instead of STRINGS.

BleEncode8

FUNCTION

This function overwrites a single byte in a string at a specified offset. If the string is not long enough, then it will be extended with the new extended block uninitialized and then the byte specified is overwritten.

If the nIndex is such that the new string length exceeds the maximum attribute length, this function fails. The maximum attribute length can be obtained using the function SYSINFO(n) where n is 2013. The Bluetooth specification allows a length between 1 and 512.

BLEENCODE8 (attr$,nData, nIndex)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
attr$	byRef *attr$* AS STRING This argument is the string that will be written to an attribute
nData	byVal nData AS INTEGER The least significant byte of this integer is saved. The rest is ignored.
nIndex	byVal *nIndex* AS INTEGER This is the zero-based index into the string attr$ where the new fragment of data is written to. If the string attr$ is not long enough to accommodate the index plus the length of the fragment, it is extended. If the extended length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function fails.
Interactive Command	No

```
//Example :: BleEncode8.sb (See in BL600CodeSnippets.zip)

DIM rc
DIM attr$

attr$="Laird"

PRINT "\nattr$=";attr$

//Remember: - 4 bytes are used to store an integer on the BL600

//write 'C' to index 2 -- '111' will be ignored
rc=BleEncode8(attr$,0x11143,2)
//write 'A' to index 0
rc=BleEncode8(attr$,0x41,0)
//write 'B' to index 1
rc=BleEncode8(attr$,0x42,1)
//write 'D' to index 3
rc=BleEncode8(attr$,0x44,3)
//write 'y' to index 7 -- attr$ will be extended
rc=BleEncode8(attr$,0x67, 7)

PRINT "\nattr$ now = ";attr$
```

Expected Output:

```
attr$=Laird
attr$ now = ABCDd\00\00g
```

BLEENCODE8 is an extension function.

BleEncode16

FUNCTION

This function overwrites two bytes in a string at a specified offset. If the string is not long enough, then it is extended with the new extended block uninitialized and then the bytes specified are overwritten.

If the nIndex is such that the new string length exceeds the maximum attribute length, this function fails. The maximum attribute length can be obtained using the function SYSINFO(n) where n is 2013. The Bluetooth specification allows a length between 1 and 512.

BLEENCODE16 (attr$,nData, nIndex)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
attr$	byRef *attr$* AS STRING This argument is the string that will be written to an attribute.
nData	byVal nData AS INTEGER The two least significant bytes of this integer is saved. The rest is ignored.
nIndex	byVal *nIndex* AS INTEGER This is the zero based index into the string attr$ where the new fragment of data is written. If the string attr$ is not long enough to accommodate the index plus the length of the fragment, it is extended. If the extended length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function fails.
Interactive Command	No

```
//Example :: BleEncode16.sb (See in BL600CodeSnippets.zip)

DIM rc, attr$
attr$="Laird"
PRINT "\nattr$=";attr$

//write 'CD' to index 2
rc=BleEncode16(attr$,0x4443,2)
//write 'AB' to index 0 - '2222' will be ignored
rc=BleEncode16(attr$,0x22224241,0)
//write 'EF' to index 3
rc=BleEncode16(attr$,0x4645,4)

PRINT "\nattr$ now = ";attr$
```

Expected Output:

```
attr$=Laird
attr$ now = ABCDEF
```

BLEENCODE16 is an extension function.

BleEncode24

FUNCTION

This function overwrites three bytes in a string at a specified offset. If the string is not long enough, then it will be extended with the new extended block uninitialized and then the bytes specified are overwritten.

If the nIndex is such that the new string length exceeds the maximum attribute length, this function fails. The maximum attribute length can be obtained using the function SYSINFO(n) where n is 2013. The Bluetooth specification allows a length between 1 and 512.

BLEENCODE24 (attr$,nData, nIndex)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
attr$	byRef *attr$* AS STRING This argument is the string that will be written to an attribute.
nData	byVal nData AS INTEGER The three least significant bytes of this integer is saved. The rest is ignored.
nIndex	byVal *nIndex* AS INTEGER This is the zero based index into the string attr$ where the new fragment of data is written. If the string attr$ is not long enough to accommodate the index plus the length of the fragment, it is extended. If the extended length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function will fail.
Interactive Command	No

```
//Example :: BleEncode24.sb (See in BL600CodeSnippets.zip)

DIM rc
DIM attr$ : attr$="Laird"

//write 'BCD' to index 1
rc=BleEncode24(attr$,0x444342,1)
//write 'A' to index 0
rc=BleEncode8(attr$,0x41,0)
//write 'EF'to index 4
rc=BleEncode16(attr$,0x4645,4)

PRINT "attr$=";attr$
```

Expected Output:

BLEENCODE24 is an extension function.

BleEncode32

FUNCTION

This function overwrites four bytes in a string at a specified offset. If the string is not long enough, then it is extended with the new extended block uninitialized and then the bytes specified are overwritten.

If the nIndex is such that the new string length exceeds the maximum attribute length, this function fails. The maximum attribute length can be obtained using the function SYSINFO(n) where n is 2013. The Bluetooth specification allows a length between 1 and 512.

BLEENCODE32(attr$,nData, nIndex)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
attr$	byRef *attr$* AS STRING This argument is the string that will be written to an attribute.
nData	byVal nData AS INTEGER The four bytes of this integer is saved. The rest is ignored.
nIndex	byVal *nIndex* AS INTEGER This is the zero based index into the string attr$ where the new fragment of data is written. If the string attr$ is not long enough to accommodate the index plus the length of the fragment, it is extended. If the extended length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function fails.
Interactive Command	No

```
//Example :: BleEncode32.sb (See in BL600CodeSnippets.zip)

DIM rc
DIM attr$ : attr$="Laird"

//write 'BCDE' to index 1
rc=BleEncode32(attr$,0x45444342,1)
//write 'A' to index 0
rc=BleEncode8(attr$,0x41,0)

PRINT "attr$=";attr$
```

Expected Output:

```
attr$=ABCDE
```

BLEENCODE32 is an extension function.

BleEncodeFLOAT

FUNCTION

This function overwrites four bytes in a string at a specified offset. If the string is not long enough, it is extended with the new extended block uninitialized and then the byte specified is overwritten.

If the nIndex is such that the new string length exceeds the maximum attribute length, this function fails. The maximum attribute length can be obtained using the function SYSINFO(n) where n is 2013. The Bluetooth specification allows a length between 1 and 512.

BLEENCODEFLOAT (attr$, nMatissa, nExponent, nIndex)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
attr$	byRef *attr$* AS STRING This argument is the string that is written to an attribute.
nMatissa	byVal nMantissa AS INTEGER This value must be in the range -8388600 to +8388600 or the function fails. The data is written in little endian so that the least significant byte is at the lower memory address. **Note:** The range is not +/- 2048 because after encoding the following two byte values have special meaning: 0x07FFFFFF NaN (Not a Number) 0x08000000 NRes (Not at this resolution) 0x07FFFFFE + INFINITY 0x08000002 - INFINITY 0x08000001 Reserved for future use
nExponent	byVal nExponent AS INTEGER This value must be in the range -128 to 127 or the function fails.
nIndex	byVal *nIndex* AS INTEGER This is the zero based index into the string attr$ where the new fragment of data is written. If the string attr$ is not long enough to accommodate the index plus the length of the fragment, it is extended. If the extended length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function fails.
Interactive Command	No

```
//Example :: BleEncodeFloat.sb (See in BL600CodeSnippets.zip)

DIM rc
DIM attr$ : attr$=""

//write 1234567 x 10^-54 as FLOAT to index 2
PRINT BleEncodeFLOAT(attr$,123456,-54,0)

//write 1234567 x 10^1000 as FLOAT to index 2 and it will fail
//because the exponent is too large, it has to be < 127
IF BleEncodeFLOAT(attr$,1234567,1000,2)!=0 THEN
  PRINT "\nFailed to encode to FLOAT"
ENDIF

//write 10000000 x 10^0 as FLOAT to index 2 and it will fail
//because the mantissa is too large, it has to be < 8388600
IF BleEncodeFLOAT(attr$,10000000,0,2)!=0 THEN
  PRINT "\nFailed to encode to FLOAT"
ENDIF
```

Expected Output:

```
0
Failed to encode to FLOAT
Failed to encode to FLOAT
```

BLEENCODEFLOAT is an extension function.

BleEncodeSFLOATEX

FUNCTION

This function overwrites two bytes in a string at a specified offset as short 16 bit float value. If the string is not long enough, it is extended with the extended block uninitialized. Then the bytes are overwritten.

If the nIndex is such that the new string length exceeds the maximum attribute length, this function fails. The maximum attribute length can be obtained using the function SYSINFO(n) where n is 2013. The Bluetooth specification allows a length between 1 and 512.

BLEENCODESFLOATEX(attr$,nData, nIndex)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
attr$	byRef *attr$* AS STRING This argument is the string that will be written to an attribute.
nData	byVal nData AS INTEGER The 32 bit value is converted into a 2 byte IEEE-11073 16 bit SFLOAT consisting of a 12 bit signed mantissa and a 4 bit signed exponent. This means a signed 32 bit value always fits in such a FLOAT enitity, but there will be a loss in significance to 12 from 32.
nIndex	byVal *nIndex* AS INTEGER This is the zero based index into the string attr$ where the new fragment of data is written. If the string attr$ is not long enough to accommodate the index plus the length of the fragment, it is extended. If the new length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function fails.
Interactive Command	No

```
//Example :: BleEncodeSFloatEx.sb (See in BL600CodeSnippets.zip)

DIM rc, mantissa, exp
DIM attr$ : attr$=""

//write 2,147,483,647 as SFLOAT to index 0
rc=BleEncodeSFloatEX(attr$,2147483647,0)
rc=BleDecodeSFloat(attr$,mantissa,exp,0)
PRINT "\nThe number stored is ";mantissa;" x 10^";exp
```

Expected Output:

```
The number stored is 214 x 10^7
```

BLEENCODESFLOAT is an extension function.

BleEncodeSFLOAT

FUNCTION

This function overwrites two bytes in a string at a specified offset as short 16 bit float value. If the string is not long enough, it is extended with the new block uninitialized. Then the byte specified is overwritten.

If the nIndex is such that the new string length exceeds the maximum attribute length, this function fails. The maximum attribute length can be obtained using the function SYSINFO(n) where n is 2013. The Bluetooth specification allows a length between 1 and 512.

BLEENCODESFLOAT(attr$, nMatissa, nExponent, nIndex)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
attr$	byRef *attr$* AS STRING This argument is the string that will be written to an attribute
nMatissa	byVal n AS INTEGER This must be in the range -2046 to +2046 or the function fails. The data is written in little endian so the least significant byte is at the lower memory address. **Note:** The range is not +/- 2048 because after encoding the following two byte values have special meaning: 0x07FF NaN (Not a Number) 0x0800 NRes (Not at this resolution) 0x07FE + INFINITY 0x0802 - INFINITY 0x0801 Reserved for future use
nExponent	byVal n AS INTEGER This value must be in the range -8 to 7 or the function fails.
nIndex	byVal *nIndex* AS INTEGER This is the zero based index into the string attr$ where the new fragment of data is written. If the string attr$ is not long enough to accommodate the index plus the length of the fragment, it is extended. If the new length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function fails.
Interactive Command	No

```
//Example :: BleEncodeSFloat.sb (See in BL600CodeSnippets.zip)

DIM rc
DIM attr$ : attr$=""

SUB Encode(BYVAL mantissa, BYVAL exp)
    IF BleEncodeSFloat(attr$,mantissa,exp,2)!=0 THEN
        PRINT "\nFailed to encode to SFLOAT"
    ELSE
        PRINT "\nSuccess"
    ENDIF
ENDSUB

Encode(1234,-4)     //1234 x 10^-4
Encode(1234,10)     //1234 x 10^10 will fail because exponent too large
Encode(10000,0)     //10000 x 10^0 will fail because mantissa too large
```

Expected Output:

```
Success
Failed to encode to SFLOAT
Failed to encode to SFLOAT
```

BLEENCODESFLOAT is an extension function.

