BL600 *smart*BASIC Module

User Manual

Release 1.5.66.0

global solutions: local support ™

Americas: +1-800-492-2320 Option 2
Europe: +44-1628-858-940
Hong Kong: +852-2923-0610
www.lairdtech.com/wireless

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

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

Revision History

Version	Revisions Date	Change History
1.0	1 Feb 2013	Initial Release
1.1.50.0r3	3 Apr 2013	Production Release
1.1.51.0	15 Apr 2013	Incorporate review comments for JG
1.1.51.5	24 Apr 2013	Engineering release
1.1.53.10	8 May 2013	Engineering release with custom service capability
1.1.53.20	12 Jun 2013	Engineering release with Virtual Serial Service capability
1.2.54.0	29 Jun 2013	Production Release
1.2.55.3	26 Jul 2013	Engineering release with PWM & FREQUENCY output
1.2.55.5	8 Aug 2013	Engineering release with VSP/Uart Bridging
1.2.55.8	12 Aug 2013	Engineering release with AT+CFG command
1.2.55.12	29 Aug 2013	Engineering release with sysinfo$()
1.3.57.0	12 Sep 2013	Engineering release with UartCloseEx
1.4.59.0	19 Dec 2013	Engineering release v1.4.59.0
1.5.62.0	4 Jan 2014	Production release v1.5.62.0 (Softdevice 6.0.0)
1.5.65.0	24 Feb 2014	Engineering release v1.5.65.0 (Softdevice 6.0.0)
1.5.66.0	28 Mar 2014	Production release v1.5.66.0 (Softdevice 6.0.0)

Contents

1. Introduction

This user manual provides detailed information on Laird Technologies *smart*BASIC language which is embedded inside the BL600 series Bluetooth Low Energy (BLE) modules. This manual is designed to make handling BLE-enabled end products a straightforward process and it includes the following:

- An explanation of the language's core and extension functions
- Instructions on how to start using the tools
- A detailed description of all language components and examples of their use

The Laird website contains many complex examples which demonstrate complete applications. For those with programming experience, *smart*BASIC is easy to use because it is derived from BASIC language.

BASIC, which stands for **B**eginners **A**ll-Purpose **S**ymbolic **I**nstruction **C**ode, was developed in the early 1960s as a tool for teaching computer programming to undergraduates at Dartmouth College in the United States. From the early 70s to the mid-80s, BASIC, in various forms, was one of the most popular programming languages and the only user programming language in the first IBM PC to be sold in the early 80s. Prior to that, the first Apple computers were also deployed with BASIC.

Both BASIC and *smart*BASIC are interpreted languages – but in the interest of run-time speed on an embedded platform which has limited resources, *smart*BASIC's program text is parsed and saved as bytecodes which are subsequently interpreted by the run-time engine to execute the application. On the BL600 module platform, the parsing from code test to bytecode is done on a Windows PC using a free cross-compiler. Other platforms with more firmware code space also offer on-board compiling capabilities.

The early BASIC implementations were based on source code statements which, because they were line numbered, resulted in non-structured applications that liberally used 'GOTO' statements.

At the outset, *smart*BASIC was developed by Laird to offer structured programming constructs. It is not line number based and it offers the usual modern constructs like subroutines, functions, **while**, **if** and **for** loops.

*smart*BASIC offers further enhancement which acknowledges the fact that user applications are always in unattended use cases. It forces the development of applications that have an event driven structure as opposed to the classical sequential processing for which many BASIC applications were written. This means that a typical *smart*BASIC application source code consists of the following:

- Variable declarations and initialisations
- Subroutine definitions
- Event handler routines
- Startup code

The source code ends with a final statement called WAITEVENT, which never returns. Once the run-time engine reaches the WAITEVENT statement, it waits for events to happen and, when they do, the appropriate handlers written by the user are called to service them.

Why Do We Need *smart*BASIC?

Programming languages are mostly designed for arithmetic operations, data processing, string manipulation, and flow control. Where a program needs to interact with the outside world, like in a BLE device, it becomes more complex due to the diversity of different input and output options. When wireless connections are involved, the complexity increases. To compound the problem, almost all wireless standards are different, requiring a deep knowledge of the specification and silicon implementations in order to make them work.

We believe that if wireless connectivity is going to be widely accepted, there must be an easier way to manage it. *smart*BASIC was developed and designed to extend a simple BASIC-like programming language with all of the tokens that control a wireless connection using modern language programming constructs.

*smart*BASIC differs from an object oriented language in that the order of execution is generally the same as the order of the text commands. This makes it simpler to construct and understand, particularly if you're not using it every day.

Our other aim in developing *smart*BASIC from the ground up is to make wireless design of products both simple and similar in look and feel for all platforms. To do this we are embedding *smart*BASIC within our wireless modules along with all of the embedded drivers and protocol stacks that are needed to connect and transfer data. A run-time engine interprets the customer applications (reduced to bytecode) that are stored there, allowing a complete product design to be implemented without the need for any additional external processing capability.

Why Write Applications?

*smart*BASIC for BLE has been designed to make wireless development quick and simple, vastly cutting down time to market. There are three good reasons for writing applications in *smart*BASIC:

- Since the module can auto launch the application each time it powers up, you can implement a complete design within the module. At one end, the radio connects and communicates while, at the other end, external interactions are available through the physical interfaces such as GPIOs, ADCs, I2C, SPI, and UART.
- If you want to add a range of different wireless options to an existing product, you can load applications into a range of modules with different wireless functionality. This presents a consistent API interface defined to your host system and allows you to select the wireless standard at the final stage of production.
- If you already have a product with a wired communications link, such as a modem, you can write a *smart*BASIC application for one of our wireless modules that copies the interface for your wired module. This provides a fast way for you to upgrade your product range with a minimum number of changes to any existing end user firmware.

In many cases, the example applications on our website and in the applications manual can be modified to speed up the development process.

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 GPIO and ADC
- Wired communication interfaces like UART, I2C, and SPI
- A *smart*BASIC run-time engine
- Program accessible flash memory which contains a robust flash file system exposing a conventional file system and a database for storing user configuration data
- Voltage regulators and brown-out detectors

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

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

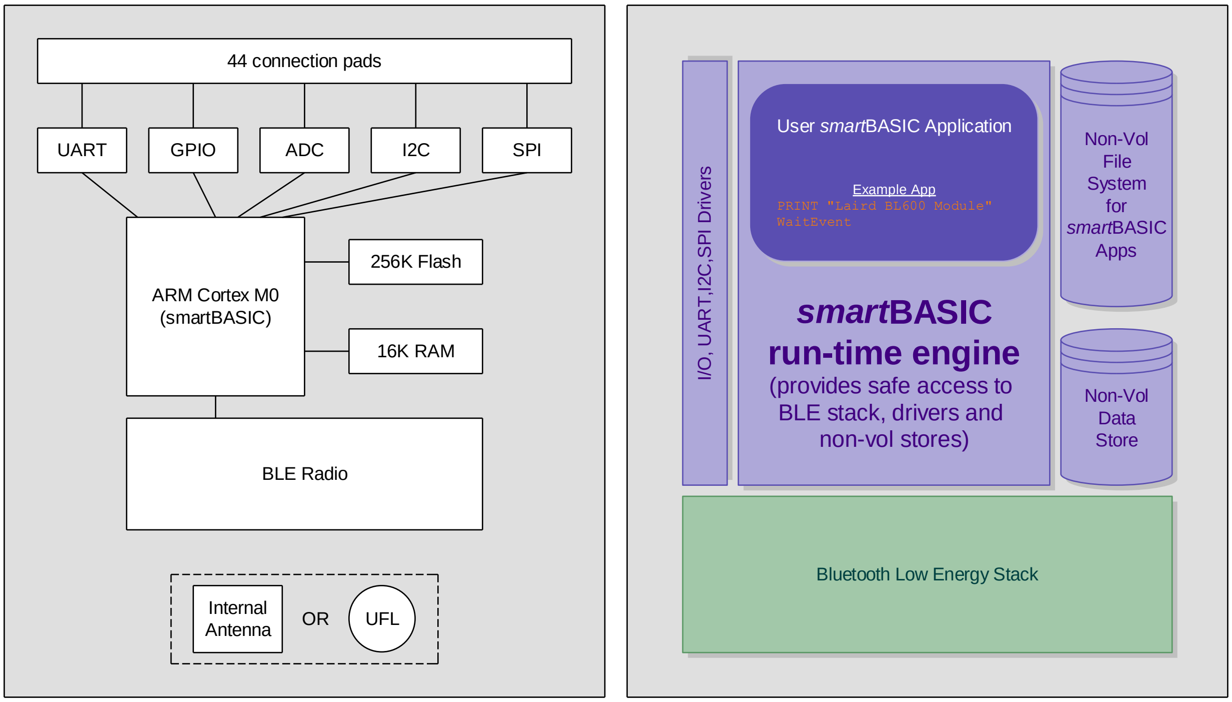


Figure 1: BLE smart *BASIC module block diagram*

*smart*BASIC Essentials

*smart*BASIC is based upon the BASIC language. It has been designed to be highly efficient in terms of memory use, making it ideal for low cost embedded systems with limited RAM and code memory.

The core language, which is common throughout all *smart*BASIC implementations, provides the standard functionality of any program, such as:

- Variables (integer and string)
- Arithmetic functions
- Binary operators
- Conditionals
- Looping
- Functions and subroutines
- String processing functions
- Arrays (single dimension only)
- I/O functions
- Memory management
- Event handling

The language on the various platforms differs by having a sophisticated set of target-specific extensions, such as BLE for the module described in this manual.

These extensions have been implemented as additional program functions that control the wireless connectivity of the module including, but not limited to, the following:

- Advertising
- Connecting
- Security – encryption and authentication
- Power management
- Wireless status

Developing with *smart*BASIC

*smart*BASIC is one of the simplest embedded environments on which to develop because much of the functionality comes prepackaged. The compiler, which can be internal or external on a Windows PC, compiles source text on a line-by-line basis into a stream of bytes (or bytecode) that can be stored to a custom-designed flash file system. Following that, the run-time engine interprets the application bytecode in-situ from flash.

To further simplify development, Laird provides its own custom developed application called UWTerminal which is a full blown customised terminal emulator for Windows, available upon request at no cost. See Chapter 2 – UWTerminal for information on writing *smart*BASIC applications using UWTerminal.

UWTerminal also embeds *smart*BASIC to automate its own functionality; the extension *smart*BASIC functions facilitate the automation of terminal emulation functionality.

*smart*BASIC Operating Modes

Any platform running *smart*BASIC has up to three modes of operation:

- Interactive Mode – In this mode, commands are sent via a streaming interface which is usually a UART, and are executed immediately. This is similiar to the behavior of a modem using AT commands. Interactive mode can be used by a host processor to directly configure the module. **It is also used to manage the download and storage of *smartBASIC* applications in the flash file system subsequently used in run-time mode.**
- Application Load Mode – This mode is only available if the platform includes the compiler in the firmware image. The BLE module has limited firmware space and so compilation is only possible outside the module using a *smart*BASIC cross-compiler (provided for free).

  If this feature is available, then the platform switches into Load mode when the compile (AT+CMP) command is sent by the host.

  In this mode the relevant application is checked for syntax correctness on a line-by-line basis, tokenised to minimise storage requirements, and then stored in a non-volatile file system as the compiled application. This application can then be run at any time and can even be designated as the application to be automatically launched upon power up.
- Run-time Mode – In Run-time mode, pre-compiled *smart*BASIC applications are read from program memory and executed in-situ from flash. The ability to run the application from flash ensures that as much RAM memory as possible is available to the user application for use as data variables.

On startup, an external GPIO input pin is checked. If the state of the input pin is asserted (high or low, depending on the platform) and **$autorun$** exists in the file system, the device enters directly into Run-time mode and the application is automatically launched. If that input pin is not asserted, then regardless of the existence of the autorun file, it enters Interactive mode.

If the auto-run application completes or encounters a STOP or END statement, then the module returns to Interactive mode.

It is therefore possible to write autorun applications that continue to run and control the module's behavior until power-down, which provides a complete embedded application.

The modes of the module and transitions are illustrated in Figure 2.

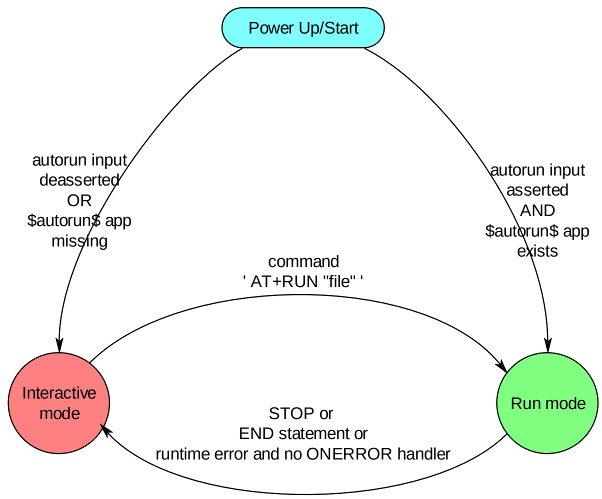


Figure 2: Module modes & transitions

Types of Applications

There are two types of applications used within a *smart*BASIC module. In terms of composition, they are the same but run at different times.

- **Autorun** – This is a normal application named **$autorun$** (case insensitive). When a *smart*BASIC module powers up, it looks for the **$autorun$** application. If it finds it and if the nAutoRUN pin of the module is at **0v**, then it executes it. Autorun applications may be used to initialise the module to a customer's desired state, make a wireless connection, or provide a complete application program. At the completion of the autorun application, which is when the last statement returns or a STOP or END statement is encountered, a *smart*BASIC module reverts to Interactive mode.

  In unattended use cases, the autorun application is expected to never terminate. It is typical for the last statement in an application to be the WAITEVENT statement.

  Developers should be aware that an autorun application does not need to complete and exit to Interactive mode. The application can be a complete program that runs within the *smart*BASIC module, *removing the requirement for an external processor*.

  Applications can access the GPIOs and ADCs and use ports (UART, I2C, and SPI, for example) to interface with peripherals such as displays and sensors.

Note: By default, when the autorun application starts up and if the STDOUT is the UART, then it will be in a closed state. If a PRINT statement is encountered which results in output, then the UART is automatically opened using default comms paramaters.

- **Other –** Applications can be loaded into the BASIC module and run under the control of an external host processor using the AT+RUN command. The flash memory supports the storage of multiple applications. Note that the storage space is module dependent. Check the individual module data sheet.

Non Volatile Memory

All *smart*BASIC modules contain user accessible flash memory. The quantity of memory varies between modules; check the relevant datasheet.

The flash memory is available for three purposes:

- **File Storage –** Files which are not applications can also be stored in flash memory certificates (for example X.501). The most common non-application files are data files for application.
- **Application Storage –** Storage of user applications and the AT+RUN command is used to select which application runs.
- **Non-volatile records –** Individual blocks of data can be stored in non-volatile memory in a flat database where each record consists of a 16 bit user defined ID and data consisting of variable length. This is useful for cases where program specific data needs to be preserved across power cycles. For example, passwords.

Using the Module's Flash File System

All *smart*BASIC modules hold data and application files in a simple flash file system which was developed by Laird and has some similarity to a DOS file system. Unlike DOS, it consists of a single directory in which all of the files are stored.

Note: When files are deleted from the flash file system, the flash memory used by that file is not released. Therefore, repeated downloads and deletions eventually fill the file system, requiring it to be completely emptied using the AT&F1 command.

The command AT I 6 returns statistics related to the flash file system when in interactive mode. From within a *smart*BASIC application, the function SYSINFO(x), where x is 601 to 606 inclusive, returns similar information.

Note: Non-volatile records are stored in a special flash segment that is capable of coping with cases where there is no free unwritten flash but there are many deleted records.

2. Getting Started

This chapter is a quick start guide for using *smart*BASIC to program an application. It shows the key elements of the BASIC language as implemented in the module and guides you through using UWTerminal (a Laird Terminal Emulation utility available for free) and Laird's Development Kit to test and debug your application.

For the purpose of this chapter, the examples are based upon Laird's BL600, a BLE module. However, the principles apply to any *smart*BASIC enabled module.

Requirements

To replicate this example, you need the following items:

- A BL600 series development kit
- UWTerminal application (contact Laird for the latest version). The UWTerninal must be at least v6.50. Save the application to a suitable directory on your PC.
- A cross-compiler application with a name typically formatted as *XComp_dddddddd_aaaa_bbbb.exe*, where dddddddd is the first non-space eight characters from the response to the AT I 0 command and aaaa/bbbb is the hexadecimal output to the command AT I 13.

Note: **aaaa/bbbb** is a hash signature of the module so that the correct cross-compiler is used to generate the bytecode for download. When an application is launched in the module, the hash value is compared against the signature in the run-time engine and, if there is a mismatch, the application is aborted.

Connecting Things Up

The simplest way to power the development board and module is to connect a USB cable to the PC. The development board regulates the USB power rail and feeds it to the module.

Note: The current requirement is typically a few mA with peak currents not exceeding 20 mA. We recommend connecting to a powered USB hub or a primary USB port.

UWTerminal

UWTerminal is a terminal emulation application with additional GUI extensions to allow easy interactions with a *smart*BASIC-enabled module. It is similar to other well-known terminal applications such as Hyperterminal. As well as a serial interface, it can also open a TCP/IP connection either as a client or as a server. This aspect of UWTerminal is more advanced and is covered in the UWTerminal User's Guide. The focus of this chapter is its serial mode.

In addition to its function as a terminal emulator it also has *smart*BASIC embedded so you can locally write and run *smart*BASIC applications. This allows you to write *smart*BASIC applications which use the terminal emulation extensions that enable you to automate the functionality of the terminal emulator.

It may be possible in the future to add BLE extensions so that when UWTerminal is running on a Windows 8 PC with Bluetooth 4.0 hardware, an application that runs on a BLE module also runs in the UwTerminal environment.

Before starting UWTerminal, note the serial port number to which the development kit is connected.

Note: The USB to serial chipset driver on the development kit generates a virtual COM port. Check the port by selecting **My Computer > Properties > Hardware > Device Manager > Ports (COM & LPT)**.

To use UWTerminal, follow the steps below. Note that the screen shots may differ slightly as it is a continually evolving Windows application:

1. Switch on the development board, if applicable.
2. Start the UWTerminal application on your PC to access the opening screen (Figure 3).

Figure 3: UWTerminal opening screen

3. Click **Accept** to open the configuration screen.

Figure 4: UWTerminal Configuration screen

4. Enter the COM port that you have used to connect the development board. The other default parameters should be:

Baudrate	9600
Parity	None
Stop Bits	1
Data Bits	8
Handshaking	CTS/RTS

Note: **Comport** (not Tcp Socket) should be selected on the left.

5. Select **Poll for port** to enable a feature that attempts to re-open the comport in the event that the devkit is unplugged from the PC causing the virtual comport to disappear.
6. In Line Terminator, select the characters that are sent when you type **ENTER**.
7. Once these settings are correct, click **OK** to bring up the main terminal screen.

Getting Around UWTerminal

Figure 5: UWTerminal tabs and status lights

The following tabs are located at the top of the UWTerminal:

- **Terminal** – Main terminal window. Used to communicate with the serial module.
- **BASIC** – *smart*BASIC window. Can be used to run BASIC applications locally without a device connected to the serial port.

Note: You can use any text editor, such as notepad, for writing your *smart* BASIC applications. However, if you use an advanced text editor or word processor you need to take care that non-standard formatting characters are not incorporated into your *smart*BASIC application.

- **Config** – Configuration window. Used to set up various parameters within UWTerminal.
- **About** – Information window that displays when you start UWTerminal. It contains command line arguments and information that can facilitate the creation of a shortcut to the application and launch the emulator directly into the terminal screen.

The four LED-type indicators below the tabs display the status of the RS-232 control lines that are inputs to the PC. The colors are red, green, or white. White signifies that the serial port is not open.

Note: According to RS-232 convention, these are inverted from the logic levels at the GPIO pin outputs on the module. A 0v on the appropriate pin at the module signifies an asserted state

- **CTS** – Clear to Send. Green indicates that the module is ready to receive data.
- **DSR** – Data Set Ready. Typically connected to the DTR output of a peripheral.
- **DCD** – Data Carrier Detect.
- **RI** – Ring Indicate.

If the module is operating correctly and there is no radio activity, then CTS should be asserted (green), while DSR, DCD, and RI are deasserted (red). Again note that if all four are white (Figure 6), it means that the serial port of the PC has not been opened and the button labelled OpenPort can be used to open the port.

Figure 6: White lights

Note: At the time of this manual being written, the DSR line on the BL600 DevKit is connected to the SIO25 signal on the module which has to be configured as an output in a *smart* BASIC application so that it drives the PC's DSR line. The DCD line (input on a PC) is connected to SIO29 and should be configured as an output in an application and finally the RI line (again an input on a PC) is connected to SIO30. Please request a schematic of the BL600 development kit to ensure that these SIO lines on the modules are correct.

Figure 7: Control options

Next to the indicators are a number of control options (Figure 7) which can be used to set the signals that appear on inputs to the module.

- **RTS** and **DTR** – The two additional control lines for the RS-232 interface.

  Note: If CTS/RTS handshaking is enabled, the RTS checkbox has no effect on the actual physical RTS output pin as it is automatically controlled via the underlying Windows driver. To gain manual control of the RTS output, disable Handshaking in the Configuration window.

- **BREAK** – Used to assert a break condition over the Rx line at the module. It must be deasserted after use. A Tx pin is normally at logic high (> 3v for RS232 voltage levels) when idle; a BREAK condition is where the Tx output pin is held low for more than the time it takes to transmit 10 bits.
  If the BREAK checkbox is ticked then the Tx output is at non-idle state and no communication is possible with the UART device connected to the serial port.
- **LocalEcho –** Enables local echoing of any characters typed at the terminal. In default operation, this option box should be selected because modules do not reflect back commands entered in the terminal emulator.
- **LineMode** – Delays transmission of characters entered into UWTerminal until you press **Enter**. Enabling LineMode means that Backspace can be used to correct mistakes. We recommend that you select this option.
- **Clear** – Removes all characters from the terminal screen.
- **ClosePort** – Closes the serial port. This is useful when a USB to serial adaptor is being used to drive the development board which has been briefly disconnected from the PC.
- **OpenPort** – Re-opens the serial port after it has been manually closed.

Useful Shortcuts

There are a number of shortcuts that help speed up the use of UWTerminal.

Each time UWTerminal starts, it asks you to acknowledge the Accept screen and to enter the COM port details. If you are not going to change these, you can skip these screens by entering the applicable command line parameters in a shortcut link.

Follow these steps to create a shortcut to UWTerminal on your desktop:

1. Locate and right-click the UwTerminal.exe file, and then drag and drop it onto your desktop. In the dialog box, select **Create Shortcut**.
2. Right-click the newly created shortcut.
3. Select **Properties**.
4. Edit the **Target** line to add the following commands (Figure 8):

   accept com=n baud=bbb linemode

   (where *n* is the COM port that is connected to the dev kit and *bbb* is the baudrate)

Figure 8: Shortcut properties

Starting UWTerminal from this shortcut launches it directly into the terminal screen. At any time, the status bar on the bottom left (Figure 9) shows the comms parameters being used at that time. The two counts on the bottom right (Tx and Rx) display the number of characters transmitted and received.

The information within **{ }** denotes the characters sent when you hit **ENTER** on the keyboard.

Figure 9: Terminal screen status bar

Using UWTerminal

The first thing to do is to check that the module is communicating with UWTerminal. To do this, follow these steps:

1. Check that the CTS light is green (DSR, DCD, and RI should be red).
2. Type **`at`**.
3. Press **Enter**. You should get a `00` response (Figure 10).

Figure 10: Interactive command access

UWTerminal supports a range of interactive commands to interact directly with the module. The following ones are typical:

- **AT** – Returns 00 if the module is working correctly.
- **AT I 3** – Shows the revision of module firmware. Check to see that it is the latest version.
- **AT I 13** – Shows the hash value of the *smart*BASIC build.
- **AT I 4** – Shows the MAC address of the module.
- **AT+DIR** – Lists all of the applications loaded on the module.
- **AT+DEL "filename"** – Deletes an application from the module.
- **AT+RUN "filename"** – Runs an application that is already loaded on the module. Please be aware that if a filename does not contain any spaces, it is possible to launch an application by just entering the filename as the command.

The next chapter lists all of the Interactive commands.

First, check to see what is loaded on the module by typing **AT+DIR** and **Enter**:

```
at+dir

06        $factory$
00
```

If the module has not been used before then you should not see any lines starting with the two digit 06 sequence.

Your First *smart*BASIC Application

Create 'Hello World' App

Let's start where every other programming manual starts… with a simple program to display "Hello World" on the screen. We use Notepad to write the *smart*BASIC application.

To write this *smart*BASIC application, follow these steps:

1. Open Notepad.
2. Enter the following text:

```
print "\nHello World\n"
```

3. Save the file with single line *test1.sb*.

Note the following:

*smart*BASIC files can have any extension. UWTerminal, which is used to download an application to the module, strips all letters including and after the first '.' when the file is downloaded to the module.

For example, a file called "this.is.my.first.file.sb" will be downloaded as "this" and so will "this.is.my.second.file.sb", but "that.is.my.other.file.sb" will get downloaded as "that". This has special significance when you want to manage the special smartBASIC file called "$autorun$" which is run automatically on power up.

