User Guide

BL652 *smart*BASIC Extensions

Release 28.9.5.0

This guide pertains to BL652-specific smartBASIC functions and routines. For information on functions and routines that apply to all smartBASIC modules, see the smartBASIC Core Manual.

Revision History

Version	Date	Notes	Contributor(s)	Approver
28.4.0.21	15 Apr 2016	Initial Release	Youssif M. Saeed	Jonathan Kaye
28.4.0.23	14 May 2016	Added NFC chapter	Youssif M. Saeed	Jonathan Kaye
28.5.0.25	07 June 2016	Added LE privacy and LE Secure Connections sections	Youssif M. Saeed	Jonathan Kaye
28.5.0.35	04 July 2016	Extension guide for engineering release v28.5.0.35	Youssif M. Saeed	Jonathan Kaye
28.6.1.2	26 Aug 2016	Production Release	Youssif M. Saeed	Youssif M. Saeed
28.6.1.2	21 Nov 2016	Added Index	Youssif M. Saeed	Youssif M. Saeed
28.6.1.4	12 Dec 2016	Added Packet Length Extension section and Long Write functions	Youssif M. Saeed	Jonathan Kaye
28.6.1.4	03 Feb 2017	Minor edits / formatting	Youssif M. Saeed	Jonathan Kaye
28.6.1.4	20 Feb 2017	Added FlashXXX section in addition to minor edits	Youssif M. Saeed	Jonathan Kaye
28.6.2.0	02 May 2017	Added LE Ping and other throughput functionality	Youssif M. Saeed	Jonathan Kaye
28.7.3.0	31 Oct 2017	Added 2M PHY, removed L2CAP, plus minor fixes	Youssif M. Saeed	Jonathan Kaye
28.8.4.0	05 Jun 2018	Added SPI Slave, HW VSP name, and minor edits; Updated to new template	Youssif M. Saeed	Jonathan Kaye
28.9.5.0	05 Sep 2018	Added 2MPHY adverts and Watchdog timer	Youssif M. Saeed	Jonathan Kaye

Contents

1 Introduction

This user guide provides detailed information on BL652-specific *smart*BASIC extensions which provide a high-level managed interface to the underlying Bluetooth stack in order to manage the following:

- Perform GAP functionality such as scanning, advertising and connections
- Perform GATT server functionality
- Perform GATT client functionality
- Perform pairing, bonding, and security manager functions
- Manage Tx power functionality
- Attribute encoding and decoding
- Perform NFC related functionality
- Events related to the above

1.1 What Does a BLE Module Contain?

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

- A highly integrated radio with an integrated antenna (external antenna options are also available)
- BLE Physical and Link layer
- Higher level stack
- Multiple SIO and ADC
- Wired communication interfaces such as 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

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.

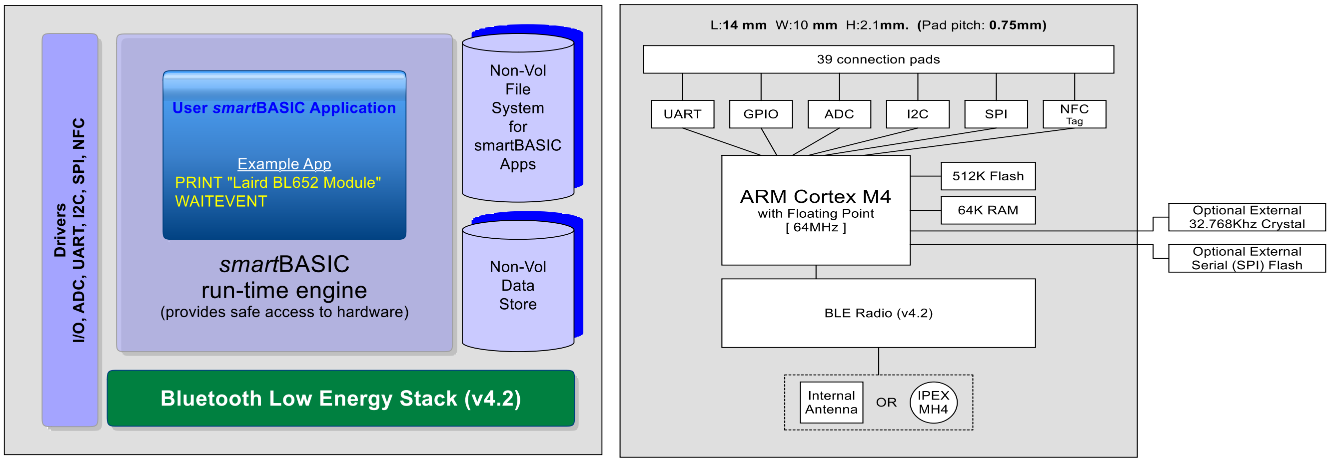


Figure 1: Bluetooth smartBASIC module block diagram

2 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. 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.

3 Interactive Mode Commands

Below are some BL652-specific AT commands.

3.1 AT I or ATI or ATIX

COMMAND

Provides compatibility with the AT command set of Laird's standard Bluetooth modules.

Note 'ATIX' will result in any integer values being displayed in hexadecimal.

AT I num

Returns	\n10\tMM\tInformation\r \n00\r Where \n = linefeed character 0x0A \t = horizontal tab character 0x09 MM = a *number* (see below) Information = string consisting of information requested associated with MM \r = carriage return character 0x0D
Arguments	
num	Integer Constant A number in the range of 0 to 65,535. Currently defined numbers are:

0	Device Name
1	BLE Stack Build Number
3	Version number of module firmware
4	Bluetooth Address
5	Chipset ID
6	File System Flash Segment Statistics
14	Static Random BLE address
16	NvRecords Flash Segment Statistics
24	If AT+MAC used to set IEEE address, then that mac address
26	BLE Bonding database segment
33	smartBASIC core version number
36	Config Keys Flash Serment Statistics

	44	Current random BLE address
	2080	Module startup time
	2081	Get time in milliseconds since reset (will overflow as 32 bit counter)
	7001	Toolchain used to build firmware
	0x8yyy	Get content of FICR register
	0x9yyy	Get content of UICR register
	0xC0FE	Displays the licence
	0xC12C	CRC of most recent file downloaded since reset – volatile The value is in hex.
Interactive Command	Yes	

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

```
'Example:

AT i 3
10   3   28.6.1.2
00
AT I 4
10   4   01 D31A920731B0
```

3.1.1 AT+CFG

COMMAND

AT+CFG is used to set a non-volatile configuration key. Configuration keys 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.

Unless otherwise stated, if a config key value is changed then a reset is required for it to take effect.

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.

<table>
<tr><th>ID</th><th>Definition</th></tr>
<tr><td>40</td><td>Maximum size of local simple variables</td></tr>
<tr><td>41</td><td>Maximum size of local complex variables</td></tr>
<tr><td>42</td><td>Maximum depth of nested user-defined functions and subroutines</td></tr>
<tr><td>43</td><td>The size of stack for storing user functions’ simple variables</td></tr>
<tr><td>44</td><td>The size of stack for storing user functions’ complex variables</td></tr>
<tr><td>45</td><td>The size of the message argument queue length</td></tr>
<tr><td>100</td><td>Enable/Disable Virtual Serial Port Service when in interactive mode. Valid values are:
<table>
<tr><td>0x0000</td><td>Disable</td></tr>
<tr><td>0x0001</td><td>Enable</td></tr>
<tr><td>0x81nn</td><td>Enable ONLY if Signal Pin nn on module is HIGH</td></tr>
<tr><td>0xC1nn</td><td>Enable ONLY if Signal Pin nn on module is LOW</td></tr>
<tr><td>ELSE</td><td>Disable</td></tr>
</table></td></tr>
<tr><td>101</td><td>In Virtual Serial Port Service, select either to use INDICATE or NOTIFY to send data to client.
<table>
<tr><td>0</td><td>Prefer Notify</td></tr>
<tr><td>ELSE</td><td>Prefer Indicate</td></tr>
</table>
This is a preference and the actual value is forced by the property of the TX characteristic of the service.</td></tr>
<tr><td>102</td><td>Advert interval in milliseconds when advertising for connections in interactive mode and AT Parse mode.
Valid values: 20 to 10240 milliseconds</td></tr>
<tr><td>103</td><td>Advert timeout in milliseconds when advertising for connections in interactive mode and AT Parse mode.
Valid values: 0 to 16383 seconds, where 0 means forever.</td></tr>
<tr><td>104</td><td>Data transfer is managed in the Virtual Serial Port service manager.
When sending data using NOTIFIES, the underlying stack uses transmission buffers of which there is a finite number. This specifies the number of transmissons to leave unused when sending a lot of data and 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</td></tr>
<tr><td>105</td><td>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.
If a value of less than 8 is specified, then the minimum value of 7.5 is selected.</td></tr>
<tr><td>106</td><td>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 the minimum specified in 105, then it is forced to the value in 105 plus 2 milliseconds.</td></tr>
<tr><td>107</td><td>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 in 106 is used.</td></tr>
<tr><td>108</td><td>When in interactive mode and connected for virtual serial port services, this is the slave latency to be negotiated with the master. An adjusted value is used if this value times the value in 106 is greater than the</td></tr>
</table>

ID	Definition
	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. The main reason for setting a low value is to ensure that in production, if *smart* BASIC applications are downloaded over the air, limited range allows many stations to be used to program devices.
110	If Virtual Serial Port Service is enabled in interactive mode (see 100), this specifies the size of the transmit ring buffer in the managed layer sitting above the service characteristic FIFO register. Valid range: 32 to 256
111	If Virtual Serial Port Service is enabled in interactive mode (see 100), this specifies the size of the receive ring buffer in the managed layer sitting above the service characteristic fifo register. Valid 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. VSP can also be configured using a $autorun$ application which does not have a waitevent statement so will exit as soon as the VSP is configured.
113	This is the advert interval in milliseconds when advertising for connections in interactive mode and UART bridge mode. VSP can also be configured using a $autorun$ application which does not have a waitevent statement so will exit as soon as the VSP is configured. **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. VSP can also be configured using a $autorun$ application which does not have a waitevent statement so will exit as soon as the VSP is configured. **Valid values:** 0 to 16383 seconds. 0 disables the timer (makes it continuous)
115	This is used to specify the UART baudrate when Virtual Serial Mode Service is active and UART bridge mode is enabled. VSP can also be configured using a $autorun$ application which does not have a waitevent statement so will exit as soon as the VSP is configured. **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.
200	Maximum number of 128-bit, Vendor Specific UUID bases to allocate
204	Gatt Table : Attribute table size in bytes. The size must be a multiple of 4
205	Max number of connections acting as a peripheral (Can be up to 1)

ID	Definition
206	Max number of connections acting as a central (Can be up to 8) **Note:** In order to configure the device to be able to have 8 connections as central, CFG 205 should be set to 0, otherwise the device will auto-adjust to have 7 connections as central and 1 as peripheral.
207	Max number of SMP instances for all connections acting as a central. It is recommended that this is left to 1 as the stack will reserve memory for its use which will only be used occasionally
208	Include the Service Changed characteristic in the Attribute Table (default is included)
209	Security manager is placed in debug mode to use the SIG defined debug key for LE Secure Connections pairing
210	Low Frequency Clock Configuration The BL652 module does not have an onboard 32.768Khz low frequency crystal and that clock is derived from an RC oscillator which is calibrated against the high frequency 32MHz crystal on a periodic basis. However the user has access to the relevant pins (SIO0 and SIO1) to fit the 32K crystal externally. This register is used to configure the LF clock source to be either one or the other or even for autodetection. **Note:** Autodetection means there is a startup delay from reset of up to half a second as opposed to about 1 to 2 milliseconds. This should be factored into any battery life calculations. This configuration register is a bitmask consisting of : ~~~ ~~~ ~~~ ~~~~~~ Bits Len Description ~~~ ~~~ ~~~ ~~~~~~ 0..7 (8) Calibration Time Interval in 1/4 second units 8..15 (8) How often (in number of calibration intervals) the RC oscillator shall be calibrated if the temperature hasn't changed. 16..26 (10) Crystal accuracy in ppm (0..1024ppm) 27..29 (3) Reserved for future use (set to 0) 30..31 (2) LF Clock Source : 00 - Autodetect 01 - RC Oscillator with Calibration against HF Clock 10 - Crystal 11 - Synthesized from HF Clock (Very power inefficient) **Note**: If bits 30-31 is '10' then bits 0-15 are ignored and likewise if 30-31 is '01' then bits 16..26 are ignored. The command AT I 2082 or from an application SYSINFO(2082) will return the actual parameters installed at the instance. So for example if autodection is specified (bits 31..31 == 00) then the value returned will be one of 01, 10 or 11. And similarly for the other parameters, if invalid values where entered.
211	Maximum ATT_MTU size. Possible values are 23 – 247 Bytes.
212	Maximum Attribute data length. Possible values are 20 – 244 Bytes.
213	Use EVCHARVALUE and EVATTRNOTIFYEX instead of the default EVCHARVAL and EVATTRNOTIFY respectively. These former events include all parameters in the event, including the string data, and therefore provide improved throughputs. For more information, see EVCHARVALUE and EVATTRNOTIFYEX.

ID	Definition
214	0: Medium bandwidth (3 packets per connection interval) is used on all connections. 1: High bandwidth (6 packets per connection interval) is used on the FIRST connection. Other connections will have medium bandwidth. Note: when high bandwidth is used, the maximum number of connections that a device can have are reduced from 8 to 6.
216	Maximum packet length a module can use (this is not the same as the current packet length). Possible values are 27-251. By default this is set to 251.
518	The default Uart TX ring buffer length
519	The default Uart RX ring buffer length
520	The baudrate to use for command mode on power up. This setting will be inherited by the $autorun$ application if a print happens before an explicit uartopen inside that application.

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

3.1.2 AT+CFGEX

COMMAND

AT+CFGEX is used to set a non-volatile string configuration key. Configuration keys 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.

Unless otherwise stated, if a config key value is changed then a reset is required for it to take effect.

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   string
```

…where *string* is the current value of the configuration key.

AT+CFGEX num value or AT+CFGEX 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	String_constant This is the new string value for the configuration key.

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
117	VSP advertisement name, the name of the device which will be seen by scanning devices when the module is in VSP mode (can be between 1-31 bytes in length). Default value is: LAIRD BL652

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

3.1.3 AT+BTD *

COMMAND

Deletes the bonded device database from the flash.

AT+BTD*

Returns	\n00\r
Arguments	*None*

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.

Example:

```
AT+BTD*
```

3.1.4 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

Returns	\n00\r
Arguments:	*None*

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

Example

```
AT+BLX
```

3.1.5 AT&F

COMMAND

AT&F provides facilities for erasing various portions of the module's non-volatile memory.

AT&F integermask

Returns	OK if flash is successfully erased
Arguments	
Integermask	Integer corresponding to a bit mask or the * character

The mask is an additive integer mask with the following acceptable values:

0x0000xxxx	Also see core user guide
1	Erases Flash File System
0x100	Erase the System Config keys Flash segment (AT+CFG)
0x10000	Erase the BLE Bonding Manager
0x10 or 0x40000	Erase the NvRecords Flash Segment
*****	Erases all data segments
Else	Not applicable to current modules

If an asterisk is used in place of a number, then the module is configured back to the factory default state by erasing all flash file segments.

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

```
AT&F 1      'delete the file system
AT&F 16     'delete the user config keys
AT&F *      'delete all data segments
```

3.1.6 AT+PROTECT

COMMAND

This command is used to enable readback protection of the flash. For this command to be issued correctly, the readback protection flag should first be enabled using 'AT+PROTECT "E"' followed by setting the protection using 'AT+PROTECT "S"'.

WARNING: Enabling readback protection is a one time only command. Exiting this mode would completely erase the firmware and would require the use of an nrfjprog command to be issued through the JTAG interface. Once erased, a new license for the module will be needed. While this mode is enabled, firmware upgrade can only be carried out over UART. DO NOT enable readback protection unless absolutely necessary.

Notes: To make note of the license, keep a copy of the response to the command AT I 14 and AT I 0xC0FE

AT+PROTECT "Char"

Returns	00 for successful execution.
Arguments:	
"Char"	A character which could be one of the following values:- E – Enable the readback protection flag. D – Disable the readback protection flag. S – Set readback protection on the module. This is an irreversible command.

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

4 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 extension manual for each target platform.

All the core functionality is described in the document *smartBASIC Core Functionality*. Additional information is also available from our Laird Embedded Wireless Solutions Support Center at http://ews-support.lairdtech.com.

Some functions have small behavioral differences from the core functionality; these are listed below.

4.1 Information Routines

4.1.1 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	Device ID. Each platform type has a unique identifier.
	3	Module firmware version number Example: W.X.Y.Z is returned as a 32-bit value made up as follows: **(W<<24) + (X<<18) + (Y<<6) + (Z)** where W is the platform and will always be 28 for the BL652 and X is changed whenever 3rd party libraries are changed. In this case the Nordic Softdevice and Y is the build number and Z is the sub-build number. Note you can check the Softdevice build number in command mode by submitting the command AT I 1
	33	BASIC core version number Example: A.B is returned as a 32 bit value made up as follows: **(A<<8) + (B)** and note the string "A.B" is returned via command mode command AT I 33

	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	Is 1 for run-time only implementation, 3 for compiler included
	1010	Module Type
	2000	Reset Reason ▪ 8 : Self-Reset due to Flash Erase ▪ 9 : ATZ ▪ 10 : Self-Reset due to smart BASIC app invoking function RESET()
	2001	Cause of last reset. This is a bit mask where the bits are defined as follows: Bit 0: Reset from pin-reset Bit 1: Reset from watchdog Bit 2: Reset from soft reset Bit 3: Reset from CPU lockup Bit 16: Reset due to wake up from System OFF mode when wakeup is triggered from GPIO Bit 19: Reset due to wake up from System OFF mode by NFC field detect
	2002	Timer resolution in microseconds
	2003	Number of timers available in a smart BASIC Application
	2004	Tick timer resolution in microseconds
	2005	LMP Version number for BT 4.0 spec
	2006	LMP Sub Version number
	2007	Chipset Company ID allocated by BT SIG
	2008	Returns the current TX power setting (see also 2018)
	2009	Number of devices in trusted device database
	2010	Number of devices in trusted device database with IRK
	2011	Number of devices in trusted device database with CSRK
	2012	Max number of devices that can be stored in trusted device database
	2013	Maximum length of a GATT Table attribute in this implementation
	2016	Radio activity of the baseband and the BT allocation is as follows:- ▪ 0 : advertising ▪ 1 : connected as slave ▪ 2 : Initiating a connection ▪ 3 : scanning for adverts ▪ 4 : connected as master
	2018	Returns the TX power while pairing in progress (see also 2008)
	2021	Stack tide mark in percent. Values near 100 are not good.
	2022	Stack size
	2023	Initial Heap size
	2024	The chipset temperature in tenth of a centigrade. E.g. 23.4 will be returned as 234

	2025	Current free heap memory. Note this is the total of all free blocks and so it is entirely possible to get a MALLOC_FAIL even though this indicates there is enough memory for your need because there may not be a block large enough to accommodate the request. Although smartBASIC does not directly expose malloc/free, they are used extensively in STRING variable operations.
	2026	Supply voltage in millivolts
	2040	Max number of devices that can be stored in trusted device database
	2041	Number of devices in trusted device database
	2042	Number of devices in the rolling device database
	2043	Maximum number of devices that can be stored in the rolling device Database
	2044	Returns a 16 bit hash of the current state of the Gatt Table Schema
	2050	Will be 0 if NFC pins are disabled and 1 if enabled
	2051	Maximum number of NDEF messages that can be created simultaneously
	2052	Maximum size of an NDEF message in bytes
	2080	The startup time from reset to just before the autorun application is launched in milliseconds
	2081	The current tick count in milliseconds
	2082	This is a bitmask value The actual Low Frequency Clock configuration submitted to the softdevice. See AT+CFG 210 description for details about the 4 bit fields in the 32 bits
	2100	Connect Scan Interval used when connecting, in milliseconds
	2101	Connect Scan Window used when connecting, in milliseconds
	2102	Connect Slave Latency default value in connection requests
	2105	Connect Multi-Link Connection Interval periodicity in milliseconds
	2150	Scan Interval used when scanning in milliseconds
	2151	Scan Window used when scanning in milliseconds
	2152	Scan Type Active or Passive (0=Passive, 1=Active)
	2203	Advert Channel Mask
	0x8000 – 0x87FF	Content of FICR register in the Nordic nrf52 chipset. In the nrf52 datasheet, in the FICR section, all the FICR registers are listed in a table with each register identified by an offset, so for example, to read the Code memory page size which is at offset 0x010, call SYSINFO(0x8010) or in interactive mode use AT I 0x8010.
	0x9000 - 0x9800	Content of UICR register in the Nordic nrf52 chipset. In the nrf52 datasheet, in the UICR section, all the UICR registers are listed in a table with each register identified by an offset, so for example, to read the NFC pins functionality which is at offset 0x20C, call SYSINFO(0x920C) or in interactive mode use AT I 0x920C.

Example:

```
// Example :: SysInfo.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 PRINT "\nSysInfo 601   = ";SYSINFO(601)    // Flash File System: Total Space (Data Segment)
 PRINT "\nSysInfo 2102   = ";SYSINFO(2102)  // Default connect slave latency
 PRINT "\nSysInfo 1002 = ";SYSINFO(1002)    // Minimum UART baud rate
```

Expected Output:

```
SysInfo 601   = 49152
SysInfo 2102   = 0
SysInfo 1002 = 1200
```

4.1.2 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 an IEEE Bluetooth address is set. 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.

Example:

```
// Example :: SysInfo$.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 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)   =
```

4.2 UART Interface

4.2.1 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)

The following baud rates are supported: 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, 76800, 115200, 230400, 250000, 460800 and 921600 baud.

4.2.2 UartSetRTS

The BL652 module does not offer the capability to control the RTS pin as the underlying hardware does not allow it.

4.2.3 UartBREAK

The BL652 module does not offer the capability to send a BREAK signal.

4.3 I2C – Two Wire Interface (TWI)

The BL652 can be only be configured as an I2C master if it is the only master on the bus and only 7-bit slave addressing is supported. See core user guide for API details.

When the I2C interface is opened using I2cOpen(), it takes a frequency parameter for the clock line. Valid values are 100KHz, 250KHz and 400KHz.

4.4 SPI Master Interface

The BL652 module can be configured as SPI master and SPI Slave. For SPI Master API details, refer to the smartBASIC Core manual as the SPI Master interface is shared between many *smart*BASIC modules.

4.5 SPI Slave Interface

This section describes all the events and routines used to interact with the SPI Slave peripheral that is available on the module. For successful SPI operations, the remote SPI master's CS, MISO, MOSI, and SCK should be connected directly to the module's CS, MISO, MOSI and SCK pins (respectively). The module's 4 SPI Slave pins can be configured using the SpiSlaveConfig() function, which by default are 11 (CS), 17 (MISO), 18 (MOSI), and 19 (SCK). Special purpose pins such as nAutorun (13) and nReset (22) cannot be configured for SPI Slave operations.

On the BL652, the SPI Slave peripheral supports the following frequencies:- 125KHz, 250KHz, 500KHz, 1MHz, 2MHz, 4MHz, and 8MHz. These frequencies are set by the SPI master and cannot be configured by the SPI Slave.

4.5.1 Events and Messages

4.5.1.1 EVSPISLAVETXRX

This event is thrown when an SPI slave transaction has been completed and the SPI slave Tx/Rx buffers have been updated. The event comes with the following parameters:-

nTxAmount – The amount of data that was read (clocked out) by the remote SPI master.

nRxAmount – The amount of data that was written by the remote SPI master into the SPI slave Rx buffer.

4.5.1.2 EVSPISLAVERXBUFFERFULL

This event is thrown when the SPI slave Rx buffer is full and as a result some data written by the remote SPI master might've been dropped. The event contains the following parameters:-

nRxAmountDropped – The amount of data that was written from the remote SPI master but dropeed due to the buffer being full.

4.5.1.3 EVSPISLAVETXBUFFEREMPTY

This event is thrown when the SPI slave Tx buffer has been emptied due to an SPI master reading out the Tx data from the SPI slave Tx buffer. The handler for this event contains no parameters.

4.5.2 SpiSlaveConfig

FUNCTION

This function is used to update the configuration options of the SPI slave peripheral. If the SPI slave peripheral is already open, then these values will not take effect until the peripheral is closed and then opened again.

SPISLAVECONFIG(nConfigId, nValue)

Returns	INTEGER, a result code. **Typical value:** 0x0000 - The Tx buffer has been updated successfully 0x5260 - Invalid configuration index
Arguments:	
nConfigId	**byVal nConfigId AS INTEGER.** The configuration ID, possible values are:-

0	SPI Slave Chip Select (CS) pin – default 11
1	SPI Slave Master In Slave Out (MISO) pin – default 17
2	SPI Slave Master Out Slave In (MOSI) pin – default 18
3	SPI Slave Clock (SCK) pin – default 19
4	SPI Slave Tx buffer size in bytes – (Possible values: 16-255, default 255)
5	SPI Slave Rx buffer size in bytes – (Possible values: 16-255, default 255)
6	SPI Slave Mode:-

Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

nValue	**byVal nValue AS INTEGER** The value to be assigned to the configuration ID.

Example:

```
// Example :: SpiSlaveConfig.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples
dim rc, nHandle

// Configure SPI Slave peripheral Tx buffer before opening
rc = SpiSlaveConfig(4, 100)
if rc == 0 then
    print "\nSPI slave tx buffer size configured"
else
```

```
    print "\nFailed to configure SPI slave tx buffer with error code ";integer.h'
rc
endif

// Open SPI Slave Periperhal
rc = SpiSlaveOpen(nHandle)
if rc == 0 then
    print "\nOpened SPI Slave peripheral with handle = ";integer.h' nHandle
else
    print "\nFailed to open SPI Slave peripheral"
endif

WaitEvent
```

Expected Output:

```
SPI slave tx buffer size configured
Opened SPI Slave peripheral with handle = 9ABCDEF0
```

4.5.3 SpiSlaveOpen

FUNCTION

This function is used to open a slave SPI peripheral in half duplex mode using the preconfigured SPI Slave values. The parameters (GPIO pins, buffer sizes, mode, etc) are inherited from the SpiSlaveConfig() function. The default parameters on the BL652 are:

SPI Slave CS Pin	11
SPI Slave MOSI Pin	17
SPI Slave MISO Pin	18
SPI Slave SCK Pin	19
SPI Slave Tx Buffer Size (in bytes)	255
SPI Slave Rx Buffer Size (in bytes)	255
SPI Slave Mode	0 (CPOL = 0, CPHL = 0)

In order to change these parameters, the SPI slave peripheral should be closed before SpiSlaveConfig() is used. After all the parameters have been successfully configured, SpiSlaveOpen can be called again at which point the new values will take effect.

SPISLAVEOPEN(nHandle)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
nHandle	**byRef nHandle AS INTEGER.** When calling this function, a variable should be given which on return will contain the handle of the opened SPI Slave peripheral if the function is successful.

Example:

```
// Example :: SpiSlaveOpen.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples
dim rc, nHandle

// Open SPI Slave Periperhal
rc = SpiSlaveOpen(nHandle)
if rc == 0 then
```

```
    print "\nOpened SPI Slave peripheral with handle = ";integer.h' nHandle
else
    print "\nFailed to open SPI Slave peripheral"
endif

WaitEvent
```

Expected Output:

```
Opened SPI Slave peripheral with handle = 9ABCDEF0
```

4.5.4 SpiSlaveClose

FUNCTION

This function is used to close the SPI slave peripheral with the given handle.

SPISLAVECLOSE(nHandle)

Returns		INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:		
	nHandle	**byRef nHandle AS INTEGER.** Handle of the SPI slave interface to close. On return, this will contain an invalid handle indicating that the SPI Slave peripheral has been successfully closed.

Example:

```
// Example :: SpiSlaveClose.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples
dim rc, nHandle

// Open SPI Slave Periperhal
rc = SpiSlaveOpen(nHandle)
if rc == 0 then
    print "\nOpened SPI Slave peripheral with handle = ";integer.h' nHandle

    rc = SpiSlaveClose(nHandle)
    if rc == 0 then
        print "\nSPI Slave successfully closed"
    endif
endif

WaitEvent
```

Expected Output:

```
Opened SPI Slave peripheral with handle = 9ABCDEF0
SPI Slave successfully closed
```

4.5.5 SpiSlaveTxBufferWrite

FUNCTION

This function is used to write the content of a string to the SPI slave Tx buffer. **This written data is only stored in the buffer and not sent to the SPI master until the SPI master selects the SPI slave chip and clock out the data from the buffer.** When the SPI master selects the chip and clocks out the data, the buffer becomes inaccessible by the app until the SPI master operation is complete.

SPISLAVETXBUFFERWRITE(nHandle, strWr$)

Returns	INTEGER, a result code. **Typical value:** 0x0000 The Tx buffer has been updated successfully 0x5206 Tx Buffer full 0x521D Resource busy (e.g. the buffer is being accessed by the remote SPI master) 0x5220 Invalid handle 0x5222 Invalid wite length (e.g. the given string is larger than the Tx buffer size)
Arguments:	
nHandle	**byVal nHandle AS INTEGER.** The handle of the SPI slave interface to write to.
strWr$	**byRef strWr$ AS STRING** Reference to a string variable to write to the SPI slave Tx buffer.

Example:

```
// Example :: SpiSlaveTxBufferWrite.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples
dim rc, nHandle
dim st$ : st$ = "SPI Slave Data"

// Open SPI Slave Periperhal
rc = SpiSlaveOpen(nHandle)
if rc == 0 then
    // Try writing data to the buffer
    rc = SpiSlaveTxBufferWrite(nHandle, st$)
    if rc == 0 then
        print "\nSPI Slave buffer updated with written data"
    else
        print "\nFailed to write SPI Slave data with error code ";integer.h' rc
    endif
endif

WaitEvent
```

Expected Output:

```
SPI Slave buffer updated with written data
```

4.5.6 SpiSlaveRxBufferRead

FUNCTION

This function is used to read the contents of the SPI slave Rx buffer. The data in the Rx buffer would have been placed by the remote SPI master in an earlier transaction. **This Rx buffer can only be accessed if the SPI slave is not selected by the remote SPI master and there is no ongoing SPI operation**. The data that is read is then removed from the buffer in order to make room for more SPI master write operations.

If data is received from the remote SPI master and the Rx buffer is full, the event EVSPISLAVERXBUFFERFULL is thrown with the amount of data that was dropped.

SPISLAVERXBUFFERREAD(nHandle, nLength, strRd$)

Returns	INTEGER, a result code. **Typical value:** 0x0000 The Rx buffer has been read successfully 0x5220 Invalid handle 0x5223 Invalid read length (e.g. the given length is larger than the Rx buffer)
Arguments:	
nHandle	**byRef nHandle AS INTEGER.** Handle of the SPI slave interface to close. On return, this will contain an invalid handle indicating that the SPI Slave peripheral has been successfully closed.
nLength	**byRef nLength AS INTEGER** Number of bytes to read from the Rx buffer. On return, this value will contain the number of data bytes that was read.
strRd$	**ByRef strRd$ AS STRING** On return, this variable will contain the string data that was read from the SPI slave Rx buffer.

Example:

```
// Example :: SpiSlaveRxBufferRead.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples
dim rc, nHandle, st$
dim nLen : nLen = 30 // Try to read 30 bytes of data

// Open SPI Slave Peripheral
rc = SpiSlaveOpen(nHandle)
if rc == 0 then
    // Try reading data from SPI slave buffer
    rc = SpiSlaveRxBufferRead(nHandle, nLen, st$)
    if rc == 0 then
        if nLen > 0 then
            print "\nSPI slave Data read: "; st$
        else
            print "\nNo SPI slave data read"
        endif
    else
        print "\nFailed to read SPI Slave data with error code ";integer.h' rc
    endif
endif

WaitEvent
```

Expected Output:

```
No SPI slave data read
```

4.6 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 entire range of modules. However, some are dependent on the actual I/O availability of each module.

There are 31 SIO (Special I/O) pins available on the BL652. All of these pins can be configured to provide additional types of functionality. However, some of the pins have set functionality that should never be changed.

Note: All of the pins can be configured as digital inputs or outputs, therefore these are not listed in the table below.

Table 1: SIO pin functionality

SIO	Functionality
0	XTAL1
1	XTAL2
2	Adc00, Vsp
3	Adc01
4	Adc02
5	UART_RTS/Adc03
6	UART_TX
7	UART_CTS
8	UART_RX
9	NFC1
10	NFC2
11	No alternate functionality
12	SFlashCS (Only when external serial SPI flash is connected, e.g. BL652 Devkit)
13	Autorun
14	SFlashMiso (Only when external serial SPI flash is connected, e.g. BL652 Devkit)
15	No alternate functionality
16	SFlashClock (Only when external serial SPI flash is connected, e.g. BL652 Devkit)
17	No alternate functionality
18	No alternate functionality
19	No alternate functionality
20	SFlashMosi (Only when external serial SPI flash is connected, e.g. BL652 Devkit)
21	Reset (Cannot be used as an SIO pin)
22	No alternative functionality
23	SpiMosi
24	SpiMiso
25	SpiClock
26	I2cData
27	I2cClock
28	Adc04
29	Adc05
30	Adc06
31	Adc07

Notes: Where Autorun or Vsp functionality is required, then that pin can only be used for that function and cannot be changed.

Pwm option outputs a fully configurable waveform; Freq option outputs a 50:50 mark space ratio waveform.

4.6.1 Events and Messages

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 BL652. 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 BL652, N can only be 0. Use GpioAssignEvent() to generate these events.

4.6.2 GpioSetFunc

FUNCTION

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

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

The nFunction argument denotes the required functionality. Use only supported values from Table 1.

The bSubFunc argument defines the configuration of the requested function.

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 SIO pin as follows: 1 = DIGITAL_IN 2 = DIGITAL_OUT 3 = ANALOG_IN
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 Default for system.

0	Use the system default: 10-bit ADC, 1/6 scaling
0x16	10-bit ADC, 1/6 scaling
0x15	10-bit ADC, 1/5 scaling
0x14	10-bit ADC, 1/4 scaling
0x13	10-bit ADC, 1/3 scaling
0x12	10-bit ADC, 1/2 scaling
0x11	10-bit ADC, 1/1 scaling (Unity)
0x21	10-bit ADC, 2/1 scaling
0x41	10-bit ADC, 4/1 scaling

Note: The internal reference voltage is the same as the module Vcc value with +/- 1.5% accuracy.

Example:

```
// Example :: GpioSetFunc.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples
```

```
PRINT GpioSetFunc(15,1,2)     //Digital In SIO 15, strong pull up resistor
PRINT GpioSetFunc(3,3,0)      //Analog In SIO 3 (Temperature Sensor), default settings
PRINT GpioSetFunc(17,2,1)     //SIO17 (LED0) digital out, initial output high
```

Expected Output:

```
000
```

4.6.3 GpioSetFuncEx

FUNCTION

This routine sets the function of the SIO pin identified by the nSigNum argument and provides for more enhanced configurability compared to the legacy function GpioSetFunc().

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

The nFunction argument denotes the required functionality. Use only supported values from Table 1.

The bSubFunc argument defines the configuration of the requested function.

GPIOSETFUNCEX (nSigNum, nFunction, subFunc$)

<table>
<tr><td>Returns</td><td>INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</td></tr>
<tr><td colspan="2">Arguments:</td></tr>
<tr><td>nSigNum</td><td>byVal nSigNum AS INTEGER.
The signal number as stated in the pinout table of the module.</td></tr>
<tr><td>nFunction</td><td>byVal nFunction AS INTEGER.
Specifies the configuration of the SIO pin as follows:
1 = DIGITAL_IN
2 = DIGITAL_OUT
3 = ANALOG_IN</td></tr>
<tr><td>subFunc$</td><td>byVal nSubFunc$ INTEGER

If nFunction == DIGITAL_IN
subFunc$ will be a string that has the following form:- “\Digital_In_Bitmask”, where Digital_In_Bitmask bits can be as follows:-

Bits 0..3
<table>
<tr><td>0x01</td><td>Pull down resistor (weak)</td></tr>
<tr><td>0x02</td><td>Pull up resistor (weak)</td></tr>
<tr><td>0x03</td><td>Pull down resistor (strong)</td></tr>
<tr><td>0x04</td><td>Pull up resistor (strong)</td></tr>
<tr><td>Else</td><td>No pull resistors</td></tr>
</table>
Bits 4, 5
<table>
<tr><td>0x10</td><td>When in deep sleep mode, awake when this pin is LOW</td></tr>
<tr><td>0x20</td><td>When in deep sleep mode, awake when this pin is HIGH</td></tr>
</table>
</td></tr>
</table>

Else	No effect in deep sleep mode

Bits 8..31

Must be 0s

If nFuncType == DIGITAL_OUT

subFunc$ will be a string that has the following form:- “\Digital_Out”, where Digital_Out consists of the following:-

Bits 0-3: Values

Bits 4-6: Drive Capacity (Only for LOW and HIGH configuration. For PWM and FREQUENCY this is always set to 0=Standard; 1=Standard)

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

The reference voltage for the analog to digital converter is 0.6 volts.

subFunc$ will be a string that has the following form:- “\Gain_hex\Resolution_hex\Acquisition_hex”

If the string is empty, then default values will be used. Otherwise, the values can be as follows:-

Gain_hex

0	Use the system default: 10-bit ADC, 1/6 scaling
0x16	1/6 scaling
0x15	1/5 scaling
0x14	1/4 scaling
0x13	1/3 scaling
0x12	1/2 scaling

	0x11	1/1 scaling (Unity)
	0x21	2/1 scaling
	0x41	4/1 scaling

For example, if you have a maximum analog voltage of 1.7 volts, then select a gain of 1/3 so that the maximum voltage into the convertor will be 1.7 * 1/3 = 0.57 which means it will not be bigger than the reference voltage of 0.6v and it will be specified in subFunc$ so that the first byte in the string is "\13"

Resolution_hex

0	Use the system default: 10-bit ADC
0x08	8-bit ADC resolution
0x0A	10-bit ADC resolution
0x0C	12-bit ADC resolution

Acquisition_hex

0	Use the system default: 10 microseconds
0x03	3 microseconds
0x05	5 microseconds
0x0A	10 microseconds
0x0F	15 microseconds
0x14	20 microseconds
0x28	40 microseconds

Any other value results in this function being rejected.

For example, selecting $1/5^{th}$ scaling, 12 bit resolution and acquisition time of 20 microseconds requires that the variable subFunc$ be initialised as "\15\0C\14"

Note: The internal reference voltage is the same as the module Vcc value with +/- 1.5% accuracy.

Example:

```
// Example :: GpioSetFuncEx.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

//Digital In SIO 15, strong pull up resistor
PRINT GpioSetFuncEx(15,1,"\02")
//Analog In SIO 3 (Temperature Sensor), default settings
PRINT GpioSetFuncEx(3,3,"")
//Analog In SIO 23, 1/6 scaling, 12-bit resolution, 3us acquisition time
PRINT GpioSetFuncEx(23,3,"\16\0C\03")
//SIO17 (LED0) digital out, initial output high
PRINT GpioSetFuncEx(17,2,"\01")
//SIO26 digital out, PWM
PRINT GpioSetFuncEx(26,2,"\02")
```

Expected Output:

```
00000
```

4.6.4 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 special PWM related peripherals in the microcontroller.

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 PWM 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. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
nMinFreqHz	**byRef nMinFreqHz AS INTEGER**. The nominal frequency of the waveform.
nMaxResolution	**byVal *nMaxResolution* AS INTEGER.** Set to same value as nMinFreqHz.

Example:

```
// Example :: GpioConfigPwm.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


dim retval
dim i
dim nFreq
dim nResolution
dim res[5] as integer
FUNCTION HandlerTimer1()
  dim TmpVal
  i=i+1
  if i==5 then
    i=0
  endif
```

```
  TmpVal = (res[i]*100/nFreq)
  PRINT "\nTimer event! PWM changed to "; TmpVal; "% duty cycle."
  GpioWrite(13,res[i])
ENDFUNC 1

i=0
nFreq=2048
nResolution=2048
res[0]=nResolution/2
res[1]=nResolution/4
res[2]=nResolution/8
res[3]=0
res[4]=nResolution

ONEVENT EVTMR1 CALL HandlerTimer1

//Configure PWM
retval = GpioConfigPWM(nFreq,nResolution)
retval = GpioSetFunc(13,2,2)

//Write the first value to the PWM out
GpioWrite(13,res[i])
PRINT "\nTimer started. PWM on 50% duty cycle."