BleEncodeTIMESTAMP

FUNCTION

This function overwrites a seven byte string into the string at a specified offset. If the string is not long enough, it is extended with the new extended block uninitialized and then the byte specified is overwritten.

The seven byte string consists of a byte each for century, year, month, day, hour, minute, and second. If (year * month) is zero, it is taken as "not noted" year and all the other fields are set zero (not noted).

For example, 5 May 2013 10:31:24 will be represented as "\14\0D\05\05\0A\1F\18"

If the nIndex is such that the new string length exceeds the maximum attribute length, this function fails. The maximum length of an attribute as implemented can be obtained using the function SYSINFO(n) where n is 2013. The Bluetooth specification allows a length between 1 and 512.

Note: When the attr$ string variable is updated, the two byte year field is converted into a 16 bit integer. Hence \14\0D gets converted to \DD\07

BLEENCODETIMESTAMP (attr$, timestamp$, nIndex)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
attr$	byRef *attr$* AS STRING This argument is the string that is written to an attribute.
timestamp$	byRef *timestamp$* AS STRING This is an exactly 7 byte string as described above. For example 5 May 2013 10:31:24 is entered "\14\0D\05\05\0A\1F\18"
nIndex	byVal *nIndex* AS INTEGER This is the zero based index into the string attr$ where the new fragment of data is written. If the string attr$ is not long enough to accommodate the index plus the length of the fragment it is extended. If the new length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function fails.
Interactive Command	No

```
//Example :: BleEncodeTimestamp.sb (See in BL600CodeSnippets.zip)

DIM rc, ts$
DIM attr$ : attr$=""

//write the timestamp <5 May 2013 10:31:24>
ts$="\14\0D\05\05\0A\1F\18"
PRINT BleEncodeTimestamp(attr$,ts$,0)
```

Expected Output:

```
0
```

BLEENCODETIMESTAMP is an extension function.

BleEncodeSTRING

FUNCTION

This function overwrites a substring at a specified offset with data from another substring of a string. If the destination string is not long enough, it is extended with the new block uninitialized. Then the byte is overwritten.

If the nIndex is such that the new string length exceeds the maximum attribute length, this function fails. The maximum length of an attribute as implemented can be obtained using the function SYSINFO(n) where n is 2013. The Bluetooth specification allows a length between 1 and 512.

BleEncodeSTRING (attr$,nIndex1 str$, nIndex2,nLen)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
attr$	byRef *attr$* AS STRING This argument is the string that will be written to an attribute
nIndex1	byVal *nIndex1* AS INTEGER This is the zero based index into the string attr$ where the new fragment of data is written If the string attr$ is not long enough to accommodate the index plus the length of the fragment it is extended. If the new length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function fails.
str$	byRef *str$* AS STRING This contains the source data which is qualified by the nIndex2 and nLen arguments that follow.
nIndex2	byVal *nIndex2* AS INTEGER This is the zero based index into the string str$ from which data is copied. No data is copied if this is negative or greater than the string.
nLen	byVal *nLen* AS INTEGER This species the number of bytes from offset nIndex2 to be copied into the destination string. It is clipped to the number of bytes left to copy after the index.
Interactive Command	No

```
//Example :: BleEncodeString.sb (See in BL600CodeSnippets.zip)
DIM rc, attr$, ts$ : ts$="Hello World"
//write "Wor" from "Hello World" to the attribute at index 2
rc=BleEncodeString(attr$,2,ts$,6,3)
PRINT attr$
```

Expected Output:

```
\00\00Wor
```

BLEENCODESTRING is an extension function.

BleEncodeBITS

FUNCTION

This function overwrites some bits of a string at a specified bit offset with data from an integer which is treated as a bit array of length 32. If the destination string is not long enough, it is extended with the new extended block uninitialized. Then the bits specified are overwritten.

If the nIndex is such that the new string length exceeds the maximum attribute length, this function fails. The maximum length of an attribute as implemented can be obtained using the function SYSINFO(n) where n is 2013. The Bluetooth specification allows a length between 1 and 512; hence the (nDstIdx + nBitLen) cannot be greater than the max attribute length times 8.

BleEncodeBITS (attr$,nDstIdx, srcBitArr , nSrcIdx, nBitLen)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
attr$	byRef *attr$* AS STRING This is the string written to an attribute. It is treated as a bit array.
nDstIdx	byVal *nDstIdx* AS INTEGER This is the zero based bit index into the string attr$, treated as a bit array, where the new fragment of data bits is written. If the string attr$ is not long enough to accommodate the index plus the length of the fragment it is extended. If the new length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function fails.
srcBitArr	byVal *srcBitArr* AS INTEGER This contains the source data bits which is qualified by the nSrcIdx and nBitLen arguments that follow.
nSrcIdx	byVal *nSrcIdx* AS INTEGER This is the zero based bit index into the bit array contained in srcBitArr from where the data bits will be copied. No data is copied if this index is negative or greater than 32.
nBitLen	byVal *nBitLen* AS INTEGER This species the number of bits from offset nSrcIdx to be copied into the destination bit array represented by the string attr$. It will be clipped to the number of bits left to copy after the index nSrcIdx.
Interactive Command	No

```
//Example :: BleEncodeBits.sb (See in BL600CodeSnippets.zip)
DIM attr$, rc, bA: bA=b'1110100001111
rc=BleEncodeBits(attr$,20,bA,7,5) : PRINT attr$ //copy 5 bits from index 7 to attr$
```

Expected Output:

```
\00\00\A0\01
```

BLEENCODEBITS is an extension function.

Attribute Decoding Functions

Data in a Characteristic is stored in a Value attribute, a byte array. Multibyte Characteristic Descriptors content are stored similarly. Those bytes are manipulated in smartBASIC applications using STRING variables.

Attibute data is stored in little endian format.

This section describes decoding functions that allow attribute strings to be read from smaller bytewise subfields more efficiently than the generic STRXXXX functions that are made available in *smart*BASIC.

Please note that CCCD and SCCD Descriptors are special cases as they are defined as having just 2 bytes which are treated as 16 bit integers mapped to INTEGER variables in smartBASIC.

BleDecodeS8

FUNCTION

This function reads a single byte in a string at a specified offset into a 32bit integer variable with sign extension. If the offset points beyond the end of the string then this function fails and returns zero.

BLEDECODES8 (attr$,nData, nIndex)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
attr$	byRef *attr$* AS STRING This references the attribute string from which the function reads.
nData	byRef nData AS INTEGER This references an integer to be updated with the 8 bit data from attr$, after sign extension.
nIndex	byVal *nIndex* AS INTEGER This is the zero based index into the string attr$ from which the data is read. If the string attr$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.
Interactive Command	No

```
//Example :: BleDecodeS8.sb (See in BL600CodeSnippets.zip)

DIM chrHandle,v1,svcHandle,rc
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
DIM attr$ : attr$="\00\01\02\03\04\85\86\87\88\89"
DIM uuid : uuid = 0x1853

//create random service just for this example
rc=BleServiceNew(1, BleHandleUuid16(uuid), svcHandle)

//create char and commit as part of service commited above
rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)

rc=BleServiceCommit(svcHandle)

rc=BleCharValueRead(chrHandle,attr$)

//read signed byte from index 2
rc=BleDecodeS8(attr$,v1,2)
PRINT "\ndata in Hex = 0x"; INTEGER.H'v1
PRINT "\ndata in Decimal = "; v1;"\n"

//read signed byte from index 6 - two's complement of -122
rc=BleDecodeS8(attr$,v1,6)
PRINT "\ndata in Hex = 0x"; INTEGER.H'v1
PRINT "\ndata in Decimal = "; v1;"\n"
```

Expected Output:

```
data in Hex = 0x00000002
data in Decimal = 2

data in Hex = 0xFFFFFF86
data in Decimal = -122
```

BLEDECODES8 is an extension function.

BleDecodeU8

FUNCTION

This function reads a single byte in a string at a specified offset into a 32bit integer variable **without** sign extension. If the offset points beyond the end of the string, this function fails.

BLEDECODEU8 (attr$,nData, nIndex)

Returns	INTEGER, the number of bytes extracted from the attribute string. Can be less than the size expected if the nIndex parameter is positioned towards the end of the string.
Arguments	
attr$	byRef *attr$* AS STRING This references the attribute string from which the function reads.
nData	byRef nData AS INTEGER This references an integer to be updated with the 8 bit data from attr$, without sign extension.
nIndex	byVal *nIndex* AS INTEGER This is the zero based index into the string attr$ from which data is read. If the string attr$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.
Interactive Command	No

```
//Example :: BleDecodeU8.sb (See in BL600CodeSnippets.zip)

 DIM chrHandle,v1,svcHandle,rc
 DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
 DIM attr$ : attr$="\00\01\02\03\04\85\86\87\88\89"
 DIM uuid : uuid = 0x1853

 rc=BleServiceNew(1, BleHandleUuid16(uuid), svcHandle)

 rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
 rc=BleCharCommit(svcHandle,attr$,chrHandle)

 rc=BleServiceCommit(svcHandle)

 rc=BleCharValueRead(chrHandle,attr$)

 //read unsigned byte from index 2
 rc=BleDecodeU8(attr$,v1,2)
 PRINT "\ndata in Hex = 0x"; INTEGER.H'v1
 PRINT "\ndata in Decimal = "; v1;"\n"

 //read unsigned byte from index 6
 rc=BleDecodeU8(attr$,v1,6)
```

```
PRINT "\ndata in Hex = 0x"; INTEGER.H'v1
PRINT "\ndata in Decimal = "; v1;"\n"
```

Expected Output:

```
data in Hex = 0x00000002
data in Decimal = 2

data in Hex = 0x00000086
data in Decimal = 134
```

BLEDECODEU8 is an extension function.

BleDecodeS16

FUNCTION

This function reads two bytes in a string at a specified offset into a 32bit integer variable with sign extension. If the offset points beyond the end of the string then this function fails.

BLEDECODES16 (attr$,nData, nIndex)

Returns	INTEGER, the number of bytes extracted from the attribute string. Can be less than the size expected if the nIndex parameter is positioned towards the end of the string.
Arguments	
attr$	byRef *attr$* AS STRING This references the attribute string from which the function reads.
nData	byRef nData AS INTEGER This references an integer to be updated with the 2 byte data from attr$, after sign extension.
nIndex	byVal *nIndex* AS INTEGER This is the zero based index into the string attr$ from which data is read. If the string attr$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.
Interactive Command	No

```
//Example :: BleDecodeS16.sb (See in BL600CodeSnippets.zip)

DIM chrHandle,v1,svcHandle,rc
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
DIM attr$ : attr$="\00\01\02\03\04\85\86\87\88\89"
DIM uuid : uuid = 0x1853

rc=BleServiceNew(1, BleHandleUuid16(uuid), svcHandle)

rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)

rc=BleServiceCommit(svcHandle)

rc=BleCharValueRead(chrHandle,attr$)

//read 2 signed bytes from index 2
rc=BleDecodeS16(attr$,v1,2)
PRINT "\ndata in Hex = 0x"; INTEGER.H'v1
PRINT "\ndata in Decimal = "; v1;"\n"
```

```
//read 2 signed bytes from index 6
rc=BleDecodeS16(attr$,v1,6)
PRINT "\ndata in Hex = 0x"; INTEGER.H'v1
PRINT "\ndata in Decimal = "; v1;"\n"
```

Expected Output:

```
data in Hex = 0x00000302
data in Decimal = 770

data in Hex = 0xFFFF8786
data in Decimal = -30842
```

BLEDECODES16 is an extension function.