It means that you can have files called "$autorun$.heart.rate.sb" and "$autorun$.blood.pressure.sb" in a single folder and yet ensure that when downloaded they get saved as "$autorun$"

We recommend always using the extension .sb to make it easier to distinguish between *smart*BASIC files and other files. You can also associate this extension with your favorite editor and enable appropriate syntax highlighting. You may also encounter files with extension .sblib which are library source files provided by Laird to make developing code easier. They are included in your application using the #include statement which is is described later in this manual.

As you start to develop more complex applications, you may want to use a more fully-featured editor such as TextPad (trial version downloadable from www.textpad.com) or Notepad++ (free and downloadable from http://notepad-plus.sourceforge.net).

Tip: Laird recommends using **TextPad** or **Notepad++** because appropriate color syntax highlighting files are available for each build of the firmware which means all tokens recognised by smartBASIC are highlighted in various colors.

If you use **Notepad++**, do the following:

1. Copy the file *smartBASIC(notepad++).xml to* the **Notepad++** install folder.
2. Launch **Notepad++**.
3. From the menu, select **Language > Define your Language**.
4. In the new dialog box, click **Import…** and select the **smartBASIC(notepad++).xml** file from the folder you saved it to. A confirmation dialog box displays stating that the import was successful.
5. Close the User defined Language dialog box and then the **Notepad++** application.
6. Reopen **Notepad++** and select **Language > smartBASIC** from the menu.

If you use **TextPad**, do the following:

1. Copy the **smartBASIC(Textpad).syn** file from the firmware upgrade zip file to the Textpad install folder (specifically, the **system** subfolder).
2. As a one-time procedure, start TextPad.
3. Ensure no documents are currently open.
4. From the menu, select **Configure > Preferences**.
5. Select **Document Classes**.
6. In the *User defined classes* list box, add **smartBASIC**.
7. Click the plus sign (+) to expand Document Classes and select **smartBASIC**.
8. In the new *Files in class smartBASIC* list box, add the following two lines:
   *.sb
   *.sblib
9. Click **+** to expand smartBASIC and select **Syntax**.
10. Select **Enable syntax highlighting** to enable it.
11. In the Syntax definition file dropdown menu, enter or select the **smartBASIC(textpad).syn** file.
12. Click **OK**.

You should now have **TextPad** configured so that any file with file extension .sb or .sblib will be displayed with color syntax highlighting. To change the colors of the syntax highlighting, do the following:

1. From the Configure/Preferences dialog box, select the Document Classes plus sign (**+**) (next to smartBASIC) and select **Colors**.
2. Change the color of any of the items as necessary.
   For example, smartBASIC FUNCTIONs are 'Keywords 2', smartBASIC SUBs are 'Keywords 3' and smartBASIC Event and Message IDs (as used in the ONEVENT statement) are 'Keywords 4'

   Figure 11 displays a sample of what a *smart*BASIC code fragment looks like in TextPad:

```
'//******************************************************************************
'// Handler definitions
'//******************************************************************************

'//==============================================================================
'// Uart Inactivity timer handler
'//==============================================================================
function handlerUartTimer() as integer
  dim rc
  '//Close the uart, and set up TX/RX/RTS lines as gpio and for a hi-lo transition
  '//on the RX line to be detected
  if UartCloseEx(1) == 0 then
    rc=GpioSetFunc(21,2,1)    '//TX  - set high on default
    rc=GpioSetFunc(23,2,0)    '//RTS - set low by default
    rc=GpioSetFunc(22,1,2)    '//RX  - Pull high input & irq on hi2lo transition
    rc=GpioAssignEvent(UART_GPIO_ASSIGN_CHANNEL,22,1)
    if rc != 0 then
      print "\nGpioAssignEvent() Failed"
    endif
  endif
endfunc 1

'//==============================================================================
'// Delay before uart is opened
'//==============================================================================
function handlerOpenDelay() as integer
  dim rc
  '// free up the level transition detection
  rc=GpioUnAssignEvent(UART_GPIO_ASSIGN_CHANNEL)
  '//Open the uart
  rc=UartOpen(9600,0,0,"CN81H")
  '//send an ack character
  print "!"
endfunc 1
```

Figure 11: Example of a smartBASIC code fragment in TextPad

Download 'Hello World' App

You must now load the compiled output of this file into the *smart*BASIC module's File System so that you can run it.

1. To manage file downloads, right click on any part of the black UWTerminal screen to display the drop-down menu (Figure 12).

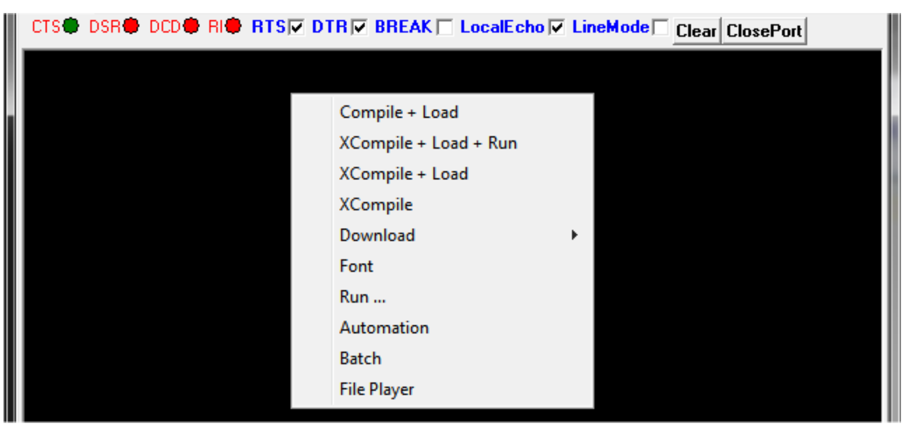


Figure 12: Right-click UWTerminal screen

2. Click **XCompile+Load** and navigate to the directory where you've stored your *test1.sb* file.

Note: Do not select **Compile+Load**.

3. Click **Open**. In UWTerminal, you should see the following display:

```
AT I 0
10     0      BL600Med
AT I 13
10     13     9E56 5F81
<<Cross Compiling [test1.sb]>>

AT+DEL "test1" +
AT+FOW "test1"
AT+FWRH "FE900002250000000000FFFFFFFF569E815FFC10"
AT+FWRH "FB70090054455354312E555743000110CE211000"
AT+FWRH "FB0009000D000A48656C6C6F20576F726C640A00"
AT+FWRH "CC211400A52000000110FD10F510"
AT+FCL
+++ DONE +++
```

Behind the scenes, the shortcut uses Interactive Commands to load the file onto the module. The first two AT I commands are used to identify the module so that the correct cross compiler can be invoked resulting in the text **<<Cross Compiling [test1.sb]>>**.

In this example, since the compilation is successful, the generated binary file must be downloaded and the **AT+DEL "filename" +** deletes any previous file with the same name that might already be on the module. The new file is downloaded using the **AT+FOW**, **AT+FWRH,** and **AT+FCL** commands. The strings following **AT+FWRH** consist of the binary data generated by the cross compiler. The **+++ DONE +++** signifies that the process of compiling and downloading was successfully accomplished.

There may be a possible failure in this process if the cross compiler cannot be located. In this case, the following window displays:

```
AT I 0
10     0      BL600Med
AT I 13
10     13     9E56 5F81
??? Cross Compiler [XComp_BL600Med_9E56_5F81.exe] not found ???
??? Please save a copy to the same folder as UwTerminal.exe ???
??? If you cannot locate the file, please contact the supplier ???
```

To fix this issue, locate the cross compiler application mentioned in between the [] brackets and save it to either the folder containing *UWTerminal.exe* or the folder that contains the *smart*BASIC application *test1.sb*

A compilation error may be another cause of failure. For example, if the print statement contains an error in the form of a missing " delimiter, then the following should display in a separate window:

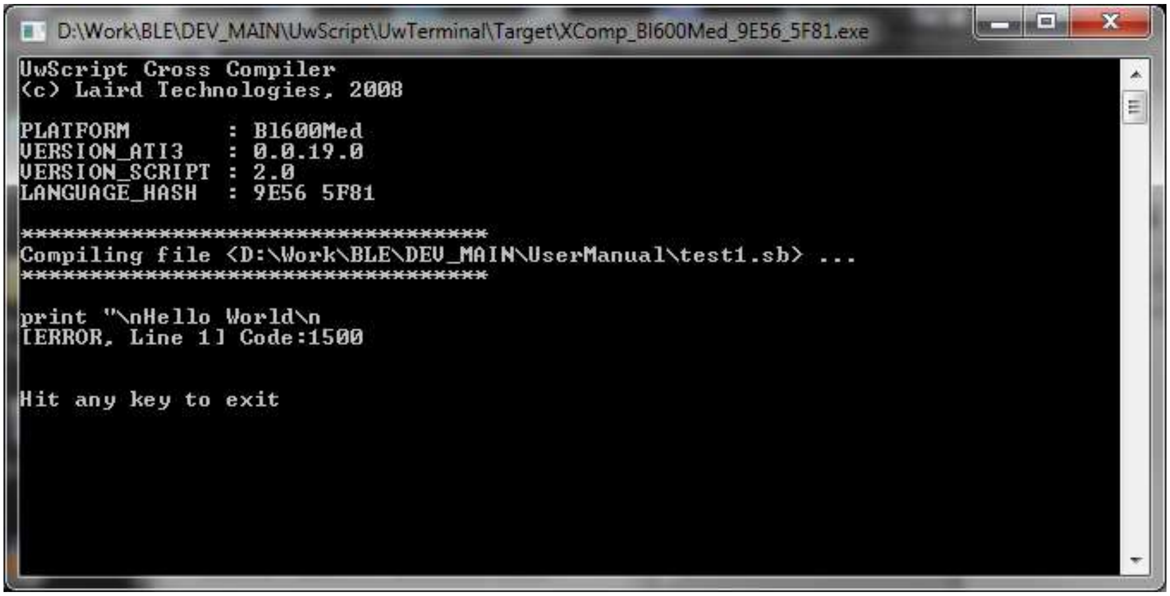


Figure 13: Compilation error window

Now that the application has been downloaded into the module, run it by issuing **test1** or **AT+RUN "test1"**.

Note: *smart*BASIC commands, variables, and filenames are not case sensitive; *smart*BASIC treats *Test1*, *test1* and *TEST1* as the same file.

The screen should display the following results (when both forms of the command are entered):

```
at+run "test1"
Hello World
00

Test1
Hello World
00
```

You can check the file system on the module by typing **AT+DIR** and pressing **Enter,** you should see:

```
06      test1
00
```

You have just written and run your first *smart*BASIC program.

To make it a little more complex, try printing "Hello World" ten times. For this we can use the conditional functions within *smart*BASIC. We also introduce the concept of variables and print formatting. Later chapters go into much more detail, but this gives a flavor of the way they work.

Before we do that, it's worth laying out the rules of the application source syntax.

*smart*BASIC Statement Format

The format of any line of *smart*BASIC is defined in the following manner:

```
{ COMMENT | COMMAND | STATEMENT | DIRECTIVE } < COMMENT > { TERMINATOR }
```

Anything in **{ }** is mandatory and anything in **< >** is optional. Within each set of **{ }** or **< >** brackets, the character **|** is used to denote a choice of values.

The various elements of each line are:

- COMMENT – A COMMENT token is a **'** or **//** followed by any sequence of characters. Any text after the token is ignored by the parser. A comment can occupy its own line or be placed at the end of a STATEMENT or COMMAND.
  COMMAND – An Interactive command; one of the commands that can be executed from Interactive mode.
- STATEMENT – A valid BASIC statement(s) separated by the **:** character if there are more than one statement.

> **Note:** When compiling an application, a line can be made of several statements which are separated by the **:** character.

- DIRECTIVE – A line starting with the **#** character. It is used as an instruction to the parser to modify its behavior. For example, #DEFINE and #INCLUDE.
- TERMINATOR – The **\r** character which corresponds to the **Enter** key on the keyboard.

The *smart*BASIC implementation consists of a command parser and a single line/single pass compiler. It takes each line of text (a series of tokens) and does one of the following (depending on its content and operating mode):

- Acts on them immediately (such as with AT commands).
- If the build includes the compiler, generates a compiled output which is stored and processed at a later time by the run-time engine. This capability is not present in the BL600 due to flash memory constraint.

*smart*BASIC has been designed to work on embedded systems where there is often a very limited amount of RAM. To make it efficient, you must declare every variable that you intend to use by using the DIM statement. The compiler can then allocate the appropriate amount of memory space.

In the following example program, we are using the variable "i" to count how many times we print "*Hello World*". *smart*BASIC allows a couple of different variable types, numbers (32 bit signed integers) and strings.

Our program (stored in a file called *HelloWorld.sb'*) looks like this:

```
//Example :: HelloWorld.sb (See in BL600CodeSnippets)

DIM i as integer            //declare our variable

for i=1 to 10               //Perform the print ten times
   print "Hello World \n"   //The \n forces a new line each time
next
```

Some notes regarding the previous program:

- Any line that starts with an apostrophe (') is a comment and is ignored by the compiler from the token onwards. In other words, the opening line is ignored. You can also add a comment to a program line by adding an apostrophe proceeded by a space to start the comment.
  If you have C++ language experience, you can also use the **//** token to indicate that the rest of the line is a comment.
- The second item of interest is the line feed character '\n' which we've added after *Hello World* in the print statement. This tells the print command to start a new line. If left out, the ten *Hello World's* would have been concatenated together on the screen. You can try removing it to see what would happen.

Compile and download the file *HelloWorld.sb* to the module (using XCompile+Load in UwTerminal) and then run the application in the usual way:

```
AT+RUN "helloworld"
```

The following output displays:

```
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

If you now change the print statement in the application to

```
print "Hello World ";i;"\n"        //The \n forces a new line each time
```

… the following output displays:

```
Hello World 1
Hello World 2
Hello World 3
Hello World 4
Hello World 5
Hello World 6
Hello World 7
Hello World 8
Hello World 9
Hello World 10
```

If you run **AT+DIR**, you will see that both of these programs are now loaded in memory. They remain there until you remove them with **AT+DEL**.

```
06    test1
06    HelloWorld
00
```

Note: All responses to interactive commands are of the format **\nNN\tOptionalText1\tOptionalText2…\r** where *NN* is always a two digit number and **\t** is the tab character and is terminated by **\r**. This format has been provided to assist with developing host algorithms that can parse these responses in a stateless fashion. The NN will always allow the host to attach meaning to any response from the module.

Autorun

One of the major features of a *smart*BASIC module is its ability to launch an application autonomously when power is applied. To demonstrate this we will use the same *HelloWorld* example.

An autorun application is identical to any other BASIC application except for its name, which must be called $autorun$. Whenever a ***smart* BASIC** module is powered up, it checks its **nAutoRUN** input line (see the BL600 module pinout) and, if it is asserted (at 0v), it looks for and executes the autorun application.

In the BL600 development kit, the **nAutoRUN** input pin of the module is connected to the **DTR** output pin of the USB to UART chip. This means the DTR checkbox in UWTerminal can be used to affect the state of that pin on the BL600 module. The DTR checkbox is always selected by default (in asserted state), which translates to a 0v at the **nAutoRUN** input of the module. This means if an autorun application exists in the module's file system, it is automatically launched on power up.

Copy the *smart*BASIC source file *HelloWorld.sb* to *$autorun$.sb* and then cross-compile and download to the module. After it is downloaded, enter the AT+DIR command and the following displays:

```
at+dir

06    test1
06    HelloWorld
06    $autorun$
00
```

TIP: A useful feature of UWTerminal is that the download function strips off the filename extension when it downloads a file into the module file system. This means that you can store a number of different autorun applications on your PC by giving them longer, more descriptive extension names.

For example:

$autorun$.HelloWorld

By doing this, each $autorun$ file on your PC is unique and the list is simpler to manage.

Note: If Windows adds a text extension, rename the file to remove it. Do not use multiple extensions in filenames (such as filename.ext1.ext2). The resulting files (after being stripped) may overwrite other files.

Clear the UWTerminal screen by clicking the Clear button on the toolbar and then enter the command **ATZ** to force the module to reset itself. You could also click **Reset** on the development kit to achieve the same outcome.

Warning: If the JLINK debugger is connected to the development kit via the ribbon, then the reset button has no effect.

The following output displays:

```
Hello World 1
Hello World 2
Hello World 3
Hello World 4
Hello World 5
Hello World 6
Hello World 7
Hello World 8
Hello World 9
Hello World 10
```

In UWTerminal, next clear the screen using the Clear button and then unselect the checkbox labelled DTR so that the nAutoRUN input of the module is not asserted. After a reset (ATZ or the button), the screen remains blank which signifies that the autorun application was NOT invoked automatically.

The reason for providing this capability (suppressing the launching of the autorun application) is to ensure that if your autorun application has the WAITEVENT as the last statement. This allows you to regain control of the module's command interpreter for further development work.

Debugging Applications

One difference with *smart*BASIC is that it does not have program labels (or line numbers). Because it is designed for a single line compilation in a memory constrained embedded environment, it is more efficient to work without them.

Because of the absence of labels, *smart*BASIC provides facilities for debugging an application by inserting breakpoints into the source code prior to compilation and execution. Multiple breakpoints can be inserted and each breakpoint can have a unique identifier associated with it. These IDs can be used to aid the developer in locating which breakpoint resulted in the break. It is up to the programmer to ensure that all IDs are unique. The compiler does not check for repeated values.

Each breakpoint statement has the following syntax:

BP nnnn

Where nnnn should be a unique number which is echoed back when the breakpoint is encountered at runtime. It is up to the developer to keep all the *nnnn's* unique as they are not validated when the source is compiled.

Breakpoints are ignored if the application is launched using the command AT+RUN (or name alone). This allows the application to be run at full speed with breaks, if required. However, if the command **AT+DBG** is used to run the application, then all of the debugging commands are enabled.

When the breakpoint is encountered, the runtime engine is halted and the command line interface becomes active. At this point, the response seen in UWTerminal is in the following form:

<linefeed>21 BREAKPOINT nnnn<carriage return>

Where **nnnn** is the identifier associated with the **BP nnnn** statement that caused the halt in execution. As the **nnnn** identifier is unique, this allows you to locate the breakpoint line in the source code.

For example, if you create an application called test2.sb with the following content:

```
//Example :: test2.sb (See in BL600CodeSnippets)

 DIM i as integer

 for i=1 to 10
    print "Hello World";i;"\n"
    if i==3 then
       bp 3333
    endif
 next
```

When you launch the application using AT+RUN, the following displays:

```
Hello World 1
Hello World 2
Hello World 3
Hello World 4
Hello World 5
Hello World 6
Hello World 7
Hello World 8
Hello World 9
Hello World 10
```

If you launch the application using AT+DBG, the following displays:

```
Hello World 1
Hello World 2
Hello World 3

21    BREAKPOINT 3333
```

Having been returned to Interactive mode, the command **? varname** can be used to interrogate the value of any of the application variables, which are preserved during the break from execution. The command **= varname *newvalue*** can then be used to change the value of a variable, if required. For example:

```
? i
08    3
00
= I 42
? i
08    42
00
```

The single step command **SO** (Step Over) can then be invoked to step through the next statements individually (note the first **SO** reruns the BP statement).

When required, the command RESUME can be used to resume the run-time engine from the current application position as shown below:

```
Hello World 1
Hello World 2
Hello World 3
21    BREAKPOINT 3333
= I 8
resume
Hello World 8
Hello World 9
Hello World 10
```

Structuring an Application

Applications must follow *smart*BASIC syntax rules. However, the single pass compiler places some restrictions on how the application needs to be arranged. This section explains these rules and suggests a structure for writing applications which should adhere to the event driven paradigm.

Typically, do something only when something happens. This *smart*BASIC implementation has been designed from the outset to feed events into the user application to facilitate that architecture and, while waiting for events, the module is designed to remain in the lowest power state.

*smart*BASIC uses a single pass compiler which can be extremely efficient in systems with limited memory. They are called "single pass" as the source application is only passed through the parser line by line once. That means that it has no knowledge of any line which it has not yet encountered and it forgets any previous line as soon as the first character of the next line arrives. The implication is that variables and subroutines need to be placed in position before they are first referenced by any function which dictates the structure of a typical application.

In practice, this results in the following structure for most applications:

- **Opening Comments** – Any initial text comments to help document the application.
- **Includes** – The cross compiler which is automatically invoked by UWTerminal allows the use of #DEFINE and #INCLUDE directives to bring in additional source files and data elements. **Variable Declarations** – Declare any global variables. Local variables can be declared within subroutines and functions.

- **Subroutines and Functions** – These should be cited here, prior to any program references. If any of them refer to other subroutines or functions, these referred ones should be placed first. The golden rule is that nothing on any line of the application should be "new". Either it should be an inbuilt *smart*BASIC function or it should have been defined higher up within the application.
- **Event and error handlers** – Normally these reference subroutines, so they should be placed here.
- **Main program** – The final part of the application is the main program. In many cases this may be as simple as an invocation of one of the user functions or subroutines and then finally the WAITEVENT statement.

An example of an application *(btn.button.led.test.sb)* which monitors button presses and reflects them to leds on the BLE development kit is as follows:

```
//******************************************************************************
// Laird Technologies (c) 2013
//
// +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// +++++                                                                    ++
// +++++  When UwTerminal downloads the app it will store it as a filenname   ++
// +++++  which consists of all characters up to the first . and excluding it ++
// +++++                                                                    ++
// +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
//
//
// Simple development board button and LED test
// Tests the functionality of button 0, button 1, LED 0 and LED 1 on the development
board
// DVK-BL600-V01
//
// 24/01/2013 Initial version
//
//******************************************************************************

//******************************************************************************
// Definitions
//******************************************************************************

//******************************************************************************
// Library Import
//******************************************************************************
//#include "$.lib.ble.sb"

//******************************************************************************
// Global Variable Declarations
//******************************************************************************

dim rc                                   // declare rc as integer variable

//******************************************************************************
// Function and Subroutine definitions
//******************************************************************************

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

```
function button0release()                  //this function is called when the button 0
is released"
gpiowrite(18,0)                            // turns LED 0 off
print "Button 0 has been released \n"      //these lines are printed to the UART when
the button is released
print "LED 0 should now go out \n\n"
endfunc 1

//==============================================================================
//==============================================================================
function button0press()                    //this function is called when the button 0
is pressed"
gpiowrite(18,1)                            // turns LED 0 on
print "Button 0 has been pressed \n"       //these lines are printed to the UART when
the button is pressed
print "LED 0 will light while the button is pressed \n"
endfunc 1

//==============================================================================
//==============================================================================
function button1release()                  //this function is called when the button 1
is released"
gpiowrite(19,0)                            //turns LED 1 off
print "Button 1 has been released \n"      //these lines are printed to the UART when
the button is released
print "LED 1 should now go out \n\n"
endfunc 1

//==============================================================================
//==============================================================================
function button1press()                    //this function is called when the button 1
is pressed"
gpiowrite(19,1)                            // turns LED 1 on
print "Button 1 has been pressed \n"       //these lines are printed to the UART when
the button is pressed
print "LED 1 will light while the button is pressed \n"
endfunc 1


//******************************************************************************
// Handler definitions
//******************************************************************************

//******************************************************************************
// Equivalent to main() in C
//******************************************************************************

rc = gpiosetfunc(16,1,2)                   //sets sio16 (Button 0) as a digital in with
a weak pull up resistor
rc = gpiosetfunc(17,1,2)                   //sets sio17 (Button 1) as a digital in with
a weak pull up resistor
rc = gpiosetfunc(18,2,0)                   //sets sio18 (LED0) as a digital out
rc = gpiosetfunc(19,2,0)                   //sets sio19 (LED1) as a digital out
rc = gpiobindevent(0,16,0)                 //binds a gpio transition high to an event.
sio16 (button 0)
rc = gpiobindevent(1,16,1)                 //binds a gpio transition low to an event.
sio16 (button 0)
rc = gpiobindevent(2,17,0)                 //binds a gpio transition high to an event.
sio17 (button 1)
rc = gpiobindevent(3,17,1)                 //binds a gpio transition low to an event.
```

```
sio17 (button 1)

onevent evgpiochan0 call button0release   //detects when button 0 is released and
calls the function
onevent evgpiochan1 call button0press     //detects when button 0 is pressed and calls
the function
onevent evgpiochan2 call button1release   //detects when button 1 is released and
calls the function
onevent evgpiochan3 call button1press     //detects when button 1 is pressed and calls
the function

print "Ready to begn button and LED test \n"   //these lines are printed to the UART
when the program is run
print "Please press button 0 or button 1 \n\n"

//------------------------------------------------------------------------------
// Wait for a synchronous event.
// An application can have multiple <WaitEvent> statements
//------------------------------------------------------------------------------
waitevent                                 //when program is run it waits here until an
event is detected
```

When this application is launched and appropriate buttons are pressed and released, the output is as follows:

3. Interactive Mode Commands

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

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

```
'Example:

AT I 3
```

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

<linefeed character> response text <carriage return>

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

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

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

Note: Inthe case of the 01 response, the "<tab>optional_verbose_explanation" will be missing in resource constrained platforms like the BL600 modules. The 'verbose explanation' is a constant string and since there are over 1000 error codes, these verbose strings can occupy more than 10 kilobytes of flash memory.

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

To get the verbose description, click on the BASIC tab (in UWTerminal) and, if the error value is hhhh, enter the command ER 0xhhhh and note the 0x prefix to 'hhhh'. This is illustrated in Figure 14.