//start a 5000 millisecond (5 second) recurring timer
TimerStart(1,5000,1)

WAITEVENT
```

Expected Output:

```
Timer started. PWM on 50% duty cycle.
Timer event! PWM changed to 25% duty cycle.
Timer event! PWM changed to 12% duty cycle.
Timer event! PWM changed to 0% duty cycle.
Timer event! PWM changed to 100% duty cycle.
```

4.6.5 GpioRead

FUNCTION

This routine reads the value from a SIO pin.

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

GPIOREAD (nSigNum)

Returns	INTEGER, the value from the signal. If the signal number is invalid, then it returns a value of 0. For digital pins, the value is 0 or 1. For ADC pins it is a value in the range 0 to M where M is the maximum value 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.

Refer to the example for GpioBindEvent.

Example:

```
// Example: GpioRead.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


//This example reads from temperature sensor, for it to work, a jumper needs to be placed
on J6 between SIO_3 and TEMP_SENS
#define GPIO_TEMP_SENS           3

dim rc, adc

//Start timer to read temperature sensor
TimerStart(0,1000,1)

//Remove resistor
rc = GpioSetFunc(GPIO_TEMP_SENS, 1, 2)

//Analogue in
rc = GpioSetFunc(GPIO_TEMP_SENS, 3, 0)

FUNCTION HandlerTimer0()
    //Read the ADC
    adc = GpioRead(GPIO_TEMP_SENS)
    PRINT "\nRaw Temperature Sensor Reading: ";adc
ENDFUNC 1
```

```
OnEvent EVTMR0 call HandlerTimer0


WAITEVENT
```

Expected output:

```
Raw Temperature Sensor Reading: 1943
Raw Temperature Sensor Reading: 1943
```

4.6.6 GpioWrite

FUNCTION

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

If the SIO pin is configured as a PWM output then the nNewValue specifies a value in the range 0 to N where N is the nMinFreqHz set in the GpioConfigPwm command. The write value controls the mark space ratio of the output waveform. A value of 0 outputs a low, a value of nMinFreqHz outputs a high, and a value in varies the mark space ratio. The higher the value, the longer the mark period.

As with the GpioConfigPwm function the nNewValue is used to calculate a hardware register value. This value must be less than the register value calculated from the GpioConfigPwm function that is used to set the PWM output frequency. Again, care must be taken to avoid non integer results or the output waveform will not be accurate.

As an indication if you divide the PWM output frequency by the value of the register calculated in the GpioConfigPwm function above, then that result is the minimum nNewValue you can enter to get a mark:space ratio. Other valid mark:space ratios are provided by integer multiples of this minimum value.

For example with a system frequency of 40 MHz and an output PWM frequency of 5 MHz then the register value to provide the output frequency will be 8. So the minimum value of nNewValue is 0.625 MHz and the remaining obtainable values are 4.375, 3.75, 3.125, 2.5, 1.875 and 1.25 MHz. Any other nNewValue entered will round down to one of these values.

GPIOWRITE (nSigNum, nNewValue)

Returns	
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 clears the pin and a non-zero value sets it. 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.

Example:

```
// Example :: GpioWrite.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


dim rc, i1, i2
i2 = 1
i1 = 1
```

```
//------------------------------------------------------------------------------
// For debugging
// --- rc = result code
// --- ln = line number
//------------------------------------------------------------------------------
Sub AssertRC(rc,ln)
    if rc!=0 then
        print "\nFail :";integer.h' rc;" at tag ";ln
    endif
EndSub

rc=GpioSetFunc(17,2,1)
AssertRC(rc,20)

rc=GpioSetFunc(19,2,1)
AssertRC(rc,23)

function HandlerTmr0()
    i1=!i1
    GpioWrite(19,i1)
    AssertRC(rc,30)
endfunc 1

function HandlerTmr1()
    i2=!i2
    GpioWrite(17,i2)
    AssertRC(rc,42)
endfunc 1

function HandlerUartRx()
endfunc 0

TimerStart(0,500,1)
TimerStart(1,1000,1)

onevent evuartrx call HandlerUartRx
onevent evtmr0   call HandlerTmr0
onevent evtmr1   call HandlerTmr1
print "\n\nPress any key to exit"
```

```
waitevent

print "\nExiting..."
```

Expected Output:

```
Press any key to exit
Exiting...
```

4.6.7 GpioBindEvent/GpioAssignEvent

FUNCTION

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

When this function is called on the BL652, the SIO pin specified by nSigNum is set up as a digital input in the underlying firmware so GpioSetFunc() does **not** need to be called beforehand.

If this function is used in your *smart*BASIC application, we recommend that you unbind all bound events by calling GpioUnbindEvent() at the end of the application. Likewise for all assigned events, GpioUnassignEvent should be called.

Note: In the BL652 module an SIO pin can only be bound to one event at a time.

GPIOBINDEVENT (nEventNum, nSigNum, nPolarity)

GPIOASSIGNEVENT (nEventNum, nSigNum, nPolarity)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
nEventNum	**byVal *nEventNum* INTEGER**. The SIO event number (in the range of 0 - N) which will result in the event EVGPIOCHANn being thrown if GpioBindEvent is used, or EVDETECTCHANn if GpioAssignEvent is used .
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 — (GpioBindEvent Only) Either a low to high or high to low transition

Note: Using GpioBindEvent provides the capability to detect any transition. However, it results in slightly higher power consumption. If power is of importance, GpioAssignEvent() should be used instead as it uses other resources to expedite an event.

Example:

```
// Example :: GpioBindEvent.sb
```

```
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples
dim rc

function HandlerBtn0()
    dim i : i = GpioRead(11)

    '//if button 0 was pressed
    if i==0 then
        print "\nButton 0 Pressed"

    '//if button 0 was released
    elseif i==1 then
        print "\nButton 0 Released"
    endif
endfunc 1

function HandlerUartRx()
endfunc 0

rc= GpioBindEvent(0,11,2)        //Bind event 0 to high or low transition on SIO11 (button 
1)
if rc==0 then
    onevent evgpiochan0 call HandlerBtn0  //When event 0 happens, call Btn0Press
    print "\nSIO11 - Button 0 is bound to event 0. Press button 0"
else
    print "\nGpioBindEvent Err: ";integer.h'rc
endif

onevent evuartrx call HandlerUartRx
print "\n\nPress any key to exit"

waitevent
rc=GpioUnbindEvent(0)
if rc==0 then
    print "\n\nEvent 0 unbound\nExiting..."
endif
```

Expected Output:

```
SIO11 - Button 0 is bound to event 0. Press button 0

Press any key to exit
Button 0 Pressed
```

```
Button 0 Released
Button 0 Pressed
Button 0 Released

Event 0 unbound
Exiting...
00
```

4.6.8 GpioUnbindEvent/GpioUnAssignEvent

FUNCTION

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

GPIOUNBINDEVENT (nEventNum)

GPIOUNASSIGNEVENT (nEventNum)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
nEventNum	**byVal *nEventNum* INTEGER.** The SIO event number (in the range of 0 - N) which will be disabled so that it no longer generates run-time events in *smart* BASIC.

See example for GpioBindEvent.

4.7 Miscellaneous Routines

This section describes all miscellaneous functions and subroutines.

4.7.1 ASSERTBL652

SUBROUTINE

This function's main use case is during *smart*BASIC source compilation and the presence of at least one instance of this statement will ensure that the *smart*BASIC application will only fully compile without errors on a BL652 module. This ensures that apps for other modules are not mistakenly loaded into the BL652.

AssertBL652 ()

Returns	Not acceptable as it is a subroutine
Arguments:	None

Example:

```
AssertBL652()//Ensure loading on BL652 only
```

4.7.2 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 SIO2 input pin is held high.

Given that SIO2 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 SIO2 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:

```
DIM rc
 rc = EraseFileSystem(1234)
 IF rc!=0 THEN
   PRINT "\nFailed to erase file system because incorrect parameter"
 ENDIF
 //Input SIO2 is low
 rc = EraseFileSystem(1)
 IF rc!=0 THEN
   PRINT "\nFailed to erase file system because SIO19 is low"
 ENDIF
```

Expected Output:

```
Failed to erase file system because incorrect parameter
Failed to erase file system because SIO19 is low
00
```

5 BLE Extensions Built-in Routines

5.1 LE Privacy

To address privacy concerns, there are four types of Bluetooth addresses in a BLE device which can change as often as required. For example, an iPhone regularly changes its BLE Bluetooth address and it always exposes only its resolvable random address.This feature is known as LE privacy. It allows the Bluetooth address within advertising packets to be replaced with a random value that can change at different time intervals. Malicious devices would not be able to track your device as it actually looks like a series of different devices.

To manage this, the usual six octet Bluetooth address is qualified on-air by a single bit which qualifies the Bluetooth address as public or random:

- Public – The format is as defined by the IEEE organisation.

- Random – The format can be up to three types and this qualification is done using the upper two bits of the most significant byte of the random Bluetooth address.

Address types:

00	Public
01	Random Static
02	Random Private Resolvable
03	Random Private Non-Resolvable
All other values are illegal	

On the BL652, the address type can be set using the function BleSetAddressTypeEx(). On the other hand, Sysinfo$(4) can be used to retrieve the Bluetooth address if it is public or random static. Due to LE privacy 1.2, if the address type is random resolvable or random non-resolvable then it cannot be retrieved by the application layer since it is fully controlled by the baseband layer.

Note: The Bluetooth address portion in *smart*BASIC is always in big endian format. If you sniff on-air packets, the same six packets will appear in 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.

5.1.1 BleSetAddressTypeEx

FUNCTION

This functions sets the current address type to be used by the LE radio scan/advert/connection requests. Type 2 and 3 can be set to be refreshed periodically.

BLESETADDRESSTYPEEX (nAddrType, nPeriodMS)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
nAddrType	**byVal *nAddrType* AS INTEGER.** Specifies the type of the LE address as follows: 0 – Public address, same as Classic. 1 – Random static address, generated first boot. 2 – Random address, resolvable with IRK, generated on call. 3 – Random address, non resolvable, generation on call
nPeriodMS	The time period for changing resolvable and non-resolvable addresses in milliseconds. If the nAddrType is 0 or 1 then this parameter is ignored. Negative values result in an error being returned. A value of 0 means the address will not change

Example:

```
// Example: BleSetAddressTypeEx.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

DIM rc, addr$

// Set the address to pulic, nPeriodMS is ignored

rc = BleSetAddressTypeEx(0,0)

addr$ = SysInfo$(4)
```

```
PRINT "\nBluetooth Address - "; StrHexize$(addr$)

// Set the address to random static, nPeriodMS is ignored
rc = BleSetAddressTypeEx(1,0)
addr$ = SysInfo$(4)
PRINT "\nBluetooth Address - "; StrHexize$(addr$)

// Set the address to be random resolvable that changes every 30 seconds
rc = BleSetAddressTypeEx(2,30000)
addr$ = SysInfo$(4)
PRINT "\nCurrent Address - "; StrHexize$(addr$)

// Set the address to be random non-resolvable that changes every 1 seconds
rc = BleSetAddressTypeEx(3,1000)
addr$ = SysInfo$(4)
PRINT "\nBluetooth Address - "; StrHexize$(addr$)
```

Expected Output:

```
Bluetooth Address - 000016A4B75201
Bluetooth Address - 01D3B61EE3F699
Bluetooth Address - 01D3B61EE3F699
Bluetooth Address - 01D3B61EE3F699
```

Note: Even though Sysinfo$(4) returns the random static address after setting address types 2 and 3, the actual address used by the radio packets are the random resolvable and the random non-resolvable addresses respectively. The reason for this is that private addresses are only known to the baseband.

5.2 Events and Messages

5.2.1 EVBLE_ADV_TIMEOUT

This event is thrown when adverts that are started using BleAdvertStart() time out.

Example:

```
// Example :: EvBle_Adv_Timeout.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 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 "\n Advert Started"
ELSE
    PRINT "\n\nAdvert not successful"
ENDIF

ONEVENT  EVBLE_ADV_TIMEOUT  CALL   HndlrBleAdvTimOut

WAITEVENT
```

Expected Output:

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

5.2.2 EVBLE_CONN_TIMEOUT

This event is thrown when a BLE connection attempt initiated by the BleConnect() function times out.

See example for BleConnect.

5.2.3 EVBLE_ADV_REPORT

This event is thrown when an advert report is received whether successfully cached or not.

See example for BleScanGetAdvReport.

5.2.4 EVBLE_FAST_PAGED

This event is thrown when an advert report is received which is of type ADV_DIRECT_IND and the advert had a target address (InitA in the spec) which matches the address of this module.

See example for BleScanGetPagerAddr.

5.2.5 EVBLE_SCAN_TIMEOUT

This event is thrown when a BLE scanning procedure initiated by the BleScanStart() function times out.

See example for BLESCANSTART.

5.2.6 EVBLEMSG

The BLE subsystem is capable of informing a *smart* BASIC application when a significant BLE related event has occurred and it does so by throwing this message (as opposed to an EVENTTable 20, which is akin to an interrupt and has no context or queue associated with it).

The message contains two parameters:

- **msgID** – Identifies what event was triggered
- **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 and, unless that queue is full, pends all messages until they are handled. Only messages that have handlers associated with them are inserted into the queue. This prevents messages that will not get handled from filling that queue. The following table lists the triggers and associated context parameters.

MsgID	Description
0	A BLE connection is established and msgCtx is the connection handle.
1	A BLE disconnection event and msgCtx identifies the handle.
4	A BLE Service Error. The second parameter contains the error code.
9	Pairing in progress and displayed Passkey supplied in msgCtx.
10	A new bond has been successfully created.
11	Pairing in progress and authentication key requested. msgCtx is key type.
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.
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.
22	Attempt to add a new bonding to the bonding database failed
23	On a BLE connection to a bonded device, if the current GATT table schema does not match what existed at the last connection, then a GATT Service Change Indication is automatically sent and the app is informed via this event
24	On a BLE connection to a bonded device, if the current gatt table schema does not match what existed at the last connection, then a GATT Service Change Indication is automatically sent and the app is informed when the client acknowledges that indication

Note: Message ID 13 is reserved for future use.

Example:

```
// Example :: EvBleMsg.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


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
        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 HndlrUartRx()
    rc=BleAdvertStop()
    PRINT "\nExiting..."
ENDFUNC 0



ONEVENT  EVBLEMSG           CALL HndlrBleMsg
ONEVENT  EVBLE_ADV_TIMEOUT  CALL HndlrBlrAdvTimOut
ONEVENT  EVUARTRX           CALL HndlrUartRx

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

```
ENDIF


WAITEVENT
```

Expected Output (When connection made with the module):

```
Adverts Started
Press any key 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 any key to exit

Advert stopped via timeout
Exiting...
```

5.2.7 EVDISCON

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

- Connection handle
- The 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 is provided in this document here.

Example:

```
// Example :: EvDiscon.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 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
```

5.2.8 EVCHARVAL

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

- **Char Handle** - Characteristic handle that was returned when the characteristic was registered using the function BleCharCommit()
- **Offset** – Offset
- **Length** – Length of the data from the characteristic value

5.2.9 EVCHARVALUE

This event is thrown when the remote device writes to a characteristic value. It differs from EVCHARVAL in that the event contains the parameters including the connection handle and the string data. If the write operation is performed on a characteristic that requires authorisation, then EVAUTHVAL is thrown instead, and the user should then authorize and read the value.

If the event is thrown with an empty string but the length has a non-zero value, then this indicates that there was not enough memory to allocate to the event.

The event comes with the following parameters:-

- **Connection Handle** – The handle of the connection that wrote to the characteristic value.
- **Char Handle** - Characteristic handle that was returned when the characteristic was registered using the function BleCharCommit()
- **Offset** – The offset at which the characteristic data was written.
- **Length** – The length of the data that was written. This should be equal to StrLen$(Data$), and can be used to detect if there was any data loss.

- **Data$** - The string data that was written to the characteristic.

Example:

```
// Example :: EvCharVal.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


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$,0x18EE,-1,-1,-1,-1,-1)
    //commit reports to GATT table - adRpt$ 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 - Thrown when AT+CFG 213=0
//==============================================================================
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

//==============================================================================
// New char value handler - Thrown when AT+CFG 213=1
//==============================================================================
FUNCTION HandlerCharValue(BYVAL nConnHandle, BYVAL charHandle, BYVAL offset, BYVAL len,
BYVAL Data$)
    DIM s$
    IF charHandle == hMyChar THEN
        PRINT "\n";len;" byte(s) have been written to char value attribute from offset
";offset

        PRINT "\nData written is :";Data$ PRINT "\nData written is :";Data$;" - Connection
Handle=";integer.h' nConnHandle


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

ONEVENT  EVCHARVAL   CALL HandlerCharVal    // This event is thrown if AT+CFG 213 = 0
ONEVENT  EVCHARVALUE CALL HandlerCharValue  // This event is thrown if AT+CFG 213 = 1
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 (AT+CFG 213=0):

```
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...
```

Expected Output (AT+CFG 213=1):

```
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
Data written is :hello - Connection Handle=0001FF00

New Char Value: Hello

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

5.2.10 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:

```
// Example :: EVCHARHVC charHandle
// See example that is provided for EVCHARCCCD
```

5.2.11 EVCHARCCCD

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

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

Example:

```
// Example :: EvCharCccd.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 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 BL652 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 BL652 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...
```

5.2.12 EVCHARSCCD

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

- The characteristic handle that is returned when the characteristic is registered using the function BleCharCommit()
- The new 16-bit value in the updated SCCD attribute

The SCCD is used to manage broadcasts of characteristic values.

Example:

```
// Example :: EvCharSccd.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


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...
```

5.2.13 EVCHARDESC

This event is thrown when the client writes to a writable descriptor of a characteristic which is not a CCCD or SCCD (they are catered for with their own dedicated messages). It comes with two parameters: the characteristic handle that was returned when the characteristic was registered using the function BleCharCommit(), and 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:

```
// Example :: EvCharDesc.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

 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))

     //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
```

5.2.14 EVAUTHVAL

This event is thrown **instead of EVCHARVAL** when a characteristic with read and/or write authorisation is being read or written to by a remote GATT client. It comes with three parameters:

- **Connection handle** – The connection handle of the GATT client
- **Char handle** –The characteristic handle that was returned when the characteristic was registered using the function BleCharCommit()
- **ReadWrite** –Will be 0x00000000 when this is a read attempt and 0x00010000 when write attempt

Call BleAuthorizeChar() to either grant or deny access.

If this a write attempt and access is granted then as soon as the function BleAuthoriseChar() returns the new value is ready to be read using BleCharValueRead().

Note: When a characteristic requires authentication and the remote device reads from it or writes to it using the WRITE_CMD (write without response), the event EVAUTHVALEX is thrown instead. The user should therefore have both EVAUTHVAL and EVAUTHVALEX events in their app and service the events appropriately. See the example below for more information.

5.2.15 EVAUTHVALEX

This event is thrown when the remote device writes to a characteristic value that requires authentication using the WRITE_CMD (write without response) command. The user should then write the data using BleCharValueWriteEx at the app layer, otherwise the value will not be updated. If the event is thrown with an empty string but the length has a non-zero value, then this indicates that there was not enough memory to allocate to the event. The event comes with three parameters:

- **Connection handle** – The connection handle of the GATT client
- **Char handle** –The characteristic handle that was returned when the characteristic was registered using the function BleCharCommit()
- **Offset** – The offset of the characteristic at which the remote is attempting to write.
- **Length** – The length of the data that the remote is attempting to write. This should be equal to StrLen$(Data$) and can be used to verify that no data loss has occurred.
- **Data$** – The string data that the remote device is attempting to write.

Note: When a characteristic requires authentication and the remote device reads from it or writes to it using a noramal WRITE, the event EVAUTHVAL is thrown instead. The user should therefore have both EVAUTHVAL and EVAUTHVALEX events in their app and service the events appropriately. See the example below for more information.

Example:

```
// Example :: EvAuthVal.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

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=BleServiceNew(1,BleHandleUuid16(0x18EE),hSvc)
    //Initialise char, write/read enabled, accept signed writes
    rc=BleCharNew(0x0A,BleHandleUuid16(1),BleAttrMetaDataex(1,1,20,8,rc),0,0)
    //Commit char initialised above, with initial value "hi" to service 'hSvc'
    rc=BleCharCommit(hSvc,attr$,hMyChar)
    //Commit changes to the service
    rc=BleServiceCommit(hSvc)
    rc=BleScanRptInit(scRpt$)
    //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

//------------------------------------------------------------------------------
// AUTHVAL - The remote has written to the characteristic using WRITE (write with response)
//------------------------------------------------------------------------------
FUNCTION HndlrAuthVal(BYVAL connHandle, BYVAL charHandle, BYVAL readWrite)
    DIM s$
    IF charHandle == hMyChar THEN
        IF readWrite!=0 THEN
            rc=BleAuthorizeChar(connHandle, charHandle, 3)  //Grant access
            rc=BleCharValueRead(hMyChar,s$)
            PRINT "\nAuthenticated char written using Write with response."
            PRINT "\nNew Char Value: ";s$
        ENDIF
```

```
    ENDIF
ENDFUNC 1

//------------------------------------------------------------------------------
// AUTHVALEX - The remote has written to the characteristic using WRITE_CMD (write without
response)
//------------------------------------------------------------------------------
FUNCTION HndlrAuthValEx(BYVAL connHandle, BYVAL charHandle, BYVAL offset, BYVAL length,
BYVAL data$ AS STRING)
    DIM s$
    IF charHandle == hMyChar THEN
        // We are OK with this connection handle, so write the characteristic
        rc = BleCharValueWriteEx(charHandle, offset, data$)
        rc=BleCharValueRead(hMyChar,s$)
        PRINT "\nAuthenticated char written using Write without response."
        PRINT "\nNew Char Value: ";s$
    ENDIF
ENDFUNC 1

//------------------------------------------------------------------------------
// Enable synchronous event handlers
//------------------------------------------------------------------------------
ONEVENT  EVBLEMSG       CALL HndlrBleMsg
ONEVENT  EVAUTHVAL      CALL HndlrAuthVal
ONEVENT  EVAUTHVALEX    CALL HndlrAuthValEx

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
```

Expected Output:

```
The characteristic's value is Hi
Write a new value to the characteristic
--- Connected to client
Authenticated char written using Write with response.
New Char Value: "Test"
Authenticated char written using Write without response.
New Char Value: "Test"
```

5.2.16 EVAUTHCCCD

This event is thrown **instead of EVCHARCCCD** when a CCCD descriptor of a characterisic with read and/or write authorisation is being read or written to by a remote GATT client. It comes with three parameters as follows:

- The connection handle of the gatt client
- The characteristic handle returned when the characteristic was registered with BleCharCommit()
- Will be 0x00000000 when this is a read attempt and 0x0001HHHH when write attempt where the new 16-bit value to be written is 0xHHHH

Call BleAuthorizeDesc() to either grant or deny access.

If this a write attempt and access is granted then as soon as the function BleAuthoriseDesc() returns the new value 0xHHHH is assumed to be written to the descriptor.

Example:

```
// Example :: EvAuthCccd.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples



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 = BleAttrMetaDataex(1,1,20,0,metaSuccess)
    DIM hSvcUuid : hSvcUuid = BleHandleUuid16(svcUuid)
    DIM mdCccd : mdCccd = BleAttrMetadataex(1,1,2,8,rc)   //CCCD metadata for char, write
auth

    //Commit svc with handle 'hSvcUuid'
    rc=BleSvcCommit(1,hSvcUuid,hSvc)
    //Initialise char, write/read enabled, accept signed writes, indicatable
    rc=BleCharNew(0x6A,charUuid,charMet,mdCccd,0)
    //Commit char initialised above, with initial value "hi" to service 'hMyChar'
    rc=BleCharCommit(hSvc,attr$,hMyChar)
    rc=BleScanRptInit(scRpt$)
    //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)
    rc=GpioBindEvent(1,16,1)     //Channel 1, bind to low transition on GPIO pin 16
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

//==============================================================================
// Handler to service button 0 pressed
//==============================================================================
FUNCTION HndlrBtn0Pr() AS INTEGER
    CloseConnections()
ENDFUNC 1

//==============================================================================
// CCCD descriptor authorisation
//==============================================================================
FUNCTION HndlrAuthCccd(BYVAL connHandle,BYVAL charHandle, BYVAL readWrite) AS INTEGER
    DIM value$

    IF charHandle==hMyChar THEN
        IF readWrite != 0x0 THEN

            rc=BleAuthorizeDesc(connHandle,charHandle, -1 ,3)  //grant access
            IF readWrite == 0x10002 THEN
                PRINT "\nSending indication..."
                value$="hello"
                IF BleCharValueIndicate(hMyChar,value$)!=0 THEN
                    PRINT "\nFailed to indicate new value"
                ENDIF
            ELSE
                PRINT "\nIndications were disabled"
            ENDIF
        ENDIF
    ELSE
        PRINT "\nThis is for some other characteristic"
    ENDIF
ENDFUNC 1

ONEVENT  EVBLEMSG    CALL HndlrBleMsg
ONEVENT  EVCHARHVC   CALL HndlrCharHvc
ONEVENT  EVAUTHCCCD  CALL HndlrAuthCccd
ONEVENT  EVGPIOCHAN1 CALL HndlrBtn0Pr

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 button 0 to exit"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF

WAITEVENT

PRINT "\nExiting..."
```

Expected Output:

```
Value of the characteristic 1818531328 is: Hi
You can write to the CCCD characteristic.
The BL600 will then indicate a new characteristic value

--- Press button 0 to exit
--- Connected to client
Sending indication...
Got confirmation of recent indication
```

5.2.17 EVAUTHSCCD

This event is thrown **instead of EVCHARSCCD** when a SCCD descriptor of a characterisic with read and/or write authorisation is being read or written to by a remote GATT client. It comes with three parameters as follows:

1. The connection handle of the gatt client
   1. The characteristic handle returned when the characteristic was registered with BleCharCommit()
   2. Will be 0x00000000 when this is a read attempt and 0x0001HHHH when write attempt where the new 16-bit value to be written is 0xHHHH

Call BleAuthorizeDesc() to either grant or deny access.

If this a write attempt and access is granted then as soon as the function BleAuthoriseDesc() returns the new value 0xHHHH is assumed to be written to the descriptor.

The SCCD is used to manage broadcasts of characteristic values.

Example:

```
// Example :: EvAuthSccd.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

DIM hMyChar,rc,at$,conHndl

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

    //Commit svc with handle 'hSvcUuid'
    rc=BleServiceNew(1,BleHandleUuid16(0x18EE),hSvc)
    //Initialise char, read enabled, accept signed writes, broadcast capable

rc=BleCharNew(0x4B,BleHandleUuid16(1),charMet,0,BleAttrMetadataex(1,1,2,8,rc2))
    //Commit char initialised above, with initial value "hi" to service 'hMyChar'
    rc=BleCharCommit(hSvc,attr$,hMyChar)
    //Commit svc
    rc=BleServiceCommit(hSvc)

    rc=BleAdvRptInit(adRpt$,0x02,0,20)
    //Add 'hSvc' and 'hMyChar' to the advert report
    rc=BleAdvRptAddUuid16(adRpt$,hSvc,hMyChar,-1,-1,-1,-1)
    //Commit reports to GATT table - adRpt$ is empty
    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
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

//==============================================================================
//handler to service button 0 pressed
//==============================================================================
FUNCTION HndlrBtn0Pr() AS INTEGER
    CloseConnections()
ENDFUNC 1

//==============================================================================
// CCCD descriptor written handler
//==============================================================================
FUNCTION HndlrAuthSccd(BYVAL connHandle,BYVAL charHandle, BYVAL readWrite) AS
INTEGER
    DIM value$

    IF charHandle==hMyChar THEN
        IF readWrite != 0x0 THEN
            rc=BleAuthorizeDesc(connHandle,charHandle, -2 ,3)  //grant access
            if readWrite == 0x10000 then
                PRINT "\nBroadcasts have been disabled by client"
            ELSE
                PRINT "\nBroadcasts have been enabled by client"
            endif
        ENDIF
    ELSE
        PRINT "\nThis is for some other characteristic"
    ENDIF
ENDFUNC 1

ONEVENT  EVBLEMSG    CALL HndlrBleMsg
```

```
ONEVENT  EVAUTHSCCD  CALL HndlrAuthSccd
ONEVENT  EVGPIOCHAN1 CALL HndlrBtn0Pr

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

WAITEVENT

PRINT "\nExiting..."
```

Expected Output:

```
Characteristic Value: Hi
You can write to the SCCD attribute.
The BL600 will then indicate a new characteristic value
--- Press button 0 to exit

--- Connected to client
Broadcasts have been enabled by client
```

5.2.18 EVAUTHDESC

This event is thrown **instead of EVCHARDESC** when a writable descriptor of a characteristic with read and/or write authorisation is being read or written by a remote GATT client. It comes with four parameters:

1. The connection handle of the gatt client
3. The characteristic handle that was returned when the characteristic was registered using the function BleCharCommit()
4. The descriptor Handle Index
5. Will be 0x00000000 when this is a read attempt and 0x00010000 when write attempt

Call BleAuthorizeChar() to either grant or deny access.

The first three parameters in the event are supplied as-is as the first three parameters to the function BleAuthizeChar().

If this event is for a write then as soon as the function BleAuthorizeDesc() returns the descriptor contains the value and so the function BleCharDescRead() can be called to read it.

Example:

```
// Example :: EvAuthDesc.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

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$
    attr$="Hi"
    DIM charMet : charMet = BleAttrMetaData(1,1,20,0,rc)
```

```
    //Commit svc with handle 'hSvcUuid'
    rc=BleServiceNew(1,BleHandleUuid16(0x18EE),hSvc)
    //Initialise char, read/write enabled, accept signed writes
    rc=BleCharNew(0x4A,BleHandleUuid16(1),charMet,0,0)

    //Add another descriptor
    attr$="descr_value"
    rc=BleCharDescAdd(0x2905,attr$,BleAttrMetadataex(1,1,20,9,rc))
    //Commit char initialised above, with initial value "hi" to service 'hMyChar'
    attr2$="char value"
    rc=BleCharCommit(hSvc,attr2$,hMyChar)
    rc=BleServiceCommit(hSvc)

    rc=BleAdvRptInit(adRpt$,0x02,0,20)
    rc=BleScanRptInit(scRpt$)
    //Get UUID handle for other descriptor
    hOtherDscr=BleHandleUuid16(0x2905)
    //Add 'hSvc','hMyChar' and the other descriptor to the advert report
    rc=BleAdvRptAddUuid16(adRpt$,hSvc,hOtherDscr,-1,-1,-1,-1)
    rc=BleAdvRptAddUuid16(scRpt$,hOtherDscr,-1,-1,-1,-1,-1)
    //Commit reports to GATT table - adRpt$ is empty
    rc=BleAdvRptsCommit(adRpt$,scRpt$)
    rc=BleAdvertStart(0,addr$,20,300000,0)
    rc=GpioBindEvent(1,16,1)     //Channel 1, bind to low transition on GPIO pin 16
ENDFUNC attr$

//==============================================================================
// 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

//==============================================================================
// Handler to service button 0 pressed
//==============================================================================
FUNCTION HndlrBtn0Pr() AS INTEGER
    CloseConnections()
ENDFUNC 1

//==============================================================================
// Client has written to writeable descriptor
//==============================================================================
FUNCTION HndlrAuthDesc(BYVAL hConn AS INTEGER, BYVAL hChar AS INTEGER, BYVAL hDesc AS
INTEGER, BYVAL rw) AS INTEGER
    dim duid,a$,rc
    IF hChar == hMyChar THEN
```

```
        rc = BleAuthorizeDesc(hConn, hChar, hDesc, 3)
        rc = BleCharDescRead(hChar,hDesc,0,512,duid,a$)
        IF rc ==0 THEN
            PRINT "\nNew value for desriptor ";hDesc;" is ";a$
        ELSE
            PRINT "\nCould not access the uuid"
        ENDIF
    ELSE
        PRINT "\nThis is for some other characteristic"
    ENDIF
ENDFUNC 1

ONEVENT  EVBLEMSG    CALL HndlrBleMsg
ONEVENT  EVAUTHDESC  CALL HndlrAuthDesc
ONEVENT  EVGPIOCHAN1 CALL HndlrBtn0Pr

PRINT "\nOther Descriptor Value: ";OnStartup$()
PRINT "\nWrite a new value \n--- Press button 0 to exit\n"

WAITEVENT

PRINT "\nExiting..."
```

Expected Output:

```
Other Descriptor Value: descr_value
Write a new value
--- Press button 0 to exit

--- Connected to client
New value for desriptor 0 is cC
```

5.2.19 EVVSPRX

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

5.2.20 EVVSPRXOVRN

This event is thrown when the Virtual Serial Port service is open, data has arrived from the peer, and there is not enough space in the receive ring buffer. This results in the appropriate amount of oldest data in the ring buffer being discarded to make room for the new data.

5.2.21 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

5.2.22 EVCONNRSSI

This event message is thrown when rssi reporting is enabled for specific connections using the function BleConnRssiStart() which takes the connection handle.

It consists of a two integers payload and the values are as follows:

- Integer 1 – The connection handle for which the rssi is being reported
- Integer 2 – The signed rssi value in units of dBm.

5.2.23 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 a finite number of these temporary buffers. If they are exhausted, the notify function or the write_with_no_resp command will fail 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:

```
// Example :: EvNotifyBuf.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

 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$,0x18EE,-1,-1,-1,-1,-1)
     //commit reports to GATT table - adRpt$ 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 BL652 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 BL652 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...
```

5.2.24 EVCONNPARAMREQ

This event is only thrown for a central role connection when a peripheral requests an update to the connection parameters via BleSetCurConnParams(). The user must turn manual parameter control to receive this message by using BleConnectConfig(8,1). In this case auto accept is disabled and full control is given to the user.

The event contains the following integer values:

nConnHandle	The handle of the connection where the peripheral is requesting a change.
nMinIntUs	The minimum acceptable connection interval in microseconds.
nMaxIntUs	The maximum acceptable connection interval in microseconds.
nSuprToutUs	The link supervision timeout for the connection in microseconds.
nSlaveLatency	The number of connection interval polls that may be ignored.

Example:

```
//Example :: EvConnParamReq.sb

// In order to get the expected output, this application should be run against
// a peripheral device. The peripheral device should request new connection
// parameters upon connection, which in turn will trigger EVCONNPARAMREQ on
// this device.

// This is the target Bluetooth device to connect to, 7 bytes in hex
#define BTAddr "000016A4B75202"

// BLE EVENT MSG IDs
#define BLE_EVBLEMSGID_CONNECT                          0 // msgCtx = connection handle
#define BLE_EVBLEMSGID_DISCONNECT                       1 // msgCtx = connection handle
#define BLE_EVBLEMSGID_CONN_PARMS_UPDATE                14  //nCtx = connection handle
#define BLE_EVBLEMSGID_CONN_PARMS_UPDATE_FAIL           15  //nCtx = connection handle

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 BLE_EVBLEMSGID_CONNECT
            PRINT "\nBLE Connection ";integer.h' nCtx;"\n"
        CASE BLE_EVBLEMSGID_DISCONNECT
```

```
            PRINT "\nDisconnected ";nCtx;"\n"
        CASE BLE_EVBLEMSGID_CONN_PARMS_UPDATE
            // The connection parameter has been updated. Read connection parameters
            dim intrvl,sprvto,slat
            rc= BleGetCurConnParms(nCtx,intrvl,sprvto,slat)
            print "--- Param Updated \n"
            print "- interval:";intrvl;" supervision timeout:";sprvto;" latency:";slat;"\n"
        CASE BLE_EVBLEMSGID_CONN_PARMS_UPDATE_FAIL
            print "--- Param Update Failed\n"
            print "- interval:";intrvl;" supervision timeout:";sprvto;" latency:";slat;"\n"
        CASE ELSE
            PRINT "\nUnknown Ble Msg"
    ENDSELECT
ENDFUNC 1

//==============================================================================
// This handler is called when peripheral requests new parameter
//==============================================================================
function HandlerParamReq(BYVAL hConn AS INTEGER, BYVAL intrvlmin AS INTEGER, BYVAL intrvlmax
AS INTEGER, BYVAL sprvto AS INTEGER, BYVAL slat AS INTEGER)

    print "--- Param Request \n"
    print "- intervalmin:";intrvlmin;" intervalmax:";intrvlmax;" supervision
timeout:";sprvto;" latency:";slat;"\n"
    // Accept the peripheral's request by changing the connection's conn parameters
    rc = BleSetCurConnParms(hConn, intrvlmin, intrvlmax, sprvto, slat)

endfunc 1

//==============================================================================
// Program starts here
//==============================================================================
// Disable auto accept so that we get an event when peripheral requests
// new connection parameteres. Set to 0 to re-enable auto accept
rc = BleConnectConfig(8,1)
// Connect to peripheral
DIM addr$ : addr$ = BTAddr
addr$ = StrDehexize$(addr$)
rc = BleConnect(addr$, 5000, 7500, 7700, 500000)

//------------------------------------------------------------------------------
// Enable synchronous event handlers
//------------------------------------------------------------------------------
ONEVENT  EVBLEMSG           CALL HndlrBleMsg
ONEVENT  EVCONNPARAMREQ     CALL HandlerParamReq

WAITEVENT
```

Expected Output:

```
BLE Connection 0001FF00
--- Param Request
- intervalmin:45000 intervalmax:50000 supervision timeout:6000000 latency:0
--- Param Updated
- interval:50000 supervision timeout:6000000 latency:0
```

5.3 Miscellaneous Functions

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

5.3.1 BleTxPowerSet

FUNCTION

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

Although this function can accept any value, the actual transmit power is determined by the internal power table which supports -40, -20, -16, -12, -8, -4, 0, and 4 dBm. When a value is set, the highest transmit power that is less than or equal to the desired power is used. SYSINFO(2008) and AT I 2008 can be used to return the power level set.

BLETXPOWERSET (nTxPower)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating 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 the radios internal power table.

Example:

```
// Example :: BleTxPowerSet.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


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= -40
Tx power : desired= -45    actual= -40
```

```
Tx power : desired= -1000  actual= -40
```

5.3.2 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 Whisper Mode Pairing. The actual value is clipped to the transmit power for normal operation which is set using BleTxPowerSet() function.

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

Although this function can accept any value, the actual transmit power is determined by the internal power table which supports -40, -20, -16, -12, -8, -4, 0, and 4 dBm, when a value is set the highest transmit power that is less than or equal to the desired power is used. SYSINFO(2008) and AT I 2008 will return the power level set, and does not reflect the transmit power level of the radio itself.

BLETXPWRWHILEPAIRING (nTxPower)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating 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 while the pairing is in progress and normal power is resumed when the transaction is complete. The actual value is determined by the radios internal power table. Please note that the tx power will be reduced to nTxPower for ALL connections, even on connections that there is no pairing in progress.

Example:

```
// Example :: BleTxPwrWhilePairing.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


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:

```
rc = 0
Tx power while pairing: desired= 1000  actual= 10
Tx power while pairing: desired= 8    actual= 8
Tx power while pairing: desired= 2    actual= 2
Tx power while pairing: desired= -10  actual= -10
Tx power while pairing: desired= -25  actual= -20
Tx power while pairing: desired= -45  actual= -20
Tx power while pairing: desired= -1000  actual= -20
```

5.3.3 BleConfigDcDc

SUBROUTINE

This routine is used to configure the DC to DC converter to one of 2 states: ENABLED or DISABLED.

BLECONFIGDCDC (nNewState)

Returns	None
Arguments	
nNewState	**byVal *nNewState* AS INTEGER.** Configure the internal DC to DC converter as follows: 0 – Disabled All other values – Enabled

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

5.4 Advertising Functions

This section describes all the advertising-related routines.

An advertisement consists of a packet of information with a header identifying it as one of four types along with 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 indicates the number of octets that follow it that belong to that record.
- Field 2 – One octet in length and is a tag value which identifies the type of payload that starts at the next octet. Hence the payload data is 'length – 1'.
- Field 3 – A special NULL AD record that consists of one field (the length field) when it contains only the 00 value.

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

Refer to the *Supplement to the Bluetooth Core Specification, Version 1, Part A* which contains the latest list of all AD records. You must register as at least an adopter, which is free, to gain access to this information. It is available at https://www.Bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=245130

5.4.1 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.

In order to advertise over 2MPHY, BleAdvertConfig() should be called beforehand to set the secondary advertising channel to 2MPHY. See BleAdvertConfig() for more details.

Extended advertising types (6-11) are only supported as experimental features in this release.

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

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation. If a 0x6A01 resultcode is received, it implies a 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. nAdvertTimeout imust be <= 1280ms because nAdvInterval is ignored and will advertise at a rate of every 3.75milliseconds which means this type of advert is not power efficient and will impact battery life. See ADV_DIRECT_LOW_DUTYCYCLE_IND for a more power efficient alternative.
2	ADV_SCAN_IND	Invites scan request for more advert data
3	ADV_NONCONN_IND	Does not accept connections/active scans
4	ADV_DIRECT_LOW_DUTYCYCLE_IND	Invites connection from addressed device. No limit on nAdvertTimeout as the advertising interval is as per nAdvInterval, like a normal advert but with the payload being the target address. See ADV_DIRECT_IND for an alternative.
5	Unused	
6	ADV_EXT_CONN_NONSCAN	Invites connection requests over the secondary

<table>
<tr><td></td><td>
<table>
<tr><td></td><td></td><td>advertising channel. This advertising type can be used for CODED PHY connections.</td></tr>
<tr><td>7</td><td>ADV_EXT_CONN_NONSCAN_DIRECTED</td><td>Invites connection from addressed devices over the secondary advertising channel. This advertising type can be used for CODED PHY connections.</td></tr>
<tr><td>8</td><td>ADV_EXT_NONCONN_SCAN</td><td>Invites scan requests over the secondary advertising channel.</td></tr>
<tr><td>9</td><td>ADV_EXT_NONCONN_SCAN_DIRECTED</td><td>Invites scan requests from addressed devices over the secondary advertising channel.</td></tr>
<tr><td>10</td><td>ADV_EXT_NONCONN_NONSCAN</td><td>Undirected nonconnecatable nonscannable advertising using extended advertising packets.</td></tr>
<tr><td>11</td><td>ADV_EXT_NONCONN_NONSCAN_DIRECTED</td><td>Directed nonconnecatable nonscannable advertising using extended advertising packets.</td></tr>
</table>
</td></tr>
<tr><td>peerAddr$</td><td>
byRef peerAddr$ 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 (for example: \00\11\22\33\44\55\66) where the first octet is the address type and the rest of the six octets is the usual Bluetooth address in big endian format (so the most significant octet of the address is at offset 1), whether public or random.
<table>
<tr><td>0</td><td>Public</td></tr>
<tr><td>1</td><td>Random Static</td></tr>
<tr><td>2</td><td>Random Private Resolvable</td></tr>
<tr><td>3</td><td>Random Private Non-Resolvable</td></tr>
</table>
All other values are illegal.
</td></tr>
<tr><td>nAdvInterval</td><td>
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.

Valid range is between 20 and 10240 milliseconds.
</td></tr>
<tr><td>nAdvTimeout</td><td>
byVal nAdvTimeout AS INTEGER.

The time after which the module stops advertising (in milliseconds). The range of this value is between 0 and 16383000 milliseconds and is rounded up to the nearest 1 seconds (1000ms).

A value of 0 means disable the timeout, but note that if limited advert modes was specified in BleAdvRptInit() then this function fails. 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 mistakenly set this to e.g. 100ms.
</td></tr>
<tr><td>nFilterPolicy</td><td>
byVal nFilterPolicy AS INTEGER.

Specifies the filter policy for the whitelist as follows:
<table>
<tr><td>0</td><td>Disable whitelist</td></tr>
<tr><td>1</td><td>Filter Policy – Filter scan request; allow connection request from any</td></tr>
<tr><td>2</td><td>Filter Policy – Filter connection request; allow scan request from any</td></tr>
<tr><td>3</td><td>Filter scan request and connection request</td></tr>
<tr><td>hhh</td><td>A whitelist handle (for more details see section “Whitelist Management Functions)</td></tr>
</table>
If the filter policy is not 0, but 1,2 or 3 the whitelist is enabled and filled with first 8 addresses and 8 identity resolving keys of devices in the trusted device database. Given the database can accommodate more devices please note that if more than 8 devices exist than a partial whitelist is activated.
</td></tr>
</table>

	To cater for that limitation, a whitelist can be manually created using the API described in the section "Whitelist Management Functions" and the handle returned from a manually created list can be supplied for this parameter

Example:

```
// Example :: BleAdvertStart.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

 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
BL652'"
 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 BL652'

Advert stopped via timeout
Exiting...
```

5.4.2 BleAdvertStop

FUNCTION

This function causes the BLE module to stop advertising.

BLEADVERTSTOP ()

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments	None

Example:

```
// Example :: BleAdvertStop.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

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(11,1,2)
 rc = GpioBindEvent(0,11,1)

 ONEVENT  EVBLE_ADV_TIMEOUT  CALL HndlrBlrAdvTimOut
 ONEVENT  EVGPIOCHAN0        CALL Btn0Press


 WAITEVENT
```

Expected Output:

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

Advertising Stopped
Exiting...
```

5.4.3 BleAdvertConfig

FUNCTION

This function is used to modify the default parameters that are used when initiating an advertise operation using BleAdvertStart().