BleDecodeU16

This function reads two bytes from a string at a specified offset into a 32bit integer variable **without** sign extension. If the offset points beyond the end of the string then this function fails.

BLEDECODEU16 (attr$,nData, nIndex)

Returns	INTEGER, the number of bytes extracted from the attribute string. Can be less than the size expected if the nIndex parameter is positioned towards the end of the string.
Arguments	
attr$	byRef *attr$* AS STRING This references the attribute string from which the function reads.
nData	byRef nData AS INTEGER This references an integer to be updated with the 2 byte data from attr$, without sign extension.
nIndex	byVal *nIndex* AS INTEGER This is the zero based index into the string attr$ from which data is read. If the string attr$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.
Interactive Command	No

```
//Example :: BleDecodeU16.sb (See in BL600CodeSnippets.zip)

DIM chrHandle,v1,svcHandle,rc
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
DIM attr$ : attr$="\00\01\02\03\04\85\86\87\88\89"
DIM uuid : uuid = 0x1853

rc=BleServiceNew(1, BleHandleUuid16(uuid), svcHandle)

rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)

rc=BleServiceCommit(svcHandle)

rc=BleCharValueRead(chrHandle,attr$)

//read 2 unsigned bytes from index 2
rc=BleDecodeU16(attr$,v1,2)
PRINT "\ndata in Hex = 0x"; INTEGER.H'v1
```

```
PRINT "\ndata in Decimal = "; v1;"\n"

//read 2 unsigned bytes from index 6
rc=BleDecodeU16(attr$,v1,6)
PRINT "\ndata in Hex = 0x"; INTEGER.H'v1
PRINT "\ndata in Decimal = "; v1;"\n"
```

Expected Output:

```
data in Hex = 0x00000302
data in Decimal = 770

data in Hex = 0x00008786
data in Decimal = 34694
```

BLEDECODEU16 is an extension function.

BleDecodeS24

FUNCTION

This function reads three bytes in a string at a specified offset into a 32bit integer variable **with** sign extension. If the offset points beyond the end of the string, this function fails.

BLEDECODES24 (attr$,nData, nIndex)

Returns	INTEGER, the number of bytes extracted from the attribute string. Can be less than the size expected if the nIndex parameter is positioned towards the end of the string.
Arguments	
attr$	**byRef *attr$* AS STRING** This references the attribute string from which the function reads.
nData	**byRef nData AS INTEGER** This references an integer to be updated with the 3 byte data from attr$, with sign extension.
nIndex	**byVal *nIndex* AS INTEGER** This is the zero based index into the string attr$ from which data is read. If the string attr$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.
Interactive Command	No

```
//Example :: BleDecodeS24.sb (See in BL600CodeSnippets.zip)

DIM chrHandle,v1,svcHandle,rc
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
DIM attr$ : attr$="\00\01\02\03\04\85\86\87\88\89"
DIM uuid : uuid = 0x1853

rc=BleServiceNew(1, BleHandleUuid16(uuid), svcHandle)

rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)
```

```
rc=BleServiceCommit(svcHandle)

rc=BleCharValueRead(chrHandle,attr$)

//read 3 signed bytes from index 2
rc=BleDecodeS24(attr$,v1,2)
PRINT "\ndata in Hex = 0x"; INTEGER.H'v1
PRINT "\ndata in Decimal = "; v1;"\n"

//read 3 signed bytes from index 6
rc=BleDecodeS24(attr$,v1,6)
PRINT "\ndata in Hex = 0x"; INTEGER.H'v1
PRINT "\ndata in Decimal = "; v1;"\n"
```

Expected Output:

```
data in Hex = 0x00040302
data in Decimal = 262914

data in Hex = 0xFF888786
data in Decimal = -7829626
```

BLEDECODES24 is an extension function.

BleDecodeU24

FUNCTION

This function reads three bytes from a string at a specified offset into a 32 bit integer variable *without* sign extension. If the offset points beyond the end of the string then this function fails.

BLEDECODEU24 (attr$,nData, nIndex)

Returns	INTEGER, the number of bytes extracted from the attribute string. Can be less than the size expected if the nIndex parameter is positioned towards the end of the string.
Arguments	
attr$	byRef *attr$* AS STRING This references the attribute string from which the function reads.
nData	byRef nData AS INTEGER This references an integer to be updated with the 3 byte data from attr$, without sign extension.
nIndex	byVal *nIndex* AS INTEGER This is the zero based index into the string attr$ from which data is read. If the string attr$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.
Interactive Command	No

```
//Example :: BleDecodeU24.sb (See in BL600CodeSnippets.zip)

DIM chrHandle,v1,svcHandle,rc
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
DIM attr$ : attr$="\00\01\02\03\04\85\86\87\88\89"
DIM uuid : uuid = 0x1853
```

```
rc=BleServiceNew(1, BleHandleUuid16(uuid), svcHandle)

rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)

rc=BleServiceCommit(svcHandle)

rc=BleCharValueRead(chrHandle,attr$)

//read 3 unsigned bytes from index 2
rc=BleDecodeU24(attr$,v1,2)
PRINT "\ndata in Hex = 0x"; INTEGER.H'v1
PRINT "\ndata in Decimal = "; v1;"\n"

//read 3 unsigned bytes from index 6
rc=BleDecodeU24(attr$,v1,6)
PRINT "\ndata in Hex = 0x"; INTEGER.H'v1
PRINT "\ndata in Decimal = "; v1;"\n"
```

Expected Output:

```
data in Hex = 0x00040302
data in Decimal = 262914

data in Hex = 0x00888786
data in Decimal = 8947590
```

BLEDECODEU24 is an extension function.

BleDecode32

FUNCTION

This function reads four bytes in a string at a specified offset into a 32bit integer variable. If the offset points beyond the end of the string, this function fails.

BLEDECODE32 (attr$,nData, nIndex)

Returns	INTEGER, the number of bytes extracted from the attribute string. Can be less than the size expected if the nIndex parameter is positioned towards the end of the string.
Arguments	
attr$	byRef *attr$* AS STRING This references the attribute string from which the function reads.
nData	byRef nData AS INTEGER This references an integer to be updated with the 3 byte data from attr$, after sign extension.
nIndex	byVal *nIndex* AS INTEGER This is the zero based index into the string attr$ from which data is read. If the string attr$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.
Interactive Command	No

```
//Example :: BleDecode32.sb (See in BL600CodeSnippets.zip)
```

```
DIM chrHandle,v1,svcHandle,rc
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
DIM attr$ : attr$="\00\01\02\03\04\85\86\87\88\89"
DIM uuid : uuid = 0x1853

rc=BleServiceNew(1, BleHandleUuid16(uuid), svcHandle)

rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)

rc=BleServiceCommit(svcHandle)

rc=BleCharValueRead(chrHandle,attr$)

//read 4 signed bytes from index 2
rc=BleDecode32(attr$,v1,2)
PRINT "\ndata in Hex = 0x"; INTEGER.H'v1
PRINT "\ndata in Decimal = "; v1;"\n"

//read 4 signed bytes from index 6
rc=BleDecode32(attr$,v1,6)
PRINT "\ndata in Hex = 0x"; INTEGER.H'v1
PRINT "\ndata in Decimal = "; v1;"\n"
```

Expected Output:

```
data in Hex = 0x85040302
data in Decimal = -2063334654

data in Hex = 0x89888786
data in Decimal = -1987541114
```

BLEDECODE32 is an extension function.

BleDecodeFLOAT

FUNCTION

This function reads four bytes in a string at a specified offset into a couple of 32 bit integer variables. The decoding results in two variables, the 24 bit signed mantissa and the 8 bit signed exponent. If the offset points beyond the end of the string, this function fails.

BLEDECODEFLOAT (attr$, nMatissa, nExponent, nIndex)

Returns	INTEGER, the number of bytes extracted from the attribute string. Can be less than the size expected if the nIndex parameter is positioned towards the end of the string.
Arguments	
attr$	byRef *attr$* AS STRING This references the attribute string from which the function reads.
nMantissa	byRef nMantissa AS INTEGER This is updated with the 24 bit mantissa from the 4 byte object. If nExponent is 0, you MUST check for the following special values: `0x007FFFFF` NaN (Not a Number) `0x00800000` NRes (Not at this resolution)

	0x007FFFFE + INFINITY 0x00800002 - INFINITY 0x00800001 Reserved for future use
nExponent	byRef nExponent AS INTEGER This is updated with the 8 bit mantissa. If it is zero, check nMantissa for special cases as stated above.
nIndex	byVal *nIndex* AS INTEGER This is the zero based index into the string attr$ from which data is read. If the string attr$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.
Interactive Command	No

```
//Example :: BleDecodeFloat.sb (See in BL600CodeSnippets.zip)

DIM chrHandle,v1,svcHandle,rc, mantissa, exp
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
DIM attr$ : attr$="\00\01\02\03\04\85\86\87\88\89"
DIM uuid : uuid = 0x1853

rc=BleServiceNew(1, BleHandleUuid16(uuid), svcHandle)
rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)

rc=BleServiceCommit(svcHandle)

rc=BleCharValueRead(chrHandle,attr$)

//read 4 bytes FLOAT from index 2 in the string
rc=BleDecodeFloat(attr$,mantissa,exp,2)
PRINT "\nThe number read is ";mantissa;" x 10^";exp

//read 4 bytes FLOAT from index 6 in the string
rc=BleDecodeFloat(attr$,mantissa,exp,6)
PRINT "\nThe number read is ";mantissa;"x 10^";exp
```

Expected Output:

```
The number read is 262914*10^-123
The number read is -7829626*10^-119
```

BLEDECODEFLOAT is an extension function.

BleDecodeSFLOAT

FUNCTION

This function reads two bytes in a string at a specified offset into a couple of 32bit integer variables. The decoding results in two variables, the 12 bit signed maintissa and the 4 bit signed exponent. If the offset points beyond the end of the string then this function fails.

BLEDECODESFLOAT (attr$, nMatissa, nExponent, nIndex)

Returns	INTEGER, the number of bytes extracted from the attribute string. Can be less than the size expected if the nIndex parameter is positioned towards the end of the string.
Arguments	

attr$	byRef *attr$* AS STRING This references the attribute string from which the function reads.
nMantissa	byRef nMantissa AS INTEGER This is updated with the 12 bit mantissa from the two byte object. If the nExponent is 0, you MUST check for the following special values: 0x007FFFFF NaN (Not a Number) 0x00800000 NRes (Not at this resolution) 0x007FFFFE + INFINITY 0x00800002 - INFINITY 0x00800001 Reserved for future use
nExponent	byRef nExponent AS INTEGER This is updated with the four bit mantissa. If it is zero, check the nMantissa for special cases as stated above.
nIndex	byVal *nIndex* AS INTEGER This is the zero based index into the string attr$ from which data is read. If the string attr$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.
Interactive Command	No

```
//Example :: BleDecodeSFloat.sb (See in BL600CodeSnippets.zip)

DIM chrHandle,v1,svcHandle,rc, mantissa, exp
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
DIM attr$ : attr$="\00\01\02\03\04\85\86\87\88\89"
DIM uuid : uuid = 0x1853

rc=BleServiceNew(1, BleHandleUuid16(uuid), svcHandle)

rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)

rc=BleServiceCommit(svcHandle)

rc=BleCharValueRead(chrHandle,attr$)

//read 2 bytes FLOAT from index 2 in the string
rc=BleDecodeSFloat(attr$,mantissa,exp,2)
PRINT "\nThe number read is ";mantissa;" x 10^";exp

//read 2 bytes FLOAT from index 6 in the string
rc=BleDecodeSFloat(attr$,mantissa,exp,6)
PRINT "\nThe number read is ";mantissa;"x 10^";exp
```

Expected Output:

```
The number read is 770 x 10^0
The number read is 1926x 10^-8
```

BLEDECODESFLOAT is an extension function.