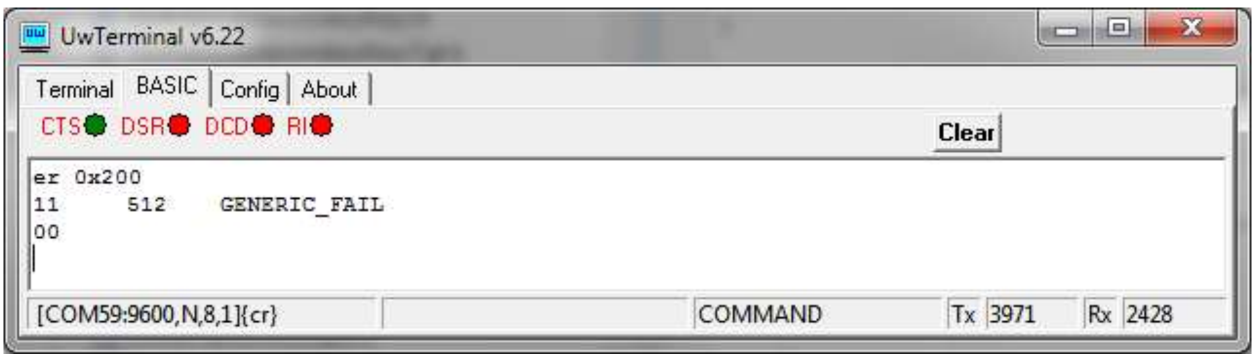


Figure 14: Optional verbose explanation

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

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

AT

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

It performs no action other than to respond with "\n00\r". It exists to emulate the behaviour of a device which is controlled using the AT protocol. This is a good command to use to check if the UART has been correctly configured and connected to the host.

AT I or ATI

Provided to give compatibility with the AT command set of Laird's standard Bluetooth modules.

AT i num

Command

Returns

\n10\tMM\tInformation\r
\n00\r

Where

\n = linefeed character 0x0A
\t = horizontal tab character 0x09
MM = a *number* (see below)
Information = sting consisting of information requested associated with MM
\r = carriage return character 0x0D

Arguments

num — *Integer Constant* - A number in the range 0 to 65,535. Currently defined numbers are:

0	Name of device
3	Version number of Module Firmware
4	MAC address in the form TT AAAAAAAAAAAA
5	Chipset name
6	Flash File System size stats (data segment): Total/Free/Deleted
7	Flash File System size stats (FAT segment) : Total/Free/Deleted
12	Last error code
13	Language hash value
16	NvRecord Memory Store stats: Total/Free/Deleted
33	BASIC core version number
601	Flash File System: Data Segment: Total Space
602	Flash File System: Data Segment: Free Space
603	Flash File System: Data Segment: Deleted Space
604	Flash File System: FAT Segment: Total Space
605	Flash File System: FAT Segment: Free Space
606	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..1999	See SYSINFO() function definition
2000..2999	See SYSINFO() function definition

Any other number currently returns the manufacturer's name.

For ATi4 the TT in the response is the type of address as follows:-

00	Public IEEE format address
01	Random static address (default as shipped)
02	Random Private Resolvable (used with bonded devices) – **not currently available**
03	Random Private Non-Resolvable (used for reconnections) – **not currently available**

Please refer to the Bluetooth specification for a further description of the types.

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

Interactive Command: Yes

```
'Example:

AT  i  3
10   3    2.0.1.2
00
AT I 4
10   4    01 D31A920731B0
```

AT i is a core command.

The information returned by this Interactive command can also be useful from within a running application and so a built-in function called SYSINFO(cmdId) can be used to return exactly the same information and cmdid is the same value as used in the list above.

AT+DIR

COMMAND

List all application or data files in the module's flash file system.

AT+DIR <"string">

Returns

\n06\tFILENAME1\r
\n06\tFILENAME2\r
\n06\tFILENAMEn\r
\n00\r

If there are no files within the module memory, then only \n00\r is sent.

Arguments:

string — string_constant An optional pattern match string.
If included AT+DIR will only return application names which include this string.

The match string is not case sensitive.

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

Interactive Command: YES

```
'Examples:

AT+DIR

AT+DIR "new"
```

AT+DIR is a core command.

AT+DEL

COMMAND

This command deletes a file from the module's flash file system.

When the file is deleted, the space it occupied does not get marked as free for use again.

Eventually, after many deletions, the file system does not have free space for new files. When this happens, the module responds with an appropriate error code when a new file write is attempted. Use the command AT&F 1 to completely erase and reformat the file system.

At any time you can use the command **AT I 6** to get information about the file system. It respond with the following:

10 6 aaaa,bbbb,cccc

Where aaaa is the total size of the file system, bbbb is the free space available, and cccc is the deleted space.

From within a *smart* BASIC application you can get aaaa by calling SYSINFO(601), bbbb by calling SYSINFO(602), and cccc by calling SYSINFO(603).

Note: After AT&F 1 is processed, because the file system manager context is unstable, there will be an automatic self-reboot.

AT+DEL "filename" (+)

Returns OK

If the file does not exist or if it was successfully erased, it will respond with \n00\r.

Arguments:

filename string_constant.
The name of the file to be deleted. The maximum length of filename is 24 characters and should not include the following characters **:*?"<>|**

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

Adding the "+" sign to an AT+DEL command can be used to force the deletion of an open file. For example, use **AT+DEL "filename" +** to delete an application which you have just exited after running it.

Interactive Command: YES

```
'Examples:

AT+DEL "data"
AT+DEL "myapp" +
```

AT+DEL is a core command.

AT+RUN

COMMAND

AT+RUN runs a precompiled application that is stored in the module's flash file system. Debugging statements in the application are disabled when it is launched using AT+RUN.

AT+RUN "filename"

Returns If the filename does not exists the AT+RUN will respond with an error response starting with a 01 and a hex value describing the type of error. When the application aborts or if the application reaches its end, a deferred \n00\r response is sent.

If the compiled file was generated with a non-matching language hash then it will not run with an error value of 0707 or 070C

Arguments:

filename string_constant.
The name of the file to be run. The maximum length of filename is 24 characters and should not include the following characters :*?"<>|

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

Note: Debugging is disabled when using AT+RUN, hence all **BP nnnn** statements are inactive. To run an application with debugging active, use AT+DBG.

If any variables exist from a previous run, they are destroyed before the specified application is serviced.

Note: The application "filename" can also be invoked by entering the name if it does not contain any spaces.

Interactive Command: YES

```
'Examples:

AT+RUN "NewApp"
or
NewApp
```

AT+RUN is a core command.

AT+DBG

COMMAND

AT+DBG runs a precompiled application that is stored in the flash file system. In contrast to AT+RUN, debugging is enabled.

AT+DBG "filename"

Returns If the filename does not exists the AT+DBG will respond with an error response. When the application aborts or if the application reaches its end, a deferred \n00\r response is sent.

Arguments:

filename string_constant.
The name of the file to be run. The maximum length of filename is 24 characters and should not include the following characters :*?"<>|

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

Debugging is enabled when using AT+DBG, which means that all **BP nnnn** statements are active. To launch an application without the debugging capability, use **AT+RUN**. You do not need to recompile the application, but this is at the expense of using more memory to store the application.

If any variables exist from a previous run, they are destroyed before the specified application is serviced.

Interactive Command: YES

```
'Examples:

AT+DBG "NewApp"
```

AT+DBG is a core command.

AT+SET

This command has been deprecated, please use the new presentation command **AT+CFG num value** instead.

AT+GET

This command has been deprecated, please use the new command **AT+CFG num ?** instead.

AT+CFG

COMMAND

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

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

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

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

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

AT+CFG num value **or** AT+CFG num ?

Returns If the config key is successfully updated or read, the response is \n00\r.

Arguments:

num Integer Constant
The ID of the required configuration key. All of the configuration keys are stored as an array of 16 bit words.

value Integer_constant
This is the new value for the configuration key and the syntax allows decimal,

octal, hexadecimal or binary values.

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

The following Configuration Key IDs are defined.

40	Maximum size of locals simple variables
41	Maximum size of locals complex variables
42	Maximum depth of nested user defined functions and subroutines
43	The size of stack for storing user functions simple variables
44	The size of stack for storing user functions complex variables
45	The size of the message argument queue length
100	Enable/Disable Virtual Serial Port Service when in interactive mode. Valid values are: 0x0000 Disable 0x0001 Enable 0x80nn Enable ONLY if Signal Pin 'nn' on module is HIGH 0xC0nn Enable ONLY if Signal Pin 'nn' on module is LOW 0x81nn Enable ONLY if Signal Pin 'nn' on module is HIGH and auto-bridged to uart when connected 0xC1nn Enable ONLY if Signal Pin 'nn' on module is LOW and auto-bridged to uart when connected ELSE Disable
101	Virtual Serial Port Service to use INDICATE or NOTIFY to send data to client. 0 Prefer Notify ELSE Prefer Indicate This is a preference and the actual value is forced by the property of the TX characteristic of the service.
102	This is the advert interval in milliseconds when advertising for connections in interactive mode and AT Parse mode. Valid values are: 20 to 10240 milliseconds
103	This is the advert timeout in milliseconds when advertising for connections in interactive mode and AT Parse mode. Valid values are: 1 to 16383 seconds
104	In the virtual serial port service manager data transfer is managed. When sending data using NOTIFIES, the underlying stack uses transmission buffers of which there are a finite number. This specifies the number of transmissons to leave unused when sending a lot of data. This allows other services to send notifies without having to wait for them. The total number of transmission buffers can be determined by calling SYSINFO(2014) or in interactive mode submitting the command ATi 2014
105	When in interactive mode and connected for virtual serial port services, this is the minimum connection interval in milliseconds to be negotiated with the master. Valid value is 0 to 4000 ms and if a value of less than 8 is specified, then the minimum value of 7.5 is selected.
106	When in interactive mode and connected for virtual serial port services, this is the maximum connection interval in milliseconds to be negotiated with the master. Valid value is 0 to 4000 ms and if a value of less the minimum specified in 105, then it is forced to the value in 105 + 2 ms
107	When in interactive mode and connected for virtual serial port services, this is the connection supervision timeout in milliseconds to be negotiated with the master. The valid range is 0 to 32000 and if the value is less than the value in 106, then a value double that specified in 106 is used.

108	When in interactive mode and connected for virtual serial port services, this is the slave latency to be negotiated with the master. An adjusted value is used if this value times the value in 106 is greater than the supervision timeout in 107
109	When in interactive mode and connected for virtual serial port services, this is the Tx power used for adverts and connections. The main reason for setting a low value is to ensure that in production, if *smart*BASIC applications are downloaded over the air, then limited range allows many stations to be used to program devices.
110	If Virtual Serial Port Service is enabled in interactive mode (see 100), then this specifies the size of the transmit ring buffer in the managed layer sitting above the service characteristic fifo register. It must be a value in the range 32 to 256
111	If Virtual Serial Port Service is enabled in interactive mode (see 100), then this specifies the size of the receive ring buffer in the managed layer sitting above the service characteristic fifo register. It must be a value in the 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 a per the modified Laird's service. See more details of the service definition here.
113	This is the advert interval in milliseconds when advertising for connections in interactive mode and UART Bridge mode. Valid values are: 20 to 10240 milliseconds
114	This is the advert timeout in milliseconds when advertising for connections in interactive mode and UART Bridge mode. Valid values are: 0 to 16383 seconds, and 0 disables the timer hence continuous
115	This is used to specify the UART baudrate when Virtual Serial Mode Service is active and UART bridge mode is enabled. Valid values are 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, 76800, 115200, 230400, 250000, 460800, 921600, 1000000. 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 transfer 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. 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 sent immediately.

Interactive Command: YES

AT+CFG is a core command.

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

AT+FOW

COMMAND

AT+FOW opens a file to allow it to be written with raw data. The group of commands (AT+FOW, AT+FWR, AT+FWRH and AT+FCL) are typically used for downloading files to the module's flash filing system. For example, web pages, x.509 certificates, or BLE data.

AT+FOW "filename"

Returns If the filename is valid, AT+FOW responds with \n00\r.

Arguments:

filename string_constant.
The name of the file to be opened. The maximum length of filename is 24 characters and should not include the following characters **:*?"<>|**

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

Interactive Command: YES

```
'Examples:

AT+FOW "myapp"
```

AT+FOW is a core command.

AT+FWR

COMMAND

AT+FWR writes a string to a file that has previously been opened for writing using AT+FOW. The group of commands (AT+FOW, AT+FWR, AT+FWRH and AT+FCL) are typically used for downloading files to the module's flash filing system. For example, web pages, x.509 certificates, or BLE data.

AT+FWR "string"

Returns If the string is successfully written, AT+FWR will respond with \n00\r.

Arguments:

string string_constant – A string that is appended to a previously opened file. Any \NN or \r or \n characters present within the string are de-escaped before they are written to the file.

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

Interactive Command: YES

```
'Examples:

AT+FWR "\nhelloworld\r"
AT+FWR "\00\01\02"
```

AT+FWR is a core command.

AT+FWRH

COMMAND

AT+FWRH writes a string to a file that has previously been opened for writing using AT+FOW. The group of commands (AT+FOW, AT+FWR, AT+FWRH and AT+FCL) are typically used for downloading files to the module's flash filing system. For example, web pages, x.509 certificates, or BLE data.

AT+FWRH "string"

Returns If the string is successfully written, AT+FWRH will respond with \n00\r.

Arguments

string string_constant – A string that is appended to a previously opened file. Only hexadecimal characters are allowed and the string is first converted to binary and then appended to the file.

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

Interactive Command: YES

```
'Examples:

AT+FWRH "FE900002250DEDBEEF"
AT+FWRH "000102"

'Invalid example

AT+FWRH "hello world"    'because not a valid hex string
```

AT+FWRH is a core command.

AT+FCL

COMMAND

AT+FCL closes a file that has previously been opened for writing using AT+FOW. The group of commands; AT+FOW, AT+FWR, AT+FWRH and AT+FCL are typically used for downloading files to the module's flash filing system.

AT+FCL

Returns If the filename exists, AT+FCL responds with \n00\r.

Arguments:

None

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

Interactive Command: YES

```
'Examples:

AT+FCL
```

AT+FCL is a core command.

? (Read Variable)

COMMAND

When an application encounters a STOP, BPnnn, or END statement, it falls into the Interactive mode of operation and does not discard any global variables created by the application. This allows them to be referenced in Interactive mode.

? var <[index]>

Returns Displays the value of the variable if it had been created by the application. If the variable is an array then the element index MUST be specified using the [n] syntax.

If the variable exists and it is a simple type then the response to this command is

\n08\tnnnnnn\r
\n00\r

If the variable is a string type, then the response is

\n08\t"Hello World"\r
\n00\r

If the variable does not exist then the response to this command is

\n01\tE023\r

Where \n = linefeed, \t = horizontal tab and \r = carriage return

---

Note: If the optional type prefix is present, the output value, when it is an integer constant, is displayed in that base. For example:

? h' var   returns

\n08\tH'nnnnnn\r
\n00\r

Arguments:

Var <[n]> Any valid variable with mandatory [n] if the variable is an array.

For integer variables, the display format can be selected by prefixing the variable with one of the integer type prefixes:

D' := Decimal
H' := Hexadecimal
O' := Octal
B' := Binary

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

Interactive Command: YES

```
'Examples:
? argc
08      11
00
? h'argc
08      H'0000000B
00
```

```
? B'argc
08      B'00000000000000000000000001011
? argv[0]
08      "hello"
00
```

? is a core command.

= (Set Variable)

COMMAND

When an application encounters a STOP, BPnnn, or END statement, it falls into the Interactive mode of operation and does not discard the global variables so that they can be referenced in Interactive Mode. The = command is used to change the content of a known variable. When the application is RESUMEd, the variable contains the new value. It is useful when debugging applications.

= var<[n]> value

Returns If the variable exists and the value is of a compatible type then the variable value is overwritten and the response to this command is:

\n00\r

If the variable exists and it is NOT of compatible type then the response to this command is

\n01\tE027\r

If the variable does not exist then the response to this command is

\n01\tE023\r

If the variable exists but the new value is missing, then the response to this command is

\n01\tE26\r

Where \n = linefeed, \t = horizontal tab and \r = carriage return

Arguments:

Var<[n]> The variable whose value is to be changed

value A string_constant or integer_constant of appropriate form for the variable.

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

Interactive Command: YES

Examples: (after an app exits which had DIM'd a global variable called 'argc')

```
? argc
08      11
00
= argc 23
00
? argc
08      23
00
```

= is a core command.

SO

SO (Step Over) is used to execute the next line of code in Interactive Mode after a break point has been encountered when an application had been launched using the AT+DBG command.

Use this command after a breakpoint is encountered in an application to process the next statement. SO can then be used repeatedly for single line execution

SO is normally used as part of the debugging process after examining variables using the ? Interactive Command and possibly the **=** command to change the value of a variable.

See also the BP nnnn, AT+DBG, ABORT, and RESUME commands for more details to aid debugging.

SO is a core function.

RESUME

COMMAND

RESUME is used to continue operation of an application from Interactive Mode which had been previously halted. Normally this occurs as a result of execution of a STOP or BP statement within the application. On execution of RESUME, application operation continues at the next statement after the STEP or BP statement.

If used after a SO command, application execution commences at the next statement.

RESUME

Returns If there is nothing to resume (e.g. immediately after reset or if there are no more statements within the application), then an error response is sent.

\n01\tE029\r

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

Interactive Command: YES

```
'Examples:

RESUME
```

RESUME is a core function.

ABORT

COMMAND

Abort is an Interactive Mode command which is used to abandon an application, whose execution has halted because it has processed a STOP or BP statement.

ABORT

Returns Abort is an Interactive Mode command which is used to abandon an application, whose execution has halted because it had processed a STOP or BP statement. If there is nothing to abort then it will return a success 00 response.

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

Interactive Command: YES

```
'Examples:
'(Assume the application someapp.sb has a STOP statement somewhere which will invoke interactive mode)

AT+RUN "someapp"
ABORT
```

ABORT is a core command.

AT+REN

COMMAND

Renames an existing file.

AT+REN "oldname" "newname"

Returns	OK if the file is successfully renamed.
Arguments	
oldname	string_constant. The name of the file to be renamed.
Newname	string_constant. The new name for the file. The maximum length of filename is 24 characters.

oldname and *newname* must contain a valid filename, which cannot contain the following seven characters

: * ? " < > |

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

Interactive Command: YES

```
'Examples:

AT+REN "oldscript.txt" "newscript.txt"
```

AT+REN is a core command.

AT&F

COMMAND

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

AT&F integermask

Returns	OK if file successfully erased.
Arguments	
Integermask	Integer corresponding to a bit mask or the "*" character

The mask is an additive integer mask, with the following meaning:

1	Erases normal file system and system config keys (see AT+CFG for examples of config keys)
16	Erases the User config keys only
*	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.

Interactive Command: YES

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

AT&F is a core command.

AT Z or ATZ

Resets the CPU.

AT Z

Returns \n00\r

Arguments: None

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

Interactive Command: YES

```
'Examples:

AT Z
```

AT Z is a core command.

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.

Interactive Command: YES

```
'Examples:

AT+BTD*
```

AT+BTD* is an extension command

AT + MAC "12 hex digit mac address"

COMMAND

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

> **Notes:** If the module has an invalid licence then this address will not be visible.
> If the address "000000000000" is written then it will be treated as invalid and prevent a new address from being entered.

AT + MAC "12 hex digits"

Returns \n00\r
or
\n01 192A\r

Where the error code 192A is "NVO_NVWORM_EXISTS" meaning an IEEE mac address already exists, which can be read using the command AT I 24

Arguments:

A string delimited by "" which shall be a valid 12 hex digit mac address that is written to non-volatile memory.

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

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

Interactive Command: YES

```
'Examples:

AT+MAC "008098010203"
```

AT+MAC is an extension command

AT + BLX

COMMAND

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

AT + BLX

Command

Returns \n00\r

Arguments: *None*

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.

Interactive Command: YES

```
'Examples:

AT+BLX
```

AT+BLX is an extension command.

4. *SMART* BASIC COMMANDS

smart BASIC contains a wide variety of commands and statements. These include a core set of programming commands found in most languages and extension commands that are designed to expose specific functionality of the platform. For example, Bluetooth Low Energy's GATT, GAP, and security functions.

Because *smart* BASIC is designed to be a very efficient embedded language, you must take care of command syntax.

Syntax

smart BASIC commands are classified as one of the following:

- Functions
- Subroutines
- Statements

Functions

A function is a command that generates a return value and is normally used in an expression. For example:

newstr$ = LEFT$ (*oldstring$*, *num*)

In other words, functions cannot appear on the left side of an assignment statement (which has the equals sign). However, a function may affect the value of variables used as parameters if it accepts them as references rather than as values. This subtle difference is described further in the next section.

Subroutines

A subroutine does not generate a return value and is generally used as the only command on a line. Like a function, it may affect the value of variables used as parameters if it accepts them as references rather than values. For example:

STRSHIFTLEFT (*string$*, *num*)

This brings us to the definition of the different forms an argument can take, both for a function and a subroutine. When a function is defined, its arguments are also defined in the form of how they are passed – either as **byVal** or **byRef**.

Passing Aruments as byVal	If an argument is passed as byVal, then the function or subroutine only sees a copy of the value. While it is able to change the copy of the variable upon exit, all changes are lost.
Passing Arguments as byRef	If an argument is passed as byRef, then the function or subroutine can modify the variable and, upon exit, the variable that was passed to the routine contains the new value.

To understand, look at the *smart* BASIC subroutine **STRSHIFTLEFT**. It takes a string and shifts the characters to the left by a specified number of places:

STRSHIFTLEFT (*string$*, *num*)

It is used as a command on *string$*, which is defined as being passed as byRef. This means that when the rotation is complete, *string$* is returned with its new value. ***num*** defines the number of places that the string is shifted and is passed as byVal; the original variable ***num*** is unchanged by this subroutine.

Note: Throughout the definition of the following commands, arguments are explicitly stated as being byVal or byRef.

Functions, as opposed to subroutines, always return a value. Arguments may be either byVal or byRef. In general and by default, string arguments are passed byRef. The reason for this is twofold:

- It saves valuable memory space because a copy of the string (which may be long) does not need to be copied to the stack.
- A string copy operation is lengthy in terms of CPU execution time. However, in some cases the valuables are passed byVal and in that case, when the function or subroutine is invoked, a constant string in the form "string" can be passed to it.

Note: For arguments specified as byRef, it is not possible to pass a constant value – whether number or string.

Statements

Statements do not take arguments, but instead take arithmetic or string expression lists. The only Statements in *smart* BASIC are PRINT and SPRINT.

Exceptions

Developing a software application that is error free is virtually an impossible task. All functions and subroutines act on the data that is passed to them and there are occasions when the values do not make sense. For example, when a divide operation is requested and the divisor passed to the function is the value zero. In these types of cases it is impossible to generate a return of meaningful value, but the event needs to be trapped so that the effects of doing that operation can be lessened.

The mitigation process is via the inclusion of an ONERROR handler as explained in detail later in this manual. If the application does not provide an ONERROR handler and if an exception is encountered at run-time, then the application aborts to Interactive mode.

Note: This is disastrous for unattended use cases. A good catchall ONERROR is to invoke a handler in which the module is reset; then at least the module resets from a known condition.

Language Definitions

Throughout the rest of this manual, the following convention is used to describe *smart* BASIC commands and statements:

Command

FUNCTION / SUBROUTINE / STATEMENT

Description of the command.

COMMAND (<byRef | byval> *arg1 <AS type>*,..)

Returns			
TYPE	Description. Value that a function returns (always byVal).		
Exceptions			
ERRVAL	Description of the error.		
Arguments (a list of the arguments for the command)			
arg1	byRef	TYPE	A description, with type, of the variable.
argn	byVal	TYPE	A description, with type, of the variable.
Interactive Command		Whether the command can be run in Interactive Mode using the **!** token.	

```
'Examples:

Examples using the command.
```

Note: Always consult the release notes for a particular firmware release when using this manual. Due to continual firmware development, there may be limitations or known bugs in some commands that cause them to differ from the descriptions given in the following chapters.

Variables

One of the important rules is that variables used within an application MUST be declared before they are referenced within the application. In most cases the best place is at the start of the application. Declaring a variable can be thought of as reserving a portion of memory for it. *smart* BASIC does not support forward declarations. If an application references a variable that has not been declared, the parser reports an **ERROR** and aborts the compilation.

Variables are characterised by two attributes:

- Variable Scope
- Variable Class

DIM

The Declare statement is used to declare a number of variables of assorted types to be defined in a single statement.

If it is used within a FUNCTION or SUB block of code, then those variables will only have local scope. Otherwise they will have validity throughout the application. If a variable is declared within a FUNCTION or SUB and a variable of the same name already exists with global scope, then this declaration will take over whilst inside the FUNCTION or SUB. However, this practice should be avoided.

DIM *var<,var<,…>>*

Arguments:

Var – A complete variable definition with the syntax ***varname <AS type>***. Multiple variables can be defined in any order with each definition being separated by a comma.

Each variable (*var*) consists of one mandatory element ***varname*** and one optional element ***AS type*** separated by whitespaces and described as follows:

- ***Vaname*** – A valid variable name.
- ***AS type*** – Where 'type' is *INTEGER* or *STRING*. If this element is missing, then varname is used to define the type of the variable so that if the name ends with a $ character, then it defaults to a *STRING*; otherwise an *INTEGER* .

A variable can be declared as an array, although only one dimension is allowed. Arrays must always be defined with their size, e.g.

array [20] – The (20) with round brackets is also allowed.

The size of an array cannot be changed after it is declared and the maximum size of an array is 256.