The following lists the default values for the parameters:

Advert Channel Mask	Bit field detailing the channels to advertise on.

Note: Set channel mask Bit 0 to enable advert channel 0, Bit 1 to enable advert channel 1, and Bit 2 to enable advert channel 2.

BLEADVERTCONFIG (configID, configValue)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
configID	**byVal *configID* AS INTEGER.** This identifies the value to update as follows: 0 – Unused 1 – Unused 2 – Unused 3 – Advert Channel Mask. Set to 0 to enable channel 37, bit 1 to enable channel 38, and bit 2 to enable channel 39 4 – Primary PHY to advertise on. Possible values are:- 1 – 1MPHY All other values are invalide 5 – Secondary PHY to advertise on. Possible values are:- 1 – 1MPHY 2 – 2MPHY All other values are invalid For all other configID values the function returns an error.
configValue	**byVal *configValue* AS INTEGER.** This contains the new value to set in the parameters indentified by configID.

5.4.4 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. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
advRpt$	**byRef *advRpt$* AS STRING.** This contains 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.

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 zero (0) then the device name is not included.

Example:

```
// Example :: BleAdvRptInit.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 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
```

5.4.5 BleScanRptInit

FUNCTION

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

This report is for use with SCAN_RESPONSE packets.

BLESCANRPTINIT (scanRpt)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
scanRpt	**byRef *scanRpt* ASSTRING.** This contains a scan report.

Example:

```
// Example :: BleScanRptInit.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 DIM scnRpt$ : scnRpt$=""
```

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

Expected Output:

```
Scan report initialised
```

5.4.6 BleAdvRptGetSpace

FUNCTION

This function returns the free space in the advert advRpt$.

BLEADVRPTGETSPACE(advRpt)

Returns	INTEGER, the free space in bytes.
Arguments:	
advRpt$	**byRef *advRpt$ AS* STRING.** This contains an advert/scan report.

Example:

```
// Example :: BleAdvRptGetSpace.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

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
```

5.4.7 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 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. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
AdvRpt$	**byRef *AdvRpt AS* STRING**. The advert report onto which the 16-bit uuids AD record is added.
nUuid1	***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.
nUuid2	***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.
nUuid3	***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.
nUuid4	***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.
nUuid5	***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.
nUuid6	***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 and then all further UUID arguments will also be ignored.

Example:

```
// Example :: BleAdvAddUuid16.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 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
```

5.4.8 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. The most typical value is 0x0000, indicating 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 a function such as BleHandleUuid128() or some other function which returns one.

Example:

```
// Example :: BleAdvAddUuid128.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


DIM uuid$ , hUuidCustom
DIM tx$,scRpt$,adRpt$,addr$, hndl
 scRpt$=""
 PRINT BleScanRptInit(scRpt$)

//create a custom uuid for my ble widget
 uuid$ = "ced9d91366924a1287d56f2764762b2a"
 uuid$ = StrDehexize$(uuid$)
 hUuidCustom = BleHandleUuid128(uuid$)

 //Advertise the 128 bit uuid in a scan report
 PRINT BleAdvRptAddUuid128(scRpt$, hUuidCustom)
 adRpt$=""
 PRINT BleAdvRptsCommit(adRpt$,scRpt$)
 addr$=""  //because we are not doing a DIRECT advert
 PRINT BleAdvertStart(0,addr$,20,30000,0)
```

Expected Output:

```
00000
```

5.4.9 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. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
AdvRpt	**byRef *AdvRpt AS* STRING**. The advert report onto which the AD record is to be appended.
nTag	***byVal nTag AS INTEGER***

	nTag should be in the range 0 to FF and is the TAG field for the record.
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.

Example:

```
// Example :: BleAdvRptAppendAD.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

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
```

5.4.10 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. Both strings can be empty. In that case, 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. The most typical value is 0x0000, indicating 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.

Tip: You can commit advert reports to update your advertisement data **while advertising**.

Example:

```
// Example :: BleAdvRptsCommit.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples
```

```
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
```

5.5 Scanning Functions

When a peripheral advertises, the advert packet consists type of advert, address, RSSI, and some user data information.

A central role device enters scanning mode to receive these advert packets from any device that is advertising.

For each advert that is received, the data is cached in a ring buffer, if space exists, and the EVBLE_ADV_REPORT event is thrown to the *smart*BASIC application so that it can invoke the function BleScanGetAdvReport() to read it.

The scan procedure ends when it times out (timeout parameter is supplied when scanning is initiated) or when explicity instructed to abort or stop.

Note: While scanning for a long period of time, it is possible that a peripheral device is advertising for a connection to it using the ADV_DIRECT_IND advert type. When this happens, it is good practice for the central device to stop scanning and initiate the connection. To cater for this specific scenario, which would normally require the central device to look out for that advert type and the self address, the EVBLE_FAST_PAGED event is thrown to the application. This means that all the user app needs to do is to install a handler for that event which stops the scan procedure and immediately starts a connection procedure.

For more information about adverts see the section Advertising Functions.

5.5.1 BleScanStart

FUNCTION

This function is used to start a scan for adverts which may result in at least one of the following events being thrown:

EVBLE_SCAN_TIMEOUT	End of scanning
EVBLE_ADV_REPORT	Advert report received
EVBLE_FAST_PAGED	Peripheral inviting a connection to this module

- **EVBLE_ADV_REPORT** – Received when an advert has been successfully cached in a ring buffer. The handler should call the function BleScanGetAdvReport() repeatedly to read all the advert reports that have been cached until the cache is empty, otherwise there is a risk that advert reports will be discarded. The output parameter nDiscarded returns the number of discarded reports, if any.

- **EVBLE_FAST_PAGED** – Received when a peripheral has sent an advert with the address of this module. The handler should stop scanning using BleScanStop() and then initiate a connection using BleConnect().

There are three parameters used when initiating a scan that are configurable using BleScanConfig(), otherwise default values are used:

- Scan Interval – Specify the duty cycle for listening for adverts. Default value: 80 milliseconds.
- Scan Window – Specify the duty cycle for listening for adverts. Default value: 40 milliseconds.
- Scan Type – Default scan type: Active

  Active scanning means that for each advert received (if it is ADV_IND or ADV_DISCOVER_IND) a SCAN_REQ is sent to the advertising device so that the data in the scan response can be appended to the data that has already been received for the advert.

The values for these default parameters can be changed prior to invoking this function by calling the function BleScanConfig() appropriately.

Note: Be aware that scanning is a memory intensive operation and so heap memory is used to manage a cache. If the heap is fragmented, it is likely this function will fail with an appropriate resultcode returned. If that happens, call reset() and then attempt the scan start again. The memory that is allocated to manage this scan process is NOT released when the scanning times out. To force release of that memory, we recommend that you start the scan and then immediately call BleScanStop().

Connections may not be established during a scan operation. If a continued scan is required, stop the scan or let it timeout, connect, then restart the scan.

BLESCANSTART (scanTimeoutMs, nFilterHandle)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
scanTimeoutMs	**byVAL *scanTimeoutMs* AS INTEGER.** The length of time in milliseconds the scan for adverts lasts. If the timer times out then the event EVBLE_SCAN_TIMEOUT is thrown to the *smart*BASIC application. Valid range is 0 to 65535000 milliseconds (about 18 hours). If 0 is supplied, a timer is not started and scanning can only be stopped by calling either BleScanAbort() or Ble ScanStop().
nFilterHandle	**byVAL nFilterHandle AS INTEGER** This must be zero (0) to specify no filtering of adverts. **Note:** In this current firmware version, this is only a placeholder.

Example:

```
// Example :: BleScanStart.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

DIM rc

 '//Scan for 20 seconds with no filtering
 rc = BleScanStart(20000, 0)

 IF rc==0 THEN
     PRINT "\nScanning"
 ELSE
     PRINT "\nError: "; INTEGER.H'rc
 ENDIF
```

```
'//This handler will be called when scanning times out
FUNCTION HndlrScanTO()
    PRINT "\nScan timeout"
ENDFUNC 0

ONEVENT EVBLE_SCAN_TIMEOUT CALL HndlrScanTO

WAITEVENT
```

Expected Output:

```
Scanning
Scan timeout
```

5.5.2 BleScanAbort

FUNCTION

This function is used to cancel an ongoing scan for adverts which has not timed out. It takes no parameters as there can only be one scan in progress.

Use the value returned by SYSINFO(2016) to determine if there is an ongoing scan operation in progress. The value is a bit mask where:

- **bit 0** is set if advertising is in progress
- **bit 1** is set if there is already a connection in a peripheral role
- **bit 2** is set if there is a current ongoing connection attempt
- **bit 3** is set when scanning
- **bit 4** is set if there is already a connection to a peripheral

There is also BleScanStop() which cancels an ongoing scan. The difference is that, by calling BleScanAbort(), the memory that was allocated from heap by BleScanStart() is not released back to the heap. The scan manager retains it for the next scan operation.

BLESCANABORT ()

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments	None

Example:

```
// Example :: BleScanAbort.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples
DIM rc, startTick


'//Scan for 20 seconds with no filtering
rc = BleScanStart(20000, 0)


IF rc==0 THEN
    PRINT "\nScanning"
ELSE
    PRINT "\nError: "; INTEGER.H'rc
ENDIF
```

```
'//Wait 2 seconds before aborting scan
startTick = GetTickCount()
WHILE GetTickSince(startTick) < 2000
ENDWHILE

'//If scan in progress, abort
IF SysInfo(2016) == 0x08 THEN
    PRINT "\nAborting scan"
    rc = BleScanAbort()
    IF SysInfo(2016) == 0 THEN
        PRINT "\nScan aborted"
    ENDIF
ENDIF
```

Expected Output:

```
Scanning
Aborting scan
Scan aborted
```

5.5.3 BleScanStop

FUNCTION

This function is used to cancel an ongoing scan for adverts which has not timed out. It takes no parameters, as there can only be one scan in progress.

Use the value returned by SYSINFO(2016) to determine if there is an ongoing scan operation in progress. The value is a bit mask where:

- **bit 0** is set if advertising is in progress
- **bit 1** is set if there is already a connection in a peripheral role
- **bit 2** is set if there is a current ongoing connection attempt
- **bit 3** is set when scanning
- **bit 4** is set if there is already a connection to a peripheral

There is also BleScanAbort() which cancels an ongoing scan. The difference is that, by calling BleScanStop(), the memory that was allocated from heap by BleScanStart() is released back to the heap. The scan manager must reallocate the memory if BleScanStart() is called again.

BLESCANSTOP ()

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments	None

Example:

```
// Example :: BleScanStop.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples
```

```
DIM rc, startTick

'//Scan for 20 seconds with no filtering
rc = BleScanStart(20000, 0)

IF rc==0 THEN
    PRINT "\nScanning"
ELSE
    PRINT "\nError: "; INTEGER.H'rc
ENDIF

'//Wait 2 seconds before aborting scan
startTick = GetTickCount()
WHILE GetTickSince(startTick) < 2000
ENDWHILE

'//If scan in progress, abort
IF SysInfo(2016) == 0x08 THEN
    PRINT "\nStop scanning. Freeing up allocated memory"
    rc = BleScanStop()
    IF SysInfo(2016) == 0 THEN
        PRINT "\nScan stopped"
    ENDIF
ENDIF
```

Expected Output:

```
Scanning
Stop scanning. Freeing up allocated memory
Scan stopped
```

5.5.4 BleScanFlush

FUNCTION

This function is used to flush the ring buffer which stores incoming adverts which are later read.

BLESCANFLUSH ()

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments	None

Example:

```
// Example :: BleScanFlush.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples
```

```
DIM rc, startTick

'//Scan for 20 seconds with no filtering
rc = BleScanStart(20000, 0)

IF rc==0 THEN
    PRINT "\nScanning"
ELSE
    PRINT "\nError: "; INTEGER.H'rc
ENDIF

'//Wait 2 seconds before aborting scan
startTick = GetTickCount()
WHILE GetTickSince(startTick) < 2000
ENDWHILE

'//If scan in progress, abort
IF SysInfo(2016) == 0x08 THEN
    PRINT "\nAborting scan"
    rc = BleScanAbort()
    IF SysInfo(2016) == 0 THEN
        PRINT "\nScan aborted"
    ENDIF

    '//Free up memory
    rc = BleScanFlush()
    IF (rc == 0) THEN
       PRINT "\nScan results flushed."
    ENDIF
ENDIF
```

Expected Output:

```
Scanning
Aborting scan
Scan aborted
Scan results flushed.
```

5.5.5 BleScanConfig

FUNCTION

This function is used to modify the default parameters that are used when initiating a scan operation using BleScanStart().

The following are the default values for the parameters:

Scan Interval	80 milliseconds
Scan Window	40 milliseconds
Scan Type (Active/Passive)	Active
Minimum Reports in Cache	4

Note: The default Scan Window and Interval give a 50% duty cycle. The 50% duty cycle attempts to ensure that connection events for existing connections are missed as infrequently as possible.

BLESCANCONFIG (configID, configValue)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
configID	**byVal *configID* AS INTEGER.** This identifies the value to update as follows: 0 – Scan Interval in milliseconds (range 0..10240) 1 – Scan Window in milliseconds (range 0..10240) 2 – Scan Type (0=Passive, 1=Active) 3 – Advert Report Cache SIze For all other configID values the function returns an error.
configValue	**byVal *configValue* AS INTEGER.** This contains the new value to set in the parameters indentified by configID.

Example:

```
// Example :: BleScanConfig.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 DIM rc, startTick

 PRINT "\nScan Interval: "; SysInfo(2150)    //get current scan interval
 PRINT "\nScan Window: "; SysInfo(2151)      //get current scan window
 PRINT "\nScan Type: ";
 IF SysInfo(2152)==0 THEN                   //get current scan type
     PRINT "Passive"
 ELSE
     PRINT "Active"
 ENDIF
 PRINT "\nReport Cache Size: "; SysInfo(2153) //get report cache size

 PRINT "\n\nSetting new parameters..."
 rc = BleScanConfig(0, 100)                //set scan interval to 100
 rc = BleScanConfig(1, 50)                 //set scan window to 50
 rc = BleScanConfig(2, 0)                  //set scan type to passive
 rc = BleScanConfig(3, 3)                  //set report cache size
```

```
PRINT "\n\n--- New Parameters:"
PRINT "\nScan Interval: "; SysInfo(2150)     //get current scan interval
PRINT "\nScan Window: "; SysInfo(2151)       //get current scan window
PRINT "\nScan Type: ";
IF SysInfo(2152)==0 THEN                    //get current scan type
    PRINT "Passive"
ELSE
    PRINT "Active"
ENDIF
PRINT "\nReport Cache Size: "; SysInfo(2153) //get report cache size
```

Expected Output:

```
Scan Interval: 80
Scan Window: 40
Scan Type: Active
Report Cache Size: 4

Setting new parameters..

--- New Parameters:
Scan Interval: 100
Scan Window: 50
Scan Type: Passive
Report Cache Size: 3
```

5.5.6 BleScanGetAdvReport

FUNCTION

When a scan is in progress after having called BleScanStart() for each advert report, the information is cached in a queue buffer and an EVBLE_ADV_REPORT event is thrown to the *smart*BASIC application.

This function is used by the *smart*BASIC application to extract it from the queue for further processing in the handler for the EVBLE_ADV_REPORT event.

The retrieved information consists of the address of the peripheral that sent the advert, the data payload, the number of adverts (all, not just from that peripheral) that have been discarded since the last time this function was called and the RSSI value for that packet.

Note: The RSSI can be used to determine the closest device. However, due to fading and reflections, it is possible that a device further away could result in a higher RSSI value.

BLESCANGETADVREPORT (periphAddr$, advData$, nDiscarded, nRssi)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
periphAddr$	**byREF *periphAddr$* AS STRING** On return, this parameter is updated with the address of the peripheral that sent the advert.

advData$	**byREF advData$ AS STRING** On return, this parameter is updated with the data payload of the advert which consists of multiple AD elements.
nDiscarded	**byREF nDiscarded AS INTEGER** On return, this parameter is updated with the number of adverts that were discarded because there was no space in the internal queue.
nRssi	**byREF nRssi AS INTEGER** On return, this parameter is updated with the RSSI as reported by the stack for that advert. **Note:** This is NOT a value that is sent by the peripheral but a value that is calculated by the receiver in this module.

Note: This code snippet was tested with another BL652 running the iBeacon app (see in smartBASIC_Sample_Apps folder) on peripheral firmware.

Example:

```
// Example :: BleScanGetAdvReport.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples
 DIM rc

 '//Scan for 20 seconds with no filtering
 rc = BleScanStart(5000, 0)

 IF rc==0 THEN
     PRINT "\nScanning"
 ELSE
     PRINT "\nError: "; INTEGER.H'rc
 ENDIF

 '//This handler will be called when scanning times out
 FUNCTION HndlrScanTO()
     PRINT "\nScan timeout"
 ENDFUNC 0

 '//This handler will be called when an advert is received
 FUNCTION HndlrAdvRpt()
     DIM periphAddr$, advData$, nDiscarded, nRssi

     '//Read all cached advert reports
     rc=BleScanGetAdvReport(periphAddr$, advData$, nDiscarded, nRssi)
     WHILE (rc == 0)
         PRINT "\n\nPeer Address: "; StrHexize$(periphAddr$)
```

```
        PRINT "\nAdvert Data: ";StrHexize$(advData$)
        PRINT "\nNo. Discarded Adverts: ";nDiscarded
        PRINT "\nRSSI: ";nRssi
        rc=BleScanGetAdvReport(periphAddr$, advData$, nDiscarded, nRssi)
    ENDWHILE

    PRINT "\n\n --- No more adverts in cache"
ENDFUNC 1



ONEVENT EVBLE_SCAN_TIMEOUT CALL HndlrScanTO
ONEVENT EVBLE_ADV_REPORT   CALL HndlrAdvRpt

WAITEVENT
```

Expected Output:

```
Scanning

Peer Address: 01D8CFCF14498D
Advert Data: 0201061AFF4C000215E2C56DB5DFFB48D2B060D0F5A71096E012345678C4
No. Discarded Adverts: 0
RSSI: -97

Peer Address: 01D8CFCF14498D
Advert Data: 0201061AFF4C000215E2C56DB5DFFB48D2B060D0F5A71096E012345678C4
No. Discarded Adverts: 0
RSSI: -97

 --- No more adverts in cache

Peer Address: 01D8CFCF14498D
Advert Data: 0201061AFF4C000215E2C56DB5DFFB48D2B060D0F5A71096E012345678C4
No. Discarded Adverts: 0
RSSI: -92

Peer Address: 01D8CFCF14498D
Advert Data: 0201061AFF4C000215E2C56DB5DFFB48D2B060D0F5A71096E012345678C4
No. Discarded Adverts: 0
RSSI: -92

 --- No more adverts in cache
Scan timeout
```

5.5.7 BleScanGetAdvReportEx

When a scan is in progress after having called BleScanStart() for each advert report, the information is cached in a queue buffer and an EVBLE_ADV_REPORT event is thrown to the *smart*BASIC application.

This function is used by the *smart*BASIC application to extract it from the queue for further processing in the handler for the EVBLE_ADV_REPORT event.

The retrieved information consists of the address of the peripheral that sent the advert, the data payload, the number of adverts (all, not just from that peripheral) that have been discarded since the last time this function was called and the RSSI value for that packet, in addition to the advert type and the channel number on which the advert was received.

BLESCANGETADVREPORTEX (nAdvertType, periphAddr$, advData$, nDiscarded, nRssi, nChannel)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
nAdvertType	**byREF *nAdvertType* AS STRING** On return, this parameter will contain the type of the advert that was read. Possible values are as follows:- 0 — ADV_IND — Invites connection requests 1 — ADV_DIRECT_IND — Invites connection from addressed device 2 — ADV_SCAN_IND — Invites scan request for more advert data 3 — ADV_NONCONN_IND — Does not accept connections/active scans
periphAddr$	**byREF *periphAddr$* AS STRING** On return, this parameter is updated with the address of the peripheral that sent the advert.
advData$	**byREF advData *$* AS STRING** On return, this parameter is updated with the data payload of the advert which consists of multiple AD elements.
nDiscarded	**byREF nDiscarded AS INTEGER** On return, this parameter is updated with the number of adverts that were discarded because there was no space in the internal queue.
nRssi	**byREF nRssi AS INTEGER** On return, this parameter is updated with the RSSI as reported by the stack for that advert. **Note:** This is NOT a value that is sent by the peripheral but a value that is calculated by the receiver in this module.
nChannel	**byREF nChannel AS INTEGER** On return, this parameter is set to the channel on which the advert has arrived. Valid values are 0, 1, or 2.

```
//Example :: BleScanGetAdvReportEx.sb

DIM rc

'//Scan for 5 seconds with no filtering
rc = BleScanStart(5000, 0)

IF rc==0 THEN
    PRINT "\nScanning"
ELSE
    PRINT "\nError: "; INTEGER.H'rc
ENDIF

'//This handler will be called when scanning times out
FUNCTION HndlrScanTO()
    PRINT "\nScan timeout"
ENDFUNC 0

'//This handler will be called when an advert is received
FUNCTION HndlrAdvRpt()
    DIM nAdvType, periphAddr$, advData$, nDiscarded, nRssi, nChannel
```

```
    '//Read all cached advert reports
    rc=BleScanGetAdvReportEx(nAdvType, periphAddr$, advData$, nDiscarded, nRssi, nChannel)
    WHILE (rc == 0)
       PRINT "\n\nAdvert Type: "; nAdvType
       PRINT "\nPeer Address: "; StrHexize$(periphAddr$)
       PRINT "\nAdvert Data: ";StrHexize$(advData$)
       PRINT "\nNo. Discarded Adverts: ";nDiscarded
       PRINT "\nRSSI: ";nRssi
       PRINT "\nChannel: ";nChannel
       rc=BleScanGetAdvReportEx(nAdvType, periphAddr$, advData$, nDiscarded, nRssi, nChannel)
    ENDWHILE

    PRINT "\n\n --- No more adverts in cache"
ENDFUNC 1


ONEVENT EVBLE_SCAN_TIMEOUT CALL HndlrScanTO
ONEVENT EVBLE_ADV_REPORT   CALL HndlrAdvRpt

WAITEVENT
```

```
Scanning

Advert Type: 2
Peer Address: 01CDBD40C5A79A
Advert Data: 0201061AFF4C000215E2C56DB5DFFB48D2B060D0F5A71096E012345678C40409526F6E
No. Discarded Adverts: 0
RSSI: -81
Channel: 1

 --- No more adverts in cache
Scan timeout
00
```

5.5.8 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 then it is treated as not existing. For example, it is 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. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
nIndex	**byVAL *nIndex AS* INTEGER** This is a zero-based index of the AD element that is copied into the output data parameter ADval$.
rptData$	**byREF *rptData$ AS* STRING**. This parameter is a string that contains concatenated AD elements which were either constructed for an outgoing advert or were received in a scan.
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.

Example:

```
// Example :: BleGetADbyIndex.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 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
```

5.5.9 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 is treated as nonexistent. For example, it is 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. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
rptData$	**byREF *rptData$ AS* STRING**. This parameter is a string that contains concatenated AD elements which were either constructed for an outgoing advert or were received in a scan.
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.

Example:

```
// Example :: BleGetADbyTag.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 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
```

5.5.10 BleScanGetPagerAddr

FUNCTION

When a scan is in progress after calling BleScanStart(), an EVBLE_FAST_PAGED event is thrown whenever an ADV_DIRECT_IND advert is received with the address of this module, requesting a connection to it.

This function returns the address of the peripheral requesting a connection and the RSSI. It should be used in the handler of the EVBLE_FAST_PAGED event to get the peripheral's address. Scanning should then be stopped using either BleScanAbort() or BleScanStop(). You can then use the address supplied by this function to connect to the peripheral using BleConnect() if that is the desired use case. The Bluetooth specification does NOT mandate a connection.

BLESCANGETPAGERADDR (periphAddr$, nRssi)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
periphAddr$	**byREF *periphAddr$* AS STRING** On return, this parameter is updated with the address of the peripheral that sent the advert.
nRssi	**byREF nRssi AS INTEGER** On return, this parameter is updated with the RSSI as reported by the stack for that advert. **Note:** This is NOT a value that is sent by the peripheral but a value that is calculated by the receiver in this module.

Example:

```
// Example :: BleScanGetPagerAddr.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 DIM rc

 '//Scan for 20 seconds with no filtering
 rc = BleScanStart(10000, 0)

 IF rc==0 THEN
     PRINT "\nScanning"
 ELSE
     PRINT "\nError: "; INTEGER.H'rc
 ENDIF

 '//This handler will be called when scanning times out
 FUNCTION HndlrScanTO()
     PRINT "\nScan timeout"
 ENDFUNC 0

 '//This handler will be called when an advert is received requesting a connection to  this
module
FUNCTION HndlrFastPaged()
     DIM periphAddr$, nRssi
     rc = BleScanGetPagerAddr(periphAddr$, nRssi)
     PRINT "\nAdvert received from peripheral "; StrHexize$(periphAddr$); " with RSSI
";nRssi
     PRINT "\nrequesting a connection to this module"
     rc = BleScanStop()
```

```
ENDFUNC 0

ONEVENT EVBLE_SCAN_TIMEOUT CALL HndlrScanTO
ONEVENT EVBLE_FAST_PAGED   CALL HndlrFastPaged

WAITEVENT
```

Expected Output:

```
Scanning
Advert received from peripheral 01D8CFCF14498D with RSSI -96
requesting a connection to this module
```

5.6 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.

5.6.1 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

5.6.2 BleConnect

FUNCTION

This function is used to make a connection to a device in peripheral mode which is actively advertising.

Note: The peripheral device MUST be advertising with either ADV_IND or ADV_DIRECT_IND type of advert to be able to successfully connect.

In the case of multiple connections, it is recommended that this function is not called in quick succession so that the underlying stack is given time to complete the setup of the new connection before moving on to establish a new connection. Calling this function in quick succession may cause newly established connections to be dropped.

In order to perform connections over 2MPHY, BleConnectConfig() should be called beforehand to set the connection PHYs to 1MPHY or 2MPHY (3) and enable extended connection. See BleConnectConfig() for more details.

When the connection is complete, a EVBLEMSG message with msgId = 0 and context containing the handle are thrown to the *smart*BASIC runtime engine.

If the connection times out, then the event EVBLE_CONN_TIMEOUT is thrown to the *smart*BASIC application.

When a connection is attempted, there are other parameters that are used and the default values for those are assumed; for example, scan window, scan interval, and periodicity. The default values for those can be changed using the BleConnectConfig() function. At any time, the current settings can be obtained via the SYSINFO() command.

BLECONNECT (periphAddr$, connTimeoutMs, minConnIntUs, maxConnIntUs, nSuprToutUs)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
periphAddr$	**byRef *periphAddr$* AS STRING** The Bluetooth address of the device to connect to which MUST be properly formatted and is exactly seven bytes long.
connTimeoutMs	**byVal *connTimeoutMs* AS INTEGER.** The length of time in milliseconds that the connection attempt lasts. If the timer times out then the event EVBLE_CONN_TIMEOUT is thrown to the *smart*BASIC application.
minConnIntUs	**byVal *minConnIntUs* AS INTEGER.** The minimum connection interval in microseconds. Valid range is between 7500 and 4000000 microseconds.
maxConnIntUs	**byVal *maxConnIntUs* AS INTEGER.** The maximum connection interval in microseconds. Valid range is between 7500 and 4000000 microseconds
nSuprToutUs	**byVal *nSuprToutUs* AS INTEGER.** The link supervision timeout for the connection in microseconds.

Example:

```
// Example :: BleConnect.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 DIM rc, periphAddr$


 '//Scan indefinitely
 rc=BleScanStart(0, 0)


 IF rc==0 THEN
     PRINT "\nScanning"
 ELSE
     PRINT "\nError: "; INTEGER.H'rc
 ENDIF
```

```
'//This handler will be called when an advert is received
FUNCTION HndlrAdvRpt()
   DIM advData$, nDiscarded, nRssi

    '//Read an advert report and connect to the sender
    rc=BleScanGetAdvReport(periphAddr$, advData$, nDiscarded, nRssi)
    rc=BleScanStop()

    '//Connect to device with Bluetooth address obtained above with 5s connection timeout,
    '//20ms min connection interval, 75 max, 5 second supervision timeout.
    rc=BleConnect(periphAddr$, 5000, 20000, 75000, 5000000)
    IF rc==0 THEN
        PRINT "\n--- Connecting"
    ELSE
        PRINT "\nError: "; INTEGER.H'rc
    ENDIF
ENDFUNC 1

'//This handler will be called in the event of a connection timeout
FUNCTION HndlrConnTO()
    PRINT "\n--- Connection timeout"
    rc=BleScanStart(0, 0)
ENDFUNC 1

'//This handler will be called when there is a BLE message
FUNCTION HndlrBleMsg(nMsgId, nCtx)
    IF nMsgId == 0 THEN
        PRINT "\n--- Connected to device with Bluetooth address "; StrHexize$(periphAddr$)
        PRINT "\n--- Disconnecting now"
        rc=BleDisconnect(nCtx)
    ENDIF
ENDFUNC 1

'//This handler will be called when a disconnection happens
FUNCTION HndlrDiscon(nCtx, nRsn)
ENDFUNC 0


ONEVENT EVBLEMSG           CALL HndlrBleMsg
```

```
ONEVENT EVDISCON            CALL HndlrDiscon
ONEVENT EVBLE_ADV_REPORT    CALL HndlrAdvRpt
ONEVENT EVBLE_CONN_TIMEOUT CALL HndlrConnTO

WAITEVENT
```

Expected Output:

```
Scanning
--- Connecting
--- Connected to device with Bluetooth address 01D8CFCF14498D
--- Disconnecting now
```

5.6.3 BleConnectCancel

FUNCTION

This function is used to cancel an ongoing connection attempt which has not timed out. It takes no parameters as there can only be one attempt in progress.

Use the value returned by SYSINFO(2016) to determine if there is an ongoing scan operation in progress. The value is a bit mask where:

- **bit 0** is set if advertising is in progress
- **bit 1** is set if there is already a connection in a peripheral role
- **bit 2** is set if there is a current ongoing connection attempt
- **bit 3** is set when scanning
- **bit 4** is set if there is already a connection to a peripheral

BLECONNECTCANCEL ()

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments	None

Example:

```
// Example :: BleConnectCancel.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

DIM rc, periphAddr$

'//Scan indefinitely
rc=BleScanStart(0, 0)

IF rc==0 THEN
    PRINT "\nScanning"
ELSE
    PRINT "\nError: "; INTEGER.H'rc
ENDIF
```

```
'//This handler will be called when an advert is received
FUNCTION HndlrAdvRpt()
    DIM advData$, nDiscarded, nRssi

    '//Read an advert report and connect to the sender
    rc=BleScanGetAdvReport(periphAddr$, advData$, nDiscarded, nRssi)
    rc=BleScanStop()

    '//Wait until module stops scanning
    WHILE SysInfo(2016)==8
    ENDWHILE

    '//Connect to device with Bluetooth address obtained above with 5s connection timeout,
    '//20ms min connection interval, 75 max, 5 second supervision timeout.
    rc=BleConnect(periphAddr$, 5000, 20000, 75000, 5000000)
    IF rc==0 THEN
        PRINT "\n--- Connecting \nCancel"
    ELSE
        PRINT "\nError: "; INTEGER.H'rc
    ENDIF

    '//Cancel current connection attempt
    rc=BleConnectCancel()

    PRINT "\n--- Connection attempt cancelled"
ENDFUNC 0

ONEVENT EVBLE_ADV_REPORT   CALL HndlrAdvRpt

WAITEVENT
```

Expected Output:

```
Scanning
--- Connecting
Cancel
--- Connection attempt cancelled
```

5.6.4 BleConnectConfig

FUNCTION

This function is used to modify the default parameters that are used when attempting a connection using BleConnect(). At any time they can be read by adding the configID to 2100 and then passing that value to SYSINFO().

When connecting, the central device must scan for adverts and then, when the particular peer address is encountered, it can send the connection message to that peripheral.

Therefore, a connection attempt requires the underlying stack API to be supplied with a scan interval and scan window. In addition, when multiple connections are in place, the radio has to be shared as efficiently as possible; one potential scheme is to have all connection parmeters being integer multiples of a 'base' value. For the purpose of this documentation, this parameter is referred to as *multi-link connection interval periodicity*.

The following are the default settings for these parameters:

Multi-link Connection Interval Periodicity	20 milliseconds
Scan Interval	80 milliseconds
Scan Window	40 milliseconds
Slave Latency	0

Notes: The Scan Window and Interval are multiple integers of the periodicity (although not required to be). The scanning has a 50% duty cycle. The 50% duty cycle attempts to ensure that connection events for existing connections are missed as infrequently as possible.

The Scan Window and Interval are internally stored in units of 0.625 milliseconds slots so reading back via SYSINFO() does not accurately return the value you set.

BLECONNECTCONFIG (configID, configValue)

<table>
<tr><td>Returns</td><td colspan="2">INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</td></tr>
<tr><td colspan="3">Arguments:</td></tr>
<tr><td rowspan="10">configID</td><td colspan="2">byVal configID AS INTEGER.
The following are the values to update:</td></tr>
<tr><td>0</td><td>Scan interval in milliseconds (range 0..10240)</td></tr>
<tr><td>1</td><td>Scan Window in milliseconds (range 0..10240)</td></tr>
<tr><td>2</td><td>Slave Latency (0..1000)</td></tr>
<tr><td>5</td><td>Multi-Link Connection Interval Periodicity (20..200)</td></tr>
<tr><td>8</td><td>Turn manual control for connection parameter update. See EvConnParamReq for more details.</td></tr>
<tr><td>9</td><td>Action to take when a PHY change request is received from remote device as follows:-
0: Automatically ccept incoming PHY change request from remote device. This is the default operation.
1: Throw an event to the smartBASIC app to allow the user to accept or reject incoming PHY change request. The event thrown is EVBLE_PHY_REQUEST. See LE 2M PHY for more information.</td></tr>
<tr><td>10</td><td>BLE PHY to perform the connection on. Possible values are:-
1 - 1MPHY
3 – 1MPHY or 2MPHY
All other values are invalid</td></tr>
<tr><td>11</td><td>Extended Connection. Possible values are:-
0 - Connect to device sending out legacy adverts
1 - Connect to device sending out legacy or extended adverts</td></tr>
<tr><td colspan="2">For all other configID values, the function returns an error.</td></tr>
<tr><td>configValue</td><td colspan="2">byVal configValue AS INTEGER.
This contains the new value to set in the parameters indentified by configID.</td></tr>
</table>

Example:

```
// Example :: BleConnectConfig.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

 DIM rc, startTick

 SUB GetParms()
     //get default scan interval for connecting
     PRINT "\nConn Scan Interval: "; SysInfo(2100);"ms"
     //get default scan window for connecting
     PRINT "\nConn Scan Window: "; SysInfo(2101);"ms"
     //get default slave latency for connecting
     PRINT "\nConn slave latency: "; SysInfo(2102)
     //get current multi-link connection interval periodicity
     PRINT "\nML Conn Interval Periodicity: "; SysInfo(2105);"ms"
 ENDSUB

 PRINT "\n\n--- Current Parameters:"
 GetParms()

 PRINT "\n\nSetting new parameters..."
 rc = BleConnectConfig(0, 60)          //set scan interval to 60
 rc = BleConnectConfig(1, 13)          //set scan window to 13 (will round to 12)
 rc = BleConnectConfig(2, 3)           //set slave latency to 1
 rc = BleConnectConfig(5, 30)          //set ML connection interval periodicity to 30
 PRINT "\n"; integer.h'rc

 PRINT "\n\n--- New Parameters:"
 GetParms()
```

Expected Output:

```
--- Current Parameters:
Conn Scan Interval: 80ms
Conn Scan Window: 40ms
Conn slave latency: 0
ML Conn Interval Periodicity: 20ms

Setting new parameters...

--- New Parameters:
Conn Scan Interval: 60ms
Conn Scan Window: 12ms
Conn slave latency: 3
```

```
ML Conn Interval Periodicity: 30ms
```

5.6.5 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 is thrown to the *smart*BASIC runtime engine.

BLEDISCONNECT (nConnHandle)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation
Arguments:	
nConnHandle	**byVal *nConnHandle* AS INTEGER.** Specifies the handle of the connection that must be disconnected.

Example:

```
// Example :: BleDisconnect.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


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
```

5.6.6 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 are thrown to the *smart*BASIC runtime engine if it is 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. The most typical value is 0x0000, indicating 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 that granted the 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 50 msec, for example, 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.