BleDecodeTIMESTAMP

FUNCTION

This function reads seven bytes from string an offset into an attribute string. If the offset plus seven bytes points beyond the end of the string then this function fails.

The seven byte string consists of a byte each for century, year, month, day, hour, minute, and second. If (year * month) is zero, it is taken as "not noted" year and all the other fields are set zero (not noted).

For example 5 May 2013 10:31:24 will be represented in the source as "\DD\07\05\05\0A\1F\18" and the year will be translated into a century and year so that the destination string will be "\14\0D\05\05\0A\1F\18"

BLEDECODETIMESTAMP (attr$, timestamp$, nIndex)

Returns	INTEGER, the number of bytes extracted from the attribute string. Can be less than the size expected if the nIndex parameter is positioned towards the end of the string.
Arguments	
attr$	byRef *attr$* AS STRING This references the attribute string from which the function reads.
timestamp$	byRef *timestamp$* AS STRING On exit this is an exact 7 byte string as described above. For example 5 May 2013 10:31:24 is stored as "\14\0D\05\05\0A\1F\18"
nIndex	byVal *nIndex* AS INTEGER This is the zero based index into the string attr$ from which data is read. If the string attr$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.
Interactive Command	No

```
//Example :: BleDecodeTimestamp.sb (See in BL600CodeSnippets.zip)

DIM chrHandle,v1,svcHandle,rc, ts$
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
//5th May 2013, 10:31:24
DIM attr$ : attr$="\00\01\02\DD\07\05\05\0A\1F\18"
DIM uuid : uuid = 0x1853

rc=BleServiceNew(1, BleHandleUuid16(uuid), svcHandle)

rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)

rc=BleServiceCommit(svcHandle)

rc=BleCharValueRead(chrHandle,attr$)

//read 7 byte timestamp from the index 3 in the string
rc=BleDecodeTimestamp(attr$,ts$,3)
PRINT "\nTimestamp = "; StrHexize$(ts$)
```

Expected Output:

```
Timestamp = 140D05050A1F18
```

BLEENCODETIMESTAMP is an extension function.

BleDecodeSTRING

FUNCTION

This function reads a maximum number of bytes from an attribute string at a specified offset into a destination string. This function will not fail as the output string can take truncated strings.

BLEDECODESTRING (attr$, nIndex, dst$, nMaxBytes)

Returns	INTEGER, the number of bytes extracted from the attribute string. Can be less than the size expected if the nIndex parameter is positioned towards the end of the string.
Arguments	
attr$	byRef *attr$* AS STRING This references the attribute string from which the function reads.
nIndex	byVal *nIndex* AS INTEGER This is the zero based index into string attr$ from which data is read.
dst$	byRef dst$ AS STRING This argument is a reference to a string that will be updated with up to nMaxBytes of data from the index specified. A shorter string will be returned if there are not enough bytes beyond the index.
nMaxBytes	byVal nMaxBytes AS INTEGER This specifies the maximum number of bytes to read from attr$.
Interactive Command	No

```
//Example :: BleDecodeString.sb (See in BL600CodeSnippets.zip)

DIM chrHandle,v1,svcHandle,rc, ts$,decStr$
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
//"ABCDEFGHIJ"
DIM attr$ : attr$="41\42\43\44\45\46\47\48\49\4A"
DIM uuid : uuid = 0x1853

rc=BleServiceNew(1, BleHandleUuid16(uuid), svcHandle)

rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)

rc=BleServiceCommit(svcHandle)

rc=BleCharValueRead(chrHandle,attr$)

//read max 4 bytes from index 3 in the string
rc=BleDecodeSTRING(attr$,3,decStr$,4)
PRINT "\nd$=";decStr$

//read max 20 bytes from index 3 in the string - will be truncated
rc=BleDecodeSTRING(attr$,3,decStr$,20)
PRINT "\nd$=";decStr$

//read max 4 bytes from index 14 in the string - nothing at index 14
rc=BleDecodeSTRING(attr$,14,decStr$,4)
PRINT "\nd$=";decStr$
```

Expected Output:

```
d$=CDEF
d$=CDEFGHIJ
d$=
```

BLEDECODESTRING is an extension function.

BleDecodeBITS

FUNCTION

This function reads bits from an attribute string at a specified offset (treated as a bit array) into a destination integer object (treated as a bit array of fixed size of 32). This implies a maximum of 32 bits can be read. This function will not fail as the output bit array can take truncated bit blocks.

BLEDECODEBITS (attr$, nSrcIdx, dstBitArr, nDstIdx,nMaxBits)

Returns	INTEGER, the number of bits extracted from the attribute string. Can be less than the size expected if the nSrcIdx parameter is positioned towards the end of the source string or if nDstIdx will not allow more to be copied.
Arguments	
attr$	byRef *attr$* AS STRING This references the attribute string from which to read, treated as a bit array. Hence a string c 10 bytes will be an array of 80 bits.
nSrcIdx	byVal *nSrcIdx* AS INTEGER This is the zero based bit index into the string attr$ from which data is read. E.g. the third bit in the second byte is index number 10.
dstBitArr	byRef *dstBitArr* AS INTEGER This argument references an integer treated as an array of 32 bits into which data is copied. Only the written bits are modified.
nDstIdx	byVal *nDstIdx* AS INTEGER This is the zero based bit index into the bit array dstBitArr where the data is written to.
nMaxBits	byVal *nMaxBits* AS INTEGER This argument specifies the maximum number of bits to read from attr$. Due to the destination being an integer variable, it cannot be greater than 32. Negative values are treated as zero.
Interactive Command	No

```
//Example :: BleDecodeBits.sb (See in BL600CodeSnippets.zip)

DIM chrHandle,v1,svcHandle,rc, ts$,decStr$
DIM ba : ba=0
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
//"ABCDEFGHIJ"
DIM attr$ : attr$="41\42\43\44\45\46\47\48\49\4A"
DIM uuid : uuid = 0x1853

rc=BleServiceNew(1, BleHandleUuid16(uuid), svcHandle)

rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)
```

```
rc=BleServiceCommit(svcHandle)

rc=BleCharValueRead(chrHandle,attr$)

//read max 14 bits from index 20 in the string to index 10
rc=BleDecodeBITS(attr$,20,ba,10,14)
PRINT "\nbit array = ", INTEGER.B' ba

//read max 14 bits from index 20 in the string to index 10
ba=0x12345678
PRINT "\n\nbit array = ",INTEGER.B' ba

rc=BleDecodeBITS(attr$,14000,ba,0,14)
PRINT "\nbit array now = ", INTEGER.B' ba
//ba will not have been modified because index 14000
//doesn't exist in attr$
```

Expected Output:

```
bit array =      00000000000100001101000000000000

bit array =      00010010001101000101011001111000
bit array now = 00010010001101000101011001111000
```

BLEDECODEBITS is an extension function.

Pairing/Bonding Functions

This section describes all functions related to the pairing and bonding manager which manages trusted devices. The database stores information like the address of the trusted device along with the security keys. At the time of writing this manual a maximum of 4 devices can be stored in the database.

The command AT I 2012 or at runtime SYSINFO(2012) returns the maximum number of devices that can be saved in the database

The type of information that can be stored for a trusted device is:

- The MAC address of the trusted device.
- The eDIV and eRAND for the long term key.
- A 16 byte Long Term Key (LTK).
- The size of the long term key.
- A flag to indictate if the LTK is authenticated – Man-In-The-Middle (MITM) protection.
- A 16 byte Indentity Resolving Key (IRK).
- A 16 byte Connection Signature Resolving Key (CSRK)

Whisper Mode Pairing

BLE provides for simple secure pairing with or without man-in-the-middle attack protection. To enhance security while a pairing is in progress the specification has provided for Out-of-Band pairing where the shared secret information is exchanged by means other than the Bluetooth connection. That mode of pairing is currently not exposed.

Laird have provided an additional mechanism for bonding using the standard inbuilt simple secure pairing which is called Whisper Mode pairing. In this mode, when a pairing is detected to be in progress, the transmit

power is automatically reduced so that the 'bubble' of influence is reduced and thus a proximity based enhanced security is achieved.

To take advantage of this pairing mechanism, use the function BleTxPwrWhilePairing() to reduce the transmit power for the short duration that the pairing is in progress.

Tests have shown that setting a power of -55 using BleTxPwrWhilePairing() will create a 'bubble' of about 30cm radius, outside which pairing will not succeed. This will be reduced even further if the BL600 module is in a case which affects radio transmissions.

BleBondMngrErase

Note: For firmware versions prior to 1.4.X.Y, this subroutine has a bug. It occurs when the subroutine is called during radio activity.

Workaround when advertising:

1. Stop adverts by calling BleAdvertStop()
2. Call BleBondMngrErase()
3. Restart adverts using BleAdvertStart()

SUBROUTINE

This subroutine deletes the entire trusted device database if the supplied parameter is 0. Other values of the parameter are reserved for future use.

Note: In Interactive Mode, the command AT+BTD* can also be used to delete the database.

BLEBONDMNGRERASE (nFutureUse)

Arguments	
nFutureUse	byVal nFutureUse AS INTEGER This shall be set to 0
Interactive Command	No

Workaround for FW **1.3.57.0 and earlier** when there is radio activity:

```
//Example :: BleBondMngrErase.sb (See in BL600CodeSnippets.zip)

DIM rc

rc=BleAdvertStop()
BleBondMngrErase(0)
```

For FW **1.4.X.Y and newer:**

```
//Example :: BleBondMngrErase.sb (See in BL600CodeSnippets.zip)

DIM rc

BleBondMngrErase(0)
```

BLEBONDMNGRERASE is an extension function.

BleBondMngrGetInfo

FUNCTION

This function retrieves the MAC address and other information from the trusted device database via an index.

Note: Do not rely on a device in the database mapping to a static index. New bondings will change the position in the database.

BLEBONDMNGRGETINFO (nIndex, addr$, nExtraInfo)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
nIndex	byVal nIndex AS INTEGER This is an index in the range 0 to 1, less than the value returned by SYSINFO(2012).
addr$	byRef addr$ AS STRING On exit ,if nIndex points to a valid entry in the database, this variable contains a MAC address exactly seven bytes long. The first byte identifies public or private random address. The next six bytes are the address.
nExtraInfo	byRef nExtraInfo AS INTEGER On exit if nIndex points to a valid entry in the database, this variable contains a composite integer value where the lower 16 bits are the eDIV. Bit 16 is set if the IRK (Identity Resolving Key) exists for the trusted device and bit 17 is set if the CSRK (Connection Signing Resolving Key) exists for the trusted device.
Interactive Command	No

```
//Example :: BleBondMngrGetInfo.sb (See in BL600CodeSnippets.zip)
#define BLE_INV_INDEX           24619
DIM rc, addr$, exInfo
rc = BleBondMngrGetInfo(0,addr$,exInfo)  //Extract info of device at index 1

IF rc==0 THEN
    PRINT "\nMAC address: ";addr$
    PRINT "\nInfo: ";exInfo
ELSEIF rc==BLE_INV_INDEX   THEN
    PRINT "\nInvalid index"
ENDIF
```

Expected Output when valid entry present in database:

```
MAC address: \00\BC\B1\F3x3\AB
Info: 97457
```

Expected Output with invalid index:

```
Invalid index
```

BLEBONDMNGRGETINFO is an extension function.