Interactive Command: NO

```
//Example :: DimEx1.sb (See in BL600CodeSnippets.zip)

DIM temp1 AS INTEGER
DIM temp2                   //Will be an INTEGER by default
DIM temp3$ AS STRING
DIM temp4$                  //Will be a STRING by default
DIM temp5$ AS INTEGER       //Allowed but not recommended practice as there
//is a $ at end of name
DIM temp6 AS STRING         //Allowed but not recommended practice as no $
//at end of name
DIM a1,a2,a3$,a4            //3 INTEGER variables and 1 STRING variable

print "We will now print each varaible on screen \n"
print temp1, temp2, temp3$, temp4$, temp5$, temp6, a1, a2, a3$, a4

//Since the variables have not been instantiated, they hold default values
//The comma inserts a TAB
```

Expected Output:

```
We will now print each varaible on screen
0       0                       0               0       0               0
```

Variable Scope

The scope of a variable defines where it can be used within an application.

- **Local Variable** – The most restricted scope. These are used within functions or subroutines and are only valid within the function or subroutine. They are declared within the function or subroutine.
- **Global Variable** – Any variables not declared in the body of a subroutine or a function and are valid from the place they are declared within an application. Global Variables remain in scope at the end of an application, which allows the user or host processor to interrogate and modify them using the **?** and **=** commands respectively.
  As soon as a new application is run, they are discarded.

Note: If a local variable has the same name as a global variable, then within a function or a subroutine, that global variable cannot be accessed.

Variable Class

smart BASIC supports two generic classes of variables:

- **Simple** – Numeric variables. There are currently two types of simple variables: INTEGER, a signed 32-bit variable (which also has the alias LONG), and ULONG, an unsigned 32-bit variable.

  Simple variables are scalar and can be used within arithmetic expressions as described later.

- **Complex** – Non-numeric variables. There is currently only one type STRING.

  STRING is an object of concatenated byte characters of any length up to a maximum of 65280 bytes but for platforms with limited memory, it is further limited and that value can be obtained by submitting the AT I 1004 command when in Interactive mode and using the SYSINFO(1004) function from within an application.

  For example, in the BLE module, the limit is 512 bytes since it is always the largest data length for any attribute.

  Complex variables can be used in expressions which are dedicated for that type of variable. In the current implementation of *smart* BASIC, the only general purpose operator that can be used with strings is the '+' operator which is used to concatenate strings.

```
//Example :: DimEx2.sb (See in BL600CodeSnippets.zip)
DIM i$ as STRING
DIM a$ as STRING
a$ = "Laird"
i$ = a$ + "Rocks!"      //Here we are concatenating the two strings
print i$
```

Expected Output:

```
LairdRocks!
```

Note: To preserve memory, *smart* BASIC only allocates memory to string variables when they are first used and not when they are allocated. If too many variables and strings are declared in a limited memory environment it is possible to run out of memory at run time. If this occurs an *ERROR* is generated and the module will return to Interactive Mode. The point at which this happens depends on the free memory so will vary between different modules.

This return to Interactive Mode is NOT desirable for unattended embedded systems. To prevent this, every application MUST have an *ONERROR* handler which is described later in this user manual.

Note: Unlike in the "C" programming language, strings are not null terminated.

Arrays

Variables can be created as arrays of **single dimensions**; their size (number of elements) must be explicitly stated when they are first declared using the nomenclature [x] or (x) after the variable name, e.g.

DIM array1 [10] AS STRING

DIM array2(10) AS STRING

```
//Example :: ArraysEx1.sb (See in BL600CodeSnippets.zip)

DIM nCmds AS INTEGER
DIM stCmds[20] AS STRING   //declare an array as a string with 20 elements
//Not recommended because we are only using 7 elements as you will see below

//Setting the values for 7 of the elements
stCmds[0]="\rATS0=1\r"
stCmds[1]="ATS512=4\r"
stCmds[2]="ATS501=1\r"
stCmds[3]="ATS502=1\r"
stCmds[4]="ATS503=1\r"
stCmds[5]="ATS504=1\r"
stCmds[6]="AT&W\r"
nCmds=6

//Print the 7 elements above in order
DIM i AS INTEGER
for i=0 to nCmds step 1
   print stCmds[i]
next
```

Expected Output:

```
ATS0=1
ATS512=4
ATS501=
ATS502=1
ATS503=1
ATS504=1

AT&W
```

General Comments on Variables

Variable Names begin with 'A' to 'Z' or '_' and then can have any combination of 'A' to 'Z', '0' to '9' '$' and '_'.

Note: Variable names are not case sensitive (for example, *test$* and *TEST$* are the same variable).

smart BASIC is a strongly typed language and so if the compiler encounters an incorrect variable type then the compilation will fail.

Declaring Variables

Variables are normally declared individually at the start of an application or within a function or subroutine.

```
DIM string$ AS STRING
DIM str1$                      // the $ at the end of the name implies a string
                               // so AS STRING not necessary
DIM temp1 AS INTEGER
DIM alarmstate                 // no $ at the of the name implies an integer
                               // so AS INTEGER not necessary
DIM array [10] AS STRING
```

Constants

Numeric Constants

Numeric Constants can be defined in decimal, hexadecimal, octal, or binary using the following nomenclature:

Decimal	D'1234	or	1234 (default)
Hex	H'1234	or	0x1234
Octal	O'1234		
Binary	B'01010101		

Note: By default, all numbers are assumed to be in decimal format.

The maximum decimal signed constant that can be entered in an application is 2147483647 and the minimum is -2147483648.

A hexadecimal constant consists of a string consisting of characters 0 to 9, and A to F (a to f). It must be prefixed by the two character token H' or h' or 0x.

```
H'1234
h'DEADBEEF
0x1234
```

An octal constant consists of a string consisting of characters 0 to 7. It must be prefixed by the two character token O' or o'.

```
O'1234
o'5643
```

A binary constant consists of a string consisting of characters 0 and 1. It must be prefixed by the two character token B' or b'.

```
B'11011100
b'11101001
```

A binary constant can consist of 1 to 32 bits and is left padded with 0s.

String Constants

A string constant is any sequence of characters starting and ending with the " character. To embed the " character inside a string constant specify it twice.

```
"Hello World"
"Laird_""Rocks"""   // in this case the string is stored as Laird_"Rocks"
```

Non-printable characters and print format instructions can be inserted within a constant string by escaping using a starting '\' character and two hexadecimal digits. Some characters are treated specially and only require a single character after the '\' character.

The table below lists the supported characters and the corresponding string.

Character	Escaped String	Character	Escaped String
Linefeed	\n	"	\22 or ""
Carriage return	\r	A	\41
Horizontal Tab	\t	B	\42
\	\5C	etc…	

Compiler Related Commands and Directives

#SET

The *smart*BASIC complier converts applications into an internally compiled program on a line by line basis. It has strict rules regarding how it interprets commands and variable types. In some cases, it is useful to modify this default behaviour, particularly within user defined functions and subroutines. To allow this, a special directive is provided - #SET.

#SET is a special directive which instructs the complier to modify the way that it interprets commands and variable types. In normal usage you should never have to modify any of the values.

#SET **must** be asserted before the source code that it affects, or the compiler behaviour will not be altered.

#SET can be used multiple times to change the tokeniser behaviour throughout a compilation.

#SET commandID, commandValue

Arguments	
cmdID	Command ID and valid range is 0..10000
cmdValue	Any valid integer value

Currently *smart*BASIC supports the following cmdIDs:

CmdID	MinVal	MaxVal	Default	Comments
1	0	1	0	Default Simple Arguments type for routines. 0 = ByVal, 1=ByRef
2	0	1	1	Default Complex Arguments type for routines. 0 = ByVal, 1=ByRef
3	8	256	32	Stack length for Arithmetic expression operands
4	4	256	8	Stack length for Arithmetic expression constants
5	16	65535	1024	Maximum number of simple global variables per application
6	16	65535	1024	Maximum number of complex global variables per application
7	2	65535	32	Maximum number of simple local variables per routine in an application
8	2	65535	32	Maximum number of complex local variables per routine in an application
9	2	32767	256	Max array size for simple variables in DIM
10	2	32767	256	Max array size for complex variables in DIM

Note: Unlike other commands, #SET may not be combined with any other commands on a line.

```
'Example

#set 1 1   'change default simple args to byRef
#set 2 0   'change default complex args to byVal
```

Arithmetic Expressions

Arithmetic expressions are a sequence of integer constants, variables, and operators. At runtime the arithmetic expression, which is normally the right hand side of an = sign, is evaluated. Where it is set to a variable, then the variable takes the value and class of the expression (such as INTEGER).

If the arithmetic expression is invoked in a conditional statement, its default type is an INTEGER.

Variable types should not be mixed.

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

DIM sum1,bit1,bit2
bit1 = 2
```

```
bit2 = 3

DIM volume,height,area
height = 5
area = 20

sum1 = bit1 + bit2
volume = height * area

print "\nSum1 = ";sum1
print "\nVolume = ";volume;"\n"
```

Expected Output:

```
Sum1 = 5
Volume = 100
```

Arithmetic operators can be unitary or binary. A unitary operator acts on a variable or constant which follows it, whereas a binary operator acts on the two entities on either side.

Operators in an expression observe a precedence which is used to evaluate the final result using reverse polish notation. An explicit precedence order can be forced by using **(** and **)** in the usual manner.

The following is the order of precedence within operators:

- Unitary operators have the highest precedence

!	logical NOT
~	bit complement
-	negative (negate the variable or number – multiplies it by -1)
+	positive (make positive – multiplies it by +1)

- Precedence then devolves to the binary operators in the following order:

*	Multiply
/	Divide
%	Modulus
+	Addition
-	Subtraction
<<	Arithmetic Shift Left
>>	Arithmetic Shift Right
<	Less Than (results in a 0 or 1 value in the expression)
<=	Less Than Or Equal (results in a 0 or 1 value in the expression)
>	Greater Than (results in a 0 or 1 value in the expression)
>=	Greater Than Or Equal (results in a 0 or 1 value in the expression)

==	Equal To (results in a 0 or 1 value in the expression)
!=	Not Equal To (results in a 0 or 1 value in the expression)
&	Bitwise AND
^	Bitwise XOR (exclusive OR)
|	Bitwise OR
&&	Logical AND (results in a 0 or 1 value in the expression)
^^	Logical XOR (results in a 0 or 1 value in the expression)
||	Logical OR (results in a 0 or 1 value in the expression)

Conditionals

Conditional functions are used to alter the sequence of program flow by providing a range of operations based on checking conditions.

Note: *smart* BASIC does not support program flow functionality based on unconditional statements, such as JUMP or GOTO. In most cases where a GOTO or JUMP might be employed, ONERROR conditions are likely to be more appropriate.

Conditional blocks can be nested. This applies to combinations of DO, UNTIL, DOWHILE, FOR, IF, WHILE, and SELECT. The depth of nesting depends on the build of *smart* BASIC but in general, nesting up to 16 levels is allowed and can be modified using the AT+CFG command.

DO / UNTIL

This DO/UNTIL construct allows a block of one or more statements to be processed until a condition becomes true.

DO
statement block
UNTIL *arithmetic expr*

- **Statement block** – A valid set of program statements. Typically several lines of application.
- **Arithmetic expression** – A valid arithmetic or logical expression. Arithmetic precedence is defined in the section '<u>Arithmetic Expressions'</u>.

For DO / UNTIL, if the arithmetic expression evaluates to zero, then the statement block is executed again. Care should be taken to ensure this does not result in infinite loops.

Interactive Command: NO

```
//Example :: DoUntil.sb (See in BL600CodeSnippets.zip)
DIM a AS INTEGER  //don't really need to supply AS INTEGER
a=1
DO
  a = a+1
  PRINT a
UNTIL a==10       //loop will end when A gets to the value 10
```

Expected Output:

```
2345678910
```

DO / UNTIL is a core function.

DO / DOWHILE

This DO / DOWHILE construct allows a block of one or more statements to be processed while the expression in the DOWHILE statement evaluates to a true condition.

DO
statement block
DOWHILE *arithmetic expr*

- **Statement block** – A valid set of program statements. Typically several lines of application
- **Arithmetic expression** – A valid arithmetic or logical expression. Arithmetic precedence is defined in the section 'Arithmetic Expressions'.

For DO / DOWHILE, if the arithmetic expression evaluates to a non-zero value, then the statement block is executed again. Care should be taken to ensure this does not result in infinite loops.

Interactive Command: NO

```
//Example :: DoWhile.sb (See in BL600CodeSnippets.zip)
DIM a AS INTEGER  //don't really need to supply AS INTEGER
a=1
DO
  a = a+1
  PRINT a
DOWHILE a<10       //loop will end when A gets to the value 10
```

Expected Output:

```
2345678910
```

DO / DOWHILE is a core function.

FOR / NEXT

The FOR / NEXT composite statement block allows program execution to be controlled by the evaluation of a number of variables. Using the tokens TO or DOWNTO determines the order of execution. An optional STEP condition allows the conditional function to step at other than unity steps. Given the choice of either TO/DOWNTO and the optional STEP, there are four variants:

FOR var = *arithexpr1* TO *arithexpr2*
statement block
NEXT

FOR var = *arithexpr1* TO *arithexpr2* STEP arithexpr3
statement block
NEXT

FOR var = *arithexpr1* DOWNTO *arithexpr2*
statement block
NEXT

FOR var = *arithexpr1* DOWNTO *arithexpr2* STEP arithexpr3
statement block
NEXT

- **Statement block** – A valid set of program statements. Typically several lines of application which can include nested conditional statement blocks.
- **Var** – A valid INTEGER variable which can be referenced in the statement block
- ***Arithexpr1*** – A valid arithmetic or logical expression. ***arithexpr1*** is enumerated as the starting point for the FOR NEXT loop.
- ***Arithexpr2*** – A valid arithmetic or logical expression. ***arithexpr2*** is enumerated as the finishing point for the FOR NEXT loop.
- ***Arithexpr3*** – A valid arithmetic or logical expression. ***arithexpr3*** is enumerated as the step in variable values in processing the FOR NEXT loop. If STEP and ***arithexpr3*** are omitted, then a unity step is assumed.

Note: Arithmetic precedence, is as defined in the section 'Arithmetic Expressions'

The lines of code comprising the ***statement block*** are processed with ***var*** starting with the value calculated or defined by **arithexpr1**. When the **NEXT** command is reached and processed, the **STEP** value resulting from ***arithexpr3*** is added to ***var*** if **TO** is specified, or subtracted from ***var*** if **DOWNTO** is specified.

The function continues to loop until the variable ***var*** contains a value less than or equal to ***arithexpr2*** in the case where TO is specified, or greater than or equal to ***arithexpr2*** in the alternative case where **DOWNTO** is specified.

Note: In *smart* BASIC the Statement Block is ALWAYS executed at least once.

Interactive Command: NO

```
//Example :: ForNext.sb (See in BL600CodeSnippets.zip)
DIM a
FOR a=1 TO 2
  PRINT "Hello"
NEXT

print "\n"

FOR a=2 DOWNTO 1
  PRINT "Hello"
NEXT

print "\n"

FOR a=1 TO 4   STEP 2
  PRINT "Hello"
NEXT
```

Expected Output:

```
HelloHello
HelloHello
HelloHello
```

FOR / NEXT is a core function.

IF THEN / ELSEIF / ELSE / ENDIF

The IF statement construct allows a block of code to be processed depending on the evaluation of a condition expression. If the statement is true (equates to non-zero), then the following block of application is processed until an ENDIF, ELSE, or ELSEIF command is reached.

Each ELSEIF allows an alternate statement block of application to be executed if that conditional expression is true and any preceding conditional expressions were untrue.

Multiple ELSEIF commands may be added, but only the statement block immediately following the first true conditional expression encountered is processed within each IF command.

The final block of statements is of the form ELSE and is optional.

IF *arithexpr_1* THEN
statement block A
ENDIF

IF *arithexpr_1* THEN
statement block A
ELSE
statement block B
ENDIF

IF *arithexpr_1* THEN
statement block A
ELSEIF *arithexpr_2* THEN
statement block B
ELSE
statement block C
ENDIF

- **Statement block A|B|C** – A valid set of zero or more program statements.
- **Arithexpr_n** – A valid arithmetic or logical expression. A valid arithmetic or logical expression. Arithmetic precedence, is as defined in the section 'Arithmetic Expressions'.

All IF constructions must be terminated with an ENDIF statement.

Note: As the arithmetic expression in an IF statement is making a comparison, rather than setting a variable, the double == operator MUST be used, e.g.

IF i==3 THEN : SLEEP(200)

See the Arithmetic Expressions section for more options.

Interactive Command: NO

```
//Example :: IfThenElse.sb (See in BL600CodeSnippets.zip)
DIM n
```

```
n=1
IF n>0 THEN
  PRINT "Laird Rocks\n"
ENDIF
IF n==0 THEN
  PRINT "n is 0"
ELSEIF n==1 THEN
  PRINT "n is 1"
ELSE
  PRINT "n is not 0 nor 1"
ENDIF
```

Expected Output:

```
Laird Rocks
N is 1
```

IF is a core function.

WHILE / ENDWHILE

The WHILE command tests the arithmetic expression that follows it. If it equates to non-zero then the following block of statements is executed until an ENDWHILE command is reached. If it is zero, then execution continues after the next ENDWHILE.

WHILE *arithexpr*
statement block
ENDWHILE

- **Statement block** – A valid set of zero or more program statements.
- **Arithexpr** – A valid arithmetic or logical expression. Arithmetic precedence, is as defined in the section 'Arithmetic Expressions'.

All WHILE commands must be terminated with an ENDWHILE statement.

Interactive Command: NO

```
//Example :: While.sb (See in BL600CodeSnippets.zip)
DIM n
n=0

//now print "Hello" ten times

WHILE n<10
  PRINT " Hello " ;n
  n=n+1
ENDWHILE
```

Expected Output:

```
Hello 0 Hello 1 Hello 2 Hello 3 Hello 4 Hello 5 Hello 6 Hello 7 Hello 8
Hello 9
```

WHILE is a core function.

SELECT / CASE / CASE ELSE / ENDSELECT

SELECT is a conditional command that uses the value of an arithmetic expression to pass execution to one of a number of blocks of statements which are identified by an appropriate CASE nnn statement, where nnn is an integer constant. After completion of the code, which is marked by a CASE nnn or CASE ELSE statement, execution of the application moves to the line following the ENDSELECT command. In a sense, it is a more efficient implementation of an IF block with many ELSEIF statements.

An initial block of code can be included after the SELECT statement. This is always processed. When the first CASE statement is encountered, execution moves to the CASE statement corresponding to the computed value of the arithmetic expression in the SELECT command.

After selection of the appropriate CASE, the relevant statement block is executed until a CASE, BREAK or ENDSELECT command is encountered. If a match is not found, then the CASE ELSE statement block is run.

It is mandatory to include a final CASE ELSE statement as the final CASE in a SELECT operation.

SELECT *arithexpr*
  unconditional statement block
CASE integerconstA
  statement block A
CASE integerconstB
  statement block B
CASE integerconstc,integerconstd, integerconste, integerconstf, …
  statement block C
CASE ELSE
  statement block
ENDSELECT

- **Unconditional statement block –** An optional set of program statements, which are always executed.
- **Statement block –** A valid set of zero or more program statements.
- **Arithexpr –** A valid arithmetic or logical expression. Arithmetic precedence, is as defined in the section 'Arithmetic Expressions'.
- **IntegerconstX** – One or more comma seperated integer constants corresponding to one of the possible values of ***arithexpr*** which identifies the block that will get processed.

Interactive Command: NO

```
//Example :: SelectCase.sb (See in BL600CodeSnippets.zip)
DIM a,b,c
a=3 : b=4               //Use ":" to write multiple commands on one line
SELECT a*b
    CASE 10
        c=10
    CASE 12              //this block will get processed
        c=12
    CASE 14,156,789,1022
        c=-1
    CASE ELSE
        c=0
ENDSELECT
PRINT c
```

Expected Output:

```
12
```

SELECT is a core function.

BREAK

BREAK is relevant in a WHILE/ENDWHILE, DO/UNTIL, DO/DOWHILE, FOR/NEXT, or SELECT/ENDSELECT compound construct. It forces the program counter to exit the currently processing block of statements.

For example, in a WHILE/ENDWHILE loop, the statement BREAK stops the loop and forces the command immediately after the ENDWHILE to be processed. Similarly, in a DO/UNTIL, the statement immediately after the UNTIL is processed.

BREAK

Interactive Command: NO

```
//Example :: Break.sb (See in BL600CodeSnippets.zip)
DIM n
n=0

WHILE n<10
  n=n+1
  IF n==5 THEN
    BREAK
  ENDIF
  PRINT "Hello " ;n
ENDWHILE

PRINT "\nFinished\n"
```

Expected Output:

```
Hello 1Hello 2Hello 3Hello 4
Finished
```

BREAK is a core function.

CONTINUE

CONTINUE is used within a WHILE/ENDWHILE, DO/UNTIL, DO/DOWHILE, or FOR/NEXT compound construct, where it forces the program counter to jump to the beginning of the loop.

CONTINUE

Interactive Command: YES

```
//Example :: Continue.sb (See in BL600CodeSnippets.zip)
DIM n
n=0

WHILE n<10
  n=n+1
```

```
  IF n==5 THEN
    CONTINUE
  ENDIF
  PRINT "Hello " ;n
ENDWHILE

PRINT "\nFinished\n"
```

Expected Output:

```
Hello 1Hello 2Hello 3Hello 4Hello 6Hello 7Hello 8Hello 9Hello 10
Finished
```

CONTINUE is a core function.

Error Handling

Error handling functions are provided to allow program control for instances where exceptions are generated for errors. These allow graceful continuation after an error condition is encountered and are recommended for robust operation in an unattended embedded use case scenario.

In an embedded environment, it is recommended to include at least one ONERROR and one ONFATALERROR statement within each application. This ensures that if the module is running unattended, then it can reset and restart itself without the need for operator intervention.

ONERROR

ONERROR is used to redirect program flow to a handler function that can attempt to modify operation or correct the cause of the error. Three different options are provided in conjunction with ONERROR: REDO, NEXT, and EXIT.

The GETLASTERROR() command should be used in the handler routine to determine the type of error that was generated.

ONERROR REDO routine	On return from the routine, the statement that originally caused the error is reprocessed.
ONERROR NEXT routine	On return from the routine, the statement that originally caused the error is skipped and the following statement is processed.
ONERROR EXIT	If an error is encountered, the application will exit and return operation to Interactive Mode.

Arguments:

Routine – The handler SUB that is called when the error is detected. This must be a SUB routine which takes no parameters. It must not be a function. It must exist within the application PRIOR to this ONERROR command being compiled.

Interactive Command: NO

```
//Example :: OnError.sb (See in BL600CodeSnippets.zip)
DIM a,b,c
SUB HandlerOnErr()              //Do this when an error occurs
   DIM le
   le = GetLastError()
   PRINT "Error code 0x";le;" denotes a Divide by zero error.\n"
   PRINT "Let's make b equal 25 instead of 0\n\n"
   b=25
ENDSUB
a=100 : b=0
ONERROR REDO HandlerOnErr       //Calls the "HandlerOnErr" routine.
                                //After that, the error causing statement
                                //(below) is reprocessed
c=a/b
print "c now equals ";c
```

Expected Output:

```
Error code 0x1538 denotes a Divide by zero error.
Let's make b equal 25 instead of 0

c now equals 4
```

ONERROR is a core function.

ONFATALERROR

ONFATALERROR is used to redirect program flow to a subroutine that can attempt to modify operation or correct the cause of a fatal error. Three different options are provided – REDO, NEXT, and EXIT.

The GETLASTERROR() command should be used in the subroutine to determine the type of error that was generated.

ONFATALERROR REDO routine	On return from the routine, the statement that originally caused the error is reprocessed.
ONFATALERROR NEXT routine	On return from the routine, the statement that originally caused the error is skipped and the following statement is processed.
ONFATALNERROR EXIT	If an error is encountered, the application will exit and return the operation to Interactive Mode.

Please Note: At present, no fatal errors are thrown in the BL600 module.

ONFATALERROR is a core function.Event Handling

An application written for an embedded platform is left unattended and in most cases waits for something to happen in the real world, which it detects via an appropriate interface. When something happens it needs to react to that event. This is unlike sequential processing where the program code order is written in the expectation of a series of preordained events. Real world interaction is not like that and so this implementation of *smart* BASIC has been optimised to force the developer of an application to write applications as a group of handlers used to process events in the order as and when those events occur.

This section describes the statements used to detect and manage those events.

WAITEVENT

WAITEVENT is used to wait for an event, at which point an event handler is called. The event handler must be a function that takes no arguments and returns an INTEGER.

If the event handler returns a zero value, then the next statement after WAITEVENT is processed. Otherwise WAITEVENT continues to wait for another event.

WAITEVENT

Interactive Command: NO

```
FUNCTION Func0()
  PRINT "\nEV0"
ENDFUNC 1

FUNCTION Func1()
  PRINT "\nEV1"
ENDFUNC 0

ONEVENT EV0 CALL Func0
ONEVENT EV1 CALL Func1

WAITEVENT               //wait  for an event  to occur

PRINT "\n Got here because EV1 happened"
```

WAITEVENT is a core function.

ONEVENT

ONEVENT is used to redirect program flow to a predefined FUNCTION that can respond to a specific event when that event occurs. This is commonly an external event, such as an I/O pin change or a received data packet, but can be a software generated event too.

ONEVENT symbolic_name CALL routine	When a particular event is detected, program execution is directed to the specified function.
ONEVENT symbolic_name DISABLE	A previously declared ONEVENT for an event is unbound from the specified subroutine. This allows for complex applications that need to optimise runtime processing by allowing an alternative to using a SELECT statement.

Events are detected from within the run-time engine – in most cases via interrupts - and are only processed by an application when a WAITEVENT statement is processed.

Until the WAITEVENT, all events are held in a queue.

Note: When WAITEVENT services an event handler, if the return value from that routine is non-zero, then it continues to wait for more events. A zero value forces the next statement after WAITEVENT to be processed.

Arguments:

Routine – The FUNCTION that is called when the event is detected. This must be a function which returns an INTEGER and takes no parameters. It must not be a SUB routine. It must exist within the application PRIOR to this ONEVENT command.

Symbolic_Name – A symbolic event name which is predefined for a specific *smart* BASIC module.

Some Symbolic Event Names:

A partial list of symbolic event names are as follows:-

EVTMRn	Timer n has expired (see Timer Events)
EVUARTRX	Data has arrived in UART interface
EVUARTTXEMPTY	The UART TX ring buffer is empty

Note: Some symbolic names are specific to a particular hardware implementation.

Interactive Command: NO