Example:

```
// Example :: BleSetCurConnParms.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples
```

```
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 BL652"
ELSE
    PRINT "\n\nAdvertisement not successful"
ENDIF

WAITEVENT
```

Expected Output (Unsuccessful Negotiation):

```
Adverts Started

Make a connection to the BL652
 --- 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 BL652
 --- 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: The first set of parameters differ depending on your central device.

5.6.7 BleGetCurConnParms

FUNCTION

This function gets the current connection parameters for the connection identified by the connection handle. Given there are 3 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. The most typical value is 0x0000, indicating a successful operation.

Arguments:	
nConnHandle	**byVal *nConnHandle* AS INTEGER.** Specifies the handle of the connection to read the connection parameters of
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.** 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.

See previous example.

5.6.8 BleConnMngrUpdCfg

FUNCTION

This function is used to initialise the connection manager for slave/peripheral role.

BLECONNMNGRUPDCFG (nConnUpdateFirstDelay, nConnUpdateNextDelay, nConnUpdateMaxRetry)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
nConnUpdateFirstDelay	**byVal *nConnUpdateFirstDelay* AS INTEGER.** In milliseconds 100 to 32000
nConnUpdateNextDelay	***BYVAL nConnUpdateNextDelay AS INTEGER*** In milliseconds 100 to 32000
nConnUpdateMaxRetry	**BYVAL nConnUpdateMaxRetry AS INTEGER** In number of retries

Example:

```
dim rc
#define CONN_UPD_FIRST_DELAY 500
#define CONN_UPD_NEXT_DELAY 800
#define CONN_UPD_MAX_RETRY 800

rc=BleConnMngrUpdCfg(CONN_UPD_FIRST_DELAY, CONN_UPD_NEXT_DELAY, CONN_UPD_MAX_RETRY)
if rc == 0 then
    print "\nConnection manager successfully initialised"
else
    print "\nError: ";integer.h'rc
endif
```

Expected Output:

```
Connection manager successfully initialised
```

5.6.9 BleGetConnHandleFromAddr

FUNCTION

This function is used to get the connection handle from a specified Bluetooth address.

BLEGETCONNHANDLEFROMADDR (BtAddrBE$, nConnHandle)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
BtAddrBE$	**byRef *BtAddrBE$* AS STRING.** The Bluetooth address of the connected remote device.
nConnHandle	**byRef *nConnHandle* AS INTEGER.** Returned connection handle.

Example:

```
// Example :: BleGetConnHandleFromAddr.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


DIM rc, periphAddr$

'//Scan indefinitely
rc=BleScanStart(0, 0)

IF rc==0 THEN
    PRINT "\nScanning"
ELSE
    PRINT "\nError: "; INTEGER.H'rc
ENDIF



'//This handler will be called when an advert is received
FUNCTION HndlrAdvRpt()
    DIM advData$, nDiscarded, nRssi

    '//Read an advert report and connect to the sender
    rc=BleScanGetAdvReport(periphAddr$, advData$, nDiscarded, nRssi)
    rc=BleScanStop()

    '//Connect to device with MAC address obtained above with 5s connection timeout,
    '//20ms min connection interval, 75 max, 5 second supervision timeout.
    rc=BleConnect(periphAddr$, 5000, 20000, 75000, 5000000)
    IF rc==0 THEN
        PRINT "\n--- Connecting"
    ELSE
```

```
        PRINT "\nError: "; INTEGER.H'rc
    ENDIF
ENDFUNC 1

'//This handler will be called in the event of a connection timeout
FUNCTION HndlrConnTO()
    PRINT "\n--- Connection timeout"
    rc=BleScanStart(0, 0)
ENDFUNC 1

'//This handler will be called when there is a BLE message
FUNCTION HndlrBleMsg(nMsgId, nCtx)
    IF nMsgId == 0 THEN
        dim h
        rc=BleGetConnHandleFromAddr(periphAddr$, h)
        PRINT "\n--- Connected to device with MAC address "; StrHexize$(periphAddr$);"
Handle: ";h
        PRINT "\n--- Disconnecting now"
        rc=BleDisconnect(nCtx)
    ENDIF
ENDFUNC 1

'//This handler will be called when a disconnection happens
FUNCTION HndlrDiscon(nCtx, nRsn)
ENDFUNC 0

ONEVENT EVBLEMSG          CALL HndlrBleMsg
ONEVENT EVDISCON          CALL HndlrDiscon
ONEVENT EVBLE_ADV_REPORT   CALL HndlrAdvRpt
ONEVENT EVBLE_CONN_TIMEOUT CALL HndlrConnTO

WAITEVENT
```

Expected Output:

```
Scanning
--- Connecting
--- Connected to device with MAC address 000016A4093A64 Handle: 261888
--- Disconnecting now
00
```

5.6.10 BleGetAddrFromConnHandle

FUNCTION

This function is used to get the Bluetooth address of a device from a connection handle.

BLEGETADDRFROMCONNHANDLE (nConnHandle, BtAddrBE$)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
nConnHandle	**byRef *nConnHandle* AS INTEGER.** Connection handle from which to get Bluetooth address
BtAddrBE$	**byRef *BtAddrBE$* AS STRING.** Returned Bluetooth address.

Example:

```
// Example :: BleGetAddrFromConnHandle.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples
DIM rc, periphAddr$


'//Scan indefinitely
rc=BleScanStart(0, 0)


IF rc==0 THEN
    PRINT "\nScanning"
ELSE
    PRINT "\nError: "; INTEGER.H'rc
ENDIF


'//This handler will be called when an advert is received
FUNCTION HndlrAdvRpt()
    DIM advData$, nDiscarded, nRssi


    '//Read an advert report and connect to the sender
    rc=BleScanGetAdvReport(periphAddr$, advData$, nDiscarded, nRssi)
    rc=BleScanStop()


    '//Connect to device with MAC address obtained above with 5s connection timeout,
    '//20ms min connection interval, 75 max, 5 second supervision timeout.
    rc=BleConnect(periphAddr$, 5000, 20000, 75000, 5000000)
    IF rc==0 THEN
        PRINT "\n--- Connecting"
    ELSE
        PRINT "\nError: "; INTEGER.H'rc
    ENDIF
ENDFUNC 1


'//This handler will be called in the event of a connection timeout
```

```
FUNCTION HndlrConnTO()
    PRINT "\n--- Connection timeout"
    rc=BleScanStart(0, 0)
ENDFUNC 1

'//This handler will be called when there is a BLE message
FUNCTION HndlrBleMsg(nMsgId, nCtx)
    IF nMsgId == 0 THEN
        dim addr$
        rc=BleGetAddrFromConnHandle(nCtx,addr$)
        PRINT "\n--- Connected to device with MAC address "; StrHexize$(addr$)
        PRINT "\n--- Disconnecting now"
        rc=BleDisconnect(nCtx)
    ENDIF
ENDFUNC 1

'//This handler will be called when a disconnection happens
FUNCTION HndlrDiscon(nCtx, nRsn)
ENDFUNC 0

ONEVENT EVBLEMSG          CALL HndlrBleMsg
ONEVENT EVDISCON          CALL HndlrDiscon
ONEVENT EVBLE_ADV_REPORT   CALL HndlrAdvRpt
ONEVENT EVBLE_CONN_TIMEOUT CALL HndlrConnTO

WAITEVENT
```

Expected Output:

```
Scanning
--- Connecting
--- Connected to device with MAC address 000016A4093A64
--- Disconnecting now
00
```

5.6.11 BleConnRssiStart

FUNCTION

This function is used to enable RSSI reporting for a particular connection. Given an RSSI value is generated for every connection event, this can result in a flood of events which will result in increased power consumption as the CPU will need to be in active mode for longer to process them. To mitigate this, this function also takes a threshold dBm value and a skipcount to reduce and manage these events.

The threshold dBm parameter ensures that a report is only generated if the change in detected RSSI value is greater or less than the most reported value by this amount and the skipcount is how many times this condition has to occur for the event to be thrown to the application.

BLECONNRSSISTART (nConnHandle, nThresholdDbm, nSkipCount)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
nConnHandle	**byVal *nConnHandle* AS INTEGER.** Specifies the handle of the connection for which rssi reporting is to be enabled
nThresholdDbm	**byVal *nThresholdDbm* AS INTEGER.** The minimum change in dBm before triggering the EVCONNRSSI event
nSkipCount	**byRef *nSkipCount* AS INTEGER.** The number of RSSI samples with a change of nThresholdDbm or more before triggering the EVCONNRSSI event

Example:

```
// Example :: BleConnRssiStart.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples
DIM rc,conHndl
DIM addr$ : addr$=""
//==============================================================================
// Initialise
//==============================================================================
FUNCTION OnStartup()
    rc=BleAdvertStart(0,addr$,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"
        rc=BleConnRssiStart(conHndl,4,10)
    ENDIF
ENDFUNC 1
//==============================================================================
// Connection related RSSI events
//==============================================================================
FUNCTION HndlrConnRssi(BYVAL charHandle, BYVAL rssi) AS INTEGER
```

```
    PRINT "\nRSSI=";rssi;" for connection "; integer.h' charHandle
    IF rssi < -80 then
        //too far away so stop monitoring the rssi (this is just an example)
        //in reality use some other reason to stop
        rc=BleConnRssiStop(conHndl)
    ENDIF
ENDFUNC 1
//==============================================================================
//==============================================================================
ONEVENT  EVBLEMSG    CALL HndlrBleMsg
ONEVENT  EVCONNRSSI  CALL HndlrConnRssi

IF OnStartup()!=0 THEN
    PRINT "\nFailure OnStartup"
ENDIF

//Wait for events
WAITEVENT
```

5.6.12 BleConnRssiStop

FUNCTION

This function is used to disable RSSI reporting for a particular connection which was enabled using the function BleConnRssiStart described above.

On disconnection, reporting will automatically stop.

BLECONNRSSISTOP (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 for which rssi reporting is to be enabled

For example, see description of BleConnRssiStart() above.

5.7 Whitelist Management Functions

This section describes routines which are used to manage whitelists.

A whitelist is a list of Bluetooth addresses and Identity Resolving Keys (IRKs) which the baseband radio will use to gate incoming packets upwards to the stack as they are received.

If the whitelist is active, then any radio packet whose source Bluetooth address is not in the list will be rejected. However, note that in BLE for privacy reasons, resolvable Bluetooth addresses can be used and so the address will not match with one in the list and so for that type of address the list of Indentity Resolving Keys in the whitelist is also consulted to see if the resolvable address is a trusted device.

A trusted device by definition will have supplied its IRK key when the pairing and bonding happened in the past.

Hence treat this group of functions as a means of creating, maintaining and destroying that list of addresses and IRKs.

The operation that enables whitelisting is the function that starts advertising and scanning. So refer to the functions BleAdvertStart() and BleScanStart().

5.7.1 BleWhitelistCreate

FUNCTION

This function is used to create a new whitelist to which addresses and identity resolving keys can be added using BleWhitelistAddAddr() or BleWhitelistAddIndex().

BLEWHITELISTCREATE (hWlist, nMaxAddrs, nMaxIrks, nPktFilterMask)

Returns	INTEGER, a result code. **Typical value:** 0x0000 indicates a successful operation 0x605E indicates too many whitelists already created.
Arguments	
hWlist	**byRef *hWlist* AS INTEGER.** If an empty whitelist is successfully created then this will be updated with a valid handle. If not then this will contain -1 (0xFFFFFFFF)
nMaxAddrs	**byVal *nMaxAddrs* AS INTEGER.** Maximum addresses that will be stored in this whitelist
nMaxIrks	**byVal *nMaxIrks* AS INTEGER.** Maximum Identity Resolving Keys (IRKs) that will be stored in this whitelist
nPktFilterMask	**byVal *nPktFilterMask* AS INTEGER.** This is a bit mask which specifies what type of incoming packets this list will apply to, as follows: ▪ Bit 0 : Set to 1 for Scan Request packets ▪ Bit 1 : Set to 1 for Connection Request packets ▪ Bit 2 : Set to 1 for Advert Report Packets ▪ Bits 3 to 31 : reserved for future use **Note**: If all bits are 0, then a default mask of 7 is used for the BL652.

Example:

```
// Example :: BleWhitelist.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


DIM rc,conHndl,hWlist, val
DIM addr$ : addr$=""


//==========================================================================
```

```
//==============================================================================
sub AssertRC(byval tag as integer)
  if rc!=0 then
    print "\nFailed with ";integer.h' rc;" at tag ";tag
  endif
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

//==============================================================================
// This handler is called when there is an advert report waiting to be read
//==============================================================================
function HandlerAdvRpt() as integer
  dim ad$,dta$,ndisc,rsi
  rc = BleScanGetAdvReport(ad$,dta$,ndisc,rsi)
  while rc==0
    print "\nADV:";strhexize$(ad$);" ";strhexize$(dta$);" ";ndisc;" ";rsi
    rc = BleScanGetAdvReport(ad$,dta$,ndisc,rsi)
  endwhile
endfunc 1

//==============================================================================
// This handler is called when there is an advert report waiting to be read
//==============================================================================
sub WhiteListInit()
  //set invalid whitelist handle
  hWlist=-1
  //now check maximum whitelists that can be defined and for that valid handle
```

```
  //is not required
  rc=BleWhiteListInfo(hWlist,0, val)  //get max number of whitelists allowes
  AssertRC(100)
  print "\n Max allowed whitelists = "; val

  //create a whitelist
  rc=BleWhitelistCreate(hWlist,8,8,0)
  IF rc==0 THEN
    //Add address we want to specifically look for
    addr$="000016A40B1623"
    rc=BleWhitelistAddAddr(hWlist,addr$)
    AssertRC(110)
    //Made a mistake so clear it
    rc=BleWhitelistClear(hWlist)
    AssertRC(120)
    //now add the correct address
    addr$="000016A40B1642"
    rc=BleWhitelistAddAddr(hWlist,addr$)
    AssertRC(130)
    //now add first one in the trusted database
    rc=BleWhitelistAddIndex(hWlist,0)
    AssertRC(140)
    //Change the filter property from default used in the create function
    //so that connection requests are disallowed
    rc=BleWhitelistSetFilter(hWlist,1)
    AssertRC(150)
    //now check the whitelist by interogating the whitelist handle
    rc=BleWhiteListInfo(hWlist,101, val)  //get current number of mac addresses
    AssertRC(160)
    print "\n Current number of addresses = "; val
  ENDIF
endsub

//==============================================================================
//==============================================================================
ONEVENT  EVBLEMSG             CALL HndlrBleMsg
OnEvent  EVBLE_ADV_REPORT     CALL HandlerAdvRpt

//Initiliase a whitelist
```

```
WhiteListInit()

//start adverts with whitelisting
addr$=""
rc=BleAdvertStart(0,addr$,50,0,hWlist)
AssertRC(910)

//Wait for events
WAITEVENT

//destroy the whitelist
BleWhitelistDestroy(hWlist)
```

5.7.2 BleWhitelistDestroy

FUNCTION

This function is used to destroy an existing whitelist identified by a valid handle previously returned from BleWhitelistCreate() so that new addresses and Identity Resolving Keys (IRKs) can be added. This function completely destroys the whitelist of the given handle, and a new one will need to be created if necessary (using BleWhitelistCreate).

BLEWHITELISTDESTROY (hWlist)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
hWlist	**byRef *hWlist* AS INTEGER.** This is the handle of the whitelist and is passed as a reference so that on exit it will have an invalid handle value so cannot be used inadvertently. The handle will have been returned by BleWhitelistCreate()

For example, see description of BleWhitelistCreate() above.

5.7.3 BleWhitelistClear

FUNCTION

This function is used to clear an existing whitelist identified by a valid handle previously returned from BleWhitelistCreate() so that new addresses and Identity Resolving Keys (IRKs) can be added. The handle of the whitelist is still valid so data can be added to the whitelist without having to call BleWhitelistCreate again.

BLEWHITELISTCLEAR (hWlist)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
hWlist	**byVal *hWlist* AS INTEGER.** This is the handle of the whitelist to clear and will have been returned by BleWhitelistCreate()

For example, see description of BleWhitelistCreate() above.

5.7.4 BleWhitelistSetFilter

FUNCTION

This function is used to change the filter policy mask associated with the whitelist object identified by the handle.

BLEWHITELISTSETFILTER (hWlist, nPktFilterMask)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
hWlist	**byRef *hWlist* AS INTEGER**. This is the handle of the whitelist and will have been returned by BleWhitelistCreate()
nPktFilterMask	**byVal *nPktFilterMask* AS INTEGER.** This is a bit mask which specifies what type of incoming packets this list will apply to, as follows: ▪ Bit 0 : Set to 1 for Scan Request packets ▪ Bit 1 : Set to 1 for Connection Request packets ▪ Bit 2 : Set to 1 for Advert Report Packets ▪ Bits 3 to 31 : reserved for future use **Note:** If all bits are 0, then a default mask of 7 is used for the BL652.

For example, see description of BleWhitelistCreate() above.

5.7.5 BleWhitelistAddAddr

FUNCTION

This function is used to add a 7 byte BT address to the whitelist identified by the handle supplied. The function will automatically check if the BT address is trusted by interrogating the trusted device database and if it is, then the address stored there along with the IRK is added instead of the address supplied. This means that in smartphones with Android and iOS (which make heavy use of resolvable addresses) there is seemless and hassle free integration.

BLEWHITELISTADDADDR (hWlist, addr$)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
hWlist	**byVal *hWlist* AS INTEGER**. This is the handle of the whitelist and will have been returned by BleWhitelistCreate()
addr$	**byRef addr$ AS STRING.** This is the address that is to be added to the whitelist. It will be checked for presence in trusted device database and if trusted, the IRK will also be added automatically to the whitelist

For example, see description of BleWhitelistCreate() above.

5.7.6 BleWhitelistAddIndex

FUNCTION

This function is used to add the Nth indexed device in the trusted device database to the whitelist identified by the handle supplied. If that Nth record exists in the database then the Identity Resolving Key will also be added automatically.

BLEWHITELISTADDINDEX (hWlist, nIndex)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
hWlist	**byVal *hWlist* AS INTEGER**. This is the handle of the whitelist and will have been returned by BleWhitelistCreate()
nIndex	**byVal nIndex AS INTEGER.** This is the Nth index (zero based) of the record in the trusted device database to add to the whitelist. The IRK will also be added automatically to the whitelist. The index is the same entity per the function BleBondMngrGetInfo()

For example, see description of BleWhitelistCreate() above.

5.7.7 BleWhitelistInfo

FUNCTION

This function is used to return information about the whitelist provided. This may be invalid for certain nInfoID values, as that is information about the whitelist manager in general.

BLEWHITELISTINFO (hWlist, nInfoID, nValue)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
hWlist	**byVal *hWlist* AS INTEGER**. This is the handle of the whitelist and will have been returned by BleWhitelistCreate()
nInfoID	**byVal *nInfoID* AS INTEGER.** This is ID of the information to be returned as follows: ▪ 0 : maximum number of whitelists (hWlist is ignored) ▪ 1 : maximum number of Bluetooth addresses (hWlist is ignored) ▪ 2 : maximum number of IRKs (hWlist is ignored) ▪ 101 : current number of addresses added ▪ 102 : current number of IRKs added **Note:** For 101 and 102, the values will be cleared to 0 if BleWhitelistClear() is called.
nValue	**byRef *nValue* AS INTEGER.** The information value is returned in this variable

For example, see description of BleWhitelistCreate() above.

5.8 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 is 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. 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. Because these descriptions are concise and difficult to understand, the following section attempts to familiarise you with these concepts using the *smart*BASIC programming environment perspective.

To help understand 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 a descriptor as the color of the pot, whether it has a lid, whether the lid has a lock, 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 *smart*BASIC API functions if you decide to add those to your characteristic definition.

You can consider a service as a carrier bag to hold a group of related characterisics together where the printing on the carrier bag is a UUID. From a *smart*BASIC developer's perspective, a set of characteristics is what you 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 difference 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 an offset plus a base 128-bit UUID defined and reserved by the Bluetooth SIG); custom service or characteristics have the full 128-bit UUID. The logic behind this is that a 16-bit UUID implies that a specification has 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 the requirement for a central register is important to understand in the sense that, if a custom service or characteristic must 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 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 intends to create custom services and/or characteristics then it is recommended that a single UUID is generated and 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 allows up to 65536 custom services and characteristics to be created, with the added advantage that it is 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 published a specification which defines that service or characteristic and there is a requirement that any device claiming to support them has proof 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 the provider and implementer.

A service is an abstraction of some collectivised functionality which, if broken down further, would cease to provide the intended behaviour. Two 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 such as Systolic Pressure, Diastolic Pressure, and Pulse Rate. Likewise, a Heart Rate service 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, it should be defined and described in a similar fashion; 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 have 128-bit (16-byte) UUIDs and API functions are provided to handle those.

Note: If you intend to create a custom service or characteristic and adopt the recommendation of a single 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 remainder of this introduction focuses 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 was described as a carrier bag that groups related characteristics together and a characteristic is 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.

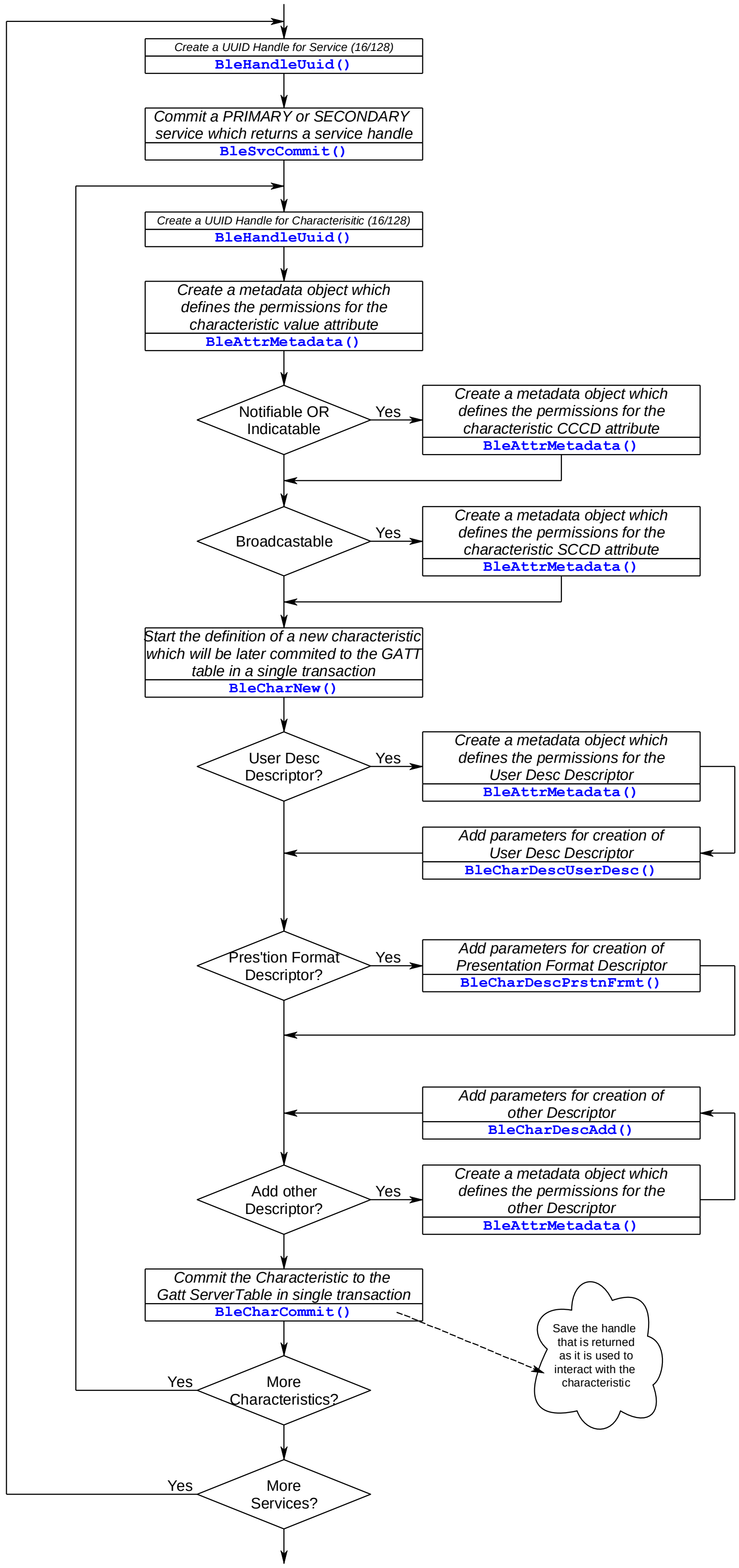
The GATT client (remote end of the wireless connection) msut see those carrier bags to determine the groupings and, once it has identified the pots, it only needs to keep a list of references to the pots it is interested in. Once that list is made at the client end, it can 'throw away the carrier bag'.

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 *smart*BASIC API function, a receipt is returned and is referred to as a char_handle. The developer must then keep those handles to be able to interact with that 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 *smart*BASIC does not care.

From the *smart*BASIC application developer's logical perspective, a GATT table looks nothing like the table that is presented in most BLE literature. Instead, the GATT table is simply 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 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 is 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 on the next page.



Example:

```
// Example :: ServicesAndCharacteristics.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 //===============================================================================
 //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 are detected and processed as follows:

```
//------------------------------------------------------------------------------
// 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, 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, 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.

5.8.1 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.

5.8.2 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 must 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. The most typical value is 0x0000, indicating a successful operation
Arguments:	
deviceName	**byRef *deviceName* AS STRING** The name of the device (such as 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. We recommend 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

Example:

```
// Example :: BleGapSvcInit.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

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
```

5.8.3 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 may 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

Example:

```
// Example :: BleGetDeviceName$.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


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


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


// Changing device name manually
dvcNme$= "My BL652"
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 BL652
--- New DevName : My BL652
```

5.8.4 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 is always set to **BL652: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$, pnpId$)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating 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$* AS STRING** The device system ID as defined in the specifications. Can be set empty to omit submission. Otherwise it shall be a string exactly eight octets long, where: Byte 0..4 := Manufacturer Identifier Byte 5..7 := Organisationally Unique Identifier If the string is one character long and contains @, the system ID is created from the Bluetooth address if (and only if) an IEEE public address is set. If the address is the random static variety, this characteristic is omitted.
regDataList$	**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)

Example:

```
// Example :: BleSvcRegDevInfo.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

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

manfNme$ = "Laird Technologies"
mdlNum$ = "BL652"
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
```

5.8.5 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 (16-byte) 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 is treated by the developer as an opaque entity and no further logic is based on the bit content, apart from all zeros which represent 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 is treated as an offset into the Bluetooth SIG 128 bit base UUID

Example:

```
// Example :: BleHandleUuid16.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


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)
```

5.8.6 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 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.

Note: Ensure that you use a 16-byte UUID that has been generated using a random number generator with sufficient entropy to minimise duplication 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 entropy to ensure that it is random. The first byte of the string is the MSB of the UUID (big endian format).

Example:

```
//Example :: BleHandleUuid128.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 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)
```

5.8.7 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 zeroes (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 is treated as an offset into the 128-bit base UUID referenced by nUuidHandle.

Example:

```
// Example :: BleHandleUuidSibling.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples
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)
```

5.8.8 BleServiceNew

FUNCTION

As explained in GATT Server Functions, 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 will 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 is the UUID that identifies 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 only receives 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 is 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 BleSvcCom() 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. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
nSvcType	**byVal *nSvcType* AS INTEGER** This is zero for a SECONDARY service and 1 for a PRIMARY service. All other values are reserved for future use and 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 has 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 contains 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.

Example:

```
// Example :: BleServiceNew.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

#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
```

5.8.9 BleServiceCommit

This function in the BL652 is used to commit a defined service using BleServiceNew() to the GATT table and should be called after the last characteristic/description has been created/commited for that service.

BLESERVICECOMMIT (hService)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
hService	**byVal hService AS INTEGER** This handle is returned from BleServiceNew().

See example for BleCharCommit().

5.8.10 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 is usually, but not necessarily, a SECONDARY service which is virtually identical to a PRIMARY service from the GATT server perspective. The only difference is that, when a GATT client queries a device for all services, it does not receive mention of SECONDARY services.

When a GATT client encounters an INCLUDED SERVICE object when querying a particular service it performs 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. 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.

Note: If a service has INCLUDED services, then they is 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. The most typical value is 0x0000, indicating a successful operation
Arguments:	
hService	**byVal hService AS INTEGER** This argument contains a handle that was previously created using the function BleSvcCommit().

Example:

```
// Example :: BleSvcAddIncludeSvc.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples
 #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)
```

5.8.11 BleAttrMetadataEx

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 two bytes of data and currently only two 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 two bytes of data and currently only one 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.

BLEATTRMETADATAEX (nReadRights, nWriteRights, nMaxDataLen, nFlags, 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)
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)
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.

nFlags	**byVal *nFlags* AS INTEGER** This is a bit mask where the bits are defined as follows: ▪ Bit 0: Set this to 1 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. ▪ Bit 1: Set this to 1 to ensure that the memory for the attribute is allocated from User space (and hence less memory available for smartBASIC) so that a larger gatt table can be created. This bit is ignored for all attributes other than for characteristic value. ▪ Bit 2: Set this to 1 to require authorisation for reads. When an attempt to read is made by the client then one of the events EVAUTHVAL, EVAUTHCCCD, EVAUTHSCCD or EVAUTHDESC is thrown to the app and in the handler for that event, either BleAuthorizeChar() or BleAuthorizeDesc() is called with appropriate parameters to grant or deny access. ▪ Bit 3: Set this to 1 to require authorisation for writes. When an attempt to write is made by the client then one of the events EVAUTHVAL, EVAUTHCCCD, EVAUTHSCCD or EVAUTHDESC is thrown to the app and in the handler for that event, either BleAuthorizeChar() or BleAuthorizeDesc() is called with appropriate parameters to grant or deny access.
resCode	**byRef resCode AS INTEGER** This variable is updated with a result code which is 0 if a metadata object was successfully returned by this call. Any other value implies a metadata object did not get created.

Example:

```
// Example :: BleAttrMetadata.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 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=BleAttrMetadataEx(17,0,20,0,rc)
 //There is a CCCD and SCCD in this characteristic
 mdCccd=BleAttrMetadataEx(1,2,2,0,rc)
```

```
mdSccd=BleAttrMetadataEx(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
```

5.8.12 BleCharNew

FUNCTION

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

This function is the first function that is 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)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
nCharProps	**byVal *nCharProps* AS INTEGER** This variable contains a bit mask to specify the following high level properties for the characteristic that is added to the GATT table: **Bit** – **Description** 0 – Broadcast capable (SCCD descriptor must be present) 1 – Can be read by the client 2 – Can be written by the client without a response 3 – Can be written 4 – Can be notifiable (CCCD descriptor must be present) 5 – Can be indicatable (CCCD descriptor must be present) 6 – Can accept signed writes 7 – Reliable writes
nUuidHandle	**byVal *nUuidHandle* AS INTEGER** This specifies the UUID that is 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 used to define the properties of the Value attribute that is created in the characteristic and is pre-created with help from 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 is created in the characteristic and is 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 treat the descriptor so that read and write access is open.
mdSccd	**byVal *mdSccd* AS INTEGER** This is an optional metadata that is used to define the properties of the SCCD descriptor attribute that is created in the characteristic and is 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 treat the descriptor so that read and write access is open.

Example:

```
// Example :: BleCharNew.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 DIM rc
 DIM charUuid : charUuid = BleHandleUuid16(2)       //Characteristic's UUID
 DIM mdVal : mdVal = BleAttrMetadataEx(1,0,20,0,rc)   //Metadata for value attribute
 DIM mdCccd : mdCccd = BleAttrMetadataEx(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
```

5.8.13 BleCharDescUserDesc

FUNCTION

This function adds an optional User Description Descriptor to a Characteristic and can only be called after BleCharNew() starts 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 as such. 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. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
userDesc$	**byRef *userDesc$* AS STRING** The user description string with which to initiliase the descriptor. If the length of the string exceeds the maximum length of an attribute then this function aborts 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 pre-created using the help of BleAttrMetadata(). If the write rights are set to 1 or greater, the attribute is marked as writable and the client is able to provide a user description that overwrites the one provided in this call.

Example:

```
// Example :: BleCharDescUserDesc.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 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
```

5.8.14 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. The most typical value is 0x0000, indicating 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. The following is the enumeration list at the time of writing: (see table below)
nExponent	**byVal *nExponent* AS INTEGER** 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.* Valid range -128 to 127
nUnit	**byVal *nUnit* AS INTEGER** 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 Valid range 0 to 65535.
nNameSpace	**byVal *nNameSpace* AS INTEGER** 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

0x00	RFU	0x01	boolean
0x02	2bit	0x03	nibble
0x04	unit8	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		

	Valid range 0 to 255.
nNSdesc	**byVal *nNSdesc* AS INTEGER** This value is a description of the organisation specified by nNameSpace. Valid range 0 to 65535.

Example:

```
// Example :: BleCharDescPrstnFrmt.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

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
```

5.8.15 BleCharDescAdd

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. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
nUuid16	**byVal *nUuid16* AS INTEGER** This is a value in the range 0x2905 to 0x2999 **Note:** This is the actual UUID value, NOT the handle. The highest value at the time of writing is 0x290E, defined for the Report Reference Descriptor. See http://developer.Bluetooth.org/GATT/descriptors/Pages/DescriptorsHomePage.aspx for a list of Descriptors defined and adopted by the Bluetooth SIG.
attr$	**byRef *attr$* AS STRING** This is the data that is 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 is created in the Characteristic and was 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 the client is able to modify the attribute value.

Example:

```
// Example :: BleCharDescAdd.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 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
```

5.8.16 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 references an internal opaque object that is a linked list of all the attribute handles in the characteristic which by definition implies that there is a minimum of 1 (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 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. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
hService	**byVal hService AS INTEGER** This is the handle of the service to which the characteristic belongs, 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 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 *smart*BASIC 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 for which characteristic the message is intended. See event messages: EVCHARHVC, EVCHARVAL, EVCHARCCCD, EVCHARSCCD, EVCHARDESC.

Example:

```
// Example :: BleCharCommit.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 #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
```

5.8.17 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. This function will most often be accessed from the handler that services that event.

BLECHARVALUEREAD (charHandle, attr$)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating 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.

Example:

```
// Example :: BleCharValueRead.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 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$,0x18EE,-1,-1,-1,-1,-1)
     //commit reports to GATT table - adRpt$ 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 BL652 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 BL652 and send a new value

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

5.8.18 BleCharValueWrite

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. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
charHandle	**byVal *charHandle* AS INTEGER** This is the handle to the characteristic whose value must be updated which was returned when BleCharCommit() was called.
attr$	**byRef attr$ AS STRING** String variable, contains new value to write to the characteristic.

Example:

```
// Example :: BleCharValueWrite.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

 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...
```

5.8.19 BleCharValueWriteEx

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(). It differs from the original BleCharValueWrite in that the offset at which to write the data can now be specified.

BLECHARVALUEWRITEEX (charHandle, offset, attr$)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
charHandle	**byVal *charHandle* AS INTEGER** This is the handle to the characteristic whose value must be updated which was returned when BleCharCommit() was called.
offset	**byVal *charHandle* AS INTEGER** This is the offset at which to write the characteristic value.
attr$	**byRef attr$ AS STRING** String variable, contains new value to write to the characteristic.

See example for EVAUTHVALEX

5.8.20 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 is returned by the function BleCharCommit().

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

BLECHARVALUENOTIFY (charHandle, attr$)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	

charHandle	**byVal *charHandle* AS INTEGER** This is the handle to the characteristic whose value must be updated which is returned when BleCharCommit() is 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.

Example:

```
// Example :: BleCharValueNotify.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

 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$,0x18EE,-1,-1,-1,-1,-1)
     //commit reports to GATT table - adRpt$ 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 BL652 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 BL652 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...
```

5.8.21 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 is presented to the *smart*BASIC application as the EVCHARHVC event.

BLECHARVALUEINDICATE (charHandle, attr$)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
charHandle	**byVal *charHandle* AS INTEGER** This is the handle to the characteristic whose value must be updated which is returned 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.

Example:

```
// Example :: BleCharValueIndicate.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples
 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$,0x18EE,-1,-1,-1,-1,-1)
     //commit reports to GATT table - adRpt$ 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 BL652 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 BL652 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...
```

5.8.22 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. The typical value is 0x0000, indicating a successful operation.
Arguments:	
charHandle	**byVal *charHandle* AS INTEGER** This is the handle to the characteristic whose descriptor must be read which is returned when BleCharCommit() is called and is 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 is 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 is 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.

Example:

```
// Example :: BleCharDescRead.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples
 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$,0x2AFF,-1,-1,-1,-1,-1)
    //commit reports to GATT table - adRpt$ 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...
```

5.8.23 BleAuthorizeChar

FUNCTION

This function is used to grant or deny a read or write access of characteristic and is called in the handler for the event EVAUTHVAL. When the function returns and if write access was requested and granted then the characteristic value is deemed to be updated and so function BleCharValueRead() can be used to get the new value.

BLEAUTHORIZECHAR (connHandle, charHandle, readWrite)

Returns	**INTEGER, a result code.** **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
connHandle	**byVal conn*Handle* AS INTEGER** This is the connection handle of the gatt client requesting the read or write access and will have been supplied in the EVAUTHVAL message.
charHandle	**byVal *charHandle* AS INTEGER** This is the handle to the characteristic whose value must be read which was returned when BleCharCommit() was called and will have been supplied in the EVAUTHVAL event message.
readWrite	**byVal *readWrite* AS INTEGER** This will be to • 0 to deny read access • 1 to allow read access • 2 to deny write access • 3 to allow write access

```
//Example :: See description for EVAUTHVAL
```

5.8.24 BleAuthorizeDesc

FUNCTION

This function is used to grant or deny a read or write access of characteristic descriptor and is called in the handler for the three events EVAUTHCCCD, EVAUTHSCCD and EVAUTHDESC. When the function returns and if write access was requested and granted then the characteristic descriptor value is deemed to be updated and so function BleCharDescRead() can be used to get the new value of the descriptor when the event is EVAUTHDESC. For events EVAUTHCCCD and EVAUTHSCCD the event itself will have supplied the new value.

BLEAUTHORIZEDESC (connHandle, charHandle, nDescType, readWrite)

Returns	INTEGER, a result code. **Typical value:** 0x0000 (indicates a successful operation)
Arguments	
connHandle	**byVal conn*Handle* AS INTEGER** This is the connection handle of the gatt client requesting the read or write access and will have been supplied in the EVAUTHVAL message.
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 EVAUTHVAL event message.
nDescType	**byVal nDescType AS INTEGER** This is as was supplied in the EVAUTHDESC event
readWrite	**byVal *readWrite* AS INTEGER** This will be to • 0 to deny read access • 1 to allow read access • 2 to deny write access • 3 to allow write access

```
//Example :: See description for EVAUTHCCCD, EVAUTHSCCD or EVAUTHDESC
```

5.8.25 BleServiceChangedNtfy

FUNCTION

This function causes an indication of the Service Changed Characteristic of the GATT Service and specifies a start attribute handle and an end attribute handle, which the client shall mark as changed so that it can update it's cache if need be.

The EVBLEMSG event will be thown with subevent ID set to **BLE_EVBLEMSGID_SRVCCHNG_IND_CNF** when other indications can be sent.

Note that if on connection to a bonded device the CCCD CRC does not match with the current GATT table then a Service Change Indication is automatically sent to the client. Additionally, the local application is sent the event **BLE_EVBLEMSGID_SRVCCHNG_IND_SENT.**

BLESERVICECHANGEDNTFY (nConnHandle, nStartHandle, nEndHandle)

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.
nStartHandle	**byVal *nStartHandle* AS INTEGER.** Specifies the start attribute handle of GATT table that has changed. Set to 0 to mark the entire table as changed.
nEndHandle	**byVal *nEndHandle* AS INTEGER.** Specifies the end attribute handle of GATT table that has changed. Set to 0 to mark the entire table as changed.

5.9 GATT Client Functions

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

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. It is best to see it as a table consisting of many rows and three visible columns (handle, type, value) and at least one more invisible column whose content affects 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.

A table should be scanned from top to bottom; the specification stipulates that the 16-bit handle field contains values in the range 1 to 65535 and SHALL be in ascending order. Gaps are allowed.

When scanning, if a row is encountered with the value 0x2800 or 0x2801 in the Type column, then it is understood as the start of a primary or secondary service which in turn contains 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 contains the value for that characteristic; afterwards, there may be zero or more descriptors.

This means each characteristic consists of at least two 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)
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. There are three services (at handles 0x0001,0x0008 and 0x000B) because there are three rows where the Type = 0x2800. 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, there is one or more characteristics where Type=0x2803. For example the service beginning at handle 0x0008 has one characteristic which contains two 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, the characteristic at row with handle = 0x0004 has the mandatory value row and then two descriptors.

The Bluetooth specification allows for multiple instances of the same service or characteristics or descriptors and they are differentiated by the unique handle. This ensures 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 stipulate that once the handle values are allocated, they are 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 its cache and rescan the GATT table.

When a connection is first established, there is no prior knowledge as to which services exist or their handles. Therefore, 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 *smart*BASIC functions which encapsulate and manage those procedures to enable a *smart*BASIC 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 *smart*BASIC internal engine also maintains data objects so that it is possible to interact with descriptors associated with the characteristic.

Basically, the table scanning process reveals characteristic handles (as handles of handles) which are used in other GATT client related *smart*BASIC functions to interact with the table to, for example, read/write or accept and process incoming notifications and indications.

This approach ensures 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 is delivered asynchronously as an event for which a handler must be specified in the user *smart*BASIC application.

The rest of this chapter details all GATT client commands, responses, and events along with example code demonstrating usage and expected output.

5.9.1 Events and Messages

The nature of GATT client operation consists of multiple queries and acting on the responses. Because the connection intervals are slower than the CPU speed, responses can arrive many tens of milliseconds after the procudure is triggered; these are delivered to an application using an event or message. Since these event/messages are tightly coupled with the appropriate commands, all but one is 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 is dropped as no further GATT client transaction can be initiated.

5.9.1.1 EVGATTCTOUT

This event message is thrown if a GATT client transaction takes longer than 30 seconds. It contains one INTEGER parameter:

- Connection Handle

Example:

```
// Example :: EVGATTCTOUT.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

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
. . .
. . .
```

5.9.1.2 EVDISCPRIMSVC

This event message is thrown if either BleDiscServiceFirst() or BleDiscServiceNext() returns a success. The message contains the following four INTEGER parameters:

- Connection Handle
- Service UUID Handle
- Start Handle of the service in the GATT table
- End Handle for the service

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

5.9.1.3 *EVDISCCHAR*

This event message is thrown if either BleDiscCharFirst() or BleDiscCharNext() returns a success. The message contains the following 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 contain zero 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

'**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 is always 0.

5.9.1.4 *EVDISCDESC*

This event message is thrown if either BleDissDescFirst() or BleDiscDescNext() returns a success. The message contains the following 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 contain zero apart from the Connection Handle.

'**Descriptor Uuid Handle**' contains the UUID of the descriptor and is supplied as a handle.

'**Handle for the Descriptor in the remote GATT Table**' is the handle for the descriptor as well as the value to store to keep track of important characteristics in a GATT server for later read/write operations.

5.9.1.5 EVFINDCHAR

This event message is thrown if BleGATTcFindChar() returns a success. The message contains the following 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 contain zero apart from the Connection Handle.

'**Characteristic Properties**' contains the properties of the characteristic and is a bit mask as follows:

Bit	Description
0	Set if BROADCAST is enabled
1	Set if READ is enabled
2	Set if WRITE_WITHOUT_RESPONSE is enabled
3	Set if WRITE is enabled
4	Set if NOTIFY is enabled
5	Set if INDICATE is enabled
6	Set if AUTHENTICATED_SIGNED_WRITE is enabled
7	Set if RELIABLE_WRITE is enabled
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 is always 0.

5.9.1.6 EVFINDDESC

This event message is thrown if BleGATTcFindDesc() returned a success. The message contains the following 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 contain zero 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, CCCDs to enable notifications and/or indications.

5.9.1.7 EVATTRREAD

This event message is thrown if BleGattcRead() returns a success. The message contains the following 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().

```
Hex      Dec   Description
```

```
0x0000  0    Success
0x0001  1    Unknown or not applicable status
0x0100  256  ATT Error: Invalid Error Code
0x0101  257  ATT Error: Invalid Attribute Handle
0x0102  258  ATT Error: Read not permitted
0x0103  259  ATT Error: Write not permitted
0x0104  260  ATT Error: Used in ATT as Invalid PDU
0x0105  261  ATT Error: Authenticated link required
0x0106  262  ATT Error: Used in ATT as Request Not Supported
0x0107  263  ATT Error: Offset specified was past the end of the attribute
0x0108  264  ATT Error: Used in ATT as Insufficient Authorisation
0x0109  265  ATT Error: Used in ATT as Prepare Queue Full
0x010A  266  ATT Error: Used in ATT as Attribute not found
0x010B  267  ATT Error: Attribute cannot be read or written using read/write blob requests
0x010C  268  ATT Error: Encryption key size used is insufficient
0x010D  269  ATT Error: Invalid value size
0x010E  270  ATT Error: Very unlikely error
0x010F  271  ATT Error: Encrypted link required
0x0110  272  ATT Error: Attribute type is not a supported grouping attribute
0x0111  273  ATT Error: Encrypted link required
0x0112  274  ATT Error: Reserved for Future Use range #1 begin
0x017F  383  ATT Error: Reserved for Future Use range #1 end
0x0180  384  ATT Error: Application range begin
0x019F  415  ATT Error: Application range end
0x01A0  416  ATT Error: Reserved for Future Use range #2 begin
0x01DF  479  ATT Error: Reserved for Future Use range #2 end
0x01E0  480  ATT Error: Reserved for Future Use range #3 begin
0x01FC  508  ATT Error: Reserved for Future Use range #3 end
0x01FD  509  ATT Common Profile and Service Error: Client Characteristic Config Descriptor
                                               (CCCD)improperly configured
0x01FE  510  ATT Common Profile and Service Error:Procedure Already in Progress
0x01FF  511  ATT Common Profile and Service Error: Out Of Range
```

5.9.1.8 EVATTRWRITE

This event message is thrown if BleGattcWrite() returns a success. The message contains the following 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.

```
Hex     Dec  Description
0x0000  0    Success
0x0001  1    Unknown or not applicable status
0x0100  256  ATT Error: Invalid Error Code
0x0101  257  ATT Error: Invalid Attribute Handle
0x0102  258  ATT Error: Read not permitted
0x0103  259  ATT Error: Write not permitted
0x0104  260  ATT Error: Used in ATT as Invalid PDU
0x0105  261  ATT Error: Authenticated link required
0x0106  262  ATT Error: Used in ATT as Request Not Supported
0x0107  263  ATT Error: Offset specified was past the end of the attribute
0x0108  264  ATT Error: Used in ATT as Insufficient Authorisation
0x0109  265  ATT Error: Used in ATT as Prepare Queue Full
0x010A  266  ATT Error: Used in ATT as Attribute not found
0x010B  267  ATT Error: Attribute cannot be read or written using read/write blob requests
0x010C  268  ATT Error: Encryption key size used is insufficient
0x010D  269  ATT Error: Invalid value size
0x010E  270  ATT Error: Very unlikely error
```

```
0x010F  271  ATT Error: Encrypted link required
0x0110  272  ATT Error: Attribute type is not a supported grouping attribute
0x0111  273  ATT Error: Encrypted link required
0x0112  274  ATT Error: Reserved for Future Use range #1 begin
0x017F  383  ATT Error: Reserved for Future Use range #1 end
0x0180  384  ATT Error: Application range begin
0x019F  415  ATT Error: Application range end
0x01A0  416  ATT Error: Reserved for Future Use range #2 begin
0x01DF  479  ATT Error: Reserved for Future Use range #2 end
0x01E0  480  ATT Error: Reserved for Future Use range #3 begin
0x01FC  508  ATT Error: Reserved for Future Use range #3 end
0x01FD  509  ATT Common Profile and Service Error: Client Characteristic Config Descriptor
                                                (CCCD) improperly configured
0x01FE  510  ATT Common Profile and Service Error:Procedure Already in Progress
0x01FF  511  ATT Common Profile and Service Error: Out Of Range
```

5.9.1.9 EVNOTIFYBUF

This event message is thrown if BleGattcWriteCmd() returned a success. The message contains no parameters.