Virtual Serial Port Service – Managed Test When Dongle and Application are Available

This section describes all the events and routines used to interact with a managed virtual serial port service.

"Managed" means there is a driver consisting of transmit and receive ring buffers that isolate the BLE service from the *smart*BASIC application. This in turn provides easy to use API functions.

Note: The driver makes the same assumption that the driver in a PC makes: If the on-air connection equates to the serial cable, there is no assumption that the cable is from the same source as prior to the disconnection. This is analogous to the way that a PC cannot detect such in similar cases.

The module can present a serial port service in the local GATT Table consisting of two mandatory characteristics and two optional characteristics. One mandatory characteristic is the TX FIFO and the other is the RX FIFO, both consisting of an attribute taking up to 20 bytes. Of the optional characteristics, one is the ModemIn which consists of a single byte and only bit 0 is used as a CTS type function. The other is ModemOut, also a single byte, which is notifiable only and is used to convey an RTS flag to the client.

By default, (configurable via AT+CFG 112), Laird's serial port service is exposed with UUID's as follows:-

The UUID of the service is:	`569a`***`1101`***`-b87f-490c-92cb-11ba5ea5167c`
The UUID of the rx fifo characteristic is:	`569a`***`2001`***`-b87f-490c-92cb-11ba5ea5167c`
The UUID of the tx fifo characteristic is:	`569a`***`2000`***`-b87f-490c-92cb-11ba5ea5167c`
The UUID of the ModemIn characteristic is:	`569a`***`2003`***`-b87f-490c-92cb-11ba5ea5167c`
The UUID of the ModemOut characteristic is:	`569a`***`2002`***`-b87f-490c-92cb-11ba5ea5167c`

Note: Laird's Base 128bit UUID is `569aXXXX-b87f-490c-92cb-11ba5ea5167c` where XXXX is a 16 bit offset. We recommend, to save RAM, that you create a 128 bit UUID of your own and manage the 16 bit space accordingly, akin to what the Bluetooth SIG does with their 16 bit UUIDs.

If command AT+CFG 112 1 is used to change the value of the config key 112 to 1 then Nordic's serial port service is exposed with UUID's as follows:-

The UUID of the service is:	`6e40`***`0001`***`-b5a3-f393-e0a9-e50e24dcca9e`
The UUID of the rx fifo characteristic is:	`6e40`***`0002`***`-b5a3-f393-e0a9-e50e24dcca9e`
The UUID of the tx fifo characteristic is:	`6e40`***`0003`***`-b5a3-f393-e0a9-e50e24dcca9e`

Note: The first byte in the UUID's above is the most significant byte of the UUID.

The 'rx fifo characteristic' is for data that comes to the module and the 'tx fifo characteristic' is for data that goes out from the module. This means a GATT Client using this service will send data by writing into the 'rx fifo characteristic' and will get data from the module via a value notification.

The 'rx fifo characteristic' is defined with no authentication or encryption requirements, a maximum of 20 bytes value attribute. The following properties are enabled:

- WRITE
- WRITE_NO_RESPONSE

The 'tx fifo characteristic' value attribute is with no authentication or encryption requirements, a maximum of 20 bytes value attribute. The following properties are enabled:

- NOTIFY (The CCCD descriptor also requires no authentication/encryption)

The 'ModemIn characteristic' is defined with no authentication or encryption requirements, a single byte attribute. The following properties are enabled:

- WRITE
- WRITE_NO_RESPONSE

The 'ModemOut characteristic' value attribute is with no authentication or encryption requirements, a single byte attribute. The following properties are enabled:

- NOTIFY (The CCCD descriptor also requires no authentication/encryption)

For ModemIn, only bit zero is used, which is set by 1 when the client can accept data and 0 when it cannot (inverse logic of CTS in UART functionality). Bits 1 to 7 are for future use and should be set to 0.

For ModemOut, only bit zero is used which is set by 1 when the client can send data and 0 when it cannot (inverse logic of RTS in UART functionality). Bits 1 to 7 are for future use and should be set to 0.

Note: Both flags in ModemIn and ModemOut are suggestions to the peer, just as in a UART scenario. If the peer decides to ignore the suggestion and data is kept flowing, the only coping mechanism is to drop new data as soon as internal ring buffers are full.

Given that the outgoing data is **notified** to the client, the 'tx fifo characteristic' has a Client Configuration Characteristic (CCCD) which must be set to 0x0001 to allow the module to send any data waiting to be sent in the transmit ring buffer. While the CCCD value is not set for notifications, writes by the *smart* BASIC application result in data being buffered. If the buffer is full the appropriate write routine indicates how many bytes actually got absorbed by the driver. In the background, the transmit ring buffer is emptied with one or more indicate or notify messages to the client. When the last bytes from the ring buffer are sent, **EVVSPTXEMPTY** is thrown to the *smart* BASIC application so that it can write more data if it chooses.

When GATT Client sends data to the module by writing into the 'rx fifo characteristic' the managing driver will immediately save the data in the receive ring buffer if there is any space. If there is no space in the ring buffer, data is discarded. After the ring buffer is updated, event **EVVSPRX** is thrown to the *smart* BASIC runtime engine so that an application can read and process the data.

Similarly, given that ModemOut is **notified** to the client, the ModemOut characteristic has a Client Configuration Characteristic (CCCD) which must be set to 0x0001. By default, in a connection the RTS bit in ModemOut is set to 1 so that the VSP driver assumes there is buffer space in the peer to send data. The RTS flag is affected by the thresholds of 80 and 120 which means the when opening the VSP port the rxbuffer cannot be less than 128 bytes.

It is intended that in a future release it will be possible to register a 'custom' service and bind that with the virtual service manager to allow that service to function in the managed environment. This allows the application developer to interact with any GATT client implementing a serial port service, whether one currently deployed or one that the Bluetooth SIG adopts.

VSP Configuration

Given that VSP operation can happen in command mode the ability to configure it and save the new configuration in non-volatile memory is available. For example, in bridge mode, the baudrate of the uart can be specified to something other than the default 9600. Configuration is done using the AT+CFG command and refer to the section describing that command for further details. The configuration id pertinent to VSP are 100 to 116 inclusive

Command and Bridge Mode Operation

Just as the physical UART is used to interact with the module when it is not running a *smart* BASIC application, it is also possible to have *limited* interaction with the module in interactive mode. The limitation applies to NOT being able to launch *smart* BASIC applications using the AT+RUN command. If bridge mode is enabled then any incoming VSP data is retransmitted out via the UART. Conversely, any data arriving via the UART is transmitted out the VSP service. This latter functionality provides a cable replacement function.

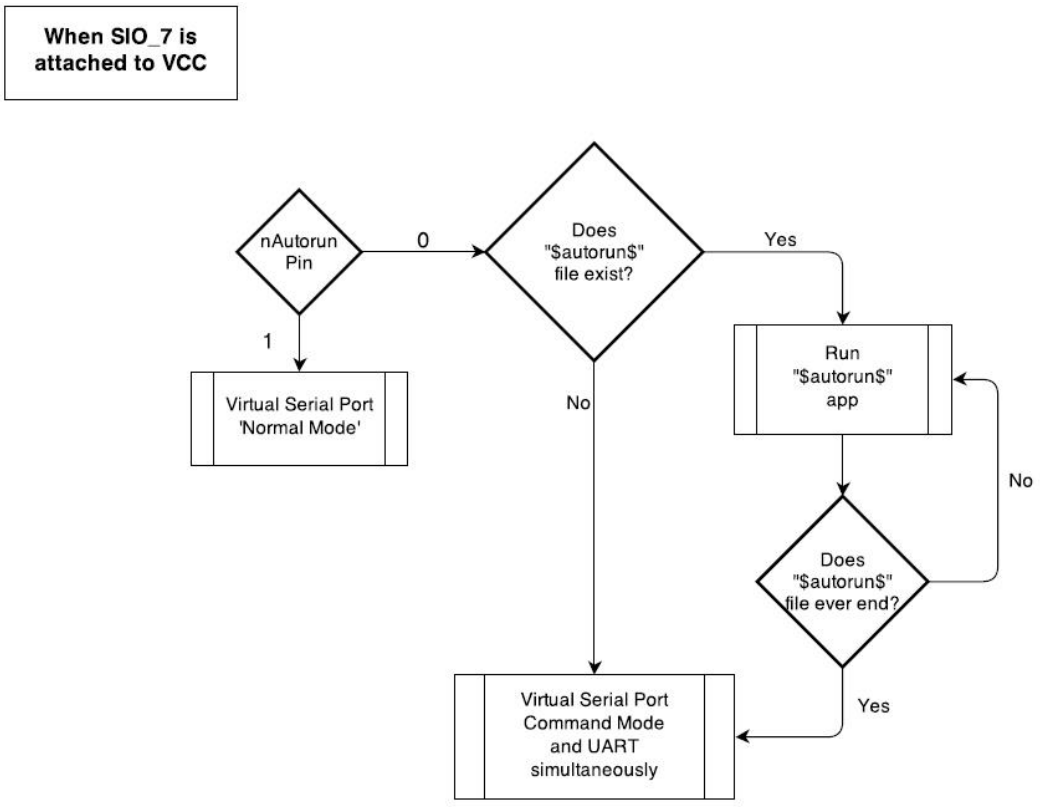
Selection of Command or Bridge Mode is done using the nAutorun input signal. When nAutorun is low, interactive mode is enabled. When it is high, and bit 8 in the config register 100 accessed by AT+CFG 100 is set, bridge mode is selected the defaule value of config register 100 is 0x8107 which means by default, bridge mode is enabled if SIO7 is held high and nAutorun is high too.

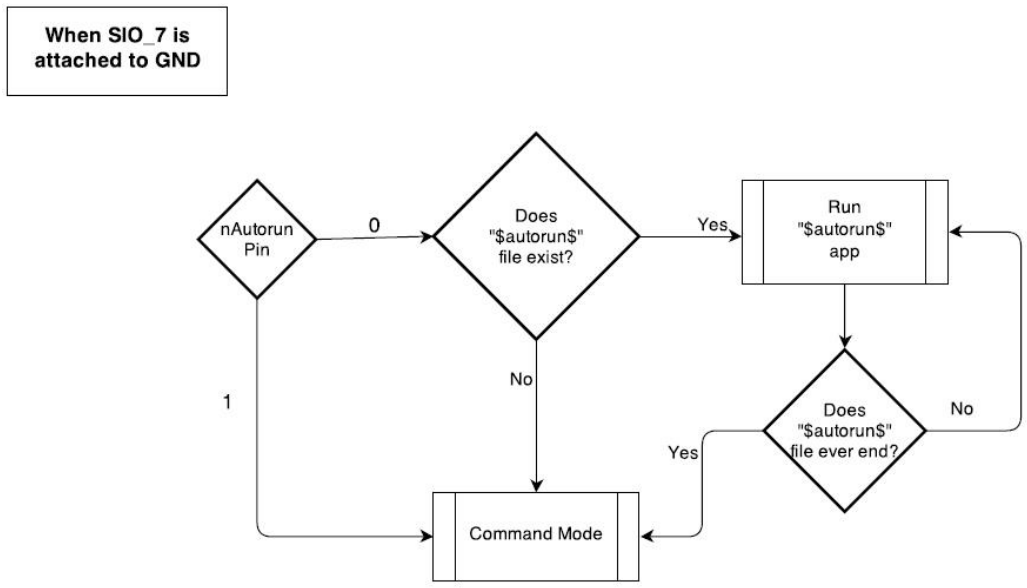
Note: If $autorun$ file exists in the file system, the bridge mode is always suppressed regardless of the state of the nAutorun input signal.