```
//Example :: OnEvent.sb (See in BL600CodeSnippets)
DIM rc

FUNCTION Btn0press()
  PRINT "\nButton 0 has been pressed"
ENDFUNC 1                                //Will continue waiting for an event

FUNCTION Btn0rel()
  PRINT "\nButton 0 released. Resume waiting for an event\n"
ENDFUNC 1

FUNCTION Btn1press()
    PRINT "\nButton 1 has been pressed"
ENDFUNC 1

FUNCTION Btn1rel()
    PRINT "\nButton 1 released. No more waiting for events\n"
ENDFUNC 0

rc = gpiobindevent(0,16,0)           //binds gpio transition high on sio16 (button 0)
to event 0
rc = gpiobindevent(1,16,1)          //binds gpio transition low on sio16 (button 0)
to event 1
rc = gpiobindevent(2,17,0)          //binds gpio transition high on sio16 (button 1)
to event 2
rc = gpiobindevent(3,17,1)          //binds gpio transition low on sio16 (button 2)
to event 3

onevent evgpiochan0 call Btn0rel     //detects when button 0 is released and calls
the function
onevent evgpiochan1 call Btn0press   //detects when button 0 is pressed and calls the
function
```

```
 onevent evgpiochan2 call Btn1rel     //detects when button 1 is released and calls
the function
 onevent evgpiochan3 call Btn1press   //detects when button 1 is pressed and calls the
function

 PRINT "\nWaiting for an event...\n"
 WAITEVENT                        //wait  for an event  to occur

 PRINT "\nGot here because evgpiochan2 happened"
```

Expected Output:

ONEVENT is a core function.

Miscellaneous Commands

PRINT

The PRINT statement directs output to an output channel which may be the result of multiple comma or semicolon separated arithmetic or string expressions. The output channel is a UART interface in most platforms.

PRINT *exprlist*

Arguments:

exprlist An expression list which defines the data to be printed consisting of comma or semicolon separated arithmetic or string expressions.

Formatting with PRINT – Expression Lists

Expression lists are used for outputting data – principally with the PRINT and the SPRINT command. Two types of Expression lists are allowed – arithmetic and string. Multiple valid Expression lists may be concatenated with a comma or a semicolon to form a complex Expression list.

The use of a comma forces a TAB character between the Expression lists it separates and a semicolon generates no output. The latter results in the output of two expressions being concatenated without any white space.

Numeric Expression Lists

Numeric variables are formatted in the following form:

<type.base> arithexpr <separator>

Where,

- **Type** – Must be INTEGER for integer variables
- **base** – Integers can be forced to print in decimal, octal, binary, or hexadecimal by prefixing with D', O', B', or H' respectively.
  For example, **INTEGER.h' somevar** will result in the content of somevar being output as a hexadecimal string.
- **Arithexpr –** A valid arithmetic or logical expression.
- **Separator** – One of the characters **,** or **;** which have the following meaning:

  , Insert a tab before the next variable.

  ; Print the next variable without a space.

String Expression Lists

String variables are formatted in the following form:

<type . minchar> strexpr< separator>

- **Type** – Must be STRING for string variables. The ***type*** must be followed by a full stop to delineate it from the width field that follows.
- **Minchar** – An optional parameter which specifies the number of characters to be printed for a string variable or expression. If necessary, leading spaces are filled with spaces.
- **strexpr –** A valid string or string expression.
- **Separator** – One of the characters , or ; which have the following meaning:

  , Insert a tab before the next variable.

  ; Print the next variable without a space.

Interactive Command: YES

```
//Example :: Print.sb (See in BL600CodeSnippets.zip)
PRINT  "Hello \n"
DIM a
a=100
PRINT a
PRINT "\nIn Hex", "0x"; INTEGER.H' 100 ;"\n"
PRINT "In Octal ", INTEGER.O' 100 ;"\n"
PRINT "In Binary ", INTEGER.B' 100 ;"\n"
```

Expected Output:

```
Hello
100
In Hex 0x00000064
In Octal      00000000144
In Binary     00000000000000000000000001100100
```

PRINT is a core function.

SPRINT

The SPRINT statement directs output to a string variable, which may be the result of multiple comma or semicolon separated arithmetic or string expressions.

It is very useful for creating strings with formatted data.

SPRINT #stringvar, *exprlist*

Arguments:

Stringvar – A pre-declared string variable.

Exprlist – An expression list which defines the data to be printed; consisting of comma or semicolon separated arithmetic or string expressions.

Formatting with SPRINT – Expression Lists

Expression lists are used for outputting data – principally with the PRINT command and the SPRINT command. Two types of Expression lists are allowed – arithmetic and string. Multiple valid Expression lists may be concatenated with a comma or a semicolon to form a complex Expression list.

The use of a comma forces a TAB character between the Expression lists it separates and a semicolon generates no output. The latter results in the output of two expressions being concatenated without any whitespace.

Numeric Expression Lists

Numeric variables are formatted in the following form:

<type.base> arithexpr <separator>

Where,

- **Type** – Must be INTEGER for integer variables
- **base** – Integers can be forced to print in decimal, octal, binary, or hexadecimal by prefixing with D', O', B', or H' respectively.
  For example, **INTEGER.h' somevar** will result in the content of somevar being output as a hexadecimal string.
- **Arithexpr –** A valid arithmetic or logical expression.
- **Separator** – One of the characters **,** or **;** which have the following meaning:

  , Insert a tab before the next variable.

  ; Print the next variable without a space.

String Expression Lists

String variables are formatted in the following form:

<type . minchar> strexpr< separator>

- **Type** – Must be STRING for string variables. The **type** must be followed by a full stop to delineate it from the width field that follows.
- **minchar** - An optional parameter which specifies the number of characters to be printed for a string variable or expression. If necessary, leading spaces are filled with spaces.
- **strexpr –** A valid string or string expression.
- **separator** – One of the characters , or ; which have the following meaning:

,	Insert tab before next variable
;	Print next variable without a space

Interactive Command: YES

```
//Example :: SPrint.sb (See in BL600CodeSnippets.zip)
DIM a,s$ : a=100
//Note: SPRINT replaces the content of s$ with exprlist each time it is used
SPRINT #s$,a                  //s$ now contains 100
PRINT "\n";s$;"\n"
SPRINT #s$,INTEGER.H'a        //s$ now contains 64
PRINT s$;"\n"
SPRINT #s$,INTEGER.O'a        //s$ now contains 144
PRINT s$;"\n"
SPRINT #s$,INTEGER.B'a        //s$ now contains 1100100
PRINT s$;"\n"
```

Expected Output:

```
100
00000064
00000000144
00000000000000000000000001100100
```

SPRINT is a core function.

STOP

STOP is used within an application to stop it running so that the device falls back into Interactive Command line mode.

STOP

It is normally limited to use in the prototyping and debugging phases.

Once in Interactive Mode, the command RESUME is used to restart the application from the next statement after the STOP statement.

Interactive Command: NO

```
//Example :: Stop.sb (See in BL600CodeSnippets.zip)
DIM a, s$
a=100

//Note: SPRINT replaces the content of s$ with exprlist each time it is used
SPRINT #s$,a                  //s$ now contains 100
PRINT "\n";s$;"\n"
SPRINT #s$,INTEGER.H'a        //s$ now contains 64
STOP
PRINT s$;"\n"
SPRINT #s$,INTEGER.O'a        //s$ now contains 144
PRINT s$;"\n"
SPRINT #s$,INTEGER.B'a        //s$ now contains 1100100
PRINT s$;"\n"
```

Expected Output:

```
100

01     0702
resume
00000064
00000000144
00000000000000000000000001100100
```

STOP is a core function.

BP

COMMAND

The BP (Breakpoint) statement is used to place a BREAKPOINT in the body of an application. The integer constant that is associated with each breakpoint is a developer supplied identifier which gets echoed to the standard output when that breakpoint is encountered. This allows the application developer to locate which breakpoint resulted in the output. Execution of the application is then paused and operation passed back to Interactive mode.

BP nnnn

After execution is returned to Interactive mode, either RESUME can be used to continue execution or the Interactive mode command SO can be used to step through the next statements.

Note: The next state is the BP statement itself, hence multiple SO commands may need to be issued.

Arguments

nnnn A constant integer identifier for each breakpoint in the range 0 to 65535. The integers should normally be unique to allow the breakpoint to be determined, but this is the responsibility of the programmer. There is no limit to the number of breakpoints that can be inserted into an application other than ensuring that the maximum size of the compiled code does not exceed the 64 Kword limit.

Note: It is helpful to make the integer identifiers relevant to the program structure to help the debugging process. A useful tip is to set them to the program line.

Interactive Command: NO

```
//Example :: BP.sb (See in BL600CodeSnippets.zip)
PRINT "hello"
BP 1234
PRINT "world"
PRINT "Laird"
PRINT "Rocks"
BP 5678
PRINT "the"
PRINT "world"
```

Expected Output (Depending on what order you use the commands SO and RESUME):

BP is a core function.

5. 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 next chapter.

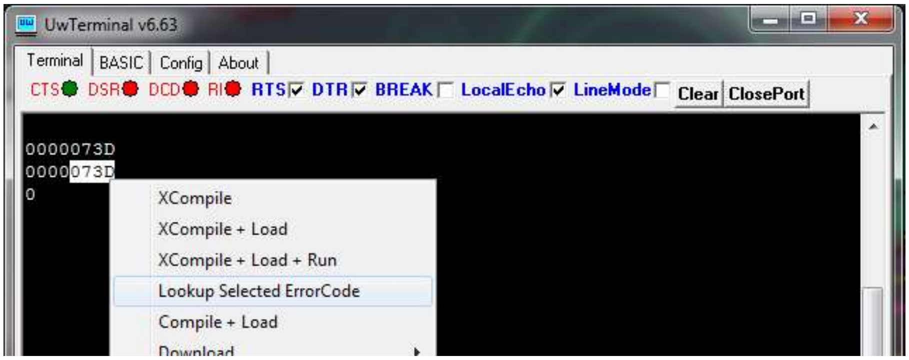
Result Codes

Some of these built-in routines are subroutines, and some are functions. Functions always return a value, and for some of these functions the value returned is a result code, indicating success or failure in executing that function. A failure may not necessarily result in a run-time error (see GetLastError() and ResetLastError()), but may lead to an unexpected output.

Being able to see what causes a failure greatly helps with the debugging process. If you declare an integer variable e.g. 'rc' and set it's value to your function call, after the function is executed you can print rc and see the result code. For it to be useful, it has to be in Hexadecimal form, so prefix your result code variable with " INTEGER.H' " when printing it. You can also save a bit of memory by printing the return value from the function directly, without the use of a variable.

```
//Example :: ResultCodes.sb (See in BL600CodeSnippets.zip)
DIM cB,nItems,rc,s$

rc=CircBufItems(cB,nItems)
PRINT INTEGER.H'rc

PRINT "\n";             //New line

//Printing return value directly
PRINT INTEGER.H'CircBufItems(cB,nItems)

//To remove the leading zeros
SPRINT #s$, INTEGER.H'CircBufItems(cB,nItems)
StrShiftLeft(s$,4) : PRINT s$
```

Now highlight the last 4 characters of the result code in UwTerminal and select "Lookup Selected ErrorCode":



Expected Output:

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

Information Routines

GETLASTERROR

FUNCTION

GETLASTERROR is used to find the value of the most recent error and is most useful in an error handler associated with ONERROR and ONFATALERROR statements which were described in the previous section.

You can get a verbose error description by printing the error value, then highliting it in UwTerminal, and selecting ‘Lookup Selected ErrorCode’.

GETLASTERROR ()

Returns INTEGER Last error that was generated.

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments None

Interactive Command: NO

```
//Example :: GetLastError.sb (See in BL600CodeSnippets.zip)
DIM err
err = GETLASTERROR()
PRINT "\nerror  = 0x" ; INTEGER.H'err
```

Expected Output (If no errors from last application run):

```
error  = 0x00000000
```

GETLASTERROR is a core function.

RESETLASTERROR

SUBROUTINE

Resets the last error, so that calling GETLASTERROR() returns a success.

RESETLASTERROR ()

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments None

Interactive Command: NO

```
//Example :: ResetLastError.sb (See in BL600CodeSnippets.zip)
DIM err : err = GETLASTERROR()
RESETLASTERROR()
PRINT "\nerror  = 0x" ; INTEGER.H'err
```

Expected Result:

```
error  = 0x00000000
```

RESETLASTERROR is a core function.

SYSINFO

FUNCTION

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

SYSINFO(varId)

Returns INTEGER .Value of information corresponding to integer ID requested.

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

varId *byVal* varId *AS INTEGER*

An integer ID which is used to determine which information is to be returned as described below.

0	ID of device, for the BL600 module the value will be 0x42460600
3	Version number of Module Firmware. For example W.X.Y.Z will be returned as a 32 bit value made up as follows: (W<<26) + (X<<20) + (Y<<6) + (Z) where Y is the Build number and Z is the 'Sub-Build' number
33	BASIC core version number
601	Flash File System: Data Segment: Total Space
602	Flash File System: Data Segment: Free Space

603 Flash File System: Data Segment: Deleted Space
611 Flash File System: FAT Segment: Total Space
612 Flash File System: FAT Segment: Free Space
613 Flash File System: FAT Segment: Deleted Space
631 NvRecord Memory Store Segment: Total Space
632 NvRecord Memory Store Segment: Free Space
633 NvRecord Memory Store Segment: Deleted Space
1000 BASIC compiler HASH value as a 32 bit decimal value
1001 How RAND() generates values: 0 for PRNG and 1 for hardware assist
1002 Minimum baudrate
1003 Maximum baudrate
1004 Maximum STRING size
1005 Will be 1 for run-time only implementation, 3 for compiler included
2000 Reset Reason
8 : Self-Reset due to Flash Erase
9 : ATZ
10 : Self-Reset due to *smart* BASIC app invoking function RESET()
2002 Timer resolution in microseconds
2003 Number of timers available in a *smart* BASIC Application
2004 Tick timer resolution in microseconds
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
2014 Total number of transmission buffers for sending attribute NOTIFIES
2015 Number of transmission buffers for sending attribute NOTIFIES – free
2016 Radio activity of the baseband
0 : no activity
1 : advertising
2 : connected
3 : broadcasting and connected
2018 Returns the TX power while pairing in progress (see also 2008)
2019 Default ring buffer length for notify/indicates in gatt client manager (see BleGattcOpen function)
2020 Maximum ring buffer length for notify/indicates in gatt client manager (see BleGattcOpen function)
2021 Stack tide mark in percent. Values near 100 is not good.
2022 Stack size
2023 Initial Heap size
0x8000 to 0x81FF
Content of FICR register in the Nordic nrf51 chipset. In the nrf51 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.

Interactive Command: No

```
//Example :: SysInfo.sb (See in BL600CodeSnippets.zip)
PRINT "\nSysInfo 1000   = ";SYSINFO(1000)    // BASIC compiler HASH value
PRINT "\nSysInfo 2003   = ";SYSINFO(2003)    // Number of timers
PRINT "\nSysInfo 0x8010 = ";SYSINFO(0x8010)  // Code memory page size from FICR
```

Expected Output (For BL600):

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

SYSINFO is a core language function.

SYSINFO$

FUNCTION

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

SYSINFO$(varId)

Returns STRING .Value of information corresponding to integer ID requested.

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

varId *byVal* varId *AS INTEGER*

An integer ID which is used to determine which information is to be returned as described below.

4 The Bluetooth address of the module. It is seven bytes long. First byte is 00 for IEEE public address and 01 for random public address. Next six bytes are the address.

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

Interactive Command: No

```
//Example :: SysInfo$.sb (See in BL600CodeSnippets.zip)
PRINT "\nSysInfo$(4)   = ";SYSINFO$(4)   // address of module
PRINT "\nSysInfo$(14)  = ";SYSINFO$(14)  // public random address
PRINT "\nSysInfo$(0)   = ";SYSINFO$(0)
```

Expected Output:

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

SYSINFO$ is a core language function.

Event & Messaging Routines

SENDMSGAPP

FUNCTION

This function is used to send an EVMSGAPP message to your application so that it can be processed by a handler from the WAITEVENT framework. It is useful for serialised processing.

For messages to be processed, the following statement must be processed so that a handler is associated with the message.

ONEVENT EVMSGAPP CALL HandlerMsgApp

Where a handler such as the following has been defined prior to the ONEVENT statement as follows:

```
FUNCTION HandlerMsgApp(BYVAL nMsgId AS INTEGER, BYVAL nMsgCtx AS INTEGER) AS INTEGER
    //do something with nMsgId and nMsgCtx
ENDFUNC 1
```

SENDMSGAPP(msgId, msgCtx)

Returns INTEGER 0000 if successfully sent.

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

msgId — *byVal* msgId *AS INTEGER*

Will be presented to the EVMSGAPP handler in the msgId field

msgCtx — *byVal* msgCtx *AS INTEGER*

Will be presented to the EVMSGAPP handler in the msgCtx field.

Interactive Command: NO

```
//Example :: SendMsgApp.sb (See in BL600CodeSnippets.zip)
DIM rc

FUNCTION HandlerMsgApp(BYVAL nMsgId AS INTEGER, BYVAL nMsgCtx AS INTEGER) AS INTEGER
  PRINT "\nId=";nMsgId;" Ctx=";nMsgCtx
ENDFUNC 1
```

```
ONEVENT EVMSGAPP CALL HandlerMsgApp
rc = SendMsgApp(100,200)
WAITEVENT
```

Expected Output:

```
Id=100 Ctx=200
```

SENDMSGAPP is a core function.

Arithmetic Routines

ABS

FUNCTION

Returns the absolute value of its INTEGER argument.

ABS (var)

Returns INTEGER Absolute value of *var*.

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow
- If the value of var is 0x80000000 (decimal -2,147,483,648) then an exception is thrown as the absolute value for that value causes an overflow as 33 bits are required to convey the value.

Arguments:

var *byVal* var *AS INTEGER*

The variable whose absolute value is required.

Interactive Command: No

```
//Example :: ABS.sb (See in BL600CodeSnippets.zip)
DIM s1 as INTEGER,s2 as INTEGER
s1 = -2 : s2 = 4
PRINT s1, ABS(s1);"\n";s2, Abs(s2)
```

Expected Output:

```
-2      2
4       4
```

ABS is a core language function.

MAX

FUNCTION

Returns the maximum of two integer values.

MAX (var1, var2)

Returns INTEGER The returned variable is the arithmetically larger of *var1* and *var2*.

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

var1 — *byVal* var1 *AS INTEGER*
The first of two variables to be compared.

var2 — *byVal* var2 *AS INTEGER*
The second of two variables to be compared.

Interactive Command: No

```
//Example :: MAX.sb (See in BL600CodeSnippets.zip)
DIM s1,s2
s1=-2 : s2=4
PRINT s1,s2
PRINT "\n The Maximum of these two integers is "; MAX(s1,s2)
```

Expected Output:

```
-2     4
The Maximum of these two integers is 4
```

MAX is a core language function.

MIN

FUNCTION

Returns the minimum of two integer values.

MIN (var1, var2)

Returns INTEGER The returned variable is the arithmetically smaller of *var1* and *var2*.

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

var1 — *byVal* var1 *AS INTEGER*
The first of two variables to be compared.

var2 — *byVal* var2 *AS INTEGER*
The second of two variables to be compared.

Interactive Command: No

```
//Example :: MIN.sb (See in BL600CodeSnippets.zip)
DIM s1,s2
s1=-2 : s2=4
PRINT s1,s2
PRINT "\nThe Minimum of these two integers is "; MIN(s1,s2)
```

Expected Output:

```
-2     4
The Maximum of these two integers is -2
```

MIN is a core language function.

String Routines

When data is displayed to a user or a collection of octets need to be managed as a set, it is useful to represent them as strings. For example, in BLE modules there is a concept of a database of 'attributes' which are just a collection of octets of data up to 512 bytes in length.

To provide the ability to deal with strings, *smart* BASIC contains a number of commands that can operate on STRING variables.

LEFT$

Retrieves the leftmost n characters of a string.

LEFT$(string,length)

Function

Returns STRING The leftmost 'length' characters of string as a STRING object.

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow
- Memory Heap Exhausted

Arguments:

string — *byRef string AS STRING*
The target string which cannot be a const string.

length — *byVal length AS INTEGER*
The number of leftmost characters that are returned.

If 'length' is larger than the actual length of ***string*** then the entire string is returned

Notes: **string** cannot be a string constant, e.g. "the cat", but must be a string variable and so if you must use a const string then first save it to a temp string variable and then pass it to the function

Interactive Command: NO

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

```
DIM newstring$
DIM s$

s$="Arsenic"
newstring$ = LEFT$s$,4)
print newstring$; "\n"
```

Expected Output:

```
Arse
```

LEFT$ is a core language function.

MID$

FUNCTION

Retrieves a string of characters from an existing string. The starting position of the extracted characters and the length of the string are supplied as arguments.

If 'pos' is positive then the extracted string starts from offset 'pos'. If it is negative then the extracted string starts from offset 'length of string – abs(pos)'

MID$(string, pos, length)

Returns STRING The 'length' characters starting at offset 'pos' of **string**.

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow
- Memory Heap Exhausted

Arguments:

string — *byRef string AS STRING*
The target string which cannot be a const string.

pos — *byVal pos AS INTEGER*
The position of the first character to be extracted. The leftmost character position is 0 (see examples).

length — *byVal length AS INTEGER*
The number of characters that are returned.

If 'length' is larger than the actual length of ***string*** then the entire string is returned from the position specified. Hence pos=0, length=65535 returns a copy of ***string.***

Note: **string** cannot be a string constant, e.g. "the cat", but must be a string variable and so if you must use a const string then first save it to a temp string variable and then pass it to the function.

Interactive Command: NO

```
//Example :: MID.sb (See in BL600CodeSnippets.zip)
DIM s$ : s$="Arsenic"
DIM new$ : new$ = MID$(s$,2,4)
```

```
PRINT new$; "\n"
```

Expected Output:

```
Arse
abcdef
cdefg
hij
```

MID$ is a core language function.

RIGHT$

FUNCTION

Retrieves the rightmost n characters from a string.

RIGHT$(string, len)

Returns STRING The rightmost segment of length *len* from *string.*

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow
- Memory Heap Exhausted

Arguments:

string — *byRef string AS STRING*
The target string which cannot be a const string.

length — *byVal length AS INTEGER*
The rightmost number of characters that are returned.

Note: **string** cannot be a string constant, e.g. "the cat", but must be a string variable and so if you must use a const string then first save it to a temp string variable and then pass it to the function

---

If 'length' is larger than the actual length of *string* then the entire string is returned.

Interactive Command: NO

```
//Example :: RIGHT$.sb (See in BL600CodeSnippets.zip)
DIM s$ : s$="Parse"
DIM new$ : new$ = RIGHT$(s$,4)
PRINT new$; "\n"
```

Expected Output:

```
arse
```

RIGHT$ is a core function.

STRLEN

FUNCTION

STRLEN returns the number of characters within a string.

STRLEN (string)

Returns INTEGER The number of characters within the string.

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

string *byRef string AS STRING*
The target string which cannot be a const string.

Interactive Command: NO

```
//Example :: StrLen$.sb (See in BL600CodeSnippets.zip)
DIM s$ : s$="HelloWorld"
PRINT "\n";s$;" is ";StrLen(S$);" bytes long"
```

Expected Output:

```
HelloWorld is 10 bytes long
```

STRLEN is a core function.

STRPOS

FUNCTION

STRPOS is used to determine the position of the first instance of a string within another string. If the string is not found within the target string a value of -1 is returned.

STRPOS (string1, string2, startpos)

Returns INTEGER Zero indexed position of ***string2*** within ***string1***.

>=0 If ***string2*** is found within ***string1*** and specifies the location where found
-1 If ***string2*** is not found within ***string1***

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

string1 *byRef string AS STRING*
The target string in which string2 is to be searched for.

string2 *byRef string AS STRING*
The string that is being searched for within string1. This may be a single character string.

startpos — *byVAL startpos AS INTEGER*
Where to start the position search.

Note: STRPOS does a case sensitive search.

Note: **string1** and **string2** cannot be a string constant, e.g. “the cat”, but must be a string variable and so if you must use a const string then first save it to a temp string variable and then pass it to the function

Interactive Command: NO

```
//Example :: StrPos.sb (See in BL600CodeSnippets.zip)
DIM s1$,s2$
s1$="Are you there"
s2$="there"
PRINT "\nIn '";S1$;"' the word '";S2$;"' occurs at position ";StrPos(S1$,S2$,0)
```

Expected Output:

```
In 'Are you there' the word 'there' occurs at position 8
```

STRPOS is a core function.

STRSETCHR

FUNCTION

STRSETCHR allows a single character within a string to be replaced by a specified value. STRSETCHR can also be used to append characters to an existing string by filling it up to a defined index.

If the nIndex is larger than the existing string then it is extended.

The use of STRSETCHR and STRGETCHR, in conjunction with a string variable allows an array of bytes to be created and manipulated.

STRSETCHR (string, nChr, nIndex)

Returns INTEGER Represents command execution status.

0 If the block is successfully updated
-1 If *nChr* is greater than 255 or less than 0
-2 If the string length cannot be extended to accommodate *nIndex*
-3 If the resultant string is longer than allowed.

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow
- Memory Heap Exhausted

Arguments:

string *byRef string AS STRING*
The target string.

nChr *byVal nCHr AS INTEGER*
The character that will overwrite the existing characters. *nChr* must be within the range 0 and 255.

nindex *byVal nIndex AS INTEGER*
The position in the string of the character that will be overwritten, referenced to a zero index.

Note: **string** cannot be a string constant, e.g. "the cat", but must be a string variable and so if you must use a const string then first save it to a temp string variable and then pass it to the function

---

Interactive Command: NO

```
//Example :: StrSetChar.sb (See in BL600CodeSnippets.zip)
DIM s$ : s$="Hello"
PRINT StrSetChr(s$,64,0)        //64 is the ASCII decimal code for the char '@'
PRINT StrSetChr(s$,64,8)        //s$ will be extended
PRINT "\n";s$
```

Expected Output:

```
000
@ello@@@@
```

STRSETCHR is a core function.

STRGETCHR

FUNCTION

STRGETCHR is used to return the single character at position nIndex within an existing string.

STRGETCHR (string, nIndex)

Returns INTEGER The ASCII value of the character at position *nIndex* within *string*, where *nIndex* is zero based. If *nIndex* is greater than the number of characters in the string or <=0 then an error value of -1 is returned.

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

string *byRef string AS STRING*
The string from which the character is to be extracted.

nindex *byVal nIndex AS INTEGER*
The position of the character within the string (zero based – see example).

Note: **string** cannot be a string constant, e.g. "the cat", but must be a string variable and so if you must use a const string then first save it to a temp string variable and then pass it to the function

Interactive Command: NO

```
//Example :: StrGetChar.sb (See in BL600CodeSnippets.zip)
DIM s$ : s$="Hello"
PRINT s$;"\n"
PRINT StrGetChr(s$,0), "-> ASCII value for 'H' \n"
PRINT StrGetChr(s$,1), "-> ASCII value for'e' \n"
PRINT StrGetChr(s$,-100), "-> error \n"
PRINT StrGetChr(s$,6), "-> error \n"
```

Expected Output:

```
Hello
72      -> ASCII value for 'H'
101     -> ASCII value for'e'
-1      -> error
-1      -> error
```

STRGETCHR is a core function.

STRSETBLOCK

FUNCTION

STRSETBLOCK allows a specified number of characters within a string to be filled or overwritten with a single character. The fill character, starting position and the length of the block are specified.

STRSETBLOCK (string, nChr, nIndex, nBlocklen)

Function

Returns INTEGER Represents command execution status.

0 If the block is successfully updated
-1 If nChr is greater than 255
-2 If the string length cannot be extended to accommodate *nBlocklen*
-3 if the resultant string will be longer than allowed
-4 If *nChr* is greater than 255 or less than 0
-5 if the nBlockLen value is negative

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

string *byRef string AS STRING*
The target string to be modified

nChr *byVal nChr AS INTEGER*
The character that will overwrite the existing characters.
nChr must be within the range 0 – 255

nindex *byVal nIndex AS INTEGER*
The starting point for the filling block, referenced to a zero index.

nBlocklen *byVal nBlocklen AS INTEGER*
The number of characters to be overwritten

Note: **string** cannot be a string constant, e.g. "the cat", but must be a string variable and so if you must use a const string then first save it to a temp string variable and then pass it to the function

Interactive Command: NO

```
//Example :: StrSetBlock.sb (See in BL600CodeSnippets.zip)
DIM s$ : s$="HelloWorld"
PRINT s$;"\n"
PRINT StrSetBlock(s$,64,4,2) : PRINT "\n";s$;"\n"
PRINT StrSetBlock(s$,300,4,200) : PRINT "\n";s$
```

Expected Output:

```
HelloWorld
0
Hell@@orld
-4
Hell@@orld
```

STRSETBLOCK is a core function.

STRFILL

FUNCTION

STRFILL is used to erase a string and then fill it with a number of identical characters.

STRFILL (string, nChr, nCount)

Returns INTEGER Represents command execution status.

0 If successful
-1 If *nChr* is greater than 255 or less than 0
-2 If the string length cannot be extended due to lack of memory
-3 If the resultant string is longer than allowed or *nCount* is <0.

STRING

string contains the modified string

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow
- Memory Heap Exhausted

Arguments:

string *byRef string AS STRING*

	The target string to be filled
nChr	*byVal nChr AS INTEGER* ASCII value of the character to be inserted. The value of nChr should be between 0 and 255 inclusive.
nCount	*byVal nCount AS INTEGER* The number of occurrences of ***nChr*** to be added.

The total number of characters in the resulting string must be less than the maximum allowable string length for that platform.

Note: **string** cannot be a string constant, e.g. "the cat", but must be a string variable and so if you must use a const string then first save it to a temp string variable and then pass it to the function

Interactive Command: NO

```
//Example :: StrFill.sb (See in BL600CodeSnippets.zip)
DIM s$ : s$="hello"
PRINT s$;"\n"
PRINT StrFill(s$,64,7);"\n"
PRINT s$;"\n"
PRINT StrFill(s$,-23,7)
```

Expected Output:

```
hello
7
@@@@@@@
-1
```

STRFILL is a core function.

STRSHIFTLEFT

SUBROUTINE

STRSHIFTLEFT shifts the characters of a string to the left by a specified number of characters and drops the leftmost characters. It is a useful subroutine to have when managing a stream of incoming data, as for example, a UART, I2C or SPI and a string variable is used as a cache and the oldest N characters need to be dropped.

STRSHIFTLEFT (string, numChars)

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

string	*byRef string AS STRING* The string to be shifted left.

numChrs — *byVal numChrs AS INTEGER*
The number of characters that the string is shifted to the left.
If *numChrs* is greater than the length of the string, then the returned string will be empty.

Note: **string** cannot be a string constant, e.g. "the cat", but must be a string variable and so if you must use a const string then first save it to a temp string variable and then pass it to the function

Interactive Command: NO

```
//Example :: StrShiftLeft.sb (See in BL600CodeSnippets.zip)
DIM s$ : s$="123456789"
PRINT s$;"\n"
StrShiftLeft(s$,4)        //drop leftmost 4 characters
PRINT s$
```

Expected Output:

```
123456789
56789
```

STRSHIFTLEFT is a core function.

STRCMP

FUNCTION

Compares two string variables.

STRCMP(string1, string2)

Returns INTEGER A value indicating the comparison result:

0 – if *string1* exactly matches *string2* (the comparison is case sensitive)

1 – if the ASCII value of *string1* is greater than *string2*

-1 - if the ASCII value of *string1* is less than *string2*

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

string1 — *byRef string1 AS STRING*
The first string to be compared.

string2 — *byRef string2 AS STRING*
The second string to be compared.

Note: **string1** and **string2** cannot be a string constant, e.g. "the cat", but must be a string variable and so if you must use a const string then first save it to a temp string variable and then pass it to the function

---

Interactive Command: NO

```
//Example :: StrCmp.sb (See in BL600CodeSnippets.zip)
DIM s1$,s2$
s1$="hello"
s2$="world"
PRINT StrCmp(s1$,s2$);"\n"
PRINT StrCmp(s2$,s1$);"\n"
PRINT StrCmp(s1$,s1$);"\n"
```

Expected Output:

```
-1
1
0
```

STRCMP is a core function.

STRHEXIZE$

FUNCTION

This function is used to convert a string variable into a string which contains all the bytes in the input string converted to 2 hex characters. It will therefore result in a string which is exactly double the length of the original string.

STRHEXIZE$ (string)

Returns STRING A printable version of ***string*** which contains only hexadecimal characters and exactly double the length of the input string.

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow
- Memory Heap Exhausted

Arguments:

String *byRef string AS STRING*
The string to be converted into hex characters.

Interactive Command: NO

Note: **string** cannot be a string constant, e.g. "the cat", but must be a string variable and so if you must use a const string then first save it to a temp string variable and then pass it to the function

---

Associated Commands: STRHEX2BIN

```
//Example :: StrHexize$.sb (See in BL600CodeSnippets.zip)
DIM s$,t$
s$="Laird"
PRINT s$;"\n"
t$=StrHexize$(s$)
PRINT StrLen(s$);"\n"
PRINT t$;"\n"
PRINT StrLen(t$);"\n"
```

Expected Output:

```
Laird
5
4C61697264
10
```

STRHEXIZE$ is a core function.

STRDEHEXIZE$

STRDEHEXISE$ is used to convert a string consisting of hex digits to a binary form. The conversion stops at the first non hex digit character encountered.

STRDEHEXIZE$ (string)

Function

Returns STRING A dehexed version of **string**

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

string *byRef string AS STRING*

The string to be converted in-situ.

If a parsing error occurs, a nonfatal error is generated which must be handled or the application aborts.

Note: **string** cannot be a string constant, e.g. "the cat", but must be a string variable and so if you must use a const string then first save it to a temp string variable and then pass it to the function

Interactive Command: NO

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

DIM s$ : s$="40414243"

PRINT "\nHex data: ";s$
PRINT "\nDehexized: "; StrDehexize$(s$)

//Will stop at first non hex digit 'h'
s$="4041hello4243"
PRINT "\n";s$;" Dehexized: "; StrDehexize$(s$)
```

Expected Output:

```
Hex data: 40414243
Dehexized: @ABC
4041hello4243 Dehexized: @A
```

STRDEHEXIZE$ is a core function.

STRHEX2BIN

This function is used to convert up to 2 hexadecimal characters at an offset in the input string into an integer value in the range 0 to 255.

STRHEX2BIN (string,offset)

Function

Returns INTEGER A value in the range 0 to 255 which corresponds to the (up to) 2 hex characters at the specified offset in the input string.

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

string — *byRef string AS STRING*
The string to be converted into hex characters.

offset — *byVal offset AS INTEGER*
This is the offset from where up to 2 hex characters will be converted into a binary number.

Interactive Command: NO

Note: **string** cannot be a string constant, e.g. “the cat”, but must be a string variable and so if you must use a const string then first save it to a temp string variable and then pass it to the function

Associated Commands: STRHEXIZE

```
//Example :: StrHex2Bin.sb (See in BL600CodeSnippets.zip)
DIM s$
s$="0102030405"
PRINT StrHex2Bin(s$,4);"\n"
s$="4C61697264"
PRINT StrHex2Bin(s$,2);"\n"
```

Expected Output:

```
3
97
```

STRHEX2BIN is a core function.

STRESCAPE$

FUNCTION

STRESCAPE$ is used to convert a string variable into a string which contains only printable characters using a 2 or 3 byte sequence of escape characters using the \NN format.

STRESCAPE$ (string)

Returns STRING A printable version of ***string*** which means at best the returned string is of the same length and at worst not more than three times the length of the input string.

The following input characters are escaped as follows:

carriage return	\r
linefeed	\n
horizontal tab	\t
\	\\
"	\"
chr < ' '	\HH
chr >= 0x7F	\HH

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow
- Memory Heap Exhausted

Arguments:

string *byRef string AS STRING*
The string to be converted.

If a parsing error is encountered a nonfatal error will be generated which needs to be handled otherwise the script will abort.

Interactive Command: NO

Note: **string** cannot be a string constant, e.g. "the cat", but must be a string variable and so if you must use a const string then first save it to a temp string variable and then pass it to the function

Associated Commands: STRDEESCAPE

```
//Example :: StrEscape$.sb (See in BL600CodeSnippets.zip)
DIM s$,t$
s$="Hello\00world"
t$=StrEscape$(s$)
PRINT StrLen(s$);"\n" : PRINT StrLen(t$);"\n"
```

Expected Output:

```
11
13
```

STRESCAPE$ is a core function.

STRDEESCAPE

SUBROUTINE

STRDEESCAPE is used to convert an escaped string variable in the same memory space that the string exists in. Given all 3 byte escape sequences are reduced to a single byte, the result is never longer than the original.

STRDEESCAPE (string)

Returns None

string now contains de-escaped characters converted as follows:

\r	carriage return
\n	linefeed
\t	horizontal tab
\\	\
""	"
\HH	ascii byte HH

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow
- String De-Escape Error (E.g chrs after the \ are not recognized)

Arguments:

string *byRef string AS STRING*
The string to be converted in-situ.

If a parsing error occurs, a nonfatal error is generated which must be handled or the application will abort.

Note: **string** cannot be a string constant, e.g. "the cat", but must be a string variable and so if you must use a const string then first save it to a temp string variable and then pass it to the function

Interactive Command: NO

```
//Example :: StrDeescape.sb (See in BL600CodeSnippets.zip)
DIM s$,t$
s$="Hello\5C40world"
PRINT s$;"\n"; StrLen(s$);"\n"
StrDeescape(s$)
PRINT s$;"\n"; StrLen(s$);"\n"
```

Expected Output:

```
Hello\40world
13
Hello@world
11
```

STRDEESCAPE is a core function.

STRVALDEC

FUNCTION

STRVALDEC converts a string of decimal numbers into the corresponding INTEGER signed value. All leading whitespaces are ignored and then conversion stops at the first non-digit character

STRVALDEC (string)

Function

Returns INTEGER Represents the decimal value that was contained within string.

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

string *byRef string AS STRING*
The target string

If STRVALDEC encounters a non-numeric character within the string it will return the value of the digits encountered before the non-decimal character.

Any leading whitespace within the string is ignored.

Note: **string** cannot be a string constant, e.g. "the cat", but must be a string variable and so if you must use a const string then first save it to a temp string variable and then pass it to the function

Interactive Command: NO

```
//Example :: StrValDec.sb (See in BL600CodeSnippets.zip)
DIM s$
s$="  1234"
```

```
PRINT "\n";StrValDec(s$)
s$=" -1234"
PRINT "\n";StrValDec(s$)
s$=" +1234"
PRINT "\n";StrValDec(s$)
s$="  2345hello"
PRINT "\n";StrValDec(s$)
s$="  hello"
PRINT "\n";StrValDec(s$)
```

Expected Output:

```
1234
-1234
1234
2345
0
```

STRVALDEC is a core function.

STRSPLITLEFT$

FUNCTION

STRSPLITLEFT$ returns a string which consists of the leftmost n characters of a string object and then drops those characters from the input string.

STRSPLITLEFT$ (string, length)

Returns STRING The leftmost 'length' characters are returned, and then those characters are dropped from the argument list.

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow
- Memory Heap Exhausted

Arguments:

string *byRef string AS STRING*
The target string which cannot be a const string.

length *byVal length AS INTEGER*
The number of leftmost characters that are returned before being dropped from the target string.

Note: **string** cannot be a string constant, e.g. "the cat", but must be a string variable and so if you must use a const string then first save it to a temp string variable and then pass it to the function

Interactive Command: NO

```
//Example :: StrSplitLeft$.sb (See in BL600CodeSnippets.zip)
DIM origStr$
origStr$ = "12345678"
```

```
PRINT StrSplitLeft$ (origStr$, 3);"\n"
PRINT origStr$
```

Expected Output:

```
123
45678
```

STRSPLITLEFT$ is a core function.

STRSUM

This function identifies the substring starting from a specified offset and specified length and then does an arithmetic sum of all the unsigned bytes in that substring and then finally adds the signed initial value supplied.

For example, if the string is "\01\02\03\04\05" and offset is 1 and length is 2 and initial value is 1000, then the output will be 1000+2+3=1005.

STRSUM (string, nIndex, nBytes, initVal)

Function

Returns INTEGER The result of the arithmetic sum operation over the bytes in the substring. If nIndex or nBytes are negative, then the initVal will be returned.

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

string *byRef string AS STRING*
String that contains the unsigned bytes which need to be arithmetically added

nIndex *byVal nIndex AS INTEGER*
Index of first byte into the string

nBytes *ByVal nBytes AS INTEGER*
Number of bytes to process

initVal *ByVal initVal AS INTEGER*
Initial value of the sum

Note: **string** cannot be a string constant, e.g. "the cat", but must be a string variable and so if you must use a const string then first save it to a temp string variable and then pass it to the function

---

Interactive Command: NO

```
//Example :: StrSum.sb (See in BL600CodeSnippets.zip)
DIM s$
s$="0aA%<"
PRINT StrSum(s$,0,5,0);"\n"      //48+97+65+37+60+0
PRINT StrSum(s$,0,5,10);"\n"     //48+97+65+37+60+10
PRINT StrSum(s$,4,1,100);"\n"    //60+100
```

Expected Output:

```
307
317
160
```

STRSUM is a core function.

STRXOR

This function identifies the substring starting from a specified offset and specified length and then does an arithmetic exclusive-or (XOR) of all the unsigned bytes in that substring and then finally XORs the signed initial value supplied.

For example, if the string is "\01\02\03\04\05" and offset is 1 and length is 2 and initial value is 1000, then the output will be 1000 ^ 2 ^ 3=1001.

STRXOR (string, nIndex, nBytes, initVal)

Function

Returns INTEGER The result of the xor operation over the bytes in the substring. If nIndex or nBytes are negative, then the initVal will be returned.

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

string — *byRef string AS STRING*
String that contains the unsigned bytes which need to be XOR'd

nIndex — *byVal nIndex AS INTEGER*
Index of first byte into the string

nBytes — *ByVal nBytes AS INTEGER*
Number of bytes to process

initVal — *ByVal initVal AS INTEGER*
Initial value of the XOR

Note: **string** cannot be a string constant, e.g. "the cat", but must be a string variable and so if you must use a const string then first save it to a temp string variable and then pass it to the function

Interactive Command: NO

```
//Example :: StrXOR.sb (See in BL600CodeSnippets.zip)
DIM number$
number$="01234"
PRINT StrXOR(number$,0,5,0)          //XOR: 48,49,50,51,52,0
PRINT StrXOR(number$,0,5,10)         //XOR: 48,49,50,51,52,10
PRINT StrXOR(number$,0,5,1000)       //XOR: 48,49,50,51,52,1000
```

Expected Output:

```
52
62
988
```

STRXOR is a core function.

EXTRACTSTRTOKEN

This function takes a sentence in the first parameter and extracts the leftmost string token from it and passes it back in the second paremeter. The token is removed from the sentence and is not post processed in any way. The function will return the length of the string in the token. This means if 0 is returned then there are no more tokens in the sentence.

It makes it easy to create custom protocol for commands send by a host over the uart for your application.

For example, if the sentence is "**My name is BL600, from Laird**" then the first call of this function will return "**My**" and the sentence will be adjusted to "**name is BL600, from Laird**". Note that "**BL600,**" will result in "**BL600**" and then "**,**"

The parser logic is exactly the same as when in the command mode. If you are not sure which alphabet character is a token in its own right, then the quickest way to get an answer is to actually try it.

NOTE: any text after either ' or // will be taken as a comment just like the behaviour in the command mode.

EXTRACTSTRTOKEN (sentence$,token$)

Function

Returns INTEGER
The length of the extracted token. Will be 0 of there are no more tokens to extract.

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

sentence$* *byRef sentence$ AS STRING
String that contains the sentence containing the tokens to be extracted

token$* *byRef token$ AS STRING
The leftmost token from the sentence and will have been removed from the sentence.

Note: **string** cannot be a string constant, e.g. "the cat", but must be a string variable and so if you must use a const string then first save it to a temp string variable and then pass it to the function

---

Interactive Command: NO

```
//Example :: ExtractStrToken.sb (See in BL600CodeSnippets.zip)
DIM sentence$, token$, tknlen
sentence$="My name is BL600, from     Laird"
PRINT "\nSentence is :";sentence$
DO
  tknlen = ExtractStrToken(sentence$,token$)
  PRINT "\nToken (len ";tknlen;") = :";token$
UNTIL tknlen==0
```

Expected Output:

ExtractStrToken is a core function.

EXTRACTINTTOKEN

This function takes a sentence in the first parameter and extracts the leftmost set of tokens that make an integer number (hex or binary or octal or decimal) from it and passes it back in the second paremeter. The tokens are removed from the sentence. The function will return the number of characters extracted from the left side f the sentence. This means if 0 is returned then there are no more tokens in the sentence.

For example, if the sentence is "**0x100 is a hex,value**" then the first call of this function will return 256 in the second parameter and the sentence will be adjusted to "**is a hex value**". Note that "**hex,value,**" whill result in "**hex**" then "**,**" and then "**value**"

The parser logic is exactly the same as when in the command mode. If you are not sure which alphabet character is a token in its own right, then the quickest way to get an answer is to actually try it.

NOTE: any text after either ' or // will be taken as a comment just like the behaviour in the command mode.

EXTRACTINTTOKEN (sentence$,intValue)

Function

Returns INTEGER
The length of the extracted token. Will be 0 of there are no more tokens to extract.

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

sentence$* — *byRef sentence$ AS STRING
String that contains the sentence containing the tokens to be extracted

intValue — *byRef intValue AS STRING*
The leftmost set of tokens constituting a legal integer value is extracted from the sentence and will be removed from the sentence.

Note: **string** cannot be a string constant, e.g. "the cat", but must be a string variable and so if you must use a const string then first save it to a temp string variable and then pass it to the function

Interactive Command: NO

```
//Example :: ExtractIntToken.sb (See in BL600CodeSnippets.zip)
DIM sentence$
DIM intValue, bytes
DIM token$, tknlen
sentence$="0x100 is a hex,value"
PRINT "\nSentence is :";sentence$
bytes = ExtractIntToken(sentence$,intValue)
PRINT "\nintValue (bytes ";bytes;") = :";intValue
DO
  tknlen = ExtractStrToken(sentence$,token$)
  PRINT "\nToken (len ";tknlen;") = :";token$
UNTIL tknlen==0
```

Expected Output:

ExtractIntToken is a core function.

Table Routines

Tables provide associative array (or in other words lookup type) functionality within *smart* BASIC programs. They are typically used to allow lookup features to be implemented efficiently so that, for example, parsers can be implemented.

Tables are one dimensional string variables, which are configured by using the TABLEINIT command.

Tables should not be confused with Arrays. Tables provide the ability to perform pattern matching in a highly optimised manner. As a general rule, use tables where you want to perform efficient pattern matching and arrays where you want to automate setup strings or send data using looping variables.

TABLEINIT

FUNCTION

TABLEINIT initialises a string variable so that it can be used for storage of multiple TLV tokens, allowing a lookup table to be created.

TLV = Tag, Length, Value

TABLEINIT (string)

Returns INTEGER Indicates success of command:

0 Successful initialisation
<>0 Failure

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

string *byRef string AS STRING*

String variable to be used for the Table. Since it is byRef the compiler will not allow a constant string to be passed as an argument. On entry the string can be non-empty, on exit the string will be empty.

Interactive Command: NO

Note: **string** cannot be a string constant, e.g. “the cat”, but must be a string variable and so if you must use a const string then first save it to a temp string variable and then pass it to the function

Associated Commands: **TABLEADD, TABLELOOKUP**

```
//Example :: TableInit.sb (See in BL600CodeSnippets.zip)
DIM t$ :t$="Hello"
PRINT "\n";"[";t$;"]"
PRINT "\n";TableInit(t$)
PRINT "\n";"[";t$;"]"     //String now blank after being initialised as a table
```

Expected Output:

```
[Hello]
0
[]
```

TABLEINIT is a core function.

TABLEADD

FUNCTION

TABLEADD adds the token specified to the lookup table in the string variable and associates the index specified with it. There is no validation to check if nIndex has been duplicated as it is entirely valid that more than one token generate the same ID value

TABLEADD (string, strtok, nID)

Returns INTEGER Indicates success of command:

- 0 Signifies that the token was successfully added
- 1 Indicates an error if *nID* > 255 or < 0
- 2 Indicates no memory is available to store token
- 3 Indicates that the token is too large
- 4 Indicates the token is empty

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

string — *byRef string AS STRING*
A string variable that has been initialised as a table using TABLEINIT.

strtok — *byVal strtok AS STRING*
The string token to be added to the table.

nID — *byVal nID AS INTEGER*
The identifier number that is associated with the token and should be in the range 0 to 255.

Note: **string** cannot be a string constant, e.g. "the cat", but must be a string variable and so if you must use a const string then first save it to a temp string variable and then pass it to the function

---

Interactive Command: NO

Associated Commands: **TABLEINIT, TABLELOOKUP**

```
//Example :: TableAdd.sb (See in BL600CodeSnippets.zip)
DIM t$ : PRINT TableInit(t$);"\n"
PRINT TableAdd(t$,"Hello",1);"\n"
PRINT TableAdd(t$,"everyone",2);"\n"
PRINT TableAdd(t$,"to",300);"\n"
```

```
PRINT TableAdd(t$,"",3);"\n"
PRINT t$
//Tokens are stored in TLV format: \Tag\LengthValue
```

Expected Output:

TABLEADD is a core function.

TABLELOOKUP

FUNCTION

TABLELOOKUP searches for the specified token within an existing lookup table which was created using TABLEINIT and multiple TABLEADDs and returns the ID value associated with it.

It is especially useful for creating a parser, for example, to create an AT style protocol over a uart interface.