5.9.1.10 EVATTRNOTIFY

This event is 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 *smart*BASIC application is informed that it must go and service the ring buffer using the function BleGattcNotifyRead. This event is only thrown if at+cfg 213=0. See BleGattcNotifyRead for usage.

5.9.1.11 EVATTRNOTIFYEX

This message from the underlying BLE manager informs the app that the remote has sent characteristic notifications/indications. The difference between this event and EVATTRNOTIFY is that this event contains the paramers such as the connection handle and the notification data. Data_length and strLen(Data$) should be of equal length. This event is only thrown if at+cfg 213=1. See BleGattcNotifyRead for usage.

The event comes with the following parameters:-

- **Connection Handle** – The handle of the connection that wrote to the characteristic value.
- **Char Handle** – Characteristic handle for which the value is being notified.
- **Type** – 0: Invalid, 1: Notification, 2: Indication.
- **Data_Length** – The length of the data that was notified. If negative, then this value indicates the amount of data lost.
- **Data$** - The string data that was notified from the attribute.

5.9.2 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; given that a majority of BL652 use cases do not use it, the sacrifice of 300 bytes is not worth the permament allocation of memory.

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

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: When the ring buffer for the notifiable and indicatable characteristics is full, then any new messages are discarded. Depending on the flags parameter, the indicates are or are not 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. Existing GATT 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. The typical value is 0x0000, indicating a successful operation.
Arguments:	
nNotifyBufLen	**byVal *nNotifyBufLen* AS INTEGER** This is the size of the ring buffer used for incoming notifiable and indicatable characteristic 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 the buffer is full then the Handle Value Confirmation is only sent when BleGattcNotifyRead() is called to read the ring buffer. **Bit 1..31 –** Reserved for future use and must be set to 0s.

Example:

```
// Example :: BleGattcOpen.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples
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
```

5.9.3 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 ()

Returns	
Arguments	None

Example:

```
// Example :: BleGattcClose.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples
```

```
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
```

5.9.4 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. 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 scans for all primary services and the latter scans 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 enters 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 hundreds of milliseconds. While this is in progress, it is safe to do other non-GATT-related operations such as servicing sensors and displays or any of the onboard peripherals.

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. The typical value is 0x0000, indicating a successful operation. This means an EVDISCPRIMSVC event message is thrown by the *smart*BASIC runtime engine containing the results. A non-zero return value implies an EVDISCPRIMSVC message is NOT 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 is returned in the EVBLEMSG event message with msgId == 0 and msgCtx is the connection 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().

BLEDISCSERVICENEXT (connHandle)

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

Returns	**INTEGER, a result code.** The typical value is 0x0000, indicating a successful operation and it means an EVDISCPRIMSVC event message is thrown by the *smart*BASIC runtime engine containing the results. A non-zero return value implies an EVDISCPRIMSVC message is not 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 is returned in the EVBLEMSG event message with msgId == 0 and msgCtx is the connection handle

Example:

```
// Example :: BleDiscServiceFirst.Next.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

//
//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
Scan abort
Scan for service with custom uuid 112233445566778899AABBCCDDEEFF00
EVDISCPRIMSVC : cHndl=2804 svcUuid=FC033344 sHndl=4 eHndl=6
EVDISCPRIMSVC : cHndl=2804 svcUuid=FC033344 sHndl=13 eHndl=15
EVDISCPRIMSVC : cHndl=2804 svcUuid=00000000 sHndl=0 eHndl=0
Scan complete

- Disconnected
Exiting...
```

5.9.5 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. When called, a handler for the event message must be registered because the discovered characteristics information is passed back in that message.

A generic or UUID based scan can be initiated. The generic version scans for all characteristics; the UUID version 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 a GATT table has a specific service and a specific characteristic, then it is more efficient to locate details of that characteristic by using the function BleGATTcFindChar(). This function is described later.

While the scan is in progress and waiting for the next piece of data from a GATT server, the module enters 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 hundreds of milliseconds. While this is in progress, it is safe to do other non- GATT-related operations such as servicing sensors and displays or any of the onboard peripherals.

Note: It is not currently possible to scan for characteristics in included services. This is planned for a future release.

BLEDISCCHARFIRST (connHandle, charUuidHandle, startAttrHandle, endAttrHandle)

A typical pseudo code for discovering characteristic involves first calling BleDiscCharFirst() with information obtained from a primary services scan, waiting for the EVDISCCHAR event message, and (depending on the information returned in that message) calling BleDiscCharNext(). This in turn results 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. The typical value is 0x0000, indicating a successful operation and it means an EVDISCCHAR event message is thrown by the *smart*BASIC runtime engine containing the results. A non-zero return value implies an EVDISCCHAR message is not 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 is returned in the EVBLEMSG event message with msgId == 0 and msgCtx is 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 is generated either by BleHandleUuid16() or BleHandleUuid128() or BleHandleUuidSibling().
startAttrHandle	**byVal *startAttrHandle* AS INTEGER**

	This is the attribute handle from where the scan for characteristic is started and is 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 is acquired by doing a primary services scan, which returns the start and end handles of services.

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. The typical value is 0x0000, indicating a successful operation. It means an EVDISCCHAR event message is thrown by the *smart*BASIC runtime engine containing the results. A non-zero return value implies an EVDISCCHAR message is not 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 is returned in the EVBLEMSG event message with msgId == 0 and msgCtx is the connection handle.

Example:

```
// Example :: BleDiscCharFirst.Next.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

//
//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 characteristic 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 characteristic 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

- Disconnected
Exiting...
```

5.9.6 BleDiscDescFirst /BleDiscDescNext

FUNCTIONS

This pair of functions is used to scan the remote GATT server for descriptors in a characteristic with the help of the EVDISCDESC message event. When called, a handler for the event message must be registered because the discovered descriptor information is passed back in that message.

A generic or UUID-based scan can be initiated. The generic version scans for all descriptors; The UUID version scans for a descriptor with a particular UUID, the handle of which must be supplied and is generated by using either BleHandleUuid16() or BleHandleUuid128().

If a GATT table has a specific service, characteristic, and a specific descriptor, then it is more efficient to locate the characteristic's details by using the function BleGATTcFindDesc(). This is described later.

While the scan is in progress and waiting for the next piece of data from a GATT server, the module enters 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 hundreds of milliseconds. While this is in progress, it is safe to do other non-GATT-related operations such as servicing sensors and displays or any of the onboard peripherals.

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. Depending on the information returned in that message, calling BleDiscDescNext() results 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. The typical value is 0x0000, indicating a successful operation and it means an EVDISCDESC event message is thrown by the *smart*BASIC runtime engine containing the results. A non-zero return value implies an EVDISCDESC message is not 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 is returned in the EVBLEMSG event message with msgId == 0 and msgCtx is 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 is 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.

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. The typical value is 0x0000, indicating a successful operation and it means an EVDISCDESC event message is thrown by the *smart*BASIC runtime engine containing the results. A non-zero return value implies an EVDISCDESC message is not 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 is returned in the EVBLEMSG event message with msgId == 0 and msgCtx is the connection handle.

Example:

```
// Example :: BleDiscDescFirst.Next.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

//
//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 characteristic 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 characteristic 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...
```

5.9.7 BleGattcFindChar

FUNCTION

This function facilitates an efficient way of locating the details of a characteristic if the UUID is known along with the UUID of the service containing it. The results are delived in an EVFINDCHAR event message. If the GATT server table has multiple instances of the same service/characteristic 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. This means the fourth instance of a characteristic with the same UUID in the third instance of a service with the same UUID is 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 hundreds of milliseconds. While this is in progress, it is safe to do other non-GATT-related operations such as servicing sensors and displays or any of the onboard peripherals.

Note: It is not currently possible to scan for characteristics in included services. This is a future enhancement.

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. The typical value is 0x0000, indicating a successful operation and it means an EVFINDCHAR event message is thrown by the *smart*BASIC runtime engine containing the results. A non-zero return value implies an EVFINDCHAR message is not 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 is returned in the EVBLEMSG event message with msgId == 0 and msgCtx is the connection handle.
svcUuidHndl	**byVal *svcUuidHndl* AS INTEGER** Set this to the service UUID handle which is 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, and so on.
charUuidHndl	**byVal *charUuidHndl* AS INTEGER** Set this to the characteristic UUID handle which is 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, and so on.

Example:

```
// Example :: BleGATTcFindChar.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


//
//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$,uHndS,uHndC
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n- Disconnected"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n- Connected, so scan remote GATT Table for an instance of char"
        uHndS = BleHandleUuid16(0xDEAD)
        uu$ = "112233445566778899AABBCCDDEEFF00"
        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...
```

5.9.8 BleGattcFindDesc

FUNCTION

This function facilitates an 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. 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 accepts instance parameters which are indexed from 0. This means that the second instance of a descriptor in the fourth instance of a characteristic with the same UUID in the third 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 hundreds of milliseconds. While this is in progress, it is safe to do other non-GATT-related operations such as servicing sensors and displays or any of the onboard peripherals.

Note: It is not currently possible to scan for characteristics in included services. This planned for a future release.

BLEGATTCFINDDESC (connHndl, svcUuHndl, svcIdx, charUuHndl, charIdx,descUuHndl, descIdx)

A typical pseudo code for finding a descrirptor involves calling BleGATTcFindDesc() which in turn results 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. The typical value is 0x0000, indicating a successful operation and it means an EVFINDDESC event message is thrown by the smartBASIC runtime engine containing the results. A non-zero return value implies an EVFINDDESC message is not 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 is returned in the EVBLEMSG event message with msgId == 0 and msgCtx is the connection handle.
svcUuHndl	**byVal *svcUuHndl* AS INTEGER** Set this to the service UUID handle which is 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, and so on.
charUuHndl	**byVal *charUuHndl* AS INTEGER** Set this to the characteristic UUID handle which is 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, and so on.
descUuHndl	**byVal *descUuHndl* AS INTEGER** Set this to the descriptor uuid handle which is 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, and so on.

Example:

```
// Example :: BleGATTcFindDesc.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


//
//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$ = "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
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...
```

5.9.9 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 hundreds of milliseconds. While this is in progress, it is safe to do other non-GATT-related operations such as 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.

BLEGATTCREAD (connHndl, attrHndl, offset)

A typical pseudo code for reading the content of an attribute calling BleGattcRead() which in turn results 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. The typical value is 0x0000, indicating a successful operation and it means an EVATTRREAD event message is thrown by the *smart*BASIC runtime engine containing the results. A non-zero return value implies an EVATTRREAD message is not 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 is returned in the EVBLEMSG event message with msgId == 0 and msgCtx is the connection handle.
attrHndl	**byVal *attrHndl* AS INTEGER** Set to the handle of the attribute to read. It is 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.

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. The typical value is 0x0000, indicating a successful read.
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 is returned in the EVBLEMSG event message with msgId == 0 and msgCtx is the connection handle.
attrHndl	**byRef *attrHndl* AS INTEGER** The handle for the attribute that was read is returned in this variable. It is 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. It is 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.

Example:

```
// Example :: BleGATTcRead.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


//

//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...
```

5.9.10BleGattcWrite

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 is 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 hundreds of milliseconds. While this is in progress, it is safe to do other non GATT related operations such as servicing sensors and displays or any of the onboard peripherals.

BLEGATTCWRITE (connHndl, attrHndl, attrData$)

A typical pseudo code for writing to an attribute which results 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. The typical value is 0x0000, indicating a successful read.
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 is returned in the EVBLEMSG event message with msgId == 0 and msgCtx is 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.

Example:

```
// Example :: BleGATTcWrite.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


//
//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...
```

5.9.11 BleGattcWriteCmd

FUNCTION

If the handle for an attribute is known, then this function is used to write into an attribute at offset 0 when 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: The acknowledgement received for the BleGattcWrite() command is from the higher level GATT layer. Do not confuse this with the link layer ACK .

All packets are acknowledged at link layer level. If a packet fails to get through, then that condition manifests 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 hundreds of milliseconds. While this is in progress, it is safe to do other non-GATT-related operations such as servicing sensors and displays or any of the onboard peripherals.

BLEGATTCWRITECMD (connHndl, attrHndl, attrData$)

The following is a typical pseudo code for writing to an attribute which results in the EVNOTIFYBUF event:

```
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. The typical value is 0x0000, indicating a successful read.
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 is returned in the EVBLEMSG event message with msgId == 0 and msgCtx is 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.

Example:

```
// Example :: BleGATTcWriteCmd.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


//
//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...
```

5.9.12 BleGattcWritePrepare

FUNCTION

The Write Prepare and Write Execute functions are used to perform the Long Write procedure. Long Writes are used when the value handle is known, but the length of the characteristic value is longer than can be sent in a single Write Request message.

BleGattcWritePrepare requests that the GATT server prepares to write the attribute value. This function can be used multiple times as long as a BleGattcWriteExecute function is used at the end to perform the full Long Write.

BLEGATTCWRITEPREPARE (connHndl, attrHndl, offset, attrData$)

Returns	INTEGER, a result code. The typical value is 0x0000, indicating a successful read.
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 is returned in the EVBLEMSG event message with msgId == 0 and msgCtx is the connection handle.
attrHndl	**byVal *attrHndl* AS INTEGER** The handle for the attribute that is to be written to.
offset	**byVal *attrHndl* AS INTEGER** This is the offset at which the data in the attribute is to be written.
attrData$	**byRef *attrData$* AS STRING** The attribute data to write.

5.9.13 BleGattcWriteExecute

FUNCTION

The BleGattcWriteExecute function is used by the GATT client to request the server to write or cancel the write of all the values that have been prepare with the BleGattcWritePrepare function. It is used as the final step in a long write operation.

BLEGATTCWRITEEXECUTE (connHndl, Flags)

Returns	INTEGER, a result code. The typical value is 0x0000, indicating a successful read.
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 is returned in the EVBLEMSG event message with msgId == 0 and msgCtx is the connection handle.
Flags	**byVal *Flags* AS INTEGER** 0 – Cancel all prepared writes 1 – Immediately write all pending prepared values

5.9.14 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 arrives from a GATT server at any time and must be managed so that it can synchronised with the *smart*BASIC 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(); for indications, an automatic GATT acknowledgement is sent when the data is saved in the ring buffer. This acknowledgment happens even if the data is discarded because the ring buffer is full. If the data must not be acknowledged when it is discarded on a full buffer, 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 is throttled and no further data is notified or indicated by it until BleGattNotifyRead() is called to extract data from the ring buffer to create space and it triggers 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 *smart*BASIC application or the interactive mode command AT I 2019 returns the default size. Likewise SYSINFO(2020) or the command AT I 2020 returns the maximum size.

Data that arrives via notifications or indications get stored in the ring buffer. At the same time, a EVATTRNOTIFY event is thrown to the *smart*BASIC 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.

BLEGATTCNOTIFYREAD (connHndl, attrHndl, attrData$, discardCount)

The following is a typical pseudo code for handling and accessing notification/indication data:

```
Register a handler for the EVATTRNOTIFY event message

On EVATTRNOTIFY event
    BleGattcNotifyRead()  //to actually get the data
```

```
    Process the data

Enable notifications and/or indications via CCCD descriptors
```

Returns	INTEGER, a result code. The typical value is 0x0000, indicating data was successful read.
Arguments:	
connHndl	**byRef *connHndl* AS INTEGER** On exit, this is the connection handle of the GATT server that sent the notification or indication.
attrHndl	**byRef *attrHndl* AS INTEGER** On exit, this is the handle of the characteristic value attribute in the notification or indication.
attrData$	**byRef *attrData$* AS STRING** On exit, this is 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. 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().

Example:

```
// Example :: BleGATTcNotifyRead.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples
//
// 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

'//==============================================================================
'// Thrown when AT+CFG 213 = 0
'//==============================================================================
function HandlerAttrNotify() as integer
  dim chndl,aHndl,att$,dscd
    print "\nEVATTRNOTIFY Event \n"
  rc=BleGattcNotifyRead(cHndl,aHndl,att$,dscd)
  print "\n  BleGattcNotifyRead()"
  if rc==0 then
    print " Connection Handle=";cHndl
    print " Characteristic Handle=";aHndl
    print " Data=";StrHexize$(att$)
    print " Discarded=";dscd
  else
    print " failed with ";integer.h' rc
  endif
endfunc 1

'//==============================================================================
'// Thrown when AT+CFG 213 = 1
'//==============================================================================
function HandlerAttrNotifyEx(BYVAL hConn, BYVAL hChar, BYVAL nType, BYVAL nLen, BYVAL
Data$) as integer

    print "\nEVATTRNOTIFYEX Event :: "
    if nType == 1 then
```

```
        print "Notification\n"
    elseif nType == 2 then
        print "Indication\n"
    endif

    print " Connection Handle=";hConn
    print " Characteristic Handle=";hChar
    print " Data=";Data$
endfunc 1

//==============================================================================
// Main() equivalent
//==============================================================================
ONEVENT  EVBLEMSG            CALL HndlrBleMsg
OnEvent  EVATTRWRITE         call HandlerAttrWrite
OnEvent  EVATTRNOTIFY        call HandlerAttrNotify   // Thrown when AT+CFG 213 = 0
OnEvent  EVATTRNOTIFYEX      call HandlerAttrNotifyEx // Thrown when AT+CFG 213 = 1

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
```

5.10 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 in little endian format and so all data manipulation is done similarly. Little endian means that a multibyte data entity is stored so that lowest significant byte is positioned at the lowest memory address and likewise, when transported, the lowest byte is on the wire first.

This section describes all the encoding functions which allow those strings to be written 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 *smart*BASIC applications so that INTEGER variables are used to manipulate those values instead of STRINGS.

5.10.1 BleEncode8

FUNCTION

This function overwrites a single byte 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 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. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
attr$	**byRef *attr$* AS STRING** This argument is the string that is written to an attribute.
nData	**byVal nData AS INTEGER** The least significant byte of this integer is saved. The rest is ignored.
nIndex	**byVal *nIndex* AS INTEGER** This is the zero-based index into the string attr$ where the new data fragment is written to. If the string attr$ is not long enough to fit 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.

Example:

```
// Example :: BleEncode8.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


DIM rc
DIM attr$

attr$="Laird"

PRINT "\nattr$=";attr$

//Remember: - 4 bytes are used to store an integer on the BL652
//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
```

5.10.2 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. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
attr$	**byRef *attr$* AS STRING** This argument is the string that is written to an attribute.
nData	**byVal nData AS INTEGER** The two least significant bytes of this integer is saved. The rest is ignored.
nIndex	**byVal *nIndex* AS INTEGER** This is the zero based index into the string attr$ 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.

Example:

```
// Example :: BleEncode16.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

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
```

5.10.3 BleEncode24

FUNCTION

This function overwrites three 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.

BLEENCODE24 (attr$, nData, nIndex)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
attr$	**byRef *attr$* AS STRING** This argument is the string that is written to an attribute.
nData	**byVal nData AS INTEGER** The three least significant bytes of this integer is saved. The rest is ignored.
nIndex	**byVal *nIndex* AS INTEGER** This is the zero based index into the string attr$ 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 fails.

Example:

```
// Example :: BleEncode24.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


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:

```
attr$=ABCDEF
```

5.10.4 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. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
attr$	**byRef *attr$* AS STRING** This argument is the string that is written to an attribute.
nData	**byVal nData AS INTEGER** The four bytes of this integer is saved. The rest is ignored.
nIndex	**byVal *nIndex* AS INTEGER** This is the zero based index into the string attr$ 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.

Example:

```
// Example :: BleEncode32.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

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
```

5.10.5BleEncodeFLOAT

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. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
attr$	**byRef *attr$* AS STRING** This argument is the string that is written to an attribute.
nMatissa	**byVal nMantissa AS INTEGER** This value must be in the range -8388600 to +8388600 or the function fails. The data is written in little endian so that the least significant byte is at the lower memory address. **Note:** The range is not +/- 2048 because after encoding the following 2 byte values have special meaning: `0x007FFFFF` NaN (Not a Number) `0x00800000` NRes (Not at this resolution) `0x007FFFFE` + INFINITY `0x00800002` - INFINITY `0x00800001` Reserved for future use
nExponent	**byVal nExponent AS INTEGER** This value must be in the range -128 to 127 or the function fails.
nIndex	**byVal *nIndex* AS INTEGER** This is the zero based index into the string attr$ 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.

Example:

```
// Example :: BleEncodeFloat.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 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
```

5.10.6BleEncodeSFLOATEX

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. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
attr$	**byRef *attr$* AS STRING** This argument is the string that is written to an attribute
nData	**byVal nData AS INTEGER** The 32 bit value is converted into a 2-byte IEEE-11073 16-bit SFLOAT consisting of a 12-bit signed mantissa and a 4-bit signed exponent. This means a signed 32-bit value always fits in such a FLOAT enitity, but there is a loss in significance to 12 from 32.
nIndex	**byVal *nIndex* AS INTEGER** This is the zero-based index into the string attr$ 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.

Example:

```
// Example :: BleEncodeSFloatEx.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


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
```

5.10.7 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. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
attr$	**byRef *attr$* AS STRING** This argument is the string that is written to an attribute.
nMatissa	**byVal *nMantissa* AS INTEGER** This must be in the range -2046 to +2046 or the function fails. The data is written in little endian so the least significant byte is at the lower memory address. **Note:** The range is not +/- 2048 because after encoding, the following 2-byte values have special meaning: 0x007FF — NaN (Not a Number) 0x00800 — NRes (Not at this resolution) 0x007FE — + INFINITY 0x00802 — - INFINITY 0x00801 — Reserved for future use
nExponent	**byVal *nExponent* AS INTEGER** This value must be in the range -8 to 7 or the function fails.
nIndex	**byVal *nIndex* AS INTEGER** This is the zero based index into the string attr$ 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.

Example:

```
// Example :: BleEncodeSFloat.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples
 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
```

5.10.8 BleEncodeTIMESTAMP

FUNCTION

This function overwrites a 7-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 7-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 is 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. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
attr$	**byRef *attr$* AS STRING** This argument is the string that is written to an attribute.
timestamp$	**byRef *timestamp$* AS STRING** This is a 7-byte string as described above. For example 5 May 2013 10:31:24 is entered \14\0D\05\05\0A\1F\18.
nIndex	**byVal *nIndex* AS INTEGER** This is the zero based index into the string attr$ 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.

Example:

```
// Example :: BleEncodeTimestamp.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 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
```

5.10.9BleEncodeSTRING

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. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
attr$	**byRef *attr$* AS STRING** This argument is the string is written to an attribute
nIndex1	**byVal *nIndex1* AS INTEGER** This is the zero based index into the string attr$ 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 specifies 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.

Example:

```
// Example :: BleEncodeString.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


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
```

5.10.10 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 maximum attribute length times eight.

BleEncodeBITS (attr$, nDstIdx, srcBitArr , nSrcIdx, nBitLen)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating 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 is copied. No data is copied if this index is negative or greater than 32.
nBitLen	**byVal *nBitLen* AS INTEGER** This specifies the number of bits from offset nSrcIdx to be copied into the destination bit array represented by the string attr$. It is clipped to the number of bits left to copy after the index nSrcIdx.

Example:

```
// Example :: BleEncodeBits.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


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
```

5.11 Attribute Decoding Functions

Data in a characteristic is stored in a value attribute, a byte array. Multibyte characteristic descriptors content is stored similarly. Those bytes are manipulated in *smart*BASIC 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.

Note: CCCD and SCCD descriptors are special cases as they are defined as having two bytes which are treated as 16-bit integers mapped to INTEGER variables in *smart*BASIC.

5.11.1 BleDecodeS8

FUNCTION

This function reads a single byte in a string at a specified offset into a 32-bit 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, 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$* AS STRING** This references the attribute string from which the function reads.
nData	**byRef nData AS INTEGER** This references an integer to be updated with the 8-bit data from attr$, 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.

Example:

```
// Example :: BleDecodeS8.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 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
```

5.11.2 BleDecodeU8

FUNCTION

This function reads a single byte in 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, 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.

Example:

```
// Example :: BleDecodeU8.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


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
```

5.11.3 BleDecodeS16

FUNCTION

This function reads two bytes in a string at a specified offset into a 32-bit 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.

Example:

```
// Example :: BleDecodeS16.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples
 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
```

5.11.4 BleDecodeU16

This function reads two 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.

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.

Example:

```
// Example :: BleDecodeU16.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


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
```

5.11.5 BleDecodeS24

FUNCTION

This function reads three bytes in a string at a specified offset into a 32-bit 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.

Example:

```
// Example :: BleDecodeS24.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 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
```

5.11.6 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.

Example:

```
// Example :: BleDecodeU24.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 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
```

5.11.7 BleDecode32

FUNCTION

This function reads four bytes in a string at a specified offset into a 32-bit 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.

Example:

```
// Example :: BleDecode32.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 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
```

5.11.8 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.

Example:

```
// Example :: BleDecodeFloat.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


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
```

5.11.9 BleDecodeSFLOAT

FUNCTION

This function reads two bytes in a string at a specified offset into a couple of 32-bit integer variables. The decoding results in two variables, the 12-bit signed mantissa and the 4-bit signed exponent. If the offset points beyond the end of the string then this function fails.

BLEDECODESFLOAT (attr$, nMantissa, nExponent, nIndex)

<table>
<tr><td>Returns</td><td>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.</td></tr>
<tr><td colspan="2">Arguments:</td></tr>
<tr><td>attr$</td><td>byRef attr$ AS STRING
This references the attribute string from which the function reads.</td></tr>
<tr><td>nMantissa</td><td>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:
<table>
<tr><td>0x007FFFFF</td><td>NaN (Not a Number)</td></tr>
<tr><td>0x00800000</td><td>NRes (Not at this resolution)</td></tr>
<tr><td>0x007FFFFE</td><td>+ INFINITY</td></tr>
<tr><td>0x00800002</td><td>- INFINITY</td></tr>
</table></td></tr>
</table>

	0x00800001	Reserved for future use
nExponent	**byRef nExponent AS INTEGER** This is updated with the 4-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.	

Example:

```
// Example :: BleDecodeSFloat.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 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
```

5.11.10 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 is represented in the source as \DD\07\05\05\0A\1F\18 and the year is be translated into a century and year so that the destination string is \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.

Example:

```
// Example :: BleDecodeTimestamp.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 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
```

5.11.11 BleDecodeSTRING

FUNCTION

This function reads a maximum number of bytes from an attribute string at a specified offset into a destination string. Because the output string can handle truncated bit blocks, this function does not fail.

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 is updated with up to nMaxBytes of data from the index specified. A shorter string is 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$.

Example:

```
// Example :: BleDecodeString.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 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$=
```

5.11.12BleDecodeBITS

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. Because the output bit array can handle truncated bit blocks, this function does not fail.

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$* AS STRING** This references the attribute string from which to read, treated as a bit array. Hence a string of 10 bytes is an array of 80 bits.
nSrcIdx	**byVal *nSrcIdx* AS INTEGER** This is the zero based bit index into the string attr$ from which data is read. For example, 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 to where the data is written.
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.

Example:

```
// Example :: BleDecodeBits.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 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
```

5.12 Bonding and Bonding Database Functions

5.12.1 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 16 devices can be stored in the database and 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 Bluetooth address of the trusted device (and it will be the non-resolvable address if the connection was originally established by the central device using its resolvable key – like iOS devices).
- A 16 byte key, eDIV and eRAND for the long term key, called LTK. Up to 2 instances of this LTK can be stored. One which is supplied by the central device and the other is the one supplied by the peripheral. This means in a connection, the device will check which role (peripheral or central) it is connected as and pick the appropriate key for subsequent encryption requests.
- 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)

5.12.2 Bonding Table Types: Rolling & Persist

The bonding database contains two tables of bonds where both tables have the same structure in terms of what each record can store and from a BLE perspective are equal in meaning.

For the purpose of clarity both in this manual and in smartBASIC, one table is called the 'Rolling' table and the other is called 'Persistent' table.

When a new bonding occurs the information is ALWAYS guaranteed to be saved in the 'Rolling' table, and if it is full, then the oldest 'Rolling' bond is automatically deleted to make space for the new one.

The 'Persistent' table can only be populated by transferring a bond from the 'Rolling' table using the function BleBondingPersistKey.

Use the function BleBondingEraseKey to delete a key and the function will look for it in both tables and when found delete it. There is no need to know which table it belongs to when deleting. The database manager ensures there is only one instance of a bond and so a device cannot occur in both.

The total number of bonds in the 'Rolling' and 'Persistent' tables will always be less than or equal to the capacity of the database which is returned as explained above using AT I 2012 or SYSINFO(2012).

The number of 'Rolling' or 'Persistent' bonds (or maximum capacity) at any time can be obtained by calling the function BleBondingStats. The 'Persistent' total is the difference between the 'total' and 'rolling' variables returned by that routine.

At any time, the capacity of the 'Rolling' table is the difference between the absolute total capacity and the number of bonds in the 'Persistent' table. See the function BleBondingStats which returns information that can be used to determine this.

Bonds in the 'Rolling' table can be transferred to 'Persistent' unless the 'Persist' table is full. The capacity of the 'Persistent' table is returned by AT I 2043 or SYSINFO(2043) and at the time of writing this manual it is 12, which corresponds to 75% of the total capacity.

If a bond exists and it happens to be in the 'Persistent' table and new bonding provides new information then the record is updated.

If a bond exists and it happens to be in the 'Rolling' table and new bonding provides new information then the record is updated and in addition, the age list is updated to that the device is marked the 'youngest' in the age list.

It is expected that a smartBASIC application wanting to manage trusted device will use a combination of the functions : BleBondMngrGetInfo, BleBondingIsTrusted, BleBondingPersistKey and BleBondingEraseKey.

5.12.3 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.

5.12.3.1 Events and Messages

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

MsgId	Description
10	A new bond has been successfully created
16	The device has successfully connected to a bonded master
17	The bonding information in the bonding database have been updated
22	Adding the paired device and its information to the bonding database has failed

5.12.4 BleBondingStats

FUNCTION

This function retrieves statistics of the bonding manager which consists of the total capacity as the return value and the rolling and total bonds via the arguments. By implication, the number of persistent bonds is the difference between nTotal and nRolling.

BLEBONDINGSTATS (nRolling, nPersistent)

Returns	The total capacity of the database
Arguments:	
nRolling	**byREF *nRolling* AS INTEGER** On return, this integer contains the total number of bonds in the rolling database.
nPersistent	**byREF *nPersistent* AS INTEGER** On return, this integer contains the total number of bonds in the persistent database.

Example:

```
// Example :: BleBondingStats.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


dim rc, nRoll, nPers
print "\n:Bonding Manager Database Statistics:"
print "\nCapacity:  ","", BleBondingStats(nRoll, nPers)
print "\nRolling: ","",nRoll
print "\nPersistent: ",nPers
```

Expected Output:

```
:Bonding Manager Database Statistics:
Capacity:         16
Rolling:          2
Persistent:             0
```

BLEBONDINGSTATS is a built-in function.

5.12.5 BleBondingPersistKey

FUNCTION

This function is used to make a bonding link key persistent. Its entry is moved from the rolling database to the persistent database so that it is never automatically overwritten.

BLEBONDINGPERSISTKEY (bdAddr$)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
bdAddr$	**byREF *bdAddr$* AS STRING** Bluetooth address in big endian. Must be exactly seven bytes long.

Example:

```
// Example :: BleBondingPersistKey.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


dim rc, i, j, k, adr$, inf


'//Loop through the bonding manager. Make all entries persistent
for i=0 to BleBondingStats(j,k)
    rc=BleBondMngrGetInfo(i,adr$,inf)
    if rc==0 then
        rc=BleBondingPersistKey(adr$)
        print "\n(";i;") : ";StrHexize$(adr$);" Now Persistent"
    endif
next
```

Expected Output:

```
(0)  : 01F63627A60BEA Now Persistent
(1)  : 01D8CFCF14498D Now Persistent
```

BLEBONDINGPERSISTKEY is a built-in function.

5.12.6 BleBondingIsTrusted

FUNCTION

This function is used to check if a device identified by the address is a trusted device which means it exists in the bonding database.

BLEBONDINGISTRUSTED (addr$, fAsCentral, keyInfo, rollingAge, rollingCount)

Returns	INTEGER: Is 0 if not trusted, otherwise it is the length of the long term key (LTK)
Arguments	
addr$	**byRef addr$ AS STRING**

	This is the address of the device for which the bonding information is to be checked. If this a resolvable address and the device is trusted, then on exit this variable is replaced with the static address that was supplied at pairing time.
fAsCentral	Set to 0 if the device is to be trusted as a peripheral and non-zero if to be trusted as central.
keyInfo	This is a bit mask with bit meanings as follows: This specifies the write rights and shall have one of the following values: Bit 0 Set if MITM is authenticated Bit 1 Set if it is a rolling bond and can be automatically deleted if the database is full and a new bonding occurs Bit 2 Set if an IRK (identity resolving key) exists Bit 3 Set if a CSRK (connection signing resolving key) exists Bit 4 Set if LTK as slave exists Bit 5 Set if LTK as master exists
rollingAge	If the value is <= 0, this is not a rolling device. 1 implies it is the newest bond, 2 implies it is the second newest bond, and so on.
rollingCount	On exit this will contain the total number of rolling bonds. This provides some context with regards to how old this device is compared to other bonds in the rolling group.

Example:

```
// Example :: BleBondingIsTrusted.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

DIM rc, addr$, realaddr$, Central, KeyInfo, Age, Count

addr$ = "000016A4123456"
realaddr$ = strdehexize$(addr$)

print "Address: ";addr$;"\n"
rc = BleBondingIsTrusted(realaddr$, Central, KeyInfo, Age, Count)
print "Is Trusted: ";rc;"\n"

if (rc != 0) then
    //Output details
    if (Central == 0) then
        print "Peripheral"
    elseif (Central == 1) then
        print "Central"
    endif
    print " device, keyinfo: ";integer.b'KeyInfo
    print " Age: ";Age;" Count: ";count;"\n"
endif
```

Expected Output: (if bond is present)

```
Address: 000016A4123456
Is Trusted: 16
Peripheral device, keyinfo: 00000000000000000000000000110110 Age: 1 Count: 1
```

Expected Output: (if there is no bond)

```
Address: 000016A4123456
Is Trusted: 0
```

BLEBONDINGISTRUSTED is a built-in function.

5.12.7 BleBondingEraseKey

FUNCTION

This function is used to erase a link key from the database for the address specified.

BLEBONDINGERASEKEY (bdAddr$)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
bdAddr$	**byREF *bdAddr$* AS STRING** Bluetooth address in big endian. Must be exactly seven bytes long.

Example:

```
// Example :: BleBondingEraseKey.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


dim rc, i, adr$, inf


//delete link key at index 0
rc=BleBondMngrGetInfo(0,adr$,inf)        //get the BT address
rc=BleBondingEraseKey(adr$)
if rc==0 then
    print "\nLink key for device ";StrHexize$(adr$);" erased"
else
    print "\nError erasing link key ";integer.h'rc
endif
```

Expected Output:

```
Link key for device 01FA84D748D903 erased
```

BLEBONDINGERASEKEY is a built-in function.

5.12.8 BleBondingEraseAll

FUNCTION

This function is used to erase all bondings in the database.

Note: Calling this function when the connection supervision timeout is 100ms may cause a disconnection. The reason for this is that calling this function may prevent the radio sending ACK packets to the remote device within the supervision timeout. The supervision timeout is set at BleConnect or at BleSetCurConnParams.

BLEBONDINGERASEALL ()

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.

Example:

```
// Example :: BleBondingEraseAll.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


dim rc


//Erase all bondings in database
rc=BleBondingEraseAll()
if rc==0 then
    print "\nBonding database cleared"
endif
```

Expected Output:

```
Bonding database cleared
```

BLEBONDINGERASEALL is a built-in function.

5.12.9 BleBondMngrGetInfo

FUNCTION

This function retrieves the Bluetooth 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 change the position in the database.

BLEBONDMNGRGETINFO (nIndex, addr$, nExtraInfo)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
nIndex	**byVal nIndex AS INTEGER** This is an index into the database, 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 Bluetooth 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 for internal use and should be treated as opaque data. Bit 17 is set if the IRK (Identity Resolving Key) exists for the trusted device and bit 18 is set if the CSRK (Connection Signing Resolving Key) exists for the trusted device.

Example:

```
// Example :: BleBondMngrGetInfo.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 #define BLE_INV_INDEX           24619
 DIM rc, addr$, exInfo
 rc = BleBondMngrGetInfo(0,addr$,exInfo)  //Extract info of device at index 0
```

```
IF rc==0 THEN
    PRINT "\nBluetooth address: ";addr$
    PRINT "\nInfo: ";exInfo
ELSEIF rc==BLE_INV_INDEX   THEN
    PRINT "\nInvalid index"
ENDIF
```

Expected Output when valid entry present in database:

```
Bluetooth address: \00\BC\B1\F3x3\AB
Info: 97457
```

Expected Output with invalid index:

```
Invalid index
```

5.13 Security Manager Functions

The following is a high level overview of Bluetooth Low Energy pairing/authentication and it is encouraged that the reader access resources on the internet which give further details, like for example https://developer.bluetooth.org/TechnologyOverview/Pages/LE-Security.aspx

Pairing is the process of exchanging security keys between two connected devices to establish trust and authenticate the connection between the two devices. The exchanged keys can be used to encrypt the connection to safeguard against passive eavesdropping. Pairing in versions 4.0 and 4.1 of the Buetooth core specification is exposed through Secure Simple Pairing, which is now referred to as Legacy pairing. Security is now greatly enhanced with the release of the 4.2 specification due to the introduction of the LE Secure Connections pairing model. In this model, Elliptic Curve Diffie-Hellman (ECDH) algorithm is used for the key exchange process where the two parties can compute a shared secret without exchanging it over the BLElink.

This section describes routines which manage all aspects of BLE security such as IO capabilities, Passkey exchange, OOB data, and bonding requirements.

5.13.1 Events and Messages

5.13.1.1 EVBLEMSG

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

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 is a number in the range 0 to 999999 and 2 for OOB key which is a 16 byte key.
18	The connection has been successfully encrypted
20	The connection has been unencrypted
26	Authentication/pairing has failed
27	LE Secure Connections pairing has been successfully established

MsgId	Description
28	OOB data has been requested by the peer device during LE Secure Connections pairing

To submit a passkey, use the function BLESECMNGRPASSKEY.

5.13.1.2 EVLESCKEYPRESS

This event message is thrown when the BL652 receives notifications that the peer device is performing keypresses during passkey entry in an LE Secure Connections pairing. This event comes with two parameters:

- Connection handle
- Keypress type

Keypress Type	Description
0	Passkey entry started
1	Passkey digit entered
2	Passkey digit erased
3	Passkey cleared
4	Passkey entry completed

See example for BleSecMngrLescKeypressNotify.

5.13.1.3 EVBLE_PASSKEY

This event is thrown when there is BLE pairing in progress that requires the entry/acceptance of a passkey. The event includes the following parameters:-

- Connection handle
- The passkey that is thrown by the stack, which should then be accepted or entered by the remote device.
- Flags parameter that is reserved for future use.

Example:

```
//Example :: BleSecMngrPasskey.sb

// Definitions
#define BLE_EVBLEMSGID_CONNECT                         0 // nCtx = connection handle
#define BLE_EVBLEMSGID_DISCONNECT                      1 // nCtx = connection handle
#define BLE_EVBLEMSGID_NEW_BOND                       10 // nCtx = connection handle
#define BLE_EVBLEMSGID_UPDATED_BOND                   17 // nCtx = connection handle
#define BLE_EVBLEMSGID_ENCRYPTED                      18 // nCtx = connection handle
#define BLE_EVBLEMSGID_AUTHENTICATION_FAILED          26 // nCtx = connection handle
#define BLE_EVBLEMSGID_LESC_PAIRING                   27 // nCtx = connection handle

// Variable Declaration
DIM rc, connHandle
DIM addr$ : addr$=""

//------------------------------------------------------------------------------
// Ble event handler
//------------------------------------------------------------------------------
FUNCTION HandlerBleMsg(BYVAL nMsgId AS INTEGER, BYVAL nCtx AS INTEGER) AS INTEGER
    SELECT nMsgId
        CASE BLE_EVBLEMSGID_CONNECT
            connHandle = nCtx
            PRINT "## Ble Connection :: Handle=";integer.h' nCtx;"\n"

        CASE BLE_EVBLEMSGID_DISCONNECT
            PRINT "## Disconnected :: Handle=";integer.h' nCtx;"\n"
            EXITFUNC 0
```

```
        CASE BLE_EVBLEMSGID_ENCRYPTED
            PRINT "## Encrypted Connection :: Handle=";integer.h' nCtx;"\n"
        CASE BLE_EVBLEMSGID_NEW_BOND
            PRINT "## New Bond :: Handle=";integer.h' nCtx;"\n"
        CASE BLE_EVBLEMSGID_LESC_PAIRING
            PRINT "## LESC Pairing :: Handle=";integer.h' nCtx;"\n"
        CASE   BLE_EVBLEMSGID_AUTHENTICATION_FAILED
            PRINT "## Pairing Failed :: Handle=";integer.h' nCtx;"\n"
        CASE ELSE
            // Do nothing
    ENDSELECT
ENDFUNC 1

//--------------------------------------------------------------------------------
// Pairing attempt in progress - Passkey needs to be displayed
//--------------------------------------------------------------------------------
Function HandlerBlePasskey(BYVAL nConnHandle, BYVAL nPasskey, BYVAL nFlags)
    // The following passkey should be entered by remote
    print "## Pairing Attempt :: Handle=";integer.h' nConnHandle;"\n"
    print "## Please enter the following passkey: ";nPasskey;"\n"
Endfunc 1

//--------------------------------------------------------------------------------
// Enable synchronous event handlers
//--------------------------------------------------------------------------------
ONEVENT  EVBLEMSG        CALL HandlerBleMsg
ONEVENT  EVBLE_PASSKEY   CALL HandlerBlePasskey

// Set pairing IO capability to Display.
// Remote pairing IO capability should be keyboard
rc = BleSecMngrIoCap(3)

// Start advertising
IF BleAdvertStart(0,addr$,25,60000,0)==0 THEN
    PRINT "## Adverts Started\n"
    PRINT "## Make a connection to the BL652\n"
ELSE
    PRINT "## Advertisement not successful\n"
ENDIF

WAITEVENT
```

Expected Output:

```
## Adverts Started
## Make a connection to the BL652
## Ble Connection :: Handle=0001FF00
## Pairing Attempt :: Handle=0001FF00
## Please enter the following passkey: 242652
## Encrypted Connection :: Handle=0001FF00
## LESC Pairing :: Handle=0001FF00
## New Bond :: Handle=0001FF00
```

5.13.2 BleSecMngrLescPairingPref

FUNCTION

This function is used to set LE Secure connections to be the preferred pairing model. Both devices must support LE Secure Connections in order for it to be used during pairing.