The operation of VSP command and bridge mode is illustrated as per the diagrams on the following page (aknowledgments to Nicolas Mejia) .

The main purpose of interactive mode operation is to facilitate the download of an autorun *smart*BASIC application. This allows the module to be soldered into an end product without preconfiguration and then the application can be downloaded over the air once the product has been pre-tested. It is the *smart*BASIC application that is downloaded over the air, NOT the firmware. Due to this principle reason for use in production, to facilitate multiple programming stations in a locality the transmit power is limited to -12dBm. It can be changed by changing the 109 config key using the command AT+CFG.

The default operation of this virtual serial port service is dependent on one of the digital input lines being pulled high externally. Consult the hardware manual for more information on the input pin number. By default it is SIO7 on the module, but it can be changed by setting the config key 100 via AT+CFG.





You can interact with the BL600 over the air via the Virtual Serial Port Service using the Laird iOS or Android "BL600 Serial" app, available free on the Apple App Store and Google Play Store respectively

You may download *smart*BASIC applications using a Windows application, which will be available for free from Laird. The PC must be BLE enabled using a Laird supplied adapter. Contact your local FAE for details.

As most of the AT commands are functional, you may obtain information such as version numbers by sending the command AT I 3 to the module over the air.

Note that the module enters interactive mode only if there is no autorun application or if the autorun application exits to interactive mode by design. Hence in normal operation where a module is expected to have an autorun application the virtual serial port service will not be registered in the GATT table.

If the application requires the virtual serial port functionality then it shall have to be registered programmatically using the functions that follow in subsequent subsections. These are easy to use high level functions such as OPEN/READ/WRITE/CLOSE.

VSP (Virtual Serial Port) Events

In addition to the routines for manipulating the Virtual Serial Port (VSP) service, when data arrives via the receive characteristic it is stored locally in an underlying ring buffer and then an event is generated.

Similarly when the transmit buffer is emptied, events are thrown from the underlying drivers so that user *smart* BASIC code in handlers can perform user defined actions.

The following is a list of events generated by VSP service managed code which can be handled by user code.

EVVSPRX	This event is generated when data has arrived and has been stored in the local ring buffer to be read using BleVSpRead().
EVVSPTXEMPTY	This event is generated when the last byte is transmitted using the outgoing data characteristic via a notification or indication.

Use the iOS BL600 Serial app and connect to your BL600 to test this sample app.

```
//Example :: VSpEvents.sb (See in BL600CodeSnippets.zip)
```

```
DIM tx$,rc,x,scRpt$,adRpt$,addr$,hndl

//handler for data arrival
FUNCTION HandlerBleVSpRx() AS INTEGER
    //print the data that arrived
    DIM n,rx$
    n = BleVSpRead(rx$,20)
    PRINT "\nrx=";rx$
ENDFUNC 1

//handler when VSP tx buffer is empty
FUNCTION HandlerVSpTxEmpty() AS INTEGER
    IF x==0 THEN
        rc = BleVSpWrite(tx$)
        x=1
    ENDIF
ENDFUNC 1

PRINT "\nDevice name is "; BleGetDeviceName$()

//Open the VSP
rc = BleVSpOpen(128,128,0,hndl)
//Initialise a scan report
rc = BleScanRptInit(scRpt$)
//Advertise the VSP service in the scan report so
//that it can be seen by the client
rc = BleAdvRptAddUuid128(scRpt$,hndl)
adRpt$=""
rc = BleAdvRptsCommit(adRpt$,scRpt$)
addr$=""  //because we are not doing a DIRECT advert
rc = BleAdvertStart(0,addr$,20,300000,0)
//Now advertising so can be connectable

ONEVENT  EVVSPRX      CALL HandlerBleVSpRx
ONEVENT  EVVSPTXEMPTY  CALL HandlerVSpTxEmpty

tx$="tx buffer empty"
PRINT "\nUse the iOS 'BL600 Serial' app to test this"

//wait for events and messages
WAITEVENT
```

BleVSpOpen

FUNCTION

This function opens the default VSP service using the parameters specified. The service's UUID is: `569a`***`1101`***`-b87f-490c-92cb-11ba5ea5167c`

By default, ModemIn and ModemOut characteristics are registered in the GATT table with the Rx and Tx FIFO characteristics. To suppress Modem characteristics in the GATT table, set bit 1 in the nFlags parameter (value 2). If the virtual serial port is already open, this function fails.

BLEVSPOPEN (txbuflen,rxbuflen,nFlags,svcUuid)

Returns	INTEGER, indicating the success of command:	
	0	Opened successfully
	0x604D	Already open

	0x604E Invalid Buffer Size 0x604C Cannot register Service in Gatt Table while BLE connected
Exceptions	▪ Local Stack Frame Underflow ▪ Local Stack Frame Overflow
Arguments	
txbuflen	***byVal txbuflen AS INTEGER*** Set the transmit ring buffer size to this value. If set to 0, a default value is used by the underlying driver and use BleVspInfo(2) to determine the size.
rxbuflen	***byVal rxbuflen AS INTEGER*** Set the receive ring buffer size to this value. If set to 0, a default value is used by the underlying driver and use BleVspInfo(1) to determine the size.
nFlags	***byVal nFlags AS INTEGER*** This is a bit mask to customise the driver as follows: Bit 0: Set to 1 to try for reliable data transfer. This uses INDICATE messages if allowed and if there is a choice. Some services only allow NOTIFY and in that case, if set to 1, it is ignored. Bit 1..31: Reserved for future use. Set to 0.
svcUuid	***byRef svcUuid AS INTEGER*** On exit, this variable is updated with a handle to the service UUID which can then be subsequently used to advertise the service in an advert report. Given that there is no BT SIG adopted Serial Port Service the UUID for the service is 128 bit, so an appropriate Advert Data element can be added to the advert or scan report using the function BleAdvRptAddUuid128() which takes a handle of that type.
Related Commands	BLEVSPINFO, BLEVSPCLOSE, BLEVSPWRITE, BLEVSPREAD, BLEVSPFLUSH

```
//Example :: BleVspOpen.sb (See in BL600CodeSnippets.zip)

DIM scRpt$,adRpt$,addr$,vspSvcHndl

//Close VSP if already open
IF BleVSpInfo(0)!=0 THEN
    BleVSpClose()
ENDIF

//Open VSP
IF BleVSpOpen(128,128,0,vspSvcHndl)==0 THEN
    PRINT "\nVSP service opened"
ELSE
    PRINT "\nFailed"
ENDIF
```

Expected Output:

```
VSP service opened
```

BLEVSPOPEN is an extension function.

BleVspClose

SUBROUTINE

This subroutine closes the managed virtual serial port which had been opened with BLEVSPOPEN. This routine is safe to call if it is already closed. When this subroutine is invoked both receive and transmit buffers are flushed. If there is data in either buffer when the port is closed, it will be lost.

BLEVSPCLOSE()

Exceptions	▪ Local Stack Frame Underflow ▪ Local Stack Frame Overflow
Arguments	None
Interactive Command	No
Related Commands	BLEVSPINFO, BLEVSPOPEN, BLEVSPWRITE, BLEVSPREAD, BLEVSPFLUSH

Use the iOS "BL600 Serial" app and connect to your BL600 to test this sample app.

```
//Example :: BleVspClose.sb (See in BL600CodeSnippets.zip)

DIM tx$,rc,scRpt$,adRpt$,addr$,hndl

//handler when VSP tx buffer is empty
FUNCTION HandlerVSpTxEmpty() AS INTEGER
    PRINT "\n\nVSP tx buffer empty"
    BleVspClose()
ENDFUNC 0

PRINT "\nDevice name is "; BleGetDeviceName$()

//Open the VSP, advertise
rc = BleVSpOpen(128,128,0,hndl)
rc = BleScanRptInit(scRpt$)
rc = BleAdvRptAddUuid128(scRpt$,hndl)
adRpt$=""
rc = BleAdvRptsCommit(adRpt$,scRpt$)
addr$=""
rc = BleAdvertStart(0,addr$,20,300000,0)

//This message will send when connected to client
tx$="send this data and will close when sent"
rc = BleVSpWrite(tx$)

ONEVENT  EVVSPTXEMPTY  CALL HandlerVSpTxEmpty

WAITEVENT

PRINT "\nExiting..."
```

Expected Output:

```
Device name is LAIRD BL600

VSP tx buffer empty
Exiting...
```

BLEVSPCLOSE is an extension subroutine.

BleVSpInfo

FUNCTION

This function is used to query information about the virtual serial port, such as buffer lengths, whether the port is already open or how many bytes are waiting in the receive buffer to be read.

BLEVSPINFO (infoId)

Returns	INTEGER The value associated with the type of UART information requested
Exceptions	▪ Local Stack Frame Underflow ▪ Local Stack Frame Overflow
Arguments	
infoId	*byVal infoId AS INTEGER* This specifies the information type requested as follows if the port is open: 0 — 0 if closed, 1 if open, 3 if open and there is a BLE connection and 7 if the transmit fifo characteristic CCCD has been updated by the client to enable notifies or indications. 1 — Receive ring buffer capacity 2 — Transmit ring buffer capacity 3 — Number of bytes waiting to be read from receive ring buffer 4 — Free space available in transmit ring buffer
Related Commands	BLEVSPOPEN, BLEVSPCLOSE, BLEVSPWRITE, BLEVSPREAD, BLEVSPFLUSH
Interactive Command	No

```
//Example :: BleVspInfo.sb (See in BL600CodeSnippets.zip)

DIM hndl, rc

//Close VSP if it is open
BleVSpClose()

rc = BleVSpOpen(128,128,0,hndl)
PRINT "\nVsp State: "; BleVSpInfo(0)
PRINT "\nRx buffer capacity: "; BleVSpInfo(1)
PRINT "\nTx buffer capacity: "; BleVSpInfo(2)
PRINT "\nBytes waiting to be read from rx buffer: "; BleVSpInfo(3)
PRINT "\nFree space in tx buffer: "; BleVSpInfo(4)
BleVspClose()
PRINT "\nVsp State: "; BleVSpInfo(0)
```

Expected Output:

```
Vsp State: 1
Rx buffer capacity: 128
Tx buffer capacity: 128
Bytes waiting to be read from rx buffer: 0
Free space in tx buffer: 128
Vsp State: 0
```

BLEVSPINFO is an extension subroutine.

BleVSpWrite

FUNCTION

This function is used to transmit a string of characters from the virtual serial port.