TABLELOOKUP (string, strtok)

Returns INTEGER Indicates success of command:

- >=0 signifies that the token was successfully found and the value is the ID
- -1 if the token is not found within the table
- -2 if the specified table is invalid
- -3 if the token is empty or > 255 characters

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

string — *byRef string AS STRING*
The lookup table that is being searched

strtok — *byRef strtok AS STRING*
The token whose position is being found

Note: **string** cannot be a string constant, e.g. "the cat", but must be a string variable and so if you must use a const string then first save it to a temp string variable and then pass it to the function

Interactive Command: NO

Associated Commands: **TABLEINIT, TABLEADD**

```
//Example :: TableLookup.sb (See in BL600CodeSnippets.zip)
DIM t$
PRINT TableInit(t$);"\n\n"

PRINT TableAdd(t$,"Hello",1);"\n"
PRINT TableAdd(t$,"world",2);"\n"
PRINT TableAdd(t$,"to",3);"\n"
PRINT TableAdd(t$,"you",4);"\n\n"

PRINT TableLookup(t$,"to");"\n"
PRINT TableLookup(t$,"Hello");"\n"
PRINT TableLookup(t$,"you");"\n"
```

Expected Output:

TABLELOOKUP is a core function.

Miscellaneous Routines

This section describes all miscellaneous functions and subroutines

RESET

SUBROUTINE

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

RESET (nType)

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

nType	byVal *nType* AS INTEGER. This is for future use. Set to 0.

Interactive Command: NO

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

Expected Output:

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

RESET is a core subroutine.

ERASEFILESYSTEM

FUNCTION

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

Given that SIO7 is high, after erasing the file system, the module will reset and reboot into command mode with the virtual serial port service enabled and the module will advertise for a few seconds. See the virtual serial port service section for more details.

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

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

ERASEFILESYSTEM (nArg)

Returns INTEGER Indicates success of command:

0 Successful erasure, but you will not see it as the module will reboot
<>0 Failure

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

nArg *byVal nArg AS INTEGER*

This is for future use and MUST always be set to 1. Any other value will result in a failure.

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

```
//Input SIO7 is low
rc = EraseFileSystem(1)
IF rc!=0 THEN
  PRINT "\nFailed to erase file system because SIO7 is low"
ENDIF
```

Expected Output:

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

ERASEFILESYSTEM is an extension function.

Random Number Generation Routines

Random numbers are either generated using pseudo random number generator algorithms or using thermal noise or equivalent in hardware. The routines listed in this section provide the developer with the capability of generating random numbers.

The Interactive Mode command "AT I 1001" or at runtime SYSINFO(1001) will return 1 if the system generates random numbers using hardware noise or 0 if a pseudo random number generator.

RAND

FUNCTION

The RAND function returns a random 32 bit integer. Use the command 'AT I 1001' or from within an application the function SYSINFO(1001), to determine whether the random number is pseudo random or generated in hardware via a thermal noise generator. If 1001 returns 0 then it is pseudo random and 1 if generated using hardware.

RAND ()

Returns INTEGER A 32 bit integer.

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments: None

Depending on the platform, the RAND function can be seeded using the RANDSEED function to seed the pseudo random number generator. If used, RANDSEED must be called before using RAND. If the platform has a hardware Random Number Generator, then RANDSEED has no effect.

Interactive Command: NO

Associated Commands: RANDSEED

```
//Example :: RAND.sb (See in BL600CodeSnippets.zip)
PRINT "\nRandom number is ";RAND()
```

Expected Output:

```
Random number is -2088208507
```

RAND is a core language function.

RANDEX

FUNCTION

The RANDEX function returns a random 32 bit **positive** integer in the range 0 to X where X is the input argument. Use the command 'AT I 1001' or from within an application the function SYSINFO(1001) to determine whether the random number is pseudo random or generated in hardware via a thermal noise generator. If 1001 returns 0 then it is pseudo random and 1 if generated using hardware.

RANDEX (maxval)

Returns INTEGER A 32 bit integer.

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

maxval *byVal maxval AS INTEGER*

The return value will not exceed the absolute value of this variable

Depending on the platform, the RANDEX function can be seeded using the RANDSEED function to seed the pseudo random number generator. If used, RANDSEED must be called before using RANDEX. If the platform has a hardware Random Number Generator, then RANDSEED has no effect.

Interactive Command: NO

Associated Commands: RANDSEED

```
//Example :: RANDEX.sb (See in BL600CodeSnippets.zip)
DIM x : x=500
PRINT "\nRandom number between 0 and ";x;" is ";RANDEX(x)
```

Expected Output:

```
Random number between 0 and 500 is 193
```

RAND is a core language function.

RANDSEED

SUBROUTINE

On platforms without a hardware random number generator, the RANDSEED function sets the starting point for generating a series of pseudo random integers. To reinitialize the generator, use 1 as the seed argument. Any other value for seed sets the generator to a random starting point. RAND retrieves the pseudo random numbers that are generated.

It has no effect on platforms with a hardware random number generator.

RANDSEED (seed)

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

Seed *byVal seed AS INTEGER*

The starting seed value for the random number generator function RAND.

Interactive Command: No

Associated Commands: RAND

```
RandSeed(1234)
```

Note: Since the BL600 contains a hardware random number generator, this subroutine has no effect.

RANDSEED is a core language subroutine.

Timer Routines

In keeping with the event driven paradigm of *smart* BASIC, the timer subsystem enables *smart* BASIC applications to be written which allow future events to be generated based on timeouts. To make use of this feature up to N timers, where N is platform dependent, are made available and that many event handlers can be written and then enabled using the ONEVENT statement so that those handlers are automatically invoked. The ONEVENT statement is described in detail elsewhere in this manual.

Briefly the usage is, select a timer, register a handler for it, and start it with a timeout value and a flag to specify whether it is recurring or single shot. Then when the timeout occurs AND when the application is processing a WAITEVENT statement, the handler will be automatically called.

It is important to understand the significance of the WAITEVENT statement. In a nutshell, a timer handler callback will NOT happen if the runtime engine does not encounter a WAITEVENT statement. Events are synchronous not asynchronous like say interrupts.

All this is illustrated in the sample code fragment below where timer 0 is started so that it will recur automatically every 500 milliseconds and timer 1 is a single shot 1000ms later.

Note, as explained in the WAITEVENT section of this manual, if a handler function returns a non-zero value then the WAITEVENT statement is reprocessed, otherwise the *smart* BASIC runtime engine will proceed to process the next statement **after** the WAITEVENT statement – not after the handlers ENDFUNC or EXITFUNC statement. This means that if the WAITEVENT is the very last statement in an application and a timer handler returns a 0 value, then the application will exit the module from Run Mode into Interactive Mode which will be disastrous for unattended operation.

Timer Events

EVTMRn Where n=0 to N, where N is platform dependent, it is generated when timer n expires. The number of timers (that is, N+1) is returned by the command AT I 2003 or at runtime by SYSINFO(2003)

```
//Example :: EVTMRn.sb (See in BL600CodeSnippets.zip)
FUNCTION HandlerTimer0()
  PRINT "\nTimer 0 has expired"
ENDFUNC 1                           //remain blocked in WAITEVENT

FUNCTION HandlerTimer1()
  PRINT "\nTimer 1 has expired"
ENDFUNC 0                           //exit from WAITEVENT

ONEVENT EVTMR0 CALL HandlerTimer0
ONEVENT EVTMR1 CALL HandlerTimer1

TimerStart(0,500,1)                 //start a 500 millisecond recurring timer
PRINT "\nWaiting for Timer 0"
TimerStart(1,1000,0)                //start a 1000 millisecond timer
PRINT "\nWaiting for Timer 1"

WAITEVENT
PRINT "\nGot here because TIMER 1 expired and handler returned 0"
```

Expected Output:

TimerStart

This subroutine starts one of the built-in timers.

The command AT I 2003 will return the number of timers and AT I 2002 will return the resolution of the timer in microseconds.

When the timer expires, an appropriate event is generated, which can be acted upon by a handler registered using the ONEVENT command.

TIMERSTART (number,interval_ms,recurring)

SUBROUTINE:

Arguments:

number *byVal* *number AS INTEGER*
The number of the timer. 0 to N where N can be determined by submitting the command AT I 2003 or at runtime returned via SYSINFO(2003).

If the value is not valid, then a runtime error will be thrown with code INVALID_TIMER.

Interval_ms — *byVal interval AS INTEGER*
A valid time in milliseconds, between 1 and 2,147,493,647 (24.8 days). Note although the time is specified in milliseconds, the resolution of the hardware timer may have more granularity than that. Submit the command AT I 2002 or at runtime SYSINFO(2002) to determine the actual granularity in microseconds.

If longer timeouts are required, start one of the timers with 1000 and make it repeating and then implement the longer timeout using *smart* BASIC code.

If the interval is negative or > 2,147,493,647 then a runtime error will be thrown with code INVALID_INTERVAL

If the ***recurring*** argument is set to non-zero, then the minimum value of the interval is 10ms

recurring — *byVal recurring AS INTEGER*
Set to 0 for a once-only timer, or non-0 for a recurring timer.

When the timer expires, it will set the corresponding EVTMRn event. That is, timer number 0 sets EVTMR0, timer number 3 sets EVTMR3. The ONEVENT statement should be used to register handlers that will capture and process these events.

If the timer is already running, calling TIMERSTART will reset it to count down from the new value, which may be greater or smaller than the remaining time.

If either ***number*** or ***interval*** is invalid an Error is thrown.

Interactive Command: No

Related Commands: ONEVENT, TIMERCANCEL

```
//Example :: EVTMRn.sb (See in BL600CodeSnippets.zip)
SUB HandlerOnErr()
  PRINT "Timer Error: ";GetLastError()
ENDSUB

FUNCTION HandlerTimer1()
  PRINT "\nTimer 1 has expired"
ENDFUNC 1                              //remain blocked in WAITEVENT

FUNCTION HandlerTimer2()
  PRINT "\nTimer 2 has expired"
ENDFUNC 0                              //exit from WAITEVENT

ONERROR NEXT HandlerOnErr

ONEVENT EVTMR1 CALL HandlerTimer1
ONEVENT EVTMR2 CALL HandlerTimer2

TimerStart(0,-500,1)                   //start a -500 millisecond recurring  timer
PRINT "\nStarted Timer 0 with invalid inerval"

TimerStart(1,500,1)                    //start a 500 millisecond recurring timer
PRINT "\nWaiting for Timer 1"

TimerStart(2,1000,0)                   //start a 1000 millisecond timer
```

```
PRINT "\nWaiting for Timer 2"

WAITEVENT
PRINT "\nGot here because TIMER 2 expired and Handler returned 0"
```

Expected Output:

TIMERSTART is a core subroutine.

TimerRunning

FUNCTION

This function determines if a timer identified by an index number is still running. The command AT I 2003 will return the valid range of Timer index numbers. It returns 0 to signify that the timer is not running and a non-zero value to signify it is still running and the value is the number of milliseconds left for it to expire.

TIMERRUNNING (number)

Function

Returns: 0 if the timer has expired, otherwise the time in milliseconds left to expire.

Arguments:

number *byVal number AS INTEGER*
The number of the timer. 0 to N where N can be determined by submitting the command AT I 2003 or at runtime returned via SYSINFO(2003).

If the value is not valid, then a runtime error will be thrown with code INVALID_TIMER.

Interactive Command: No

Related Commands: ONEVENT, TIMERCANCEL

```
//Example :: TimerRunning.sb (See in BL600CodeSnippets.zip)
SUB HandlerOnErr()
  PRINT "Timer Error ";GetLastError()
ENDSUB

FUNCTION HandlerTimer0()
  PRINT "\nTimer 0 has expired"
  PRINT "\nTimer 1 has ";TimerRunning(1);" milliseconds to go"
ENDFUNC 1                                      //remain blocked in WAITEVENT

FUNCTION HandlerTimer1()
```

```
   PRINT "\nTimer 1 has expired"
ENDFUNC 0                               //exit from WAITEVENT

ONERROR NEXT HandlerOnErr

ONEVENT EVTMR0 CALL HandlerTimer0
ONEVENT EVTMR1 CALL HandlerTimer1

TIMERSTART(0,500,1)                     //start a 500 millisecond recurring timer
PRINT "\nWaiting for Timer 0"

TIMERSTART(1,2000,0)                    //start a 1000 millisecond timer
PRINT "\nWaiting for Timer 1"

WAITEVENT
```

Expected Output:

TIMERRUNNING is a core function

TimerCancel

SUBROUTINE

This subroutine stops one of the built-in timers so that it will not generate a timeout event.

TIMERCANCEL (number)

Arguments:

number — *byVal number AS INTEGER*
The number of the timer. 0 to N where N can be determined by submitting the command AT I 2003 or at runtime returned via SYSINFO(2003).

If the value is not valid, then a runtime error will be thrown with code INVALID_TIMER.

Interactive Command: NO

Related Commands: ONEVENT, TIMERCANCEL,TIMERRUNNING

```
//Example :: TimerCancel.sb (See in BL600CodeSnippets.zip)
DIM i,x
i=0 : x=1   //'x' is HandlerTimer0's return value
```

```
            //Will switch to 0 when timer0 has expired so that the application can
stop
 FUNCTION HandlerTimer0()
     PRINT "\nTimer 0 has expired, starting again"
     IF i==4 THEN
         PRINT "\nCancelling Timer 0"
         TimerCancel(0)
         PRINT "\nTimer 0 ran ";i+1;" times"
         x=0
     ENDIF
     i=i+1
 ENDFUNC x
 ONEVENT EVTMR0 CALL HandlerTimer0
 TimerStart(0,800,1)
 PRINT "\nWaiting for Timer 0. Should run 5 times"
 WAITEVENT
```

Expected Output:

TIMERCANCEL is a core subroutine.

GetTickCount

FUNCTION

There is a 31 bit free running counter that increments every 1 millisecond. The resolution of this counter in microseconds can be determined by submitting the command AT I 2004 or at runtime SYSINFO(2004) . This function returns that free running counter. It wraps to 0 when the counter reaches 0x7FFFFFFF.

GETTICKCOUNT ()

Returns: INTEGER A value in the range 0 to 0x7FFFFFFF (2,147,483,647) in units of milliseconds.

Arguments: None

Interactive Command: No

Related Commands: GETTICKSINCE

```
 //Example :: GetTickCount.sb (See in BL600CodeSnippets.zip)
 FUNCTION HandlerTimer0()
   PRINT "\n\nTimer 0 has expired"
 ENDFUNC 0

 PRINT "\nThe value on the counter is ";GetTickCount()

 ONEVENT EVTMR0 CALL HandlerTimer0
```

```
TimerStart(0,1000,0)
PRINT "\nWaiting for Timer 0"

WAITEVENT
PRINT "\nThe value on the counter is now ";GetTickCount();
```

Expected Output:

GETTICKCOUNT is a core subroutine.

GetTickSince

FUNCTION

This function returns the time elapsed since the 'startTick' variable was updated with the return value of GETTICKCOUNT(). It signifies the time in milliseconds. If 'startTick' is less than 0 which is a value that GETTICKCOUNT() will never return, then a 0 will be returned.

GETTICKSINCE (startTick)

Returns: INTEGER A value in the range 0 to 0x7FFFFFFF (2,147,483,647) in units of milliseconds.

startTickr *byVal startTick AS INTEGER*
This is a variable that was updated using the return value from GETTICKCOUNT() and it is used to calculate the time elapsed since that update.

Interactive Command: No

Related Commands: GETTICKCOUNT

```
//Example :: GetTickSince.sb (See in BL600CodeSnippets.zip)
DIM startTick, elapseMs, x
x=1
startTick = GetTickCount()

DO
    PRINT x;" x 2 = "
    x=x*2
    PRINT x;"\n"
UNTIL x==32768

elapseMs = GetTickSince(startTick)
PRINT "\n\nThe Do Until loop took ";elapseMS; " msec to process"
```

Expected Output:

GETTICKCOUNT is a core subroutine.

Circular Buffer Management Functions

It is a common requirement in applications that deal with communications to require circular buffers that can act as first-in, first-out queues or to create a stack that can store data in a push/pop manner.

This section describes functions that allow these to be created so that they can be expedited as fast as possible without the speed penalty inherited in any interpreted language. The basic entity that is managed is the INTEGER variable in smartBASIC. Hence be aware that for a buffer size of N, 4 times N is the memory that will be taken from the internal heap.

These buffers are referenced using handles provided at creation time.

CircBufCreate

FUNCTION

This function is used to create a circular buffer with a maximum capacity set by the caller. Most often it will be used as a first-in, first-out queue.

CIRCBUFCREATE (nItems, circHandle)

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

nItems — byVal *nItems* AS INTEGER
This specifies the maximum number of INTEGER values that can be stored in the buffer. If there isn't enough free memory in the heap, then this function will fail and return an appropriate result code.

circHandle **byRef circHandle AS INTEGER**
If the circular buffer is successfully created, then this variable will return a handle that should be used to interact with it.

Interactive Command: NO

```
//Example :: CircBufCreate.sb (See in BL600CodeSnippets.zip)
DIM circHandle, circHandle2, rc

rc = CircBufCreate(16,circHandle)
PRINT "\n";rc
IF rc!=0 THEN
  PRINT "\nThe circular buffer ";circHandle; "was not created"
ENDIF

rc = CircBufCreate(32000,circHandle2)
PRINT "\n\n";rc
IF rc!=0 THEN
  PRINT "\n---> The circular buffer 'circHandle2' was not created"
ENDIF
```

Expected Output:

```
0

20736
---> The circular buffer 'circHandle2' was not created
```

CIRCBUFCREATE is an extension function.

CircBufDestroy

SUBROUTINE

This function is used to destroy a circular buffer previously created using CircBufCreate.

CIRCBUFDESTROY (circHandle)

Arguments:

circHandle **byRef circHandle AS INTEGER**
A handle referencing the circular buffer that needs to be deleted. On exit an invalid handle value will be returned

Interactive Command: NO

```
//Example :: CircBufDestroy.sb (See in BL600CodeSnippets.zip)
DIM circHandle, circHandle2, rc

rc = CircBufCreate(16,circHandle)
PRINT "\n";rc
IF rc!=0 THEN
    PRINT "\nThe circular buffer ";circHandle; " was not created"
```

```
ENDIF

CircBufDestroy(circHandle)
PRINT "\nThe handle value is now ";circHandle; " so it has been destroyed"
```

Expected Output:

```
0
The handle value is now -1 so it has been destroyed
```

CIRCBUFDESTROY is an extension function.

CircBufWrite

FUNCTION

This function is used to write an integer at the head end of the circular buffer and if there is no space available to write, then it will return with a failure resultcode and NOT write the value.

CIRCBUFWRITE (circHandle, nData)

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Arguments:

circHandle — **byRef circHandle AS INTEGER**
This identifies the circular buffer to write into.

nData — **byVal *nData* AS INTEGER**
This is the integer value to write into the circular buffer

Interactive Command: NO

```
// Example :: CircBufWrite.sb (See in BL600CodeSnippets.zip)
DIM rc
DIM circHandle
DIM i

rc = CircBufCreate(16,circHandle)
IF rc != 0 then
    PRINT "\nThe circular buffer was not created\n"
ELSE
    PRINT "\nThe circular buffer was created successfully\n"
ENDIF

//write 3 values into the circular buffer
FOR i = 1 TO 3
    rc = CircBufWrite(circHandle,i)
    IF rc != 0 then
        PRINT "\nFailed to write into the circular buffer\n"
    ELSE
        PRINT i;" was successfuly written to the circular buffer\r"
    ENDIF
NEXT
```

Expected output:

```
The circular buffer was created successfully
1 was successfuly written to the circular buffer
2 was successfuly written to the circular buffer
3 was successfuly written to the circular buffer
```

CIRCBUFWRITE is an extension function.

CircBufOverWrite

FUNCTION

This function is used to write an integer at the head end of the circular buffer and if there is no space available to write, then it will return with a failure resultcode but still write into the circular buffer by first discarding the oldest item.

CIRCBUFOVERWRITE (circHandle, nData)

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation
Note if the buffer was full and the oldest value was overwritten then a non-zero value of 0x5103 will still be returned.

Arguments:

circHandle byRef circHandle AS INTEGER
This identifies the circular buffer to write into.

nData byVal *nData* AS INTEGER
This is the integer value to write into the circular buffer. It is always written into the buffer. Oldest is discarded to make space for this.

Interactive Command: NO

```
// Example :: CircBufOverwrite.sb (See in BL600CodeSnippets.zip)
DIM rc,circHandle,i

rc = CircBufCreate(4,circHandle)
IF rc != 0 THEN
    PRINT "\nThe circular buffer was not created\n"
ELSE
    PRINT "\nThe circular buffer was created successfully\n"
ENDIF

FOR i = 1 TO 5
    rc = CircBufOverwrite(circHandle,i)
    IF rc == 0x5103 THEN
        PRINT "\nOldest value was discarded to write ";i
    ELSEIF rc !=0 THEN
        PRINT "\nFailed to write into the circular buffer"
    ELSE
        PRINT "\n";i
```

```
    ENDIF
NEXT
```

Expected Output:

CIRCBUFOVERWRITE is an extension function.

CircBufRead

FUNCTION

This function is used to read an integer from the tail end of the circular buffer. A nonzero resultcode will be returned if the buffer is empty or if the handle is invalid.

CIRCBUFREAD(circHandle, nData)

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation. If 0x5102 is returned it implies the buffer was empty so nothing was read.

Arguments:

circHandle — **byRef circHandle AS INTEGER**
This identifies the circular buffer to read from.

nData — **byRef *nData* AS INTEGER**
This is the integer value to read from the circular buffer

Interactive Command: NO

```
// Example :: CircBufRead.sb (See in BL600CodeSnippets.zip)
DIM rc,circHandle,i,nData

rc = CircBufCreate(4,circHandle)
IF rc != 0 THEN
    PRINT "\nThe circular buffer was not created"
ELSE
    PRINT "\nThe circular buffer was created successfully\n"
    PRINT "Writing..."
ENDIF

FOR i = 1 TO 5
    rc = CircBufOverwrite(circHandle,i)
    IF rc == 0x5103 THEN
        PRINT "\nOldest value was discarded to write ";i;"\n"
    ELSEIF rc !=0 THEN
        PRINT "\nFailed TO write inTO the circular buffer"
    ELSE
```

```
        PRINT "\n";i
    ENDIF
NEXT

//read 4 values from the circular buffer
PRINT "\nReading...\n"
FOR i = 1 to 4
  rc = CircBufRead(circHandle,nData)
  IF rc == 0x5102 THEN
    PRINT "The buffer was empty"
  ELSEIF rc != 0 THEN
    PRINT "Failed to read from the circular buffer"
  ELSE
    PRINT nData;"\n"
  ENDIF
NEXT
```

Expected Output:

CIRCBUFREAD is an extension function.

CircBufItems

FUNCTION

This function is used to determine the number of integer items held in the circular buffer.

CIRCBUFITEMS(circHandle, nItems)

Returns: INTEGER, a result code. The typical value is 0x0000, indicating a successful operation. If 0x5102 is returned it implies the buffer was empty so nothing was read.

Arguments:

circHandle — byRef circHandle AS INTEGER
This identifies the circular buffer which needs to be queried.

nItems — byRef *nItems* AS INTEGER
This returns the total items waiting to be read in the circular buffer.

Interactive Command: NO

```
// Example :: CircBufItems.sb (See in BL600CodeSnippets.zip)
DIM rc,circHandle,i,nItems
rc = CircBufCreate(4,circHandle)
IF rc != 0 THEN
    PRINT "\nThe circular buffer was not created\n"
ELSE
    PRINT "\nThe circular buffer was created successfully\n"
ENDIF

FOR i = 1 TO 5
    rc = CircBufOverwrite(circHandle,i)
    IF rc == 0x5103 THEN
        PRINT "\nOldest value was discarded to write ";i
    ELSEIF rc !=0 THEN
        PRINT "\nFailed TO write inTO the circular buffer"
    ENDIF

    rc = CircBufItems(circHandle,nItems)
    IF rc == 0 THEN
        PRINT "\n";nItems;" items in the circular buffer"
    ENDIF
NEXT
```

Expected Output:

CIRCBUFITEMS is an extension function.

Serial Communications Routines

In keeping with the event driven architecture of *smart* BASIC, the serial communications subsystem enables *smart* BASIC applications to be written which allow communication events to trigger the processing of user *smart* BASIC code.

Note that if a handler function returns a non-zero value then the WAITEVENT statement is reprocessed, otherwise the *smart* BASIC runtime engine will proceed to process the next statement **after** the WAITEVENT statement – not after the handlers ENDFUNC or EXITFUNC statement. Please refer to the detailed description of the WAITEVENT statement for further information.

UART (Universal Asynchronous Receive Transmit)

This section describes all the events and routines used to interact with the UART peripheral available on the platform. Depending on the platform, at a minimum, the UART will consist of a transmit, a receive, a CTS

(Clear To Send) and RTS (Ready to Send) line. The CTS and RTS lines are used for hardware handshaking to ensure that buffers do not overrun.

If there is a need for the following low bandwidth status and control lines found on many peripherals, then the user is able to create those using the GPIO lines of the module and interface with those control/status lines using *smart* BASIC code.

Output	DTR	Data Terminal Ready
Input	DSR	Data Set Ready
Output/Input	DCD	Data Carrier Detect
Output/Input	RI	Ring Indicate

The lines DCD and RI are marked as Output or Input because it is possible, unlike a device like a PC where they are always inputs and modems where they are always outputs, to configure the pins to be either so that the device can adopt a DTE (Data Terminal Equipment) or DCE (Data Communications Equipment) role. *Please note that both DCD and RI have to be BOTH outputs or BOTH inputs, one cannot be an output and the other an input.*

UART Events

In addition to the routines for manipulating the UART interface, when data arrives via the receive line 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 detailed list of all events generated by the UART subsystem which can be handled by user code.