BLESECMNGRLESCPAIRINGPREF (nLescPairingPref)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
nLescPairingPref	**byVal nJustWorksConf AS INTEGER**. If set to 0, legacy pairing is used. If set to 1, LE Secure Connections with diffie-hellman key exchange is used as the pairing model. The default pairing model is LE Secure Connections pairing.

See example for BlePair().

5.13.3 BlePair

FUNCTION

This routine is used to induce the module to pair with the peer and to specify whether to bond with the peer by storing pairing information in the bonding manager. This function is likely to be used if a write attempt to an attribute fails with a status code such as 0x105. See EvAttrWrite and EvAttrRead.

BLEPAIR (hConn, nSave)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
hConn	**byRef *hConn*** AS INTEGER. This is the connection handle provided in the EVBLEMSG(0) message which informs the stack that a connection had been established.
nSave	**byVal *nSave* AS INTEGER** This flag sets whether or not to bond. **Value** – **Description** 0 – Do not store pairing information (don't bond) 1 – Store pairing information (bond)

Example:

```
// Example :: BlePair.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


dim rc, pr$, hC, hDesc
dim s$ : s$ = "\02\00"      //value to write to cccd to enable indications


//This example app was tested with a BL652 running the health thermometer sensor sample
app which requires bonding.
//It connects, tries to read from the temperature characteristic and then initiates a
bonding procedure when it fails.


#define GATT_SERVER_ADDRESS          "\01\F6\36\27\A6\0B\EA"
#define AUTHENTICATION_REQUIRED      0x0105


#define SERVICE_UUID                 0x1809
```

```
#define CHAR_UUID                      0x2a1c
#define DESC_UUID                      0x2902

'//------------------------------------------------------------------------------
'// For debugging
'// --- rc = result code
'// --- ln = line number
'//------------------------------------------------------------------------------
Sub AssertRC(rc,ln)
    if rc!=0 then
        print "\nFail :";integer.h' rc;" at tag ";ln
    endif
EndSub

'//------------------------------------------------------------------------------
'// This handler is called when there is a significant BLE event
'//------------------------------------------------------------------------------
function HndlrBleMsg(byval nMsgId as integer, byval nCtx as integer)
    select nMsgId
        case 0
            hC = nCtx
            print "\nConnected, Finding Temp Measurement Char"
            rc=BleGattcFindDesc(nCtx,  BleHandleUuid16(SERVICE_UUID), 0,
BleHandleUuid16(CHAR_UUID), 0, BleHandleUuid16(DESC_UUID), 0)
            AssertRC(rc,35)
        case 1
            print "\n\n --- Disconnected"
        case 10
            print "\nNew bond created"
            print "\n\nAttempting to enable indications again"
            rc=BleGattcWrite(hC, hDesc, s$)
            AssertRC(rc,58)
        case 11
            print "\nPair request: Accepting"
            rc=BleAcceptPairing(hC,1)
            AssertRC(rc,52)
            print "\nPairing in progress"
        case 17
            print "\nNew pairing/bond has replaced old key"
        case 18
```

```
            print "\nConnection now encrypted"
        case else
    endselect
endfunc 1



'//------------------------------------------------------------------------------
'// Called after BleGattcFindDesc returns success
'//------------------------------------------------------------------------------
function HndlrFindDesc(hConn, hD)
    if hD==0 then
        print "\nCCCD not found"
        exitfunc 0
    endif

    hDesc = hD
    print "\nTemp Measurement Char CCCD Found. Attempting to enable indications"
    rc=BleGattcWrite(hConn, hDesc, s$)
    AssertRC(rc,58)
endfunc 1

'//------------------------------------------------------------------------------
'// Called after BleGattcRead returns success
'//------------------------------------------------------------------------------
function HndlrAttrWriteExit(hConn, hAttr, nSts)
endfunc 0

'//------------------------------------------------------------------------------
'// Called after BleGattcRead returns success
'//------------------------------------------------------------------------------
function HndlrAttrWrite(hConn, hAttr, nSts)
    if nSts == 0 then
        print "\nIndications enabled"
        print "\nDisabling indications"
        s$ = "\00\00"
        rc=BleGattcWrite(hC, hDesc, s$)
        onevent evattrwrite call HndlrAttrWriteExit
        exitfunc 1
```

```
    elseif nSts == AUTHENTICATION_REQUIRED then
        print "\n\nAuthentication required."
        '//bond with the peer
        rc=BlePair(hConn, 1)
        AssertRC(rc,75)
        print " Bonding..."
    endif
endfunc 1

//******************************************************************************
// Equivalent to main() in C
//******************************************************************************
rc=BleLescPairingPref(1)        //set the pairing model to be LE Secire Connections
pairing
rc=BleSecMngrIoCap(1)           //set io capability to Yes/No

rc=BleGattcOpen(0,0)
pr$ = GATT_SERVER_ADDRESS
rc=BleConnect(pr$, 10000, 25, 100, 30000000)
AssertRC(rc,91)

//------------------------------------------------------------------------------
// Enable synchronous event handlers
//------------------------------------------------------------------------------
onevent evblemsg   call HndlrBleMsg
onevent evfinddesc call HndlrFindDesc
onevent evattrwrite call HndlrAttrWrite

waitevent

print "\nExiting..."
```

Expected Output:

```
Connected, Finding Temp Measurement Char
Temp Measurement Char CCCD Found. Attempting to enable indications

Authentication required. Bonding...
Pair request: Accepting
Pairing in progress
Connection now encrypted
New bond created

Attempting to enable indications again
Indications enabled
```

```
Disabling indications
Exiting...
```

5.13.4 BleSecMngrIoCap

FUNCTION

This function sets the user I/O capability for subsequent pairings and is used to determine if the pairing is authenticated. This is 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 4.2 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. The most typical value is 0x0000, indicating 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 and Display (authenticated pairing)

Example:

```
// Example :: BleSecMngrIoCap.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 PRINT BleSecMngrIoCap(1)
```

Expected Output:

```
0
```

See also examples for BleSecMngrPasskey() and BlePair().

5.13.5 BleAcceptPairing

FUNCTION

In legacy pairing the device can choose from Just Works, Passkey Entry, and OOB as the method of pairing depending on the input/output capabilities of the device. With Bluetooth v4.2, LE Secure connections adds the numeric comparison method to the other three. This function is used to accept or decline numeric comparison pairing.

BLEACCEPTPAIRING (nConnHandle, nAccept)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	

nConnHandle	**byVal nConnHandle AS INTEGER**. The handle of the connection for which you are accepting or rejecting a pairing request.
nAccept	**byVal nAccept AS INTEGER**. Set to 0 to reject the numeric comparison pairing request, set to 1 to accept the pairing request.

See example for BlePair().

5.13.6 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 and Messages.

BLESECMNGRPASSKEY (connHandle, nPassKey)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
connHandle	**byVal connHandle AS INTEGER**. The connection handle as received via the EVBLEMSG event with msgId set to 0.
nPassKey	**byVal *nPassKey* AS INTEGER.** The passkey to submit to the stack. Submit a value outside the range 0 to 999999 to reject the pairing.

Example:

```
// Example :: BleSecMngrPasskey.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

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

'// Called when data arrives through the UART - PIN
FUNCTION HandlerUartRxPIN()
    i = UartReadMatch(pin$,13)
    if i !=0 then
        pin$ = StrSplitLeft$(pin$,i-1)
        if strcmp(pin$,"quit")==0 || strcmp(pin$,"exit")==0 then
            rc=BleDisconnect(connHandle)
            exitfunc 0

        elseif BleSecMngrPassKey(connHandle,StrValDec(pin$))==0 then
            print "\nPasskey: ";pin$
            OnEvent  EVUARTRX  disable
        endif
        pin$=""
    endif
ENDFUNC 1
```

```
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 10
            PRINT "\n--- New bond"
        CASE 11
            PRINT "\n +++ Auth Key Request, type=";nCtx
            PRINT "\nEnter the pass key and Press Enter:\n"
            onevent evuartrx call HandlerUartRxPIN
        CASE 17
            print "\nNew pairing/bond has replaced old key"
        CASE ELSE
    ENDSELECT
ENDFUNC 1

ONEVENT  EVBLEMSG  CALL HandlerBleMsg

rc=BleSecMngrIoCap(2)  //Set i/o capability - Keyboard Only (authenticated pairing)
IF BleAdvertStart(0,addr$,25,0,0)==0 THEN
    PRINT "\nAdverts Started\n"
    PRINT "\nPair with the module"
ELSE
    PRINT "\n\nAdvertisement not successful"
ENDIF

WAITEVENT
```

Expected Output:

```
Adverts Started

Pair with the module
--- Ble Connection,    2782
 +++ Auth Key Request, type=1
Enter the pass key and Press Enter:
904096

Passkey: 904096
--- New bond
--- Disconnected 2782
```

5.13.7 BleSecMngrLescKeypressEnable

FUNCTION

This function is used to enable keypress notifications so that during LE secure connections, when keys are entered during passkey entry pairing, notifications can be sent or received to or from the peer device therefore enhancing protection against man in the middle attacks.

BLESECMNGRLESCKEYPRESSENABLE (nEnable)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
nEnable	**byVal nEnable AS INTEGER**. 0 to disable keypress notifications, 1 to enable keypress notifications

Example:

```
// Example :: BleSecMngrLescKeypressNotify.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

// Enable keypress notifications
rc = BLESECMNGRLESCKEYPRESSENABLE(1)
if rc == 0 THEN
    PRINT "Keypress notifications enabled\n"
endif
```

5.13.8 BleSecMngrLescKeypressNotify

FUNCTION

This function is used to send keypress notifications to the peer device during passkey entry in LE Secure Connections pairing.

BLESECMNGRLESCKEYPRESSNOTIFY (connHandle, nKeypressType)

<table>
<tr><td>Returns</td><td colspan="2">INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.</td></tr>
<tr><td colspan="3">Arguments:</td></tr>
<tr><td>connHandle</td><td colspan="2">byVal connHandle AS INTEGER.
This is the handle of the connection on which pairing is being perfermed</td></tr>
<tr><td rowspan="6">nKeypressType</td><td colspan="2">byRef nKeypressType AS STRING.
This is the type of the keypress, and can be one of the following values:</td></tr>
<tr><td>0</td><td>Passkey entry started</td></tr>
<tr><td>1</td><td>Passkey digit entered</td></tr>
<tr><td>2</td><td>Passkey digit erased</td></tr>
<tr><td>3</td><td>Passkey digit cleared</td></tr>
<tr><td>4</td><td>Passkey entry completed</td></tr>
</table>

Example:

```
// Example :: BleSecMngrLescKeypressNotify.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

// Keypress Types
#define BLE_GAP_KP_NOT_TYPE_PASSKEY_START       0x00   // Passkey entry started.
#define BLE_GAP_KP_NOT_TYPE_PASSKEY_DIGIT_IN    0x01   // Passkey digit entered.
#define BLE_GAP_KP_NOT_TYPE_PASSKEY_DIGIT_OUT   0x02   // Passkey digit erased.
```

```
#define BLE_GAP_KP_NOT_TYPE_PASSKEY_CLEAR        0x03    // Passkey cleared.
#define BLE_GAP_KP_NOT_TYPE_PASSKEY_END          0x04    // Passkey entry completed.

// Global variable
dim rc      // Result Code
dim ghConn  // Global connection handle

//==============================================================================
// This handler is called when data has arrived at the serial port
//==============================================================================
function HandlerUartRxCmd() as integer

    dim StrKey$     // key entered

    // Now read a single character from the UART buffer
    rc = UartReadN(StrKey$, 1)

    if (strcmp(StrKey$,"\r")==0) THEN
        // Let the user know that we are done with keypresses, then send passkey
        rc = BleSecMngrLescKeypressNotify(ghConn,BLE_GAP_KP_NOT_TYPE_PASSKEY_END)
    endif

endfunc 1

'//******************************************************************************
'// Equivalent to main() in C
'//******************************************************************************

//------------------------------------------------------------------------------
// Enable synchronous event handlers
//------------------------------------------------------------------------------
OnEvent  EVUARTRX                  call HandlerUartRxCmd

// Enable keypress notifications
rc = BLESECMNGRLESCKEYPRESSENABLE(1)
// Set LE Secure Connections to be the preffered pairing model
rc = BLESECMNGRLESCPAIRINGPREF(1)
// Set IO capability to 2: Keyboard only
rc = BLESECMNGRIOCAP(2)

WaitEvent
```

5.13.9 BleSecMngrOOBPref

FUNCTION

This function is used to set a flag to indicate to the peer during a pairing that OOB pairing is preferred.

BLESECMNGROOBPREF(nOobPreferred)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
nOobPreferred	**byVal connHandle AS INTEGER.** If set to 0, OOB pairing will not have preference. If set to 1, OOB pairing will be preferred.

Example:

```
// Example :: BleSecMngrOobPref.sb
```

```
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

dim rc
rc = BleSecMngrOobPref(1)
IF (rc == 0) THEN
    PRINT "OOB Pairing preference has been set."
ENDIF
```

Expected Output:

```
OOB Pairing preference has been set.
```

5.13.10 BleSecMngrOOBKey

FUNCTION

This function submits an OOB (Out Of Band) key to the underlying stack during a legacy pairing procedure when prompted by the EVBLEMSG with msgId set to 11 and the key type nCtx is 2, OOB. See Events & Messages.

BLESECMNGROOBKEY (connHandle, oobKey$)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating 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.
oobKey$	**byRef *oobKey$* AS STRING.** This is the OOB key to submit to the stack. Submit a 16 byte string, or a string of a different length to reject the request.

Example:

```
// Example :: BleSecMngrOOBKey.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


DIM rc, connHandle
DIM addr$ : addr$=""
DIM oob$ : oob$ = "\11\22\33\44\55\66\77\88\99\00\aa\cc\bb\dd\ee\ff"

#define OOB_KEY     2

FUNCTION HandlerBleMsg(BYVAL nMsgId AS INTEGER, BYVAL nCtx AS INTEGER) AS INTEGER
    SELECT nMsgId
        CASE 0
            connHandle = nCtx
            PRINT "\nBle Connection ",nCtx
        CASE 1
```

```
            PRINT "\nDisconnected ";nCtx;"\n"
            EXITFUNC 0
        CASE 10
            PRINT "\n--- New bond"
        CASE 11
            PRINT "\n +++ Auth Key Request, type=",nCtx
            if nCtx == OOB_KEY then
                rc=BleSecMngrOobKey(connHandle,oob$)
                print "\nOOB Key ";StrHexize$(oob$);" was used"
            endif

        CASE ELSE
            PRINT "\nUnknown Ble Msg"
    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 BL652"
ELSE
    PRINT "\n\nAdvertisement not successful"
ENDIF

WAITEVENT
```

Expected Output:

```
Adverts Started

Make a connection to the BL652
Ble Connection,  1655
 +++ Auth Key Request, type=2
OOB Key 112233445566778899 11AACCBBDDEEFF was used
--- New bond
Disconnected 1655
```

5.13.11 BleSecMngrLescOwnOobDataGet

FUNCTION

This function retrieves the OOB data that should be given to the peer device. The peer device should then use this as the out of band data during LE Secure Connections pairing. The OOB data is regenerated everytime this function is called.

BLESECMNGRLESCOWNOOBDATAGET (addr$ oobHash$, oobRand$)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
addr$	**byRef addr$ AS INTEGER**. The Bluetooth address of the local device that should be used by the remote device during LE Secure Connections pairing
oobHash$	**byRef *oobHash$* AS STRING.** The OOB hash of the local device that should be used by the remote device during LE Secure Connections pairing
oobRand$	**byRef *oobRand$* AS STRING.** The OOB randomiser of the local device that should be used by the remote device during LE Secure Connections pairing

5.13.12 BleSecMngrLescPeerOobDataSet

FUNCTION

This function is used during the pairing process to send the remote OOB data via the Bluetooth link. When EVBLEMSG is received with ID 28, indicating that the remote device is requesting it's OOB data to be sent, this function should be used to send the data that was previously exchanged out of band.

BLESECMNGRLESCPEEROOBDATASET (addr$ oobHash$, oobRand$)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
addr$	**byRef addr$ AS INTEGER**. The Bluetooth address of the remote device that was given out of band.
oobHash$	**byRef *oobHash$* AS STRING.** The OOB hash of the remote device that was given out of band.
oobRand$	**byRef *oobRand$* AS STRING.** The OOB randomiser of the remote device that was given out of band.

Example:

```
// Example :: BleSecMngrLescPeerOobDataSet.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


// In this example, the OOB data is exchanged over the UART in the form
// OOB_ADDRESS OOB_HASH OOB_RAND\r
// e.g. 000016A4B75201 63F6E834009C368612724FBC3253DDE2
8311CD946F30C785DD7EA83038A5221D\r


//BLE EVENT MSG IDs
#define BLE_EVBLEMSGID_CONNECT                              0 // msgCtx = connection handle
#define BLE_EVBLEMSGID_DISCONNECT                           1 // msgCtx = connection handle
13
#define BLE_EVBLEMSGID_ENCRYPTED                           18 // msgCtx = connection handle
#define BLE_EVBLEMSGID_AUTHENTICATION_FAILED               26 // msgCtx = connection handle
#define BLE_EVBLEMSGID_LESC_PAIRING                        27 // msgCtx = connection handle
#define BLE_EVBLEMSGID_LESC_OOB_REQUEST                    28 // msgCtx = connection handle
```

```
//Global defines

DIM rc, stRsp$

//==============================================================================
// This subroutine is called when Out of Band LESC pairing is in progress
//==============================================================================
sub HandleOobReq()

    DIM OobData$, OobAddr$, OobHash$, OobRand$
    // Get our local OOB data
    rc = BleSecMngrLescOwnOobDataGet(OobAddr$, OobHash$, OobRand$)
    // Hexize the data
    OobAddr$ = StrHexize$(OobAddr$)
    OobHash$ = StrHexize$(OobHash$)
    OobRand$ = StrHexize$(OobRand$)
    // Construct a string of the retreived data
    OobData$ = OobAddr$ + " " + OobHash$ + " " + OobRand$ + "\r"
    // Finally send the OOB data over UART
    rc = UartWrite(OobData$)
    print "Local OOB data sent over UART\n"


endsub

//==============================================================================
// This handler is called when there is a BLE message
//==============================================================================
function HandlerBleMsg(BYVAL nMsgId AS INTEGER, BYVAL nCtx AS INTEGER) as integer
  dim hz

  select nMsgId
  case BLE_EVBLEMSGID_CONNECT
    print " --- Connect: (";integer.h' nCtx;")\n"

  case BLE_EVBLEMSGID_DISCONNECT
    print " --- Disconnect: (";integer.h' nCtx;")\n"

  case BLE_EVBLEMSGID_ENCRYPTED
    print " +++ Encrypted Connection: (";integer.h' nCtx;")\n"

  case BLE_EVBLEMSGID_LESC_PAIRING
    print " +++ LESC pairing: (";integer.h' nCtx;")\n"

  case BLE_EVBLEMSGID_LESC_OOB_REQUEST
    print " +++ LESC OOB Request: (";integer.h' nCtx;")\n"
    HandleOobReq()

  case   BLE_EVBLEMSGID_AUTHENTICATION_FAILED
    print " +++ Auth Failed: (";integer.h' nCtx;"\n"

  case else

  endselect
endfunc 1

//==============================================================================
// This handler is called when data has arrived at the serial port
//==============================================================================
function HandlerUartRx() as integer

    dim nMatch
    dim OobData$, OobAddr$, OobHash$, OobRand$
```

```
    // read UART data until carriage return and save it into stRsp$
    nMatch=UartReadMatch(stRsp$,13)
    if nMatch!=0 then
        // Get the hash and randomiser from the input string
        OobData$ = strsplitleft$(stRsp$, nMatch)
        rc = ExtractStrToken(OobData$,OobAddr$)
        rc = ExtractStrToken(OobData$,OobHash$)
        rc = ExtractStrToken(OobData$,OobRand$)

        // Dehexize the data first
        OobAddr$ = StrDeHexize$(OobAddr$)
        OobHash$ = StrDeHexize$(OobHash$)
        OobRand$ = StrDeHexize$(OobRand$)
        // Now Send the remote OOB data over the BLE link
        rc = BleSecMngrLescPeerOobDataSet(OobAddr$, OobHash$, OobRand$)
        if rc==0 THEN
        print "Remote OOB data received from UART and sent over the BLE link\n"
        endif
    endif

endfunc 1

//--------------------------------------------------------------------------------
// Enable synchronous event handlers
//--------------------------------------------------------------------------------
OnEvent  EVBLEMSG                 call HandlerBleMsg
OnEvent  EVUARTRX                 call HandlerUartRx


// Initialise LE adverts
dim addr$
rc = BleAdvertStart(0,addr$,100,30000,0)
// Enable LESC pairing
rc = BleSecMngrLescPairingPref(1)

//--------------------------------------------------------------------------------
// Wait for a synchronous event.
// An application can have multiple <WaitEvent> statements
//--------------------------------------------------------------------------------
WaitEvent
```

Expected Output:

```
--- Connect: (0001FF00)
+++ LESC OOB Request: (0001FF00)
Local OOB data sent over UART
Remote OOB data received from UART and sent over the BLE link
+++ Encrypted Connection: (0001FF00)
+++ LESC pairing: (0001FF00)
```

5.13.13 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. The most typical value is 0x0000, indicating 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.

Example:

```
// Example :: BleSecMngrKeySizes.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


PRINT BleSecMngrKeySizes(8,15)
```

Expected Output:

```
0
```

5.13.14 BleSecMngrBondReq

FUNCTION

This function is used to enable or disable bonding when pairing. If enabled, and if your application requires pairing, a peer device only needs to pair with this module once. If disabled, the device needs to pair every time it connects to the module.

BLESECMNGRBONDREQ (nBondReq)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
nBondReq	**byVal *nBondReq AS* INTEGER**. 0 – Disable 1 – Enable

Example:

```
// Example :: BleSecMngrBondReq.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


IF BleSecMngrBondReq(0)==0 THEN
    PRINT "\nBonding disabled"
ENDIF
```

Expected Output:

```
Bonding disabled
```

5.13.15 BleEncryptConnection

FUNCTION

This function is used to encrypt a BLE connection with a device that the module has previously bonded with (the device is present in the bonding manager). The function can only be issued by the central device (i.e. the device that has initiated the connection request).

BLEENCRYPTCONNECTION (nConnHandle, nLtkMinSize, nMitmRequired)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
nConnHandle	**byVal nConnHandle AS INTEGER**. The handle of the connection which is obtained from an EVBLEMSG message with ID 0 indicating that a connection had been established.
nLtkMinSize	**byVal nLtkMinSize AS INTEGER**. The minimum long term key size which must be in the range 7-16.
nMitmRequired	**byVal nMitmRequired AS INTEGER**. Set to 1 if MITM protection is required, 0 if not required.

Example:

```
dim rc, pr$, hC, hDesc
#define GATT_SERVER_ADDRESS         "\01\F6\36\27\A6\0B\EA"

//This example app was tested with a BL652 running the health thermometer sensor sample app
//which the module had previously bonded with.



'//------------------------------------------------------------------------------
'// For debugging
'// --- rc = result code
'// --- ln = line number
'//------------------------------------------------------------------------------
Sub AssertRC(rc,ln)
    if rc!=0 then
        print "\nFail :";integer.h' rc;" at tag ";ln
    endif
EndSub

'//------------------------------------------------------------------------------
'// This handler is called when there is a significant BLE event
'//------------------------------------------------------------------------------
function HndlrBleMsg(byval nMsgId as integer, byval nCtx as integer)
```

```
    select nMsgId
        case 0
            hC = nCtx
            print "\nConnected"
            rc=BleEncryptConnection(hC, 16, 0)
            if rc==0 then
                print "\nEncrypting connection"
            else
                AssertRC(rc,28)
            endif
        case 1
            print "\n\n --- Disconnected"
             exitfunc 0
        case 10
            print "\nNew bond created"

        case 11
            print "\nPair request: Accepting"
            rc=BleAcceptPairing(hC,1)
            AssertRC(rc,52)
            print "\nPairing in progress"
        case 17
            print "\nNew pairing/bond has replaced old key"
        case 18
            print "\nConnection now encrypted"
            rc=BleDisconnect(hC)
        case else
    endselect
endfunc 1

rc=BleSecMngrIoCap(0)           //set io capability to just works
rc=BleSecMngrJustWorksConf(0)   //module will  not wait for confirmation (EVBLEMSG 11)
before just works pairing

pr$ = GATT_SERVER_ADDRESS
rc=BleConnect(pr$, 10000, 25, 100, 30000000)
AssertRC(rc,91)

onevent evblemsg   call HndlrBleMsg
```

```
waitevent

print "\nExiting..."
```

Expected Output:

```
Connected
Encrypting connection
Connection now encrypted

 --- Disconnected
Exiting...
```

5.14 Virtual Serial Port Service – Managed

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.

5.14.1 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 115200. 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. Additionally, the device name advertised by the VSP service can be configured using the AT+CFGEX command, by default the VSP name is "LAIRD BL652".

It is also possible to configure the command mode VSP by providing a $autorun$ application which launchs after reset automatically. In this application the baudrate, GAP service, VSP Service and advertising can be configured and adverts started. Once done, given the autorun application does not have a WAITEVENT statement it falls into command mode and that VSP configuration will be operational.

A sample autorun application is as follows:

```
//******************************************************************************
// Laird (c) 2015
//
// This application is meant to autorun on power up and so is named apprpriately.
// It PURPOSELY does not have a WAITEVENT statement at the end and so will exit
// to command mode, where the VSP fucntionality will continue to operate.
//
// +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// When UwTerminal downloads the app it will store it as $autorun$
// +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
//
//******************************************************************************


//******************************************************************************
// Debugging
//******************************************************************************
#set $cmpif,0xFFFFFFFF                 //set to 0 to disable all debugging

//******************************************************************************
// Definitions
//******************************************************************************

//------------------------------------------------------------------------------
// UART config
//------------------------------------------------------------------------------
#define UARTBAUD                        9600
#define UARTBUFLENRX                    0  //default
```

```
#define UARTBUFLENTX                    0  //default
#define UARTOPTIONS                     "CN81H"

//------------------------------------------------------------------------------
// GAP Service
//------------------------------------------------------------------------------
//DeviceName
#define GAPDEVNAME                      "autoVSP"
//DeviceName Writeable in Gap Service
#define GAPNAME_WRITEABLE               0
//Appearance in Gap Service (see BT Spec for adopted values) 512=Custom
#define GAPAPPEARANCE                   512
//Minimum Connection Interval in microseconds
#define GAPMINCONNINTus                 7500
//Maximum Connection Interval in microseconds
#define GAPMAXCONNINTus                 50000
//Link Supervision Timeout in microseconds
#define GAPLINKSUPRVSNTOUTus            2000000
//Slave Latency
#define GAPSLAVELATENCY                 0

//------------------------------------------------------------------------------
// VSP Service
//------------------------------------------------------------------------------
#define VSPSECURITY                     1  //1=Open, 2=NO_MITM, 3=WITH_MITM

#define VSPUUIDSERVICE                  "EADE1101B87f490C92CB11BA5EA5EFBE"
#define VSPUUIDRX                          0x7001  //uses base of VSPUUIDSERVICE
#define VSPUUIDTX                          0x7002  //uses base of VSPUUIDSERVICE
#define VSPUUIDMDMIN                       0x7003  //uses base of VSPUUIDSERVICE
#define VSPUUIDMDMOUT                      0x7004  //uses base of VSPUUIDSERVICE

#define VSPBUFLENRX                     0  //default
#define VSPBUFLENTX                     0  //default

//------------------------------------------------------------------------------
// Adverts
//------------------------------------------------------------------------------
#define ADVDISCOVERYFLAGS               2  //1=Limited,2=General,3=Both (0 do not define)
```

```
#define ADVMAXDEVICENAMELEN                10
#define ADVINTERVALms                      100
#define ADVTIMEOUTms                       0  //0 means infinity
#define ADVFILTERPOLICY                    0

//******************************************************************************
// Library Import
//******************************************************************************

//******************************************************************************
// Global Variable Declarations
//******************************************************************************

//------------------------------------------------------------------------------
// Misc variables
//------------------------------------------------------------------------------

dim rc                  //result code
dim hVspUuidSvc         //Contains the uuid handle of the VSP service so that it
                        //can be used to create an AD element in adverts
dim baud                //the configured baudrate

//******************************************************************************
// Function and Subroutine definitions
//******************************************************************************

//==============================================================================
// For debugging :: will inspect the global 'rc' variable
// --- ln = line number
//==============================================================================
#cmpif 0x01 : sub DbgAssertRC(ln as integer)
#cmpif 0x01 :   if rc!=0 then
#cmpif 0x01 :     print "\nFail :";integer.h' rc;" at tag ";ln
#cmpif 0x01 :   endif
#cmpif 0x01 : endsub

//==============================================================================
//==============================================================================
sub OpenUART()
```

```
  baud=UARTBAUD
  rc=UartOpen(baud,UARTBUFLENTX,UARTBUFLENRX,UARTOPTIONS)
  #cmpif 0x01 : DbgAssertRC(1050)
endsub

//================================================================================
// Device Name (writable/not)
// Connection Parameters
//================================================================================
sub ConfigServiceGAP()
  dim devicename$ : devicename$= GAPDEVNAME

rc=BleGapSvcInit(devicename$,GAPNAME_WRITEABLE,GAPAPPEARANCE,GAPMINCONNINTus,GAPMAXCONNINTus,
GAPLINKSUPRVSNTOUTus,GAPSLAVELATENCY)
  #cmpif 0x01 : DbgAssertRC(1150)
endsub

//================================================================================
// Security :: 1=Open, 2=NO_MITM, 3=WITH_MITM
//================================================================================
sub OpenVSP(vspSec)
  dim uuid$
  dim hVspUuidRx
  dim hVspUuidTx
  dim hVspUuidMdmIn
  dim hVspUuidMdmOut

  //create the advert & scan reports
  uuid$         = VSPUUIDSERVICE
  uuid$         = StrDehexize$(uuid$)
  hVspUuidSvc   = BleHandleUuid128(uuid$)
  hVspUuidRx    = BleHandleUuidSibling(hVspUuidSvc,VSPUUIDRX)
  hVspUuidTx    = BleHandleUuidSibling(hVspUuidSvc,VSPUUIDTX)
  hVspUuidMdmIn = BleHandleUuidSibling(hVspUuidSvc,VSPUUIDMDMIN)
  hVspUuidMdmOut= BleHandleUuidSibling(hVspUuidSvc,VSPUUIDMDMOUT)

  vspSec        = (vspSec & 0x7)<<2

  //finally open the VSP

rc=BleVspOpenEx(VSPBUFLENTX,VSPBUFLENRX,vspSec,hVspUuidSvc,hVspUuidRx,hVspUuidTx,hVspUuidMdmI
```

```
n,hVspUuidMdmOut)
  #cmpif 0x01 : DbgAssertRC(1410)

endsub

//==============================================================================
//==============================================================================
sub StartADVERTS()
  dim advReport$
  dim scnReport$
  dim peerAdr$ : peerAdr$=""

  rc=BleAdvRptInit(advReport$,ADVDISCOVERYFLAGS,GAPAPPEARANCE,ADVMAXDEVICENAMELEN)
  #cmpif 0x01 : DbgAssertRC(1530)
  rc=BleScanRptInit(scnReport$)
  #cmpif 0x01 : DbgAssertRC(1550)
  rc=BleAdvRptAddUuid128(scnReport$,hVspUuidSvc)
  #cmpif 0x01 : DbgAssertRC(1570)
  rc=BleAdvRptsCommit(advReport$,scnReport$)
  #cmpif 0x01 : DbgAssertRC(1590)

  //finally start the adverts
  rc=BleAdvertStart(0,peerAdr$,ADVINTERVALms,ADVTIMEOUTms,ADVFILTERPOLICY)
  #cmpif 0x01 : DbgAssertRC(1630)
endsub

//******************************************************************************
// Handler definitions
//******************************************************************************

//******************************************************************************
// Equivalent to main() in C
//******************************************************************************

//------------------------------------------------------------------------------
//Config and open UART
// See UARTxxx #defines above
//------------------------------------------------------------------------------
OpenUART()
```

```
//------------------------------------------------------------------------------
//Configure GAP Service
// See GAPxxx #defines above
//------------------------------------------------------------------------------
ConfigServiceGAP()

//------------------------------------------------------------------------------
//Config and open VSP
// See VSPxxx #defines above
//------------------------------------------------------------------------------
OpenVSP(VSPSECURITY)

//------------------------------------------------------------------------------
//Advertising
// See ADVxxx #defines above
//------------------------------------------------------------------------------
StartADVERTS()

//------------------------------------------------------------------------------
// PURPOSELY COMMENTED OUT AS WE WANT TO FALL INTO COMMAND MODE
//------------------------------------------------------------------------------
//waitevent
```

5.14.2 Command and Bridge Mode Operation

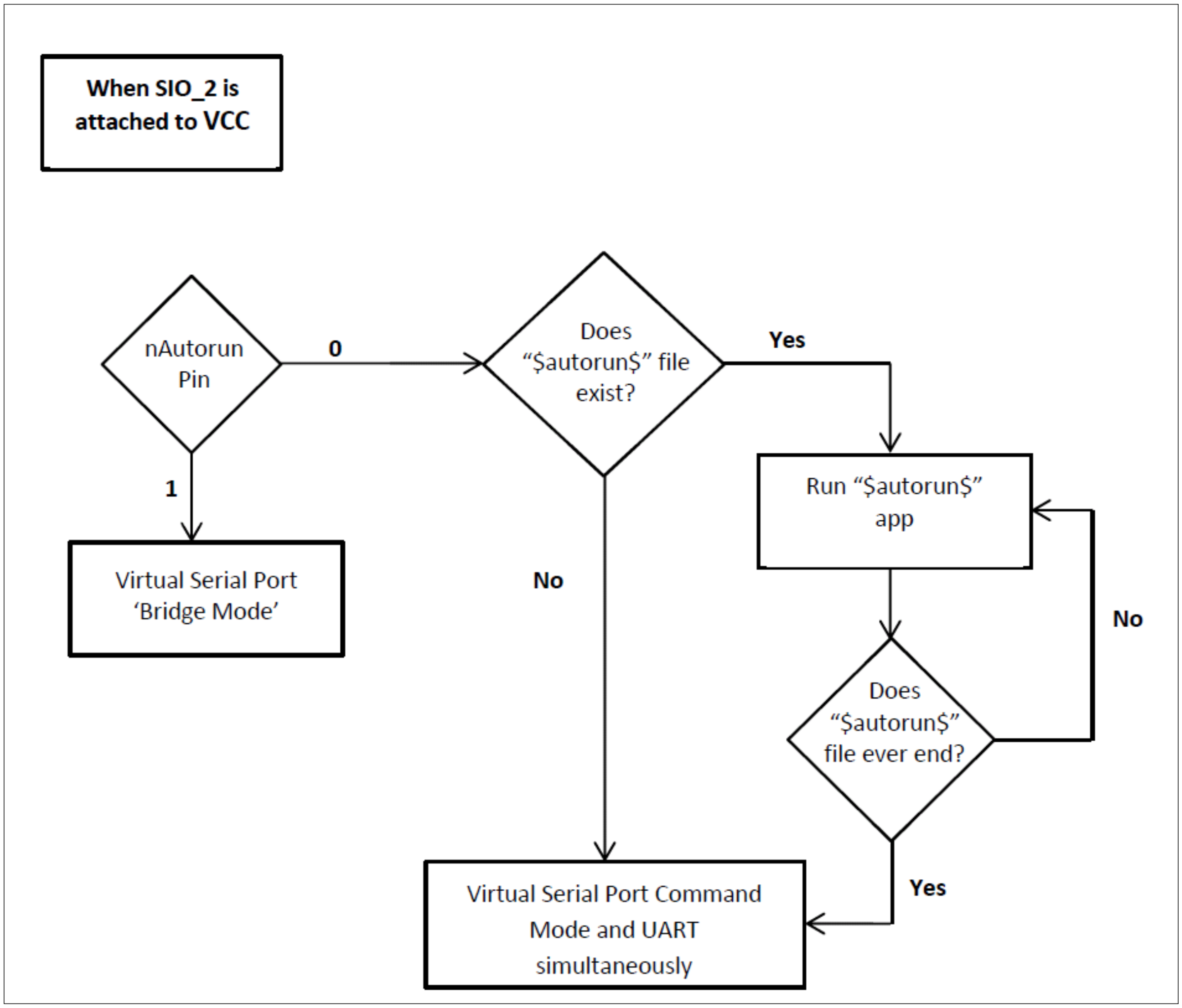
Just as the physical UART is used to interact with the module when it is not running a *smart* BASIC application, it is also possible to have *limited* interaction with the module in interactive mode. The limitation applies to NOT being able to launch *smart* BASIC applications using the AT+RUN command. 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 0x8102 which means by default, bridge mode is enabled if SIO2 is held high and nAutorun is high too.

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 SIO2 on the module, but it can be changed by setting the config key 100 via AT+CFG.



You can interact with the BL652 over the air via the Virtual Serial Port Service using the Laird Toolkit app, available for iOS or Android.

You may download smartBASIC applications onto the BL652 Over The Air using a BT900-US/BL652 devkit and a smartBASIC application from GitHub. 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.

5.14.3 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 BL6xx Serial app and connect to your BL652 to test this sample app.

Example:

```
// Example :: VSpEvents.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples
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 BL6xx Serial app to test this"

//wait for events and messages
WAITEVENT
```

5.14.4BleVSpOpen

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.

Note that the parameters specified in the first call to this function are sticky. After calling BleVspClose() if this function is recalled the parameters will be ignored and the internal state machine managing the VSP function will resume from a suspended state. This is because on a close, it is not possible to remove the service from the GATT table. If this is strictly required, perform a warm reset using RESET() and then action appropriately in the new incarnation. One way of detection a new incarnation could be by using NvRecordSet()/NvRecordGet() as that writes/reads to non-volatile memory.

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. **This is deprecated – always set to 0** **Bit 1**: Set to 1 to suppress ModemIn and ModemOut characteristics **Bits**: Security Setting for accesing characteristics 4 3 2 — Bit Number 0 0 0 — Open 0 0 1 — Open 0 1 0 — ENCRYPTED_NO_MITM 0 1 1 — ENCRYPTED_WITH_MITM 1 0 0 — SIGNED_NO_MITM (reserved for future) 1 0 1 — SIGNED_WITH_MITM (reserved for future) 1 1 0 — ENCRYPTED_NO_MITM 1 1 1 — ENCRYPTED_NO_MITM **Bit 5..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,BLEVSPOPENEX

Example:

```
// Example :: BleVspOpen.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


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
```

5.14.5 BleVSpOpenEx

FUNCTION

This function opens the a managed VSP service using the parameters specified. The service's UUID and UUIDs for the up to 4 characteristics can all be inidividually specified.

ModemIn and ModemOut characteristics are registered in the GATT table with the Rx and Tx FIFO characteristics if both UUIDMdmIn and UUIDMdmOut are not invalid (invalid handle == 0).

Note that the parameters specified in the first call to this function are sticky. After calling BleVspClose() if this function is recalled the parameters will be ignored and the internal state machine managing the VSP function will resume from a suspended state. This is because on a close, it is not possible to remove the service from the GATT table. If this is strictly required, perform a warm reset using RESET() and then action appropriately in the new incarnation. One way of detection a new incarnation could be by using NvRecordSet()/NvRecordGet() as that writes/reads to non-volatile memory.