BLEVSPWRITE (strMsg)

Returns	INTEGER 0 to N : Actual number of bytes successfully written to local transmit ring buffer.
Exceptions	▪ Local Stack Frame Underflow ▪ Local Stack Frame Overflow
Arguments	
strMsg	*byRef strMsg AS STRING* The array of bytes to be sent. STRLEN(strMsg) bytes are written to the local transmit ring buffe If STRLEN(strMsg) and the return value are not the same, it implies that the transmit buffer did not have enough space to accommodate the data. If the return value does not match the length of the original string, use STRSHIFTLEFT function drop the data from the string, so subsequent calls to this function only retry with data not placed in the output ring buffer. Another strategy is to wait for EVVSPTXEMPTY events, then resubmit data.
Interactive Command	No
Related Commands	BLEVSPOPEN, BLEVSPCLOSE, BLEVSPINFO, BLEVSPREAD, BLEVSPFLUSH

Note: strMsg cannot be a string constant, e.g. "the cat", but must be a string variable. If you must use a const string, first save it to a temp string variable and then pass it to the function

Use the iOS BL600 Serial app and connect to your BL600 to test this sample app.

```
//Example :: BleVSpWrite.sb (See in BL600CodeSnippets.zip)

DIM tx$,rc,scRpt$,adRpt$,addr$,hndl,cnt

//handler when VSP tx buffer is empty
FUNCTION HandlerVSpTxEmpty() AS INTEGER
    cnt=cnt+1
    IF cnt<= 2 THEN
        tx$="then this is sent"
        rc = BleVSpWrite(tx$)
    ENDIF

ENDFUNC 0

rc = BleVSpOpen(128,128,0,hndl)
rc = BleScanRptInit(scRpt$)
rc = BleAdvRptAddUuid128(scRpt$,hndl)
adRpt$=""
rc = BleAdvRptsCommit(adRpt$,scRpt$)
addr$=""
rc = BleAdvertStart(0,addr$,20,300000,0)
```

```
PRINT "\nDevice name is "; BleGetDeviceName$()

cnt=1
tx$="send this data and "
rc = BleVSpWrite(tx$)

ONEVENT  EVVSPTXEMPTY  CALL HandlerVSpTxEmpty

WAITEVENT

PRINT "\nExiting..."
```

Expected Output:

```
Device name is LAIRD BL600
Exiting...
```

BLEVSPWRITE is a extension subroutine.

BleVSpRead

FUNCTION

This function is used to read the content of the receive buffer and copy it to the string variable supplied.

BLEVSPREAD(strMsg,nMaxRead)

Returns	INTEGER 0 to N : The total length of the string variable. This means the caller does not need to call strlen() function to determine how many bytes in the string must be processed.
Exceptions	▪ Local Stack Frame Underflow ▪ Local Stack Frame Overflow
Arguments	
strMsg	*byRef strMsg AS STRING* The content of the receive buffer is *copied* to this string.
nMaxRead	*byVal nMaxRead AS INTEGER* The maximum number of bytes to read.
Interactive Command	No
Related Commands	BLEVSPOPEN, BLEVSPCLOSE, BLEVSPINFO, BLEVSPWRITE, BLEVSPFLUSH

Note: strMsg cannot be a string constant, e.g. "the cat", but must be a string variable and. If you must use a const string, first save it to a temp string variable and then pass it to the function

Use the iOS BL600 Serial app and connect to your BL600 to test this sample app.

```
//Example :: BleVSpRead.sb (See in BL600CodeSnippets.zip)

DIM conHndl
//Only 1 global variable because its value is used in more than 1 routine
//All other variables declared locally, inside routine that they are used in.
//More efficient because these local variables only exist in memory
```

```
//when they are being used inside their respective routines


//============================================================================
// Open VSp and start advertising
//============================================================================
SUB OnStartup()
    DIM rc, hndl, tx$, scRpt$, addr$, adRpt$ : adRpt$="" : addr$=""
    rc=BleVSpOpen(128,128,0,hndl)
    rc=BleScanRptInit(scRpt$)
    rc=BleAdvRptAddUuid128(scRpt$,hndl)
    rc=BleAdvRptsCommit(adRpt$,scRpt$)
    rc=BleAdvertStart(0,addr$,20,300000,0)
    PRINT "\nDevice name is "; BleGetDeviceName$()

    tx$="\nSend me some text \nTo exit the app, just tell me\n"
    rc = BleVSpWrite(tx$)
ENDSUB

//============================================================================
// Close connections so that we can run another app without problems
//============================================================================
SUB CloseConnections()
    DIM rc
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
    BleVspClose()
ENDSUB

//============================================================================
//  VSP Rx buffer event handler
//============================================================================
FUNCTION HandlerVSpRx() AS INTEGER
    DIM rc, rx$, e$ : e$="exit"
    rc=BleVSpRead(rx$,20)
    PRINT "\nMessage from client: ";rx$

    //If user has typed exit
    IF StrPos(rx$,e$,0) > -1 THEN
       EXITFUNC 0
    ENDIF
ENDFUNC 1

//============================================================================
// BLE event handler
//============================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\nDisconnected from client"
        EXITFUNC 0
    ENDIF
ENDFUNC 1

ONEVENT  EVVSPRX  CALL HandlerVSpRx
ONEVENT  EVBLEMSG CALL HndlrBleMsg

OnStartup()          //Calls first subroutine declared above

WAITEVENT

CloseConnections()  //Calls second subroutine declared above
```

```
PRINT "\nExiting..."
```

Expected Output:

```
Device name is LAIRD BL600
Messgae from client: (Whatever data you send from your device)
Message from client: exit
Exiting...
```

BLEVSPREAD is an extension subroutine.

BleVSpUartBridge

SUBROUTINE

This function creates a bridge between the managed Virtual Serial Port Service and the UART when both are open. Any data arriving from the VSP is automatically transferred to the UART for forward transmission. Any data arriving at the UART is sent over the air.

It should be called either when data arrives at either end or when either end indicates their transmit buffer is empty. The following events are examples: EVVSPRX, EVUARTRX, EVVSPTXEMPTY and EVUARTTXEMPTY.

Given that data can arrive over the UART a byte at a time, a latency timer specified by AT+CFG 116 command may be used to optimise the data transfer over the air. This tries to ensure that full packets are transmitted over the air. Therefore, if a single character arrives over UART, a latency timer is started. If it expires, that single character (or any more that arrive but less than 20) will be forced onwards when that timer expires.

BLEVSPUARTBRIDGE()

Exceptions	▪ Local Stack Frame Underflow ▪ Local Stack Frame Overflow
Arguments	None
Interactive Command	No
Related Commands	BLEVSPOPEN, BLEVSPCLOSE, BLEVSPINFO, BLEVSPWRITE, BLEVSPFLUSH

Related Commands:

```
//Example :: BleVSpUartBridge.sb (See in BL600CodeSnippets.zip)

DIM conHndl

//============================================================================
// Open VSp and start advertising
//============================================================================
SUB OnStartup()
    DIM rc, hndl, tx$, scRpt$, addr$, adRpt$

    rc=BleVSpOpen(128,128,0,hndl)
    rc=BleScanRptInit(scRpt$)
    rc=BleAdvRptAddUuid128(scRpt$,hndl)
    rc=BleAdvRptsCommit(adRpt$,scRpt$)
    rc=BleAdvertStart(0,addr$,20,300000,0)
```

```
    rc=GpioBindEvent(1,16,1)      //Channel 1, bind to low transition on GPIO pin 16
    PRINT "\nDevice name is "; BleGetDeviceName$();"\n"

    tx$="\nSend me some text. \nPress button 0 to exit\n"
    rc = BleVSpWrite(tx$)
ENDSUB

//================================================================================
// Close connections so that we can run another app without problems
//================================================================================
SUB CloseConnections()
    DIM rc
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
    BleVspClose()
ENDSUB

//================================================================================
// BLE event handler - connection handle is obtained here
//================================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\nDisconnected from client"
        EXITFUNC 0
    ENDIF
ENDFUNC 1

//================================================================================
//handler to service button 0 pressed
//================================================================================
FUNCTION HndlrBtn0Pr() AS INTEGER
    //just exit and stop waiting for events
ENDFUNC 0


//================================================================================
//handler to service an rx/tx event
//================================================================================
FUNCTION HandlerBridge() AS INTEGER
  // transfer data between VSP and UART ring buffers
  BleVspUartBridge()
ENDFUNC 1


ONEVENT  EVVSPRX            CALL HandlerBridge
ONEVENT  EVUARTRX           CALL HandlerBridge
ONEVENT  EVVSPTXEMPTY       CALL HandlerBridge
ONEVENT  EVUARTTXEMPTY      CALL HandlerBridge
ONEVENT  EVBLEMSG          CALL HndlrBleMsg
ONEVENT  EVGPIOCHAN1       CALL HndlrBtn0Pr


OnStartup()

WAITEVENT

CloseConnections()  //Calls second subroutine declared above
PRINT "\nExiting..."
```

BLEVSPUARTBRIDGE is an extension subroutine.

BleVSpFlush

SUBROUTINE

This subroutine flushes either or both receive and transmit ring buffers.

This is useful when, for example, you have a character terminated messaging system and the peer sends a very long message, filling the input buffer. In that case, there is no more space for an incoming termination character. A flush of the receive buffer is the best approach to recover from that situation.

BLEVSPFLUSH(bitMask)

Returns	▪ Local Stack Frame Underflow ▪ Local Stack Frame Overflow
Arguments	
bitMask	*byVal bitMask AS INTEGER* Bit 0 is set to flush the Rx buffer. Bit 1 is set to flush the Tx buffer. Set both bits to flus both buffers.
Interactive Command	No
Related Commands	BLEVSPOPEN, BLEVSPCLOSE, BLEVSPINFO, BLEVSPWRITE, BLEVSPREAD

```
//Example :: BleVSpFlush.sb (See in BL600CodeSnippets.zip)

DIM conHndl


//============================================================================
// Open VSp and start advertising
//============================================================================
SUB OnStartup()
    DIM rc, hndl, tx$, scRpt$, addr$, adRpt$ : adRpt$="" : addr$=""
    rc=BleVSpOpen(128,128,0,hndl)
    rc=BleScanRptInit(scRpt$)
    rc=BleAdvRptAddUuid128(scRpt$,hndl)
    rc=BleAdvRptsCommit(adRpt$,scRpt$)
    rc=BleAdvertStart(0,addr$,20,300000,0)
    rc=GpioBindEvent(1,16,1)     //Channel 1, bind to low transition on GPIO pin 16
    PRINT "\nDevice name is "; BleGetDeviceName$()

    tx$="\nSend me some text, I won't get it. \nTo exit the app press Button 0\n"
    rc = BleVSpWrite(tx$)
ENDSUB

//============================================================================
// Close connections so that we can run another app without problems
//============================================================================
SUB CloseConnections()
    DIM rc
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
    BleVspClose()
    BleVspFlush(2)      //Flush both buffers
ENDSUB

//============================================================================
//  VSP Rx buffer event handler
//============================================================================
FUNCTION HandlerVSpRx() AS INTEGER
```

```
    BleVspFlush(0)
    PRINT "\nRx buffer flushed"
ENDFUNC 1

//===============================================================================
//handler to service button 0 pressed
//===============================================================================
FUNCTION HndlrBtn0Pr() AS INTEGER
     //stop waiting for events and exit app
ENDFUNC 0


//===============================================================================
// BLE event handler
//===============================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\nDisconnected from client"
        EXITFUNC 0
    ENDIF
ENDFUNC 1


ONEVENT  EVVSPRX     CALL HandlerVSpRx
ONEVENT  EVBLEMSG    CALL HndlrBleMsg
ONEVENT  EVGPIOCHAN1  CALL HndlrBtn0Pr


OnStartup()          //Calls first subroutine declared above

WAITEVENT

CloseConnections()  //Calls second subroutine declared above
PRINT "\nExiting..."
```

Expected Output:

```
Device name is LAIRD BL600
Rx buffer flushed
Rx buffer flushed
Exiting...
```

BLEVSPFLUSH is an extension subroutine.