BLEVSPOPENEX (txbuflen, rxbuflen, nFlags, hUuidSvc, hUuidRx, hUuidTx, hUuidMdmIn, hUuidMdmOut)

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. **This is deprecated – always set to 0** Bit 1 — This bit is ignored. See hUuidMdmIn and hUuidMdmOut instead to manage. Bits — Security Setting for accesing characteristics 4 3 2 — Bit Number 0 0 0 — Open 0 0 1 — Open

	0 1 0 ENCRYPTED_NO_MITM 0 1 1 ENCRYPTED_WITH_MITM 1 0 0 SIGNED_NO_MITM (reserved for future) 1 0 1 SIGNED_WITH_MITM (reserved for future) 1 1 0 ENCRYPTED_NO_MITM 1 1 1 ENCRYPTED_NO_MITM Bit 5..31 Reserved for future use. Set to 0.
hUuidSvc	***byVal hUuidSvc AS INTEGER*** This is the handle for 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.
hUuidRx	***byVal hUuidRx AS INTEGER*** This is the handle for the Rx Characteristic UUID. It cannot be an invalid handle.
hUuidTx	***byVal hUuidTx AS INTEGER*** This is the handle for the Tx Characteristic UUID. It cannot be an invalid handle.
hUuidMdmIn	***byVal hUuidMdmIn AS INTEGER*** This is the handle for the MdmIn Characteristic UUID. Can be an invalid handle (0) and in that case both modem characteristic are not registered.
uUuidMdmOut	***byVal hUuidMdmOut AS INTEGER*** This is the handle for the MdmOut Characteristic UUID. . Can be an invalid handle (0) and in that case both modem characteristic are not registered.
Related Commands	BLEVSPINFO, BLEVSPCLOSE, BLEVSPWRITE, BLEVSPREAD, BLEVSPFLUSH, BLEVSPOPEN

```
//Example

 DIM scRpt$,adRpt$,addr$,hUuidSvc,hUuidRx,hUuidTx,hUuidMdmIn,hUuidMdmOut,uuid$

 uuid$ = "ced9d91366924a1287d56f2764762b2a"
 uuid$ = StrDehexize$(uuid$)
 hUuidSvc = BleHandleUuid128(uuid$)
 hUuidRx = BleHandleUuidSibling(hUuid1,0x1234)
 hUuidTx = BleHandleUuidSibling(hUuid1,0x5678)
 hUuidMdmIn  = BleHandleUuidSibling(hUuid1,0x9ABC)
 hUuidMdmOut = BleHandleUuidSibling(hUuid1,0xDEF0)

 //Open VSP
 IF BleVSpOpenEx(128,128,0, hUuidSvc,hUuidRx,hUuidTx,hUuidMdmIn,hUuidMdmOut)==0 THEN
     PRINT "\nVSP service opened with non-default UUIDs"
 ELSE
     PRINT "\nFailed"
```

```
ENDIF
```

Expected Output:

```
VSP service opened with non-default UUIDs
```

5.14.6 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.

Note that the parameters specified in the first call of BleVspOpen() are sticky. After calling this function if BleVspOpen() or BleVspOpenEx() is called again then the open parameters will be ignored and the internal state machine managing the VSP function will resume from a suspended state. This is because on a close, it is not possible to remove the service from the GATT table. If this is strictly required, perform a warm reset using RESET() and then action appropriately in the new incarnation. One way of detection a new incarnation could be by using NvRecordSet()/NvRecordGet() as that writes/reads to non-volatile memory.

BLEVSPCLOSE ()

Exceptions	▪ Local Stack Frame Underflow ▪ Local Stack Frame Overflow
Arguments	None
Related Commands	BLEVSPINFO, BLEVSPOPEN, BLEVSPWRITE, BLEVSPREAD, BLEVSPFLUSH

Use the iOS “BL6xx Serial” app and connect to your BL652 to test this sample app.

Example:

```
// Example :: BleVspClose.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples
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 BL652

VSP tx buffer empty
Exiting...
```

5.14.7 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 5 — Tx/Rx attribute size in bytes. Valid range is 20-244, and can be configured using AT+CFG 212. See Data Packet Length Extension section for more information.

Related Commands	BLEVSPOPEN, BLEVSPCLOSE, BLEVSPWRITE, BLEVSPREAD, BLEVSPFLUSH

Example:

```
// Example :: BleVspInfo.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


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)
PRINT "\nTx/Rx Characteristic Size: "; BleVSpInfo(5)     // Changed using AT+CFG
212 xx
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
Tx/Rx Characteristic Size: 20
Vsp State: 0
```

5.14.8 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 buffer. If STRLEN(strMsg) and the return value are not the same, it implies that the transmi 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 to drop the data from the string, so subsequent calls to this function only retry wi data not placed in the output ring buffer. Another strategy is to wait for EVVSPTXEMPTY events, then resubmit data.
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 Laird Toolkit app for iOS/Android and connect to your BL652 to test this sample app.

Example:

```
// Example :: BleVSpWrite.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples



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 BL652
Exiting...
```

5.14.9 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.
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 Laird Toolkit app for iOS/Android with your BL652 to test this sample app.

Example:

```
// Example :: BleVSpRead.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

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 BL652
Messgae from client: (Whatever data you send from your device)
Message from client: exit
Exiting...
```

5.14.10 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
Related Commands	BLEVSPOPEN, BLEVSPCLOSE, BLEVSPINFO, BLEVSPWRITE, BLEVSPFLUSH

Example:

```
// Example :: BleVSpUartBridge.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples
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)
```


Americas: +1-800-492-2320
Europe: +44-1628-858-940
Hong Kong: +852 2923 0610

```
    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..."
```

5.14.11 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	▪ None
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 flush both buffers.
Related Commands	BLEVSPOPEN, BLEVSPCLOSE, BLEVSPINFO, BLEVSPWRITE, BLEVSPREAD

Example:

```
// Example :: BleVSpFlush.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples


 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(3)       //Flush both buffers
ENDSUB
//==============================================================================
// VSP Rx buffer event handler
//==============================================================================
FUNCTION HandlerVSpRx() AS INTEGER
    BleVspFlush(1)
    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 BL652
Rx buffer flushed
Rx buffer flushed
```

```
Exiting...
```

5.15 Data Packet Length Extension

This section describes all the events and functions used for Data Packet Length Extension and related features to achieve higher throughputs.

5.15.1 Overview

5.15.1.1 Data Packet Length Extension

One of the major additions in Bluetooth v4.2 is LE Data Packet Length Extension. This feature allows the BLE packet size to increase from 27 to 251 bytes at the link layer, thus increasing the capacity of the data channel by approximately ten times. The benefits of of this include the following:

- **Higher Throughputs** – Less time is required to transfer the same amount of data compared to Bluetooth v4.1.
- **Lower power consumption** – Fewer transactions are required to transfer a given amount of data compared to Bluetooth v4.1. This reduces the time for which the radio is active.

In order to take full advantage of packet length extension, the device should also have an ATT_MTU greater than the default 23 bytes.

5.15.1.2 ATT_MTU

The attribute Maximum Transmission Unit (ATT_MTU) is the maximum size of any packet sent betweem a GATT client and a GATT server. It determines the maximum amount of data that can be sent over the air for GATT operations.

GATT Operation	Attribute Size	Example when ATT_MTU=23
Read	0 to (ATT_MTU-1)	The GATT client can only read 22 bytes from a GATT server's attribute data.
Write	0 to (ATT_MTU-3)	The GATT client can only write up to 20 bytes to a GATT server attribute.
Notification	0 to (ATT_MTU-3)	The GATT server can only send notifies of data up to 20 bytes long
Indications	0 to (ATT_MTU-3)	The GATT server can only send indications of data up to 20 bytes long

The MTU exchange is a subprocedure used by the GATT client to set the connection's ATT_MTU to the maximum possible value that can be supported by both devices. This means that if the ATT_MTU is set to a value larger than the default 23 bytes, larger amounts of data can be sent between the GATT server and the GATT client per transaction, therefore resulting in higher throughput. For example, when the ATT_MTU is set to 247, single read/write/notifies/indicates can be performed on attributes that are 244 bytes long.

5.15.2 CFG Keys Configuration

5.15.2.1 Maximum ATT_MTU

The maximum ATT_MTU value that the BL652 supports can be set using **AT+CFG 211 num**. Once this value is set, the BL652 should be reset (e.g. via ATZ command or a UART BREAK) for the configuration to take effect. When the *smart*BASIC application is running and if the BL652 is acting as a GATT client, the function BleGattcAttributeMtuRequest should be used to request the ATT_MTU size to change to its maximum supported value. If the BL652 is acting as a GATT server, when it receives the request it automatically responds with its maximum ATT_MTU. The connection's MTU is the minimum value between the client's and server's maximum ATT_MTU.

CFG ID	Definition
211	Maximum ATT_MTU in bytes

Example:

```
AT+CFG 211 247
```

```
00

ATZ

00

AT+CFG 211 ?

27   0x000000F7 (247)

00
```

5.15.2.2 Maximum Attribute Data Length

In order to take full advantage of the increased ATT_MTU and packet length extension, the BL652 now supports attribute data lengths in the GATT table of up to 244 bytes. The maximum attribute data length is set using **AT+CFG 212 num**. The default value is 20 bytes. Once this is set, the BL652 should be reset (e.g. via ATZ command or a UART BREAK) for the configuration to take effect. At runtime, the function BleAttrMetaDataEx can then be used to create characteristic values larger than 20 bytes.

ID	Definition
212	Maximum Attribute Data Length

Example:

```
AT+CFG 212 244

00

ATZ

00

AT+CFG 212 ?

27   0x000000F4 (244)

00
```

5.15.2.3 Maximum Packet Length

The BL652 supports a packet size of 27 bytes by default, and can be configured to support packet sizes up to 251 bytes, which is the maximum that is allowed by the Bluetooth specification. In order to increase the packet size supported by the device, the command AT+CFG 216 num should be called, where num should be in the range of 27-251 bytes long. For values less than or greater than the range, the packet length will be capped to 27 bytes or 251 bytes respectively.

Note: This function only sets the maximum packet length supported by the device. To actually change the packet length for a connection, the function BleGattcAttributeMtuRequest() during the connection, and the packet length requested will be 'ATT_MTU + 4'. For more information, refer to the example for BleGattcAttributeMtuRequest().

5.15.3 Events and Messages

5.15.3.1 EVATTRIBUTEMTU

This event is thrown when the ATT_MTU of a connection is changed. It occurs after an MTU exchange procedure has been intiated from the GATT client. The event comes with the following parameters:

- **Connection handle** – The handle of the connection for which the attribute MTU has changed.
- **Attribute MTU** – The new attribute size. This is in the range of 23-247 bytes.

For usage, see example for BleGattcAttributeMtuRequest.

5.15.3.2 EVPACKETLENGTH

This event message is thrown when the connection's data packet length changes. It is only thrown after a negotiation of the attribute MTU via the BleAttributeMtuRequest *smart*BASIC function. The event comes with the following parameters:

- **Connection handle** – The handle of the connection for which the packet length has changed.
- **Maximum Tx Octets** – The maximum number of bytes that the BL652 sends on this connection. The valid range is between 27-251 bytes.
- **Maximum Tx Time** – The maximum time that the BL652 takes to send one byte on this connection. The valid range is between 328-2120 microseconds. This value cannot be controlled by the *smart*BASIC application and is only provided for informative purposes.
- **Maximum Rx Octets** – The maximum number of bytes that the BL652 receives on this connection. The valid range is between 27-251 bytes. The default value is 27 bytes.
- **Maximum Rx Time** – The maximum time that the BL652 takes to send one byte on this connection. The valid range is betweem 328-2120 microseconds. This value cannot be controlled by the *smart*BASIC application and is only provided for informative purposes.

For usage, see example for BleGattcAttributeMtuRequest.

5.15.4 BleGattcAttributeMtuRequest

This function is used by the GATT client to request a new attribute MTU from the remote GATT server. On the BL652, the default ATT_MTU is 23 bytes. The maximum value that the BL652 can support is 247 bytes. This can be set using the config key 211.

Note: The ATT_MTU value is set using the interactive command **AT+CFG 211 num**. This value is then always used when the BleGattcAttributeMtuRequest is called.

BLEGATTCATTRIBUTEMTUREQUEST(nConnHandle)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
nConnHandle	**byVal nEnable AS INTEGER**. The connection handle for which the ATT_MTU should change

```
// Example :: BleGattcAttributeMtuRequest.sb

// IMPORTANT: before running this application, the ATT_MTU and maximum packet
// length are set using the interactive commands:
//
// AT+CFG 211 247 (This is to set the maximum ATT_MTU)
// AT+CFG 216 251 (This is to set the maximum packet length)
// ATZ            (This is to reset the device for value to take effect)
```

```
//
// In order to achieve an ATT_MTU larger than the default 23, the remote device
// should also have its maximum ATT_MTU set to a value greater than 23. If the
// remote device is a BL652, the same AT+CFG command should be used

//BLE EVENT MSG IDs
#define BLE_EVBLEMSGID_CONNECT                               0 // msgCtx = connection handle
#define BLE_EVBLEMSGID_DISCONNECT                            1 // msgCtx = connection handle

DIM rc, stRsp$, addr$

//==============================================================================
// This handler is called when there is a BLE message
//==============================================================================
function HandlerBleMsg(BYVAL nMsgId AS INTEGER, BYVAL nCtx AS INTEGER) as integer
  dim hz

  select nMsgId
  case BLE_EVBLEMSGID_CONNECT
    print " --- Connect: (";integer.h' nCtx;")\n"
    // Upon connection, request a new attribute length. The value used will be that
    // whcih was set using 'AT+CFG 211 num' before running the program
    rc = BleGattcAttributeMtuRequest(nCtx)

  case BLE_EVBLEMSGID_DISCONNECT
    print " --- Disconnect: (";integer.h' nCtx;")\n"
    // Upon disconnection, start advertising again
    rc = BleAdvertStart(0,addr$,100,0,0)

  case else

  endselect
endfunc 1

//==============================================================================
// This handler is called when the packet length is changed
//==============================================================================
function HandlerPacketLength(BYVAL hConn, BYVAL Tx_Octets, BYVAL Tx_Time, BYVAL
Rx_Octets, BYVAL Rx_Time)

    print "Packet Length Change: \n"
    print "Handle: ";integer.h' hConn;"\n"
    print "Tx_Octets=";Tx_Octets;" Tx_Time =";Tx_Time;"\n"
    print "Rx_Octets=";Rx_Octets;" Rx_Time =";Rx_Time;"\n"

endfunc 1

//==============================================================================
// This handler is called when there is an event that the attribute MTU has changed
//==============================================================================
function HandlerAttrMTU(BYVAL hCOnn AS INTEGER, BYVAL nSize AS INTEGER)

    print "Attribute MTU Changed - Handle:";integer.h' hConn;" Size:";nSize;"\n"

endfunc 1

//------------------------------------------------------------------------------
// Enable synchronous event handlers
//------------------------------------------------------------------------------
OnEvent  EVBLEMSG                    call HandlerBleMsg
OnEvent  EVATTRIBUTEMTU              call HandlerAttrMTU
OnEvent  EVPACKETLENGTH              call HandlerPacketLength
```

```
// Initialise LE routines
rc = BleAdvertStart(0,addr$,100,0,0)
// Open the gatt client. Specify the buffer size to be 251 to be able to receive
// notifications up to 244 bytes long (maximum supported by BL652 when ATT_MTU = 247)
rc = BleGattcOpen(251, 0)

//------------------------------------------------------------------------------
// Wait for a synchronous event.
// An application can have multiple <WaitEvent> statements
//------------------------------------------------------------------------------
WAITEVENT
```

Expected Output:

```
AT+CFG 211 247

00

AT+CFG 216 251

00

ATZ

00

AT+RUN "BleGattcAttributeMtuReq"


--- Connect: (0001FF00)
Attribute MTU Changed - Handle:0001FF00 Size:247
Packet Length Change:
Handle: 0001FF00
Tx_Octets=251 Tx_Time =2120
Rx_Octets=251 Rx_Time =2120
```

5.15.5 BleMaxPacketLengthSet

This function has been removed and replaced with the config key 216. To set the maximum packet length, either call '**AT+CFG 216 nSize**' (followed by 'ATZ' for the value to take effect), or at runtime calling NvCfgKeySet(216, nSize) (followed by reset(0) for the value to take effect.

5.15.6 BleMaxPacketLengthGet

This function is used to get the preferred maximum packet length on the BL652. The actual packet length change only occurs when when the attribute MTU for the connection is changed via the BleGattcAttributeMtuRequest function.

BLEMAXPACKETLENGTHSET (nSize)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
nSize	**byRef nSize AS INTEGER**.

	When the function is used, this value will contain the maximum packet length preferred by the device.

Example:

```
// Example :: BleMaxPacketLengthSet.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

// Before running the example, issue 'at+cfg 216 155' followed by 'atz'
dim rc, nSize

// Now get the maximum packet length
rc = BleMaxPacketLengthGet(nSize)
PRINT "\nThe maximum packet size is ";nSize
```

```
The maximum packet size is 155
```

5.16 LE Ping

5.16.1 Overview

The LE Ping feature can be used to verify the existence of an encrypted link with the remote device. When enabled, the BL652 sends a request to the remote device to send an encrypted packet. If a timeout occurs without the reception of a packet, an event is triggered on the BL652.

5.16.2 Events and Messages

5.16.2.1 EVBLE_PING_AUTH_TIMEOUT

This event is thrown when the ping authenticated payload timer has expired without receiving an encrypted packet. The event comes with the following parameter:-

Connection Handle – The handle of the connection for which the timeout has occurred.

For usage, see example for BlePingAuthTimeout.

5.16.3 BlePingAuthTimeout

On an encrypted connection, this function is used to monitor the time since the last reception of an encrypted packet. If the timeout is exceeded without receiving a packet, then the EVBLE_PING_AUTH_TIMEOUT is triggered. This can be used to detect if there is something wrong with the encrypted link, and therefore if the event is received, a safe action would be to disconnect.

Note: Setting nAuthTimeOut to a value less than (2*Connection Interval) will always cause the EVBLE_PING_AUTH_TIMEOUT event to be triggered. The reason for this is that two connection events are required for a packet to be sent to the remote device and then sent back, therefore having nAuthTimeout smaller than (2*Connection Interval) means that the timer will always expire before the response is received from the remote device, causing the event to be triggered.

BLEPINGAUTHTIMEOUT (hConnHanlde, nAuthTimeout)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
nConnHandle	**byVal hConnHandle AS INTEGER.**

	The connection handle for which the authenticated payload timer is to start.
nAuthTimeout	**byVal nAuthTimeout AS INTEGER**. The authentication timeout in microseconds. The range of this value is between 10000 and 480000 microseconds, and is rounded up to the nearest 10000us (10ms).

Example:

```
//Example :: BlePingAuthTimeout.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples



// Set BLE_PING_TIMEOUT to a value more than (2*connection interval)
// for the feature to work. Otherwise the event will be triggered
// because two connection events are required for a packet to be
// sent back and forth.
#define BLE_PING_TIMEOUT    10000
#define BTAddr              "000016A4B75204"

// Variable declaration
DIM hndl, rc, intrvl,sprvto,slat, pingTO

//------------------------------------------------------------------------------
// Function to handle Ble event messages
//------------------------------------------------------------------------------
#define BLE_EVBLEMSGID_CONNECT                  0   //nCtx = connection handle
#define BLE_EVBLEMSGID_DISCONNECT               1   //nCtx = connection handle
#define BLE_EVBLEMSGID_ENCRYPTED                18  //nCtx = connection handle
//------------------------------------------------------------------------------
FUNCTION HandlerBleMsg(nMsgId, nCtx)

    select nMsgId
    case BLE_EVBLEMSGID_CONNECT
        print "## Connected!\n"
        // Read connection interval
        rc = BleGetCurConnParms(nCtx,intrvl,sprvto,slat)
        print "## Connection Interval=";intrvl;"\n"
        // Pair to the remote device
        rc = BlePair(nCtx, 0)

    case BLE_EVBLEMSGID_DISCONNECT
        print "## Disconnected!\n"

    case BLE_EVBLEMSGID_ENCRYPTED
        print "## Encrypted Connection!\n"
        // Start LE Ping Authenticated Timeout
        pingTO = BLE_PING_TIMEOUT
        rc = BlePingAuthTimeout(nCtx, pingTO)
        if rc == 0 then
            print "## Ping auth timeout enabled :: Timeout=";pingTO;"\n"
        endif

    case else
    endselect
ENDFUNC 1

//------------------------------------------------------------------------------
// This handler is called when the LE Ping authentication has timed out
//------------------------------------------------------------------------------
function  HandlerLePingTimeout(BYVAL hConn AS INTEGER)
    print "## LE Ping Timeout : ";integer.h' hConn;"\n"
```

```
    // Disconnect as this is not safe, check timeout is more than 2*connection interval
    rc = BleDisconnect(hConn)
endfunc 1

//------------------------------------------------------------------------------
// Enable synchronous event handlers
//------------------------------------------------------------------------------
OnEvent  EVBLEMSG                  call HandlerBleMsg
OnEvent  EVBLE_PING_AUTH_TIMEOUT   call HandlerLePingTimeout

//Connect to remote device
DIM addr$
addr$ = BTAddr
addr$ = StrDehexize$(addr$)
rc = BleConnect(addr$, 5000, 27000, 30000, 500000)

//------------------------------------------------------------------------------
// Wait for a synchronous event.
//------------------------------------------------------------------------------
WaitEvent
```

5.17 LE 2M PHY

5.17.1 Events and Messages

5.17.1.1 EVBLE_PHY_REQUEST

By default, this event is not enabled and an incoming PHY change request is automatically accepted. This event is only enabled when the function BleConnectConfig(9, 1) is called. It is thrown when there is a request from the remote device to switch the PHY modulation. In the function handler for this event, the function BlePhySet should be used to respond with the module's PHY preferences. The event comes with the following parameters:-

Connection Handle – The handle of the connection for which there is a PHY modulation request.

BlePhyTx – The transmission PHY preference of the remote device. 1 for 1MPHY, and 2 for 2MPHY.

BlePhyRx – The reception PHY preference of the remote device. 1 for 1MPHY, and 2 for 2MPHY.

For usage, see example for BlePhyReq.

5.17.1.2 EVBLE_PHY_UPDATED

This event is thrown when the PHY modulation of the underlying connection has been updated. The event contains the following parameters:-

Connection Handle – The handle of the connection for which there is a PHY modulation has been updated.

Status – The HCI status code of the operation. 0x00 indicates a successful command. 0x01 – 0xFF indicates that the command has failed. A full list of HCI status codes can be found at the end of this document.

BlePhyTx – The new value of the transmission PHY. 1 for 1MPHY, and 2 for 2MPHY.

BlePhyRx – The new value of the transmission PHY. 1 for 1MPHY, and 2 for 2MPHY.

For usage, see example for BlePhyReq.

5.17.2 BlePhySet

This function is used to set the PHY preferences of a connection, or reply to PHY request from a remote device. When this command is initiated from the module, it triggers an EVBLE_PHY_REQUEST on the remote device, and if successful, EVBLE_PHY_UPDATED event is thrown to indicate that the PHY configuration of the connection has changed.

BLEPHYSET (hConn, nPhyTx, nPhyRx, nOptions)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments:	
hConn	**byVal hConn AS INTEGER**. The handle of the connection for which a PHY modulation update is taking place.
nPhyTx	**byVal nPhyTx AS INTEGER**. A bit field that indicates the transmission PHYs that the host prefers ▪ Bit 0 : The host prefers to use the LE 1M transmission PHY (possibly among others). ▪ Bit 1 : The host prefers to use the LE 2M transmission PHY (possibly among others). ▪ Bit 2-7: Reserved for future use.
nPhyRx	**byVal nPhyRx AS INTEGER**. A bit field that indicates the reception PHYs that the host prefers ▪ Bit 0 : The host prefers to use the LE 1M reception PHY (possibly among others). ▪ Bit 1 : The host prefers to use the LE 2M reception PHY (possibly among others). ▪ Bit 2-7: Reserved for future use.
nOptions	**byVal nPhyRx AS INTEGER**. This is reserved for future use and should always be set to 0.

```
//Example :: BlePhySet.sb

// Ensure that the remote device is advertising

#define BTAddr              "000016A4B75202"

// Variable declaration
DIM rc, hConn

//------------------------------------------------------------------------------
// Function to handle Ble event messages
//------------------------------------------------------------------------------
#define BLE_EVBLEMSGID_CONNECT                    0    //nCtx = connection handle
#define BLE_EVBLEMSGID_DISCONNECT                 1    //nCtx = connection handle
//------------------------------------------------------------------------------
FUNCTION HandlerBleMsg(nMsgId, nCtx)

    select nMsgId
    case BLE_EVBLEMSGID_CONNECT
        print "## Connected!\n"
        // Upon connection, request a change to 2MPHY
        hConn = nCtx
        dim nPhyTx : nPhyTx = 2
        dim nPhyRx : nPhyRx = 2
        dim nOptions : nOptions = 0
        rc = BlePhySet(hConn, nPhyTx, nPhyRx, nOptions)

    case BLE_EVBLEMSGID_DISCONNECT
        print "## Disconnected!\n"

    case else
    endselect
ENDFUNC 1
```

```
//------------------------------------------------------------------------------
// This handler is called when there is a connection attempt timeout
//------------------------------------------------------------------------------
function HandlerBleConnTimOut() as integer
  print "## Connection attempt stopped via timeout\n"
endfunc 1

//------------------------------------------------------------------------------
// This handler is called when remote is requesting a switch to a different PHY
//------------------------------------------------------------------------------
function  HandlerPhyRequest(BYVAL hConn, BYVAL PhyTx, BYVAL PhyRx)
    print "## BLE PHY REQUEST: \n"
    print "Handle: ";integer.h' hConn;"\n"
    print "PhyTx=";PhyTx;" PhyRx =";PhyRx;"\n"
endfunc 1

//------------------------------------------------------------------------------
// This handler is called when the BLE PHY is updated
//------------------------------------------------------------------------------
function  HandlerPhyUpdated(BYVAL hConn, BYVAL nStatus, BYVAL PhyTx, BYVAL PhyRx)
    print "## BLE PHY CHANGED: \n"
    print "Handle: ";integer.h' hConn;"\n"
    print "Status: ";integer.h' nStatus;"\n"
    print "PhyTx=";PhyTx;" PhyRx =";PhyRx;"\n"
endfunc 1

//------------------------------------------------------------------------------
// Enable synchronous event handlers
//------------------------------------------------------------------------------
OnEvent  EVBLEMSG                  call HandlerBleMsg
OnEvent  EVBLE_CONN_TIMEOUT        call HandlerBleConnTimOut
OnEvent  EVBLE_PHY_REQUEST         call HandlerPhyRequest
OnEvent  EVBLE_PHY_UPDATED         call HandlerPhyUpdated

//Connect to remote device
DIM addr$
addr$ = BTAddr
addr$ = StrDehexize$(addr$)

// Change default configuration so that EVBLE_PHY_REQUEST is thrown
rc = BleConnectConfig(9, 1)

rc = BleConnect(addr$, 30000, 27000, 30000, 500000)

//------------------------------------------------------------------------------
// Wait for a synchronous event.
//------------------------------------------------------------------------------
WaitEvent
```

Expected Output:

```
## Connected!
## BLE PHY CHANGED:
Handle: 0001FF00
Status: 00000000
PhyTx=2 PhyRx =2
```

6 Other Extension Built-in Routines

This chapter describes non BLE-related extension routines that are not part of the core *smart*BASIC language.

6.1 Near Field Communications (NFC)

This chapter provides details of all the *smart*BASIC functions and subroutines that expose the NFC functionality and also the events that are generated when in operation.

6.1.1 Overview

This section describes all the events and routines used to interact with the NFC peripheral on the BL652 which is a passive device which means it is **not** possible to establish NFC communications between two BL652 devices. In any NFC communications, one device shall be an Active device.

On the BL652 the NFC is exposed as a Tag Type 2 **Passive** interface which means it can only offer tags to be read from an **Active** NFC reader (for example, a smartphone or an Arduino based shield).

The NFC Forum has agreed on four tag types and a good definition of those NFC Tag Types is provided at http://www.nfc.cc/technology/nfc-tag-types which is reproduced as follows:

- **Type 1** – Type 1 Tag is based on ISO/IEC 14443A. This tag type is read and re-write capable. The memory of the tags can be write protected. Memory size can be between 96 bytes and 2 Kbytes. Communication Speed with the tag is 106 kbit/sec. Example: Innovision Topaz
- **Type 2 –** Type 2 Tag is based on ISO/IEC 14443A. This tag type is read and re-write capable. The memory of the tags can be write protected. Memory size can be between 48 bytes and 2 Kbytes. Communication Speed with the tag is 106 kbit/sec. Example: NXP Mifare Ultralight, NXP Mifare Ultralight
- **Type 3** – Type 3 Tag is based on the Japanese Industrial Standard (JIS) X 6319-4. This tag type is pre-configured at manufacture to be either read and re-writable, or read-only. Memory size can be up to 1 Mbyte. Communication Speed with the tag is 212 kbit/sec. Example: Sony Felica
- **Type 4** – Type 4 is fully compatible with the ISO/IEC 14443 (A \& B) standard series. This tag type is pre-configured at manufacture to be either read and re-writable, or read-only. Memory size can be up to 32 KBytes; For the communication with tags APDUs according to ISO 7816-4 can be used. Communication speed with the tag is 106 kbit/sec. Example: NXP DESfire, NXP SmartMX with JCOP.)

  Mifare Classic is not an NFC forum compliant tag, although reading and writing of the tag is supported by most of the NFC devices as they ship with an NXP chip. The specifications for the tag types are available for free from the NFC-Forum website.

The following is a high level overview of NFC communications and it is encouraged that the reader access resources on the internet which give further details, like for example http://www.nfc.cc/technology/nfc/.

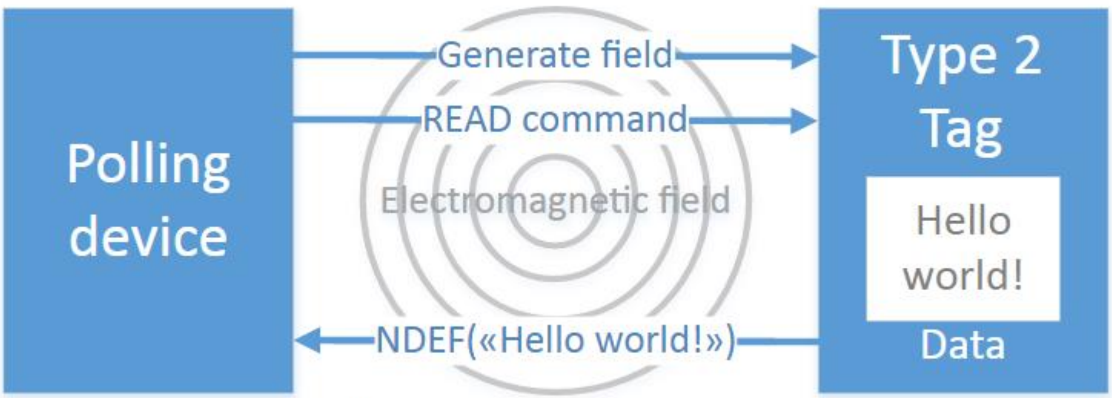
- The NFC physical layer is a half-duplex, bi-directional pipe with a typical datarate of 106kbps and can be 212 or 424 kbps. (The BL652 only provides a 106kbps datarate)
- The data is carried on a 13.56MHz carrier wave which is provided by one of the **active** devices in the peer to peer link. The signalling in the carrier is done using load modulation. “The term load modulation describes the influence of load changes on the initiators carrier field’s amplitude”
  <credit: http://www.nfc.cc/technology/nfc/>
- There is Active mode and Passive mode. At least one device (the initiator) has to be an active device which provides the 13.56MHz carrier wave.
- The data layer for Tags consists of NDEF messages. NDEF = NFC Data Exchange Format.
  Each NDEF message consists of one or more NDEF records.

Each NDEF record consists of a well defined variable length header and a payload which can be anything and the NFC forum does not specify any format.

- An NDEF Record header consists of a payload length, a Type field and an optional ID Field.
  The Type field is used to qualify the payload so that the recipient can interpret it appropriately.
  The optional ID field is typically used to give a 'name' to the record which allows other records in the message to link to.
- NFC provides for three types of communications over the physical channel and they are; Reader/Writer mode, Card Emulation mode and Peer-To-Peer mode. In the context of BL652, only reader/writer mode functionality is made available and initially only passive Tags Type 2 which means Tags can be read but not written.
  Future enhancments to the BL652 firmware may provide Tag Type 4 (which can be read or written) but that is dependent on the chipset vendor providing an appropriate stack.

The Tag Type 2 functionality exposed in the BL652 is nicely illustrated by the following diagram, for which Laird acknowledges Nordic Semiconductor, the chipset vendor.

In the diagram the 'Polling device' is an active device like an NFC enabled smartPhone or an Arduino with an Adafruit NFC shield.



Simplified overview of how NFC can be used

6.1.2 NDEF Messages

NDEF is the acronym for "NFC Data Exchange Format"

NDEF **Messages**, in the context of Tags of any type, are simply an array of 1 or more NDEF **Records**.

A Tag of any type is simply an NDEF message.

Each NDEF record consists of a **header** and a **payload** both being variable length and the length of the payload in each record can be up to 2^32 bytes long, but limited to 32K in Tag Type 4.

The header consists of:

Byte 0 : A bit mask which contains a 3 bit TNF (Type Name Format) and 5 other single bit fields. One of which specifies if the Payload length field is 1 or 4 bytes and another which specifies if the ID field in the header is present. The rest of the bits are used to specify if the record is the first, last or an in-between record in the overall NDEF message.

Byte 1 – Specifies the length of the Type field in the header which can be up to 255 bytes

Next Byte (or next 4 Bytes) – The payload length.

Next Byte – The ID Length (if the ID bit in the first byte is set)

Next N bytes – Where N is specified by Byte 1 is the the Type field

Next N Bytes – Where N is specified by the 'ID length' field and only if the ID bit in Byte 0 is set, used for the ID.

For full details please refer to the NFC Forum technical specification titled *NFC Data Exchange Foramt (NDEF)* and there are various resources online which have good explanations.

6.1.3 Arduino Based NFC Reader

The API presented in this section was tested using an Arduino Uno (www.arduino.cc/en/Main/ArduinoBoardUno) fitted with an Adafruit 'PN532 RFID/NFC Shield' (www.adafruit.com/products/789) and an Arduino application which is also available as-is without warranty and it can be freely modified called **NfcCli.ino**.

It is assumed that the reader is familiar with how to use an Arduino especially how to load apps into a target board. Please refer to online resources if not.

The Arduino application presents a uart based command line interface and currently has three commands :

- **open\r** – This opens the NFC interface
- **scan\r** – This forces a scan for tags and will timeout after about 5 seconds. If a tag is read, then it is interpreted and displayed in textual manner
- **close\r** – This closes the NFC interface

The command set allows for keeping the Arduino NFC antenna constantly in contact with the module's antenna and then allows the field to be enabled or disabled.

6.1.4 Sample Application 1

The following example application, for which the source is available, shows how to create an NDEF message for a Tag which has two text records where the Type is "T."

```
//******************************************************************************
// Example App File : nfc1.text.tag.sb
//
// This application commits an NDEF message with two text tag of type 'T' with
// a "Hello World" and "Welcome" message. Which can be read with an Arduino +
// Adafruit NFC shield running an arduino app written by Laird which is availble
// on request.
//
//******************************************************************************

//******************************************************************************
// Definitions
//******************************************************************************
#define INVALID_NDEF_HANDLE     0xFFFFFFFF

//******************************************************************************
// Register Error Handler as early as possible
//******************************************************************************
sub HandlerOnErr()
  print "\n OnErr - ";GetLastError();"\n"
endsub
onerror next HandlerOnErr

//******************************************************************************
// Debugging resource as early as possible
//******************************************************************************

//==============================================================================
//==============================================================================
sub AssertResCode(byval rc as integer,byval tag as integer)
  if rc!=0 then
    print "\nFailed with ";integer.h' rc;" at tag ";tag
```

```
  endif
endsub

//******************************************************************************
// Global Variable Declarations
//******************************************************************************

dim rc
dim nfcHandle       //returned by NfcOpoen
dim ndefHandle      //returned by NfcNdefMsgNew

dim type$
dim id$
dim engLang$
dim payload$
dim records,memTotal,memUsed

//******************************************************************************
// Initialisse Global Variable
//******************************************************************************

type$="T" : id$=""
engLang$=" en"
rc=strsetchr(engLang$,strlen(engLang$),0) //prepend the language code length + UTF type

//******************************************************************************
// Function and Subroutine definitions
//******************************************************************************


//******************************************************************************
// Handler definitions
//******************************************************************************

//==============================================================================
// This handler is called when data has arrived at the serial port
#define NFC_MSGIN_NFCFIELDOFF               (2)
#define NFC_MSGIN_NFCFIELDON                (3)
#define NFC_MSGIN_NFCTAGREAD                (7)
//==============================================================================
function HandlerNfc(msgid) as integer
  print "\nEVNFC "
  select(msgid)
  case NFC_MSGIN_NFCFIELDOFF
    print "FIELD OFF"
  case NFC_MSGIN_NFCFIELDON
    print "FIELD ON"
  case NFC_MSGIN_NFCTAGREAD
    print "TAG READ"
  case else
  endselect
endfunc 1

//******************************************************************************
//******************************************************************************
// Equivalent to main() in C
//******************************************************************************

//------------------------------------------------------------------------------
// Enable synchronous event handlers
//------------------------------------------------------------------------------
OnEvent  EVNFC  call HandlerNfc
```

```
//------------------------------------------------------------------------------
// Initialise and then wait for events
//------------------------------------------------------------------------------

//Enable NFC hardware interface, it already is, so will succeed
rc=NfcHardwareState(0,1)
AssertResCode(rc,20000)

//Open NFC and return the handle
rc=NfcOpen(0,"\00",nfcHandle)
AssertResCode(rc,20005)

//Create a new NDEF message object that has a maximum size of 16 bytes
rc=NfcNdefMsgNew(32,ndefHandle)
AssertResCode(rc,20010)

//Oops, buffer will be too small do delete and create a new one
rc=NfcNdefMsgDelete(ndefHandle)
AssertResCode(rc,20012)

//Create a new NDEF message object that has a maximum size of 128 bytes
rc=NfcNdefMsgNew(128,ndefHandle)
AssertResCode(rc,20014)

//Add a NDEF Record of type "T" and message "My World" in english language code
payload$="My World"
rc=NfcNdefRecAddGeneric(ndefHandle,1,type$,id$,engLang$,INVALID_NDEF_HANDLE,payload$)
AssertResCode(rc,20020)

//Oops, changed my mind about message so reset the ndef buffer
rc=NfcNdefMsgReset(ndefHandle)
AssertResCode(rc,20022)

//Add a NDEF Record of type "T" and message "Hello World" in english language code
payload$="Hello World"
rc=NfcNdefRecAddGeneric(ndefHandle,1,type$,id$,engLang$,INVALID_NDEF_HANDLE,payload$)
AssertResCode(rc,20024)

//Add a NDEF Record of type "T" and message "Welcome" in english language code
payload$="Welcome"
rc=NfcNdefRecAddGeneric(ndefHandle,1,type$,id$,engLang$,INVALID_NDEF_HANDLE,payload$)
AssertResCode(rc,20040)

//Inspect the status of the ndef message object
rc=NfcNdefMsgGetInfo(ndefHandle,records,memTotal,memUsed)
if rc==0 then
  print "\nNDEF Info: Records=";records;" TotalMem=";memTotal;" UsedMem=";memUsed
endif

//Commit the NDEF message to the stack
rc=NfcNdefMsgCommit(nfcHandle,ndefHandle)
AssertResCode(rc,20060)

//Enable field Sense
rc=NfcFieldSense(nfcHandle,1)
AssertResCode(rc,20080)

//------------------------------------------------------------------------------
// Wait for an event.
//------------------------------------------------------------------------------
WaitEvent
```

The output from the Arduino reader is as follows:

```
open

OK
scan

++ NDEF MESSAGE ++
NFC Forum Type 2
UID: 5F 59 28 A2 AB C6 79

Contains (2) NDEF Records.

NDEF Record 1 (Payload Length=: 14 (0xE))
  TNF: 1
  Type: T
  03656E48656C6C6F20576F726C64       .enHello World

NDEF Record 2 (Payload Length=: 10 (0xA))
  TNF: 1
  Type: T
  03656E57656C636F6D65               .enWelcome

-- NDEF MESSAGE --
OK
```

6.1.5 Sample Application 2

The following example application, for which the source is available, shows how to create an NDEF message for a Tag which has a single record defined as a 'Simplified Tag Format for a Single Bluetooth Carrier Record' as specified in the Bluetooth SIG specification "Bluetooth Secure Simple Pairing Using NFC" dated 2014-01-09.

```
//******************************************************************************
// Example App File : nfc2.text.ble.connection.handover.sb
//
// This application commits an NDEF message with a "Simplified Tag Format for a
// single Bluetooth Carrier Record" which will result in a connection and a just
// works pairing from an android device like Nexus 7 tablet.
//
// It have only been tested against a Nexus 7 (newest model)
//
//******************************************************************************

//******************************************************************************
// Definitions
//******************************************************************************
#define INVALID_NDEF_HANDLE     0xFFFFFFFF

//******************************************************************************
// Register Error Handler as early as possible
//******************************************************************************
sub HandlerOnErr()
  print "\n OnErr - ";GetLastError();"\n"
endsub
onerror next HandlerOnErr
```

```
//******************************************************************************
// Debugging resource as early as possible
//******************************************************************************

//==============================================================================
//==============================================================================
sub AssertResCode(byval rc as integer,byval tag as integer)
  if rc!=0 then
    print "\nFailed with ";integer.h' rc;" at tag ";tag
  endif
endsub

//******************************************************************************
// Global Variable Declarations
//******************************************************************************

dim rc
dim nfcHandle       //returned by NfcOpoen
dim ndefHandle      //returned by NfcNdefMsgNew

dim payload$
dim records,memTotal,memUsed
dim maxdevname : maxdevname = 12
dim appearance : appearance = 0x512
dim flags : flags = 0x2
dim role : role=2
dim oobKey$ : oobKey$=""  //no TK
dim devname$ : devname$="LAIRD BL652"
dim advRpt$, scnRpt$
dim peerAd$ : peerAd$=""
dim hConn : hConn=0xFFFFFFFF

//******************************************************************************
// Function and Subroutine definitions
//******************************************************************************


//******************************************************************************
// Handler definitions
//******************************************************************************

//==============================================================================
// This handler is called when data has arrived at the serial port
#define NFC_MSGIN_NFCFIELDOFF               (2)
#define NFC_MSGIN_NFCFIELDON                (3)
#define NFC_MSGIN_NFCTAGREAD                (7)
//==============================================================================
function HandlerNfc(msgid) as integer
  print "\nEVNFC "
  select(msgid)
  case NFC_MSGIN_NFCFIELDOFF
    print "FIELD OFF"
  case NFC_MSGIN_NFCFIELDON
    print "FIELD ON"
  case NFC_MSGIN_NFCTAGREAD
    print "TAG READ"
```

```
  case else
  endselect
endfunc 1

//==============================================================================
// This handler is called when there is a BLE message
//------------------------------------------------------------------------------
#define BLE_EVBLEMSGID_CONNECT          0
#define BLE_EVBLEMSGID_NEW_BOND        10
#define BLE_EVBLEMSGID_ENCRYPTED       18
//==============================================================================
function HandlerBleMsg(BYVAL nMsgId AS INTEGER, BYVAL nCtx AS INTEGER) as
integer

  select nMsgId
  case BLE_EVBLEMSGID_CONNECT
    hConn=nCtx
    print "\n +++ Connect: (";integer.h' hConn;")"

  case BLE_EVBLEMSGID_NEW_BOND
    print "\n +++ New Bond"
    //Disable field Sense
    rc=NfcFieldSense(nfcHandle,0)
    AssertResCode(rc,20080)
    print "\n --- NFC Field OFF"

  case BLE_EVBLEMSGID_ENCRYPTED
    print "\n +++ Encrypted Connection"

  case else
  endselect
endfunc 1

//==============================================================================
// This handler is called when there is a EVDISCON message
#define ADVTYPE     0 //ADV_IND
#define ADVINTVTL   100 //andvert interval in milliseconds
#define ADVTOUT     0 //no timoeut
//==============================================================================
function HandlerDisconnect(BYVAL nConnH AS INTEGER, BYVAL nReas AS INTEGER) as
integer
  print "\n +++ Disconnect: (";integer.h' nConnH;") reason=";nReas

  rc=BleAdvertStart(ADVTYPE,peerAd$,ADVINTVTL,ADVTOUT,0)
  AssertResCode(rc,10000)

endfunc 1

//******************************************************************************
//******************************************************************************
// Equivalent to main() in C
//******************************************************************************

//------------------------------------------------------------------------------
// Enable synchronous event handlers
//------------------------------------------------------------------------------
```

```
OnEvent  EVNFC      call HandlerNfc
OnEvent  EVBLEMSG   call HandlerBleMsg
OnEvent  EVDISCON   call HandlerDisconnect

//------------------------------------------------------------------------------
// Initialise and then wait for events
//------------------------------------------------------------------------------

//Open NFC and return the handle
rc=NfcOpen(0,"\00",nfcHandle)
AssertResCode(rc,20005)

//Create a new NDEF message object that has a maximum size of 128 bytes
rc=NfcNdefMsgNew(128,ndefHandle)
AssertResCode(rc,20014)

//Add "Simplified Tag Format for a single Bluetooth Carrier" Record
rc=NfcNdefRecAddLeOob(ndefHandle,maxdevname,appearance,role,flags,oobKey$)
AssertResCode(rc,20020)

//Inspect the status of the ndef message object
rc=NfcNdefMsgGetInfo(ndefHandle,records,memTotal,memUsed)
if rc==0 then
  print "\n *** NDEF Info: Records=";records;" TotalMem=";memTotal;"
UsedMem=";memUsed
endif

//Commit the NDEF message to the stack
rc=NfcNdefMsgCommit(nfcHandle,ndefHandle)
AssertResCode(rc,20060)

//Initialise the GAP service
rc=BleGapSvcInit(devname$,0,appearance,7500,100000,2000000,0)
AssertResCode(rc,20100)

//Initialise adverts and commit
rc=BleAdvRptInit(advRpt$,flags,appearance,maxdevname)
AssertResCode(rc,20200)
rc=BleScanRptInit(scnRpt$)
AssertResCode(rc,20210)
rc=BleAdvRptsCommit(advRpt$,scnRpt$)
AssertResCode(rc,20220)

//Start Adverts
rc=BleAdvertStart(ADVTYPE,peerAd$,ADVINTVTL,ADVTOUT,0)
AssertResCode(rc,20300)
print "\n --- Adverts ON"

//Enable field Sense
rc=NfcFieldSense(nfcHandle,1)
AssertResCode(rc,20400)
print "\n --- NFC Field ON"

//------------------------------------------------------------------------------
// Wait for an event.
//------------------------------------------------------------------------------
```

```
WaitEvent
```

The output from the Arduino reader is as follows:

```
open

OK
scan

++ NDEF MESSAGE ++
NFC Forum Type 2
UID: 5F 59 28 A2 AB C6 79

Contains (1) NDEF Record.

NDEF Record 1 (Payload Length=: 32 (0x20))
  TNF: 2
  Type: application/vnd.bluetooth.le.oob
  021C02081B83160BA416000003191205  ................
  0201060C094C4149524420424C363532  .....LAIRD BL652

-- NDEF MESSAGE --
OK
```

Where the payload 021C02…. 363532 is an array of BLE Advert Data Elements which have format Len:Tag:Data. For example 021C02 implies an AD element of length 2 and tag 1C and since 1C means 'LE Role' it corresponds to the value 2 that was passed in the variable 'role' in the function call NfcNdefRecAddLeOob() in the sample app 2 above.

6.1.6 Wake-On-NFC

When the module is in deep sleep, it is possible to wake it up when an NFC field energises it's antenna when an active reader comes into the zone.

By default this does not happen; it only wakes up if the field sense is switched on via NfcFieldSense(). To do that, a 'dummy' tag needs to be commited. The following sequence is necessary to enable this feature:

1. NfcOpen()
2. NfcNdefMsgNew()
3. NfcNdefRecAddLeOob() or NfcNdefRecAddGeneric()
4. NfcNdefMsgCommit()
5. NfcFieldSense()
6. SystemStateSet(0)

Once SystemStateSet() is processed, the module enters deep sleep ***unless the reader is already energising the NFC field which will prevent deep sleep to persist.***

Please note that when the system wakes up, it is assumed that in a normal deployed scenario there will be an $autorun$ application so after reset your application will automatically restart. In your application you could call SYSINFO(2001) which will tell you what was the reason for waking up from reset. If you logically AND the result with the value 0x80000 and you end up with 0x80000, then it implies the wakeup was due to Wake-On-NFC.

```
IF (SYSINFO(2001) & 0x80000)==0x80000 THEN
      PRINT "We woke up because of NFC"
ENDIF
```

6.1.7 Events and Messages

In addition to the routines for manipulating the NFC interface, when an active reader generates a carrier field around the module's antenna a FIELD-ON event is generated, and conversely when the carrier field collapses because the active device moves away, a FIELD-OFF event is generated. When the Tag exposed by the module is actually read, then a TAG-READ event is generated.

The following is a list of events generated by the NFC manager which can be handled by user code.

EVNFC This is an event message with one INTEGER payload which identifies the event that happened as follows:

2	FIELD OFF (reader carrier has collapsed)
3	FIELD ON (reader carrier is active)
7	TAG READ (reader has finished reading the commited NDEF message)

6.1.8 NfcHardwareState

FUNCTION

This function is used to enable or disable the NFC hardware on the device.

Note: On the BL652 the 2 pins used for the NFC antenna are multifunction so that they are either for NFC or plain GPIO. However, this is set via a non-volatile configuration register in a special region of the onchip flash. These pins are by default set for NFC functionality and have appropriate protection from over energisation from an active field. Given this is a flash register, once the NFC functionality is disabled using this function, it can only be reactivated by reloading the entire firmware using the JLINK interface. It is not possible to reset this register when firmware is uploaded using the UART interface.

NFCHARDWARESTATE (interfaceNum, newState)

Returns	INTEGER, indicating the success of command: 0 Opened successfully 0x5A00 Invalid interface number 0x5A06 Enable Fail
Exceptions	▪ Local Stack Frame Underflow ▪ Local Stack Frame Overflow
Arguments	
interfaceNum	***byVal interfaceNum AS INTEGER*** For platforms that have multiple NFC interfaces, this identifies the interface to enable or disable and for platforms with only one interface specify 0 for this argument
newState	***byVal newState AS INTEGER*** Set to 0 to disable NFC functionality. Non-zero to enable.
Related Commands	NFCFIELDSENSE, NFCCLOSE, NFCNDEFMSGCOMMIT

Example:

```
//See subsection 'Sample Application 1'
```

6.1.9 NfcOpen

FUNCTION

This function opens the NFC interface identified by the 'interfaceNum' parameter, configure it as specified in the 'config$' future extensible string parameter and will return a handle which is used in appropriate subsequent NFC related function calls.

The 'interfaceNum' parameter exists as in future other smartBASIC based can potentially have multiple physical NFC interfaces.

NFCOPEN (interfaceNum, config$, nfcHandle)

Returns	INTEGER, indicating the success of command: 0 — Opened successfully 0x5A00 — Invalid interface number 0x5A04 — NFC hardware not available
Exceptions	▪ Local Stack Frame Underflow ▪ Local Stack Frame Overflow
Arguments	
interfaceNum	***byVal interfaceNum AS INTEGER*** For platforms that have multiple NFC interfaces, this identifies the interface to open and for platforms with only one interface specify 0 for this argument
config$	***byVal config$ AS STRING*** This is an extensible argument with 0 or more bytes which is used to configure the NFC interface as follows: Byte Value Description 0 0 Tag Type 2 Functionality A 0 value specifies default functionality, and more bytes will be allocated as needed to define appropriate new functionality
nfcHandle	***byRef nfcHandle AS INTEGER*** If the function fails, then on exit this parameter is set to INVALID_HANDLE (which is 0xFFFFFFFF), and if successful a valid handle to be used in susbsequent appropriate NFC related function calls.
Related Commands	NFCFIELDSENSE, NFCCLOSE, NFCNDEFMSGCOMMIT

Example:

```
//See subsections 'Sample Application 1' and 'Sample Application 2'
```

6.1.10 NfcClose

SUBROUTINE

This function closes the NFC interface identified by the 'nfcHandle' parameter and on exit the handle will be set to 0xFFFFFFFF so that it cannot be mistakenly used.

NFCCLOSE (nfcHandle)

Returns	None
Exceptions	▪ Local Stack Frame Underflow ▪ Local Stack Frame Overflow
Arguments	
nfcHandle	***byRef nfcHandle AS INTEGER***

	If the function is successful then on exit this variable will be set to 0xFFFFFFFF
Related Commands	NFCFIELDSENSE, NFCOPEN, NFCNDEFMSGCOMMIT

Example:

```
//See subsection 'Sample Application 2'
```

6.1.11 NfcFieldSense

FUNCTION

This function is used when the device is in passive mode to enable or disable field sensing so that an active device can communicate with it.

NFCFIELDSENSE (nfcHandle, fNewState)

Returns	INTEGER, indicating the success of command: 0 Opened successfully 0x020C Invalid handle 0x5A03 NFC interface is not open 0x5AEx An underlying stack related error
Exceptions	▪ Local Stack Frame Underflow ▪ Local Stack Frame Overflow
Arguments	
nfcHandle	***byVal nfcHandle AS INTEGER*** This is the handle returned by a prior call of NfcOpen()
fNewState	***byVal fNewState AS INTEGER*** Specify 0 to disable field sensing and non-zero to enable it
Related Commands	NFCOPEN, NFCCLOSE, NFCNDEFMSGCOMMIT

Example:

```
//See subsections 'Sample Application 1' and 'Sample Application 2'
```

6.1.12 NfcNdefMsgNew

FUNCTION

An NDEF record can be as long as 4.2 billion bytes and since an NDEF message is an array of NDEF records the whole message can theoretically be multiples of 4.2 billion bytes.

In practice most tags only have a limited amount of memory (typically less than 32K). Most messages are less than a kilobyte in the context of the *smart*BASIC based device.

All the NDEF messages that will be created using the API exposed in this device will not be of the same length, but the memory must be persistent so that it can be delivered to a reader when required.

Therefore, this *smart*BASIC implementation, requires that the creation of an NDEF message starts with dynamically allocated memory which can be released as and when required.

This function is used to create a dynamic buffer in RAM. This buffer is of the minimum length specified by the 'maxMSgLen' parameter and is associated with a 'ndefHandle' for which a valid handle value is returned if the memory requested was successfully acquired from the underlying memory manager. There is also an absolute limit on this implementation with regards to maximum amount of memory that can be allocated and that value can be obtained via AT I 2052 command or from within a running app using SYSINFO(2052).

The ‘ndefHandle’ is subsequently used for various API calls to make up the full message by writing single records at a time.

Note that NDEF records are added to this buffer using variious NfcNdefRecAddXXXX() functions and at any time the function NfcNdefMsgGetInfo() can be used to see how big the buffer is and how much of that is used.

NFCNDEFMSGNEW (maxMsgLen, ndefHandle)

Returns	INTEGER, indicating the success of command: 0 — Opened successfully 0x5A09 — Invalid max memory required 0x5A0A — Memory could not be acquired SYSINFO(2052) returns max len allowed in this system 0x5A0B — No spare handles as available SYSINFO(2051) returns max ndef handles in this system
Exceptions	▪ Local Stack Frame Underflow ▪ Local Stack Frame Overflow
Arguments	
maxMsgLen	***byVal maxMsgLen AS INTEGER*** This specifies the maximum expected length of the NDEF message that will be stored in the memory acquired. If, while adding a record, it does not fit, use NfcNdefMsgDelete() function to release that memory and call this function again with a larger value and try again.
ndefHandle	***byRef ndefHandle AS INTEGER*** If the function fails, then on exit this parameter is set to INVALID_HANDLE (which is 0xFFFFFFFF), and if successful a valid handle to be used in susbsequent appropriate NDEF related function calls.
Related Commands	NFCNDEFMSGCOMMIT, NFCNDEFDELETE, NFCDEFMSGGETINFO, NFCNDEFMSGRESET,NFCNDEFRECADDLEOOB,NFCNDEFRECADDGENERIC

Example:

```
//See subsections 'Sample Application 1' and 'Sample Application 2'
```

6.1.13 NfcNdefMsgDelete

FUNCTION

This function is used to release the memory block associated with an ndefHandle that was aquired using NfcNdefMsgNew().

NFCNDEFMSGDELETE (ndefHandle)

Returns	INTEGER, indicating the success of command: 0 — Opened successfully 0x5A20 — Cannot be deleted as it has been commited and locked to the stack using NfcNdefMsgCommit() 0x5A0C — The handle is not valid
Exceptions	▪ Local Stack Frame Underflow ▪ Local Stack Frame Overflow

Arguments	
ndefHandle	***byVal ndefHandle AS INTEGER*** The handle of the memory block that was acquired using NfcNdefMsgNew
Related Commands	NFCNDEFMSGCOMMIT, NFCNDEFNEW, NFCDEFMSGGETINFO, NFCNDEFMSGRESET,NFCNDEFRECADDLEOOB,NFCNDEFRECADDGENERIC

Example:

```
//See subsections 'Sample Application 1'
```

6.1.14 NfcNdefMsgGetInfo

FUNCTION

After an NDEF message memory buffer has been acquired using NfcNdefMsgNew(), call this function to see how much of the memory is used after adding records.

This function is particularly useful during the smartBASIC app development as it allows the optimisation of memory usage after all testing has been done to then reduce the size of the buffer for final deployment.

NFCNDEFMSGGETINFO (ndefHandle, records, memTotal, memUsed)

Returns	INTEGER, indicating the success of command: 0 Opened successfully 0x5A0C The handle is not valid
Exceptions	▪ Local Stack Frame Underflow ▪ Local Stack Frame Overflow
Arguments	
ndefHandle	***byRef ndefHandle AS INTEGER*** The handle of the memory block that was acquired using NfcNdefMsgNew.
records	***byRef records AS INTEGER*** If the ndefHandle is valid, then on exit this will be updated with the number of records currently added to the message.
memTotal	***byRef MemTotal AS INTEGER*** If the ndefHandle is valid, then on exit this will be updated with the total memory allcocated for this message (value that was specified in NfcNdefMsgNew()) when the handle was acquired.
memUsed	***byRef MemUsed AS INTEGER*** If the ndefHandle is valid, then on exit this will be updated with the memory that has been used in the buffer. For deployed systems, you want this to be as close to memTotal as possible to optimise memory usage.
Related Commands	NFCNDEFMSGCOMMIT, NFCNDEFDELETE, NFCDEFMSGNEW, NFCNDEFMSGRESET,NFCNDEFRECADDLEOOB,NFCNDEFRECADDGENERIC

Example:

```
//See subsections 'Sample Application 1' and 'Sample Application 2'
```

6.1.15 NfcNdefMsgReset

FUNCTION

After an ndef message has been used, this function can be used to reset the record count and memory used to 0 so that a new message with new records can be created without releasing the memory. It eliminates a heap free and malloc and so helps mitigate heap fragmentation.

NFCNDEFMSGRESET (ndefHandle)

Returns	INTEGER, indicating the success of command: 0 — Opened successfully 0x5A20 — Cannot be deleted as it has been commited and locked to the stack using NfcNdefMsgCommit() 0x5A0C — The handle is not valid
Exceptions	▪ Local Stack Frame Underflow ▪ Local Stack Frame Overflow
Arguments	
ndefHandle	***byVal ndefHandle AS INTEGER*** The handle of the memory block that was acquired using NfcNdefMsgNew
Related Commands	NFCNDEFMSGCOMMIT, NFCNDEFNEW, NFCDEFMSGGETINFO, NFCNDEFMSGDELETE,NFCNDEFRECADDLEOOB,NFCNDEFRECADDGENERIC

Example:

```
//See subsections 'Sample Application 1'
```

6.1.16 NfcNdefRecAddLeOob

FUNCTION

This function is used to add an NDEF record to a NDEF Message.

After an NDEF message memory buffer has been acquired using NfcNdefMsgNew(), call this function to add a 'Simplified Tag Format for a Single Bluetooth Carrier Record' as specified in the Bluetooth SIG specification "Bluetooth Secure Simple Pairing Using NFC" dated 2014-01-09.

This tag is a single record in the NDEF message and will contain the following BLE AD elements (same format as in BLE adverts).

- LE Bluetooth Local Device Address
- LE Role
- Appearance
- Local Name
- (Optional) Security Manager TK Value

Please note that due to the inclusion of the local device address LE Privacy should not be enabled otherwise the NFC record will soon contain a stale address and so the smartphone/tablet will not be able to make a connection and pair.

Note: The Local Device Address and Local Name is not provided in this function call as the underlying service routine will obtain both information from the stack. With regards to the Local Name, only the maximum characters you want to add to the record need be specified. Depending on the actual device name registered with the stack using BleGapSvcInit() function the appropriate AD element tag will be automatically used.

Warning: This function adds an NDEF record as per the specification mentioned above and publishes it as a Type 2 tag. You will not be able to interact with it using any iOS devices even when the iOS device (like the iPhone 6S) has NFC which is only used for Apple Pay. With Android, you will see inconsistent behaviour between different brands and OS versions. Hence any testing you perform is best done using something like an Arduino Uno and an Adafruit NFC Shield as shown above in the context of the two sample apps.

If you wish to experiment, use the function *NfcNdefRecAddGeneric()* which will allow you to create NDEF records of any type and payload.

NFCNDEFRECADDLEOOB (ndefHandle, maxDevName, appearance, role, flags, oobKey$)

Returns	INTEGER, indicating the success of command: 0 — Opened successfully 0x5A0C — The handle is not valid 0x5A13 — Invalid Device Name Length 0x5A14 — Invalid Appearance (has to be 0 .. 0xFFFF) 0x5A15 — Invalid Role 0x5A16 — Invalid OobKey (must be 0 or 16 bytes long) 0x5A17 — Invalid Flags value 0x5A11 — Inconsistent records in message (lengths don't make sense) 0x5AEC — Not enough space in msg buffer
Exceptions	▪ Local Stack Frame Underflow ▪ Local Stack Frame Overflow
Arguments	
ndefHandle	***byRef ndefHandle AS INTEGER*** The handle of the memory block that was acquired using NfcNdefMsgNew.
maxDevName	***byVal maxDevName AS INTEGER*** This specifies the maximum length of the device name to be added to the record. The appropriate AD type tag will automatically used if the length is shorter than the actuall name registered using BleGapSvcInit().
appearance	***byVal appearance AS INTEGER*** To be consistent, this should be the same 'appearance' that was provided when BleGapSvcInit() was called. This value can be used by the phone/tablet to present an icon after it reads the NFC tag.
role	***byVal role AS INTEGER*** This is the BLE role that this device prefers and the value to specify is as follows: 0 Only Peripheral Supported 1 Only Central Supported 2 Both, Peripheral Preferred 3 Both, Central Preferred
flags	***byVal flags AS INTEGER*** This should be the same flags value as was supplied in the most recent call of the function BleAdvRptInit(). Reproduced from BleAdvRptInit() .. *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.*

oobkey$	***byRef oobKey$ AS STRING*** If this string is empty then then Security Manager TK Value AD element is not added to the record. If it is exactly 16 bytes long then it will be added.
Related Commands	NFCNDEFMSGCOMMIT, NFCNDEFDELETE, NFCDEFMSGNEW, NFCNDEFMSGRESET, NFCNDEFRECADDGENERIC, NFCNDEFMSGGETINFO

Example:

```
//See subsection 'Sample Application 2'
```

6.1.17 NfcNdefRecAddGeneric

FUNCTION

This function is used to add an NDEF record to a NDEF Message.

After an NDEF message memory buffer has been acquired using NfcNdefMsgNew(), call this function to add any record of your choice where you can specify the Type, ID and Payload.

The payload can even be another NDEF message, which means you can create records where the payload is an embedded NDEF record. That schema has been seen in few implementations. This is why the payload is specified using a prepend string parameter 'payload0$', followed by a ndef handle 'ndefHandlePayload', and lastly a postpend string parameter 'payload1$'.

It is perfectly valid for any two out of <payload0$, ndefHandlePayload, payload1$> to be empty strings or an invalid handle.

NFCNDEFRECADDGENERIC (ndefHandle, tnf, type$, id$, payload0$, ndefHandlePayload, payload1$)

Returns	INTEGER, indicating the success of command: 0 — Opened successfully 0x5A0C — Either ndefHandle or ndefHandlePayload is not valid 0x5A18 — Invalid TNF value 0x5A12 — ndefHandlePayload is valid but is empty 0x5A11 — Inconsistent records in message (lengths don't make sense) 0x5A21 — type$ is empty 0x5A22 — type$ is too big 0x5A23 — id$ is too big 0x5AEC — Not enough space in message buffer
Exceptions	▪ Local Stack Frame Underflow ▪ Local Stack Frame Overflow
Arguments	
ndefHandle	***byRef ndefHandle AS INTEGER*** The handle of the memory block that was acquired using NfcNdefMsgNew.
tnf	***byVal tnf AS INTEGER*** This can only be in the range 0 to 7 as it needs to fit in the 3 bit field of the first byte of the record.
type$	***byRef type$ AS STRING*** This is string that has to be between 1 and 255 bytes long and specifies the content of the Type field in the record header.
id$	***byRef id$ AS STRING*** This is string that has to be between 0 and 255 bytes long and specifies the content of

	the ID field in the record header. If the string is empty, then the ID field, which is optional, is not added to the record header.
Payload0$	***byRef payload0$ AS STRING*** This is string can be empty. If not it is added to the payload of the record.
ndefHandlePayload	***byVal ndefHandlePayload AS INTEGER*** This can be 0xFFFFFFFF which is designated as an invalid handle and in that is ignored. If it is not 0xFFFFFFFF and not a valid handle then this routine will exit with an error. If a valid handle, but the message buffer is empty then routine will exit with an error. Finally if the message is not empty, then it is copied in its entirety to this record (including the header, not just the payload in that message) This allows a nested mechanism and as deep as the number of ndef message handles that can be created. Note that once, the content of this embedded message is copied, this embedded handle message can be reset to create yet another message for embedding.
Payload1$	***byRef payload1$ AS STRING*** This is string can be empty. If not it is added to the payload of the record
Related Commands	NFCNDEFMSGCOMMIT, NFCNDEFDELETE, NFCDEFMSGNEW, NFCNDEFMSGRESET, NFCNDEFRECADDLEOOB, NFCNDEFMSGGETINFO

Example:

```
//See subsections 'Sample Application 1'
```

6.1.18 NfcNdefMsgCommit

FUNCTION

After a message has been created and records added, it needs to be commited so that it can be served as a tag for an active reader to access.

This function is used to do that and if successfully commited, then the ndefHandle is locked and cannot be deleted or reset using the NfcNdefMsgDelete() or NfcNdefMsgReset() function respectively.

When the tag is read, an EVNFC message is thrown with context NFC_READ.

NFCNDEFMSGCOMMIT (nfcHandle, ndefHandle)

Returns	INTEGER, indicating the success of command: 0 Opened successfully 0x5A0C The handle is not valid
Exceptions	▪ Local Stack Frame Underflow ▪ Local Stack Frame Overflow
Arguments	
ndefHandle	***byRef ndefHandle AS INTEGER*** The handle that was returned by NfcOpen().
ndefHandle	***byRef ndefHandle AS INTEGER*** The handle of the memory block that was acquired using NfcNdefMsgNew.
Related Commands	NFCNDEFDELETE, NFCDEFMSGNEW, NFCNDEFMSGGETINFO, NFCNDEFMSGRESET,NFCNDEFRECADDLEOOB,NFCNDEFRECADDGENERIC

Example:

```
//See subsections 'Sample Application 1' and 'Sample Application 2'
```

6.2 System Configuration Routines

6.2.1 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) **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:

```
// Example :: SystemStateSet.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples
 //Put the module into deep sleep
 PRINT "\n"; SystemStateSet(0)
```

6.3 Flash Routines

6.3.1 Overview

smartBASIC language provides high level API for accessing the flash, if **both of these requirements are met**:-

1. The external serial (SPI) flash **must** be connected to BL652 SIO_12 (SFLASH_CS), SIO_14 (SFLASH_MISO), SIO_16 (SFLASH_CLK), and SIO_20 (SFLASH_MOSI)
2. The external flash connected **must** be one of the two:-
   - 4 Mbit Macronix MX25R4035F
   - 8 Mbit Macronix MX25R8035F

The smartBASIC Flash routines can then be used for fast access using open/read/write API functions as described in the following sections.

Note: By default the BL652 devkit contains an optional SPI Flash (4 Mbit Macronix MX25R4035F) which can be used to demonstrate the Flash routines. However, the SPI flash is not connected. To connect the optional flash, solder bridges SB4, SB5, SB6, SB7 must be individually shorted.

6.3.2 FlashOpen

This function is used to open access to the flash memory in raw mode. It returns the total size of the memory accessible and the sector size.

FLASHOPEN (totalSize, sectorSize)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments	
totalSize	***byRef totalSize AS INTEGER*** The total memory in bytes available (will be 0 if flash is not detected).
sectorSize	***byRef sectorSize AS INTEGER*** The sector sizes in this block on memory in bytes.

Example:

```
//Example :: FlashOpen.sb
DIM rc, nTotalSize, nSectorSize
//open the flash memory in raw mode
rc = FlashOpen(nTotalSize,nSectorSize)
IF rc == 0 THEN
    PRINT "\nOpened flash successfully"
    PRINT "\nTotal Size=";nTotalSize;"  Sector Size=";nSectorSize
ENDIF
```

Expected Output:

```
Opened flash successfully
Total Size=524288  Sector Size=4096
00
```

6.3.3 FlashRead

This function is used to read from the flash exposed by a previous FlashOpen() call. The number of actual bytes s returned – which is the same as strlen(data$) and will be less than or equal to nReadLen.

FLASHREAD (nOffset, nReadLen, data$)

Returns	Will return the length of data$ on exit.
Arguments	
nOffset	***byVal nOffset AS INTEGER*** The offset to read from.
nReadLen	***byVal nReadLen AS INTEGER*** The number of bytes to read (the maximum allowed value is 1024 bytes).
Data$	***byRef data$ AS INTEGER*** The data will be read into this string.

Example:

```
//Example :: FlashRead.sb
DIM rc, nTotalSize, nSectorSize, nOffset, nReadLen, data$
//open the flash memory in raw mode
rc = FlashOpen(nTotalSize,nSectorSize)
IF rc == 0 THEN
    PRINT "\nOpened flash successfully"
ENDIF
data$ = ""
nOffset = 4088 : nReadLen = 4
rc = FlashRead(nOffset,nReadLen,data$)
PRINT "\nRead flash data: "
PRINT "\ndata=";StrHexize$(data$);" nReadLen=";nReadLen
```

Expected Output:

```
Opened flash successfully
Read flash data:
data=FFFFFFFF nReadLen=4
00
```

6.3.4 FlashWrite

This function is used to write to the bank of flash previously exposed by FlashOpen(). Please note that if the new data results in a bit reversal from 0 to 1 then the write will fail. A bit reversal from 0 to 1 can only be achieved by erasin a full sector using the function FlashErase().

FLASHWRITE (nOffset, data$, nExitInfo)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation. If FDV_VERIFY_FAIL is returned, then nExitInfo is equal to the offset that does not verify.
Arguments	
nOffset	***byVal nOffset AS INTEGER*** The offset to write to.
Data$	***byRef data$ AS INTEGER*** The data will be written from this string
nExitInfo	***byVal nExitInfo AS INTEGER*** If the return value is not 0x0000 (indicating success), then nExitInfo will contain further information about the reason of unsuccessful operation.

Example:

```
//Example :: FlashWrite.sb
DIM rc, nTotalSize, nSectorSize, nOffset, nReadLen, data$, nExitInfo
//open the flash memory in raw mode
rc = FlashOpen(nTotalSize,nSectorSize)
IF rc == 0 THEN
    PRINT "\nOpened flash successfully"
ENDIF
// Write some data
nOffset = 4088 : data$ = "ABCD"
rc = FlashWrite(nOffset,data$,nExitInfo)
IF rc == 0 THEN
    PRINT "\nWrote data to the flash successfully"
ENDIF
// clear the data$ variable before reading
data$ = ""
nOffset = 4088 : nReadLen = 4
rc = FlashRead(nOffset,nReadLen,data$)
PRINT "\nRead flash data: "
PRINT "\ndata=";data$;" nReadLen=";nReadLen
```

Expected Output:

```
Opened flash successfully
Wrote data to the flash successfully
Read flash data:
data=ABCD nReadLen=4
00
```

6.3.5 FlashErase

This function is used to erase a sector in the bank specified. The sector size in the block will have been returned in the FlashOpen function.

FLASHERASE (nOffset)

Returns	INTEGER, a result code. The most typical value is 0x0000, indicating a successful operation.
Arguments	
nOffset	***byVal nOffset AS INTEGER*** The offset in the sector with the block to erase. Any offset in that sector will suffice.

Example:

```
//Example :: FlashErase.sb
DIM rc, nOffset, nTotalSize, nSectorSize
//open the flash memory in raw mode
rc = FlashOpen(nTotalSize,nSectorSize)
IF rc == 0 THEN
    PRINT "\nOpened flash successfully"
ENDIF
// Erase flash at offset 4088
nOffset = 4088
rc = FlashErase(nOffset)
IF rc == 0 THEN
    PRINT "\nFlash erased successfully"
ENDIF
```

Expected Output:

```
Opened flash successfully
Total Size=524288  Sector Size=4096
00
```

6.3.6 FlashClose

This subroutine is used to close access to a block of flash in raw mode.

FLASHCLOSE()

Returns	Not acceptable as it is a subroutine
Arguments:	None

Example:

```
//Example :: FlashClose.sb
DIM rc, nTotalSize, nSectorSize
//open the flash memory in raw mode
rc = FlashOpen(nTotalSize,nSectorSize)
IF rc == 0 THEN
    PRINT "\nOpened flash successfully"
ENDIF
// Close access to the flash
FlashClose()
PRINT "\nClosed flash"
```

Expected Output:

```
Opened flash successfully
Closed flash
00
```

6.4 Cryptographic Routines

6.4.1 EccGeneratePubPrvKeys

This functions is used to generate public/private keypair based on the algorithm (ECC type) provided.

ECCGENERATEPUBPRVKEYS (nEccType, privKey$, pubKey$)

Returns	INTEGER, a result code. The most typical values are:- 0x0000 – Keys created successfully 0x5907 – CRYPTO_ECC_TYPE_UNKNOWN (Unknown ECC type) 0x0201 – MALLOC_FAIL (not enough memory to return the keys)
Arguments	
nEccType	***byVal nEccTypeAS INTEGER*** The ECC type to be used when calculating and generating the shared key. Possible values:- 0x10000 : Algorithm Curve 25519 (used in Eddystone EID)
privKey$	***byRef privKey$ AS STRING*** On exit, will contain the generated private key, size as appropriate for algorithm
pubKey$	***byRef pubKey$ AS STRING*** On exit, will contain the generated public key, size as appropriate for algorithm

See example for EccCalcSharedSecret().

6.4.2 EccCalcSharedSecret

This function is used to create a shared scalar value which will have the same value when the remote performs an equivalent calculation with its own local private key and this side's public key.

Essentially, calling EccGeneratePubPrvKeys() twice to create two sets of private and public keys and then calling EccPubSharedSecret() twice with the private from one and public from the other will generate the same sharedSecret$.

ECCCALCSHAREDSECRET (nEccType, privKey$, pubKey$, sharedSecret$)

Returns	INTEGER, a result code. The most typical values are:- 0x0000 – Keys created successfully 0x5907 – CRYPTO_ECC_TYPE_UNKNOWN (Unknown ECC type) 0x0201 – MALLOC_FAIL (not enough memory to return the keys)
Arguments	
nEccType	***byVal nEccTypeAS INTEGER*** The ECC type to be used when generating the public/private keypair. Possible values:- 0x10000 : Algorithm Curve 25519 (used in Eddystone EID)
privKey$	***byRef privKey$ AS STRING*** On entry contains the local private key, untouched on exit
pubKey$	***byRef pubKey$ AS STRING*** On entry contains the remote public key, untouched on exit

sharedSecret$	***byRef sharedSecret$ AS STRING*** On exit will contain the shared secret key

```
// Example :: EccCalcSharedSecret.sb

// Note: In real world scenarios, two devices generate their private/public
// key pair separately, then exchange the public key. Using the remote's
// public key and the own private key, the shared secret is generated, therefore
// ending with the same shared secret without exposing material that could be used to
// by a third party to decrypt in a reasonable amount of time.
// For simplicity, this example shows this process performed on one device only

dim rc, EccType : EccType = 0x10000
dim prvKey_A$, pubKey_A$, Secret_A$
dim prvKey_B$, pubKey_B$, Secret_B$

// Generate first Public/Private keypair
rc = EccGeneratePubPrvKeys(EccType, prvKey_A$, pubKey_A$)
if rc == 0 then
    PRINT "\rPrv Key A: "; strhexize$(prvKey_A$)
    PRINT "\rPub Key A: "; strhexize$(pubKey_A$)
endif

// Generate second Public/Private keypair
rc = EccGeneratePubPrvKeys(EccType, prvKey_B$, pubKey_B$)
if rc == 0 then
    PRINT "\rPrv Key B: "; strhexize$(prvKey_B$)
    PRINT "\rPub Key B: "; strhexize$(pubKey_B$)
endif

// Compute first shared secret using private key A and public key B
rc = EccCalcSharedSecret(EccType, prvKey_A$, pubKey_B$, Secret_A$)
if rc == 0 then
    PRINT "\rShared Secret 1: "; strhexize$(Secret_A$)
endif

// Compute second shared secret using private key B and public key A
rc = EccCalcSharedSecret(EccType, prvKey_B$, pubKey_A$, Secret_B$)
if rc == 0 then
    PRINT "\rShared Secret 2: "; strhexize$(Secret_B$)
endif

// Compare keys to check if they are the same
If StrCmp(Secret_A$, Secret_B$)==0 then
    PRINT "\rThe generated shared secret keys are identical"
else
    PRINT "\rThe generated shared secret keys do not match"
Endif
```

Expected Output:

```
Prv Key A: 3A803352CFBBE969C28952C9950706A7F807C3B3974B65FEFD69C15A258C56EF
Pub Key A: 92F2589A0B08F0A1ADBC42F38FFB3093823257607C5DC0F4AF9DDEFE85E34030
Prv Key B: 10C9D43736EC510DE317732EF1C057954EB11FBD7800B1C6D827E63FB2657B5F
Pub Key B: 91FADCE2BD6E2FE7DF7F3251B2879753753D8F7F7D85978E2F0743DB3AE20577
Shared Secret 1: 3666BE535446B3E8A99970982EB2CE79C2501312CE2D30872DDB540A46453D23
Shared Secret 2: 3666BE535446B3E8A99970982EB2CE79C2501312CE2D30872DDB540A46453D23
The generated shared keys are identical
```

6.4.3 EccHmacSha256

This function is used to generate a HMAC-SHA256 authenticated hash of the content of data$ using the key supplied which can be from 0 to 64 bytes in length.

ECCHMACSHA256 (key$, data$, hmacOut$)

Returns	INTEGER, a result code. The most typical values are:- 0x0000 – Keys created successfully 0x0201 – MALLOC_FAIL (not enough memory to return the keys)
Arguments	
Key$	***byRef key$ AS STRING*** On entry contains a key from 0 to 64 bytes long and untouched on exit
data$	***byRef data$ AS STRING*** On entry contains the data to be hashed and untouched on exit
hmacOut$	***byRef hmacOut$ AS STRING*** On exit will contain the hmac output, use strlen() to determine length

```
//Example :: EccHmacSha256.sb

dim rc, key$
dim data_A$, hmacOut_A$
dim data_B$, hmacOut_B$

key$ = "KEY"
data_A$ = "AAAAB"
data_B$ = "AAAAA"

// Generate the HMAC-SHA256 for the first data
rc = EccHmacSha256(key$, data_A$, hmacOut_A$)
if rc == 0 then
    PRINT "\rHMAC of data_A: "; strhexize$(hmacOut_A$)
endif

// Generate the HMAC-SHA256 for the second data
rc = EccHmacSha256(key$, data_B$, hmacOut_B$)
if rc == 0 then
    PRINT "\rHMAC of data_A: "; strhexize$(hmacOut_B$)
endif

// Compare the HMAC-SHA256 outputs
if StrCmp(hmacOut_A$, hmacOut_B$) == 0 then
    PRINT "\rData A matches Data B"
else
    PRINT "\rData A does not match Data B"
endif
```

Expected Output:

```
HMAC of data_A: 7DB831431B6B7CDACE411C9F51CCC550EF1C20FB0812A24B7BBE12AE4332BB20
HMAC of data_A: 7DBF238349A98AB446AB8B4596E12E3729653ADA1E1A4B9ADA57C507E2021034
Data A does not match Data B
```

6.5 Watchdog Timer

6.5.1 WdtStart

FUNCTION

This function starts a watchdog timer with nResetTimeout in seconds. If the timer is not reset within nResetTimeout seconds, the module will reset.

Returns	INTEGER, a result code and the most typical values are following:- 0x0000 := Success 0x5262 := An invalid time has been provided (i.e. the value is outside the range) 0x5263:= The watchdog timer is already running
Arguments	
nResetTimeout	***byVal nResetTimeout AS INTEGER*** The reset timeout in seconds. Valid range is between 0-131072.

```
//Example :: WdtStart
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

DIM rc, nTimeout
nTimeout = 60
// Start a timer for 60 seconds
rc = WdtStart(nTimeout)
if rc == 0 then
    PRINT "Watchdog Timer started\n"
else
    PRINT "Failed to start Watchdog Timer\n"
endif

WaitEvent
```

Expected Output:

```
Watchdog Timer started
```

6.5.2 WdtReset

FUNCTION

This function resets the watchdog timer. WdtStart should be called before this function can be used.

Returns	Will return a resultcode and the most typical value is 0x0000 indicating success.
Arguments	**None**

```
//Example :: WdtReset
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

DIM rc, nTimeout
nTimeout = 60
// Start a  watchdog timer for 60 seconds
rc = WdtStart(nTimeout)

// Start a recurring normal timer to reset the watchdog timer every 55 seconds
```

```
Timerstart(1, 55000, 1)

Function HandlerTimer1()
    rc = WdtReset()
    if rc == 0 then
        print "Watchdog timer reset successfully\n"
    endif
Endfunc 1

OnEvent EVTMR1 CALL HandlerTimer1

WaitEvent
```

Expected Output:

```
Watchdog timer reset successfully
```

6.5.3 WdtIsRunning

FUNCTION

This function starts a watchdog timer with nResetTimeout in seconds. If the timer is not reset within nResetTimeout seconds, the module will reset.

Returns	Will return the following value:- 0 := Not Running 1 := Running
Arguments	**None**

```
//Example :: WdtIsRunning
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

DIM rc, nTimeout
nTimeout = 60
// Start a timer for 60 seconds
rc = WdtStart(nTimeout)

rc = WdtIsRunning()
if rc == 1 then
    PRINT "Watchdog is running\n"
else
    PRINT "Watchdog is not running\n"
endif

WaitEvent
```

Expected Output:

```
Watchdog is running
```

6.6 Miscellaneous Routines

6.6.1 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

Example:

```
// Example :: ReadPwrSupplyMv.sb
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

 //read and print the supply voltage
 PRINT "\nSupply voltage is "; ReadPwrSupplyMv();"mV"
```

Expected Output:

```
Supply voltage is 3343mV
```

6.6.2 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.

Please note that when the power supply rises above this value and then drops again, the power fail warning event will NOT be thrown again, unless this function is called explicitly again in the event handler.

In addition, if the event is enabled by calling this function AND the supply voltage is still below the threshold then all flash write and erase operations will fail silently (for example, like pairing [with bonding] will fail to retain the keys). Likewise NvRecordSet (and all other operations that involve writing to flash memory) will silently fail and nothing will be written.

6.6.2.1 *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 (nThreshMv)

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. If 0 is supplied then low supply voltage notification is disabled which implies flash operation is no longer affected.

Example:

```
// Example :: SetPwrSupplyThreshMv.sb
```

```
// https://github.com/LairdCP/BL652-Applications/tree/master/UserGuideExamples

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…
```

7 Events and 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.

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 never needs the complexity of locking variables and objects.

The subsystems which generate events and messages relevant to the routines described in this guide are as follows:

- BLE events and messages as described here.
- Generic Characteristics events and messages as described here.

8 Miscellaneous

8.1 Bluetooth Result Codes

There are some operations and events that provide a single byte Bluetooth HCI result code (such as 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.

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_LINKKEY_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**

9 Acknowledgements

9.1 AES Encryption

The following are required acknowledgements to address our use of open source code on the BL652 to implement AES encryption. Laird's implementation includes the following files: **aes.c** and **aes.h**.

Copyright (c) 1998-2008, Brian Gladman, Worcester, UK. All rights reserved.

9.1.1 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.

9.1.2 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.

9.2 Micro-ECC

The following are required acknowledgements to address our use of open source code on the BL652 to implement Elliptic-Curve Diffie Hellman cryptography . Laird's implementation includes the following files: **uECC.c** and **uECC.h**.

Copyright (c) 2014, Kenneth MacKay

9.2.1 License Terms

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

9.2.2 Disclaimer

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

10 INDEX

Americas: +1-800-492-2320
Europe: +44-1628-858-940
Hong Kong: +852 2923 0610

Americas: +1-800-492-2320
Europe: +44-1628-858-940
Hong Kong: +852 2923 0610