6. Other Extension Built-in Routines

This chapter describes non BLE-related extension routines that are not part of the core *smart* BASIC language.

System Configuration Routines

SystemStateSet

FUNCTION

This function is used to alter the power state of the module as per the input parameter.

SYSTEMSTATESET (nNewState)

Returns	INTEGER, a result code. The typical value is 0x0000, indicating a successful operation.
Arguments	
nNewState	byVal *nNewState* AS INTEGER New state of the module as follows: 0 System OFF (Deep Sleep Mode)
Interactive Command	NO

Note: You may also enter this state when UART is open and a BREAK condition is asserted. Deasserting BREAK makes the module resume through reset i.e. power cycle.

```
//Example :: SystemStateSet.sb (See in Firmware Zip file)

//Put the module into deep sleep
PRINT "\n"; SystemStateSet(0)
```

SYSTEMSTATESET is an extension function.

Miscellaneous Routines

ReadPwrSupplyMv

FUNCTION

This function is used to read the power supply voltage and the value will be returned in millivolts.

READPWRSUPPLYMV ()

Returns	INTEGER, the power supply voltage in millivolts.
Arguments	None
Interactive Command	NO

```
//Example :: ReadPwrSupplyMv.sb (See in Firmware Zip file)

//read and print the supply voltage
PRINT "\nSupply voltage is "; ReadPwrSupplyMv();"mV"
```

Expected Output:

```
Supply voltage is 3343mV
```

READPWRSUPPLYMV is an extension function.

SetPwrSupplyThreshMv

FUNCTION

This function sets a supply voltage threshold. If the supply voltage drops below this then the BLE_EVMSG event is thrown into the run time engine with a MSG ID of BLE_EVBLEMSGID_POWER_FAILURE_WARNING (19) and the context data will be the current voltage in millivolts.

Events & Messages

MsgId	Description
19	The supply voltage has dropped below the value specified as the argument to this function in the most recent call. The context data is the current reading of the supply voltage in millivolts

SETPWRSUPPLYTHRESHMV(nThresh)

Returns	INTEGER, 0 if the threshold is successfully set, 0x6605 if the value cannot be implemented.
Arguments	
nThreshMv	byVal *nThresMv* AS INTEGER The BLE_EVMSG event is thrown to the engine if the supply voltage drops below this value. Valid values are 2100, 2300, 2500 and 2700.
Interactive Command	NO

```
//Example :: SetPwrSupplyThreshMv.sb (See in Firmware Zip file)

DIM rc
DIM mv

//============================================================================
// Handler for generic BLE messages
//============================================================================
FUNCTION HandlerBleMsg(BYVAL nMsgId, BYVAL nCtx) AS INTEGER
    SELECT nMsgId
        CASE 19
            PRINT "\n --- Power Fail Warning ",nCtx
            //mv=ReadPwrSupplyMv()
            PRINT "\n --- Supply voltage is "; ReadPwrSupplyMv();"mV"
        CASE ELSE
            //ignore this message
     ENDSELECT
ENDFUNC 1


//============================================================================
// Handler to service button 0 pressed
//============================================================================
FUNCTION HndlrBtn0Pr() AS INTEGER
    //just exit and stop waiting for events
ENDFUNC 0


ONEVENT  EVBLEMSG   CALL HandlerBleMsg
ONEVENT  EVGPIOCHAN1 CALL HndlrBtn0Pr

rc=GpioBindEvent(1,16,1)     //Channel 1, bind to low transition on GPIO pin 16
PRINT "\nSupply voltage is "; ReadPwrSupplyMv();"mV\n"
mv=2700
rc=SetPwrSupplyThreshMv(mv)

PRINT "\nWaiting for power supply to fall below ";mv;"mV"

//wait for events and messages
WAITEVENT

PRINT "\nExiting..."
```

Expected Output:

```
Supply voltage is 3343mV

Waiting for power supply to fall below 2700mV
Exiting...
```

SETPWRSUPPLYTHRESHMV is an extension function.

7. Events & Messages

smart BASIC is designed to be event driven, which makes it suitable for embedded platforms where it is normal to wait for something to happen and then respond.

To ensure that access to variables and resources doen't end up in race conditions, the event handling is done synchronously, meaning the *smart* BASIC runtime engine has to process a WAITEVENT statement for any events or messages to be processed. This guarantees that *smart* BASIC will never need the complexity of locking variables and objects.

The subsystems which generate events and messages relevant to the routines described in this manual are as follows:-

- BLE events and messages as described here.
- Generic Characteristics events and messages as described here.

8. Module Configuration

There are many features of the module that cannot be modified programmatically which relate to interactive mode operation or alter the behaviour of the smartBASIC runtime engine. These configuration objects are stored in non-volatile flash and are retained until the flash file system is erased via AT&F* or AT&F 1.

To write to these objects, which are identified by a positive integer number, the module must be in interactive mode and the command AT+CFG must be used which is described in detail here.

To read current values of these objects use the command AT+CFG, described here.

Predefined configuration objects are as listed under details of the AT+CFG command.

9. Miscellaneous

Bluetooth Result Codes

There are some operations and events that provide a single byte Bluetooth HCI result code, e.g. the EVDISCON message. The meaning of the result code is as per the list reproduced from the Bluetooth Specifications below. No guarantee is supplied as to its accuracy. Consult the specification for more.

Result codes in `grey` are not relevant to Bluetooth Low Energy operation and are unlikely to appear.

BT_HCI_STATUS_CODE_SUCCESS	**0x00**
BT_HCI_STATUS_CODE_UNKNOWN_BTLE_COMMAND	**0x01**
BT_HCI_STATUS_CODE_UNKNOWN_CONNECTION_IDENTIFIER	**0x02**
BT_HCI_HARDWARE_FAILURE	0x03
BT_HCI_PAGE_TIMEOUT	0x04
BT_HCI_AUTHENTICATION_FAILURE	**0x05**
BT_HCI_STATUS_CODE_PIN_OR_KEY_MISSING	**0x06**
BT_HCI_MEMORY_CAPACITY_EXCEEDED	**0x07**
BT_HCI_CONNECTION_TIMEOUT	**0x08**
BT_HCI_CONNECTION_LIMIT_EXCEEDED	0x09
BT_HCI_SYNC_CONN_LIMI_TO_A_DEVICE_EXCEEDED	0x0A
BT_HCI_ACL_COONECTION_ALREADY_EXISTS	0x0B
BT_HCI_STATUS_CODE_COMMAND_DISALLOWED	**0x0C**
BT_HCI_CONN_REJECTED_DUE_TO_LIMITED_RESOURCES	0x0D
BT_HCI_CONN_REJECTED_DUE_TO_SECURITY_REASONS	0x0E
BT_HCI_BT_HCI_CONN_REJECTED_DUE_TO_BD_ADDR	0x0F
BT_HCI_CONN_ACCEPT_TIMEOUT_EXCEEDED	0x10
BT_HCI_UNSUPPORTED_FEATURE_ONPARM_VALUE	0x11
BT_HCI_STATUS_CODE_INVALID_BTLE_COMMAND_PARAMETERS	**0x12**
BT_HCI_REMOTE_USER_TERMINATED_CONNECTION	**0x13**
BT_HCI_REMOTE_DEV_TERMINATION_DUE_TO_LOW_RESOURCES	**0x14**
BT_HCI_REMOTE_DEV_TERMINATION_DUE_TO_POWER_OFF	**0x15**
BT_HCI_LOCAL_HOST_TERMINATED_CONNECTION	**0x16**
BT_HCI_REPEATED_ATTEMPTS	0x17
BT_HCI_PAIRING_NOTALLOWED	0x18
BT_HCI_LMP_PDU	0x19
BT_HCI_UNSUPPORTED_REMOTE_FEATURE	**0x1A**
BT_HCI_SCO_OFFSET_REJECTED	0x1B
BT_HCI_SCO_INTERVAL_REJECTED	0x1C
BT_HCI_SCO_AIR_MODE_REJECTED	0x1D
BT_HCI_STATUS_CODE_INVALID_LMP_PARAMETERS	**0x1E**
BT_HCI_STATUS_CODE_UNSPECIFIED_ERROR	**0x1F**

BT_HCI_UNSUPPORTED_LMP_PARM_VALUE	0x20
BT_HCI_ROLE_CHANGE_NOT_ALLOWED	0x21
BT_HCI_STATUS_CODE_LMP_RESPONSE_TIMEOUT	**0x22**
BT_HCI_LMP_ERROR_TRANSACTION_COLLISION	0x23
BT_HCI_STATUS_CODE_LMP_PDU_NOT_ALLOWED	**0x24**
BT_HCI_ENCRYPTION_MODE_NOT_ALLOWED	0x25
BT_HCI_LINK_KEY_CAN_NOT_BE_CHANGED	0x26
BT_HCI_REQUESTED_QOS_NOT_SUPPORTED	0x27
BT_HCI_INSTANT_PASSED	**0x28**
BT_HCI_PAIRING_WITH_UNIT_KEY_UNSUPPORTED	**0x29**
BT_HCI_DIFFERENT_TRANSACTION_COLLISION	**0x2A**
BT_HCI_QOS_UNACCEPTABLE_PARAMETER	0x2C
BT_HCI_QOS_REJECTED	0x2D
BT_HCI_CHANNEL_CLASSIFICATION_UNSUPPORTED	0x2E
BT_HCI_INSUFFICIENT_SECURITY	0x2F
BT_HCI_PARAMETER_OUT_OF_MANDATORY_RANGE	0x30
BT_HCI_ROLE_SWITCH_PENDING	0x32
BT_HCI_RESERVED_SLOT_VIOLATION	0x34
BT_HCI_ROLE_SWITCH_FAILED	0x35
BT_HCI_EXTENDED_INQUIRY_RESP_TOO_LARGE	0x36
BT_HCI_SSP_NOT_SUPPORTED_BY_HOST	0x37
BT_HCI_HOST_BUSY_PAIRING	0x38
BT_HCI_CONN_REJ_DUETO_NO_SUITABLE_CHN_FOUND	0x39
BT_HCI_CONTROLLER_BUSY	**0x3A**
BT_HCI_CONN_INTERVAL_UNACCEPTABLE	**0x3B**
BT_HCI_DIRECTED_ADVERTISER_TIMEOUT	**0x3C**
BT_HCI_CONN_TERMINATED_DUE_TO_MIC_FAILURE	**0x3D**
BT_HCI_CONN_FAILED_TO_BE_ESTABLISHED	**0x3E**

10. ACKNOWLEDGEMENTS

The following are required acknowledgements to address our use of open source code on the BL600 to implement AES encryption.

Laird's implementation includes the following files: aes.c and aes.h.

LICENSE TERMS

The redistribution and use of this software (with or without changes) is allowed without the payment of fees or royalties providing the following:

- Source code distributions include the above copyright notice, this list of conditions and the following disclaimer;
- Binary distributions include the above copyright notice, this list of conditions and the following disclaimer in their documentation;
- The name of the copyright holder is not used to endorse products built using this software without specific written permission.

DISCLAIMER

This software is provided 'as is' with no explicit or implied warranties in respect of its properties, including, but not limited to, correctness and/or fitness for purpose.

Issue 09/09/2006

This is an AES implementation that uses only 8-bit byte operations on the cipher state (there are options to use 32-bit types if available).

The combination of mix columns and byte substitution used here is based on that developed by Karl Malbrain. His contribution is acknowledged.

INDEX