EVUARTRX	This event is generated when one or more new characters have arrived and have been stored in the local ring buffer.
EVUARTTXEMPTY	This event is generated when the last character is transferred from the local transmit ring buffer to the hardware shift register.

```
// Example :: EVUARTRX.sb (See in BL600CodeSnippets.zip)
DIM rc
FUNCTION HndlrUartRx()
  PRINT "\nData has arrived\r"
ENDFUNC 1        //remain blocked in WAITEVENT

FUNCTION Btn0Pressed()
ENDFUNC 0

rc = GPIOBindEvent(0,16,1)
PRINT "\nPress Button 0 to exit this application \n"

ONEVENT EVUARTRX    CALL HndlrUartRx
ONEVENT EVGPIOCHAN0 CALL Btn0Pressed

WAITEVENT       //wait for rx, tx and modem status events
PRINT "Exiting..."
```

Expected Output:

```
```

Note: If you type unknown commands, an E007 error displays in UwTerminal.

```
// Example :: EVUARTTXEMPTY.sb (See in BL600CodeSnippets.zip)
FUNCTION HndlrUartTxEty()
  PRINT "\nTx buffer is empty"
ENDFUNC 0

ONEVENT EVUARTTXEMPTY CALL HndlrUartTxEty

PRINT "\nSend this via uart"

WAITEVENT
```

Expected Output:

```
Send this via uart
Tx buffer is empty
```

UartOpen

Note: Until further notice, the parity parameter shall not be changed when using this function.

Function

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

If the uart is already open then this function will fail.

If this function is used to alter the communications parameters, like say the baudrate and the application exits to interactive mode, then those settings will be inherited by the interactive mode parser. Hence this is the only way to alter the communications parameters for Interactive mode.

While the uart is open, if a BREAK is sent to the module, then it will force the module into deep sleep mode as long as BREAK is asserted. As soon as BREAK is deasserted, the module will wake up through a reset as if it had been power cycled.

UARTOPEN (baudrate,txbuflen,rxbuflen,stOptions)

Returns: INTEGER Indicates success of command:

0	Opened successfully
0x5208	Invalid baudrate
0x5209	Invalid parity
0x520A	Invalid databits
0x520B	Invalid stopbits
0x520C	Cannot be DTE (because DCD and RI cannot be inputs)
0x520D	Cannot be DCE (because DCD and RI cannot be outputs)
0x520E	Invalid flow control request
0x520F	Invalid DTE/DCE role request
0x5210	Invalid length of stOptions parameter (must be 5 chrs)
0x5211	Invalid tx buffer length
0x5212	Invalid rx buffer length

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

baudrate — *byVal baudrate AS INTEGER*
The baudrate for the uart. Note that, the higher the baudrate, the more power will be drawn from the supply pins.
AT I 1002 or SYSINFO(1002) returns the minimum valid baudrate
AT I 1003 or SYSINFO(1003) returns the maximum valid baudrate

txbuflen — *byVal txbuflen AS INTEGER*
Set the transmit ring buffer size to this value. If set to 0 then a default value will be used by the underlying driver

rxbuflen — *byVal rxbuflen AS INTEGER*
Set the receive ring buffer size to this value. If set to 0 then a default value will be used by the underlying driver

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' for DTE and 'C' for DCE
1: Parity – 'N' for none, 'O' for odd and 'E' for even
2: Databits – '5','6','7','8',9'
3: Stopbits – '1','2'
4: Flow Control – 'N' for none, 'H' for CTS/RTS hardware, 'X' for xon/xof

---

Note: There will be further restrictions on the options based on the hardware as for example a PC implementation cannot be configured as a DCE role. Likewise many microcontroller uart peripherals are not capable of 5 bits per character – but a PC is.

Note: In DTE equipment DCD and RI are inputs, while in DCE they are outputs.
Interactive Command: No

Related Commands: UARTINFO, UARTCLOSE, UARTWRITE, UARTREAD, UARTREADMATCH UARTGETDSR, UARTGETCTS, UARTGETDCD, UARTGETRI, UARTSETDTR, UARTSETRTS, UARTSETDCD, UARTSETRI, UARTBREAK, UARTFLUSH

```
// Example :: UartOpen.sb (See in BL600CodeSnippets.zip)
DIM rc

FUNCTION HndlrUartRx()
    PRINT "\nData has arrived\r"
ENDFUNC 1 //remain blocked in WAITEVENT

FUNCTION Btn0Pressed()
    UartClose()
ENDFUNC 0

rc = GPIOBindEvent(0,16,1)                    //For button0

ONEVENT EVUARTRX CALL HndlrUartRx
ONEVENT EVGPIOCHAN0 CALL Btn0Pressed

UartClose()                                   //Since Uart port is already open we must
                                              //close it before opening it again with
                                              //different settings.

//--- Open comport so that DCD and RI are inputs
rc = UartOpen(9600,0,0,"CN81H")               //Open as DCE, no parity, 8 databits,
                                              //1 stopbits, cts/rts flow control

IF rc!= 0 THEN
    PRINT "\nFailed to open UART interface with error code ";INTEGER.H' rc
ELSE
    PRINT "\nUART open success"
ENDIF

PRINT "\nPress button0 to exit this application\n"

WAITEVENT                                     //wait for rx, events
PRINT "\nExiting..."
```

Expected Output:

UARTOPEN is a core function.

UartClose

FUNCTION

This subroutine is used to close a uart port which had been opened with UARTOPEN.

If after the uart is closed, a print statement is encountered, the uart will be automatically re-opened at the default rate (9600N81) so that the data generated by the PRINT statement is sent.

This routine will throw an exception if the uart is already closed, so if you are not sure then it is best to call it if UARTINFO(1) returns a non-zero value.

When this subroutine is invoked, the receive and transmit buffers are both flushed. If there is any data in either of these buffers when the UART is closed, it will be lost. This is because the execution of UARTCLOSE takes a very short amount of time, while the transfer of data from the buffers will take much longer.

In addition please note that when a *smart* BASIC application completes execution with the UART closed, it will automatically be reopened in order to allow continued communication with the module in Interactive Mode using the default communications settings.

UARTCLOSE()

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments: None

Interactive Command: No

Related Commands: UARTOPEN,UARTINFO, UARTWRITE, UARTREAD, UARTREADMATCH, UARTGETCTS, UARTGETDCD, UARTGETRI, UARTSETDTR, UARTSETRTS, UARTSETDCD, UARTSETRI, UARTBREAK, UARTFLUSH

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

IF UartInfo(0)==0 THEN
    PRINT "\nThe Uart port was closed"
ELSE
    PRINT "\nThe Uart port was not closed"
ENDIF

IF UartInfo(0)!=0 THEN
    PRINT "\nand now it is open"
ENDIF
```

Expected Output:

```
The Uart port was closed
and now it is open
```

UARTCLOSE is a core subroutine.

UartCloseEx

FUNCTION

This function is used to close a uart port which had been opened with UARTOPEN depending on the flag mask in the input parameter.

Please see UartClose() for more details

Note:

For firmware versions older than 1.3.57.3 there is a bug which means that if the rx & tx buffers are not empty an internal pointer is still set to NULL when it should. This results in unpredictable behaviour.

Workaround:

Use UartInfo(6) to check if the buffers are empty and then call UartCloseRx(1)

UARTCLOSEEX(nFlags)

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation. If 0x5231 is returned it implies one of the buffers was not empty so not closed.

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

nFlags *byVal nFlags AS INTEGER*
If Bit 0 is set, then only close if both rx and tx buffers are empty. Setting this bit to 0 has the same effect as UartClose() routine.
Bits 1 to 31 are for future use and must be set to 0.

Interactive Command: No

Related Commands: UARTOPEN,UARTINFO, UARTWRITE, UARTREAD, UARTREADMATCH, UARTGETCTS, UARTGETDCD, UARTGETRI, UARTSETDTR, UARTSETRTS, UARTSETDCD, UARTSETRI, UARTBREAK, UARTFLUSH

Workaround for FW **1.3.57.0 and earlier:**

```
//Example :: UartCloseExWA.sb (See in BL600CodeSnippets.zip)
DIM rc1
DIM rc2

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

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

For FW **1.3.57.3 and newer:**

```
//Example :: UartCloseEx.sb (See in BL600CodeSnippets.zip)
DIM rc1
DIM rc2

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

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

Expected Output:

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

UARTCLOSEEX is a core function.

UartInfo

FUNCTION

This function is used to query information about the default uart, such as buffer lengths, whether the port is already open or how many bytes are waiting in the receive buffer to be read.

UARTINFO (infoId)

Function

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 type of uart information requested as follows if the uart is open:-
0 := 1 (the port is open), 0 (the port is closed)
And the following specify the type of uart information when the port is open:-
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 := Number of bytes still waiting to be sent in transmit buffer
6 := Total number of bytes waiting in rx and tx buffer

If the uart is closed, then regardless of the value of *infoId*, a 0 will be returned.
Note: UARTINFO(0) will always return the open/close state of the uart.
Interactive Command: No

Related Commands: UARTOPEN, UARTCLOSE, UARTWRITE, UARTREAD, UARTREADMATCH UARTGETDSR, UARTGETCTS, UARTGETDCD, UARTGETRI, UARTSETDTR, UARTSETRTS, UARTSETDCD, UARTSETRI, UARTBREAK, UARTFLUSH

```
//Example :: UartInfo.sb (See in BL600CodeSnippets.zip)
DIM rc,start

UartClose()

IF UartInfo(0)==0 THEN
    PRINT "\nThe Uart port was closed\n"
ELSE
    PRINT "\nThe Uart port was not closed\n"
ENDIF

PRINT "\nReceive ring buffer capacity:          ";UartInfo(1)
PRINT "\nTransmit ring buffer capacity:     ";UartInfo(2)
PRINT "\nNo. bytes waiting in transmit buffer:  ";UartInfo(5)

start = GetTickCount()
DO
UNTIL UartInfo(5)==0
```

```
PRINT "\n\nTook ";GetTickSince(start);" milliseconds for transmit buffer to be emptied"
```

Expected Output:

UARTINFO is a core subroutine.

UartWrite

FUNCTION

This function is used to transmit a string of characters.

UARTWRITE (strMsg)

Returns: INTEGER 0 to N : Actual number of bytes successfully written to the local transmit ring buffer

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow
- Uart has not been opened using UARTOPEN (or auto-opened with PRINT statement)

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,this implies the transmit buffer did not have enough space to accommodate the data. If the return value does not match the length of the original string, then use STRSHIFTLEFT function to drop the data from the string, so that subsequent calls to this function only retries with data which was not placed in the output ring buffer.

Interactive Command: No

Note: **strMsg** cannot be a string constant, e.g. "the cat", but must be a string variable and so if you must use a const string then first save it to a temp string variable and then pass it to the function

---

Related Commands: UARTOPEN,UARTINFO, UARTCLOSE, UARTREAD, UARTREADMATCH UARTGETDSR, UARTGETCTS, UARTGETDCD, UARTGETRI, UARTSETDTR, UARTSETRTS, UARTSETDCD, UARTSETRI, UARTBREAK, UARTFLUSH

```
//Example :: UartWrite.sb (See in BL600CodeSnippets.zip)
DIM rc,str$,i,done,d

//str$ contains a lot of space so that we can satisfy the condition in the IF
statement
str$="
Hello World"

FUNCTION HndlrUartTxEty()
  PRINT "\nTx buffer is now empty"
ENDFUNC 0 //exit from WAITEVENT

rc=UartWrite(str$)

//Shift 'str$' if there isn't enough space in the buffer until 'str$' can be written
WHILE done == 0
    IF rc < StrLen(str$) THEN
        PRINT rc;" bytes written"
        PRINT "\nStill have ";StrLen(str$)-rc;" bytes to write\n"
        PRINT "\nShifting 'str$' by ";rc
        StrShiftLeft(str$,rc)
        done = 0
    ELSE
        PRINT "\nString 'str$' written successfully"
        done=1
    ENDIF
ENDWHILE

ONEVENT EVUARTTXEMPTY CALL HndlrUartTxEty

WAITEVENT
```

Expected Output:

UARTWRITE is a core subroutine.

UartRead

FUNCTION

This function is used to read the content of the receive buffer and <u>append</u> it to the string variable supplied.

UARTREAD(strMsg)

Returns: INTEGER 0 to N : The total length of the string variable – not just what got appended. This means the caller does not need to call strlen() function to determine how many bytes in the string that need to be processed.

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow
- Uart has not been opened using UARTOPENxxx

Arguments:

strMsg *byRef strMsg AS STRING*
The content of the receive buffer will get <u>appended</u> to this string.

Interactive Command: No

Note: **strMsg** cannot be a string constant, e.g. "the cat", but must be a string variable and so if you must use a const string then first save it to a temp string variable and then pass it to the function

Related Commands: UARTOPEN,UARTINFO, UARTCLOSE, UARTWRITE, UARTREADMATCH, UARTGETDSR, UARTGETCTS, UARTGETDCD, UARTGETRI, UARTSETDTR, UARTSETRTS, UARTSETDCD, UARTSETRI, UARTBREAK, UARTFLUSH

```
//Example :: UartRead.sb (See in BL600CodeSnippets.zip)
DIM rc,strLength,str$
str$="Your name is "

FUNCTION HndlrUartRx()
    TimerStart(0,100,0)      //Allow enough time for data to reach rx buffer
ENDFUNC 1

FUNCTION HndlrTmr0()
    strLength=UartRead(str$)
    PRINT "\n";str$
ENDFUNC 0

ONEVENT EVTMR0       CALL HndlrTmr0
ONEVENT EVUARTRX     CALL HndlrUartRx

PRINT "\nWhat is your name?\n"

WAITEVENT
```

Expected Output:

```
What is your name?
David

Your name is David
```

UARTREAD is a core subroutine.

UartReadN

FUNCTION

This function is used to read the content of the receive buffer and <u>append</u> it to the string variable supplied but it ensures that the string is not longer than nMaxLen.

UARTREADN(strMsg, nMaxLen)

Returns: INTEGER 0 to N : The total length of the string variable – not just what got appended. This means the caller does not need to call strlen() function to determine how many bytes in the string that need to be processed.

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow
- Uart has not been opened using UARTOPENxxx

Arguments:

strMsg — *byRef strMsg AS STRING*
The content of the receive buffer will get <u>appended</u> to this string.

nMaxLen — *byval nMaxLen AS INTEGER*
The output string strMsg will never be longer than this value. If a value less than 1 is specified, it will be clipped to 1 and if > that 0xFFFF it will be clipped to 0xFFFF.

Interactive Command: No

Note: strMsg cannot be a string constant, e.g. "the cat", but must be a string variable and so if you must use a const string then first save it to a temp string variable and then pass it to the function

Related Commands: UARTOPEN,UARTINFO, UARTCLOSE, UARTWRITE, UARTREADMATCH, UARTGETDSR, UARTGETCTS, UARTGETDCD, UARTGETRI, UARTSETDTR, UARTSETRTS, UARTSETDCD, UARTSETRI, UARTBREAK, UARTFLUSH

```
//Example
DIM rc,strLength,str$
str$="Your name is "

FUNCTION HndlrUartRx()
    TimerStart(0,100,0)      //Allow enough time for data to reach rx buffer
ENDFUNC 1
```

```
FUNCTION HndlrTmr0()
    strLength=UartReadn(str$,11)
    PRINT "\n";str$
ENDFUNC 0

ONEVENT EVTMR0      CALL HndlrTmr0
ONEVENT EVUARTRX    CALL HndlrUartRx

PRINT "\nWhat is your name?\n"

WAITEVENT
```

Expected Output:

```
What is your name?
David

Your name i
```

UARTREADN is a core subroutine.

UartReadMatch

FUNCTION

This function is used to read the content of the underlying receive ring buffer and **append** it to the string variable supplied, up to and including the first instance of the specified matching character OR the end of the ring buffer.

This function is very useful when interfacing with a peer which sends messages terminated by a constant character such as a carriage return (0x0D). In that case, in the handler, if the return value is greater than 0, it implies a terminated message arrived and so can be processed further.

UARTREADMATCH(strMsg , chr)

Returns: INTEGER Indicates the presence of the match character in **strMsg** as follows:

0 : data **may** have been appended to the string, but no matching character.
1 to N : The total length of the string variable up to and including the match **chr**.

Note: When 0 is returned you can use STRLEN(strMsg) to determine the length of data stored in the string. On some platforms with low amount of RAM resources, the underlying code may decide to leave the data in the receive buffer rather than transfer it to the string.

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow
- Uart has not been opened using UARTOPEN

Arguments:

strMsg — *byRef strMsg AS STRING*
The content of the receive buffer will get **appended** to this string up to and including the match character.

chr *byVal chr AS INTEGER*
The character to match in the receive buffer, for example the carriage return character 0x0D

Interactive Command: No

Note: **strMsg** cannot be a string constant, e.g. "the cat", but must be a string variable and so if you must use a const string then first save it to a temp string variable and then pass it to the function

---

Related Commands: UARTOPEN,UARTINFO, UARTCLOSE, UARTWRITE, UARTREAD, UARTGETDSR, UARTGETCTS, UARTGETDCD, UARTGETRI, UARTSETDTR, UARTSETRTS, UARTSETDCD, UARTSETRI, UARTBREAK, UARTFLUSH

```
//Example :: UartReadMatch.sb (See in BL600CodeSnippets.zip)
DIM rc,str$,ret,char,str2$
ret=1                                   //Function return value
char=13                                 //ASCII decimal value for 'carriage return'

str$="Your name is "
str2$="\n\nMatch character ' ' not found \nExiting.."


FUNCTION HndlrUartRx()
    TimerStart(0,10,0)                  //Allow time for data to reach rx buffer
ENDFUNC 1

FUNCTION HndlrTmr0()
    rc = UartReadMatch(str$,char)
    PRINT "\n";str$
    IF rc==0 THEN
        rc=StrSetChr(str2$,char,19) //Insert 'char', the match character
        PRINT str2$
        str2$="\n\nMatch character  not found \nExiting.."  //reset str2$
        ret=0
    ELSE
        PRINT "\n\n\nNow type something without the letter 'a'\n"
        str$="You sent "                                    //reset str$
        char=97                        //ASCII decimal value for 'a'
        ret=1
    ENDIF
ENDFUNC ret

ONEVENT EVTMR0      CALL HndlrTmr0
ONEVENT EVUARTRX    CALL HndlrUartRx

PRINT "\nWhat is your name?\n"

WAITEVENT
```

Expected Output:

UARTREADMATCH is a core subroutine.

UartFlush

SUBROUTINE

This subroutine is used to flush 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 and the input buffer fills up. In that case, there is no more space for an incoming termination character and the RTS handshaking line would have been asserted so the message system will stall. A flush of the receive buffer is the best approach to recover from that situation.

Note: Execution of UARTFLUSH is much quicker than the time taken to transmit data to/from the buffers

UARTFLUSH(bitMask)

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow
- Uart has not been opened using UARTOPEN

Arguments:

bitMask *byVal* *bitMask AS INTEGER*

This bit mask is used to choose which ring buffer to flush.

Bit	Description
0	Set to flush the rx buffer
1	Set to flush the tx buffer

Set both bits to flush both buffers.

Interactive Command: No

Related Commands: UARTOPEN,UARTINFO, UARTCLOSE, UARTWRITE, UARTREAD, UARTREADMATCH, UARTGETCTS, UARTGETDCD, UARTGETRI, UARTGETDSR, UARTSETRTS, UARTSETDCD, UARTBREAK, UARTFLUSH

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

```
FUNCTION HndlrUartRx()
    TimerStart(0,2,0)                              //Allow time for data to reach rx
buffer
ENDFUNC 1

FUNCTION HndlrTmr0()
    PRINT UartInfo(3);" bytes in the rx buffer,\n"
    UartFlush(01)                                  //clear rx buffer
    PRINT UartInfo(3);" bytes in the rx buffer after flushing"
ENDFUNC 0

ONEVENT EVUARTRX    CALL HndlrUartRx
ONEVENT EVTMR0      CALL HndlrTmr0

PRINT "\nSend me some text\n"

WAITEVENT
```

Expected Output:

```
Send me some data
Laird
6 bytes in the rx buffer,
0 bytes in the rx buffer after flushing
```

```
//Example :: UartFlushTx.sb (See in BL600CodeSnippets.zip)
DIM s$ : s$ = "Hello World"
DIM rc : rc = UartWrite(s$)

UartFlush(10)        //Will flush before all chars have been transmitted
PRINT UartInfo(5); " bytes in the tx buffer after flushing"
```

Expected Output:

```
H0 bytes in the tx buffer after flushing
```

UARTFLUSH is a core subroutine.

UartGetCTS

FUNCTION

This function is used to read the current state of the CTS modem status input line.

If the device does not expose a CTS input line, then this function will return a value that signifies an asserted line.

UARTGETCTS()

Returns: INTEGER Indicates the status of the CTS line:

0 : CTS line is NOT asserted
1 : CTS line is asserted

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow
- Uart has not been opened using UARTOPEN

Arguments: None

Interactive Command: No

Related Commands: UARTOPEN,UARTINFO, UARTCLOSE, UARTWRITE, UARTREAD, UARTREADMATCH, UARTGETDSR, UARTGETDCD, UARTGETRI, UARTSETDTR, UARTSETRTS, UARTSETDCD, UARTSETRI, UARTBREAK, UARTFLUSH

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

IF UartGetCTS()==0 THEN
    PRINT "\nCTS line is not asserted"
ELSEIF UartGetCTS()==1 THEN
    PRINT "\nCTS line is asserted"
ENDIF
```

Expected Output:

```
CTS line is not asserted
```

UARTGETCTS is a core subroutine.

UartSetRTS

SUBROUTINE

This function is used to set the state of the RTS modem control line. When the UART port is closed, the RTS line can be configured as an input or an output and can be available for use as a general purpose input/output line.

When the uart port is opened, the RTS output is automatically defaulted to the asserted state. If flow control was enabled when the port was opened then the RTS output cannot be manipulated as it is owned by the underlying driver.

UARTSETRTS(newState)

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow
- Uart has not been opened using UARTOPEN

Arguments:

newState *byVal newState AS INTEGER*
0 to deassert and non-zero to assert

Interactive Command: No

Related Commands: UARTOPEN,UARTINFO, UARTCLOSE, UARTWRITE, UARTREAD, UARTREADMATCH, UARTGETCTS, UARTGETDCD, UARTGETRI, UARTGETDSR, UARTSETDTR, UARTSETDCD, UARTSETRI, UARTBREAK, UARTFLUSH

Note: This subroutine is not implemented in the BL600

UARTSETRTS is a core subroutine.

UartBREAK

SUBROUTINE

This subroutine is used to assert/deassert a BREAK on the transmit output line. A BREAK is a condition where the line is in non idle state (that is 0v) for more than 10 to 13 bit times, depending on whether parity has been enabled and the number of stopbits.

On certain platforms the hardware may not allow this functionality, contact Laird to determine if your device has the capability. On platforms that do not have this capability, this routine has no effect.

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

UARTBREAK(state)

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow
- Uart has not been opened using UARTOPEN

Arguments:

newState *byVal newState AS INTEGER*
0 to deassert and non-zero to assert

Interactive Command: No

Related Commands: UARTOPEN, UARTINFO, UARTCLOSE, UARTWRITE, UARTREAD, UARTREADMATCH, UARTGETCTS, UARTGETDCD, UARTGETRI, UARTGETDSR, UARTSETRTS, UARTSETDCD, UARTFLUSH

Note: This subroutine is not implemented in the BL600

UARTBREAK is a core subroutine.

I2C - Also known as Two Wire Interface (TWI)

This section describes all the events and routines used to interact with the I2C peripheral available on the platform. An I2C interface is also known as a Two Wire Interface (TWI) and has a master/slave topology.

An I2C interface allows multiple masters and slaves to communicate over a shared wired-OR type bus consisting of two lines which normally sit at 5 or 3.3v.

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

The two signal lines are called SCL and SDA. The former is the clock line which is always sourced by the master and the latter is a bi-directional data line which can be driven by any device on the bus.

It is essential to remember that pull up resistors on both SCL and SDA lines are not provided in the module and MUST be provided external to the module.

A very good introduction to I2C can be found at http://www.i2c-bus.org/i2c-primer/ and the reader is encouraged to refer to it before using the api described in this section.

I2C Events

The API provided in the module is synchronous and so there is no requirement for events.

I2cOpen

FUNCTION

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

This function is used to open the main I2C peripheral using the parameters specified.

On the BL600 module the SCL signal Pin is on SIO9 and SDA signal pin is SIO8.

I2COPEN (nClockHz, nCfgFlags, nHande)

Returns: INTEGER Indicates success of command:

0	Opened successfully
0x5200	Driver not found
0x5207	Driver already open
0x5225	Invalid Clock Frequency Requested
0x521D	Driver resource unavailable
0x5226	No free PPI channel
0x5202	Invalid Signal Pins
0x5219	I2C not allowed on pins specified

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

nClockHz — *byVal nClockHz AS INTEGER*
This is the clock frequency to use, and can be one of 100000, 250000 or 400000.

nCfgFlags — *byVal nCfgFlags AS INTEGER*
This is a bit mask used to configure the I2C interface. All unused bits are allocated as for future use and MUST be set to 0. Used bits are as follows:-

Bit	Description
0	If set, then a 500 microsecond low pulse will NOT be sent on open. This low pulse is used to create a start and stop condition on the bus so that any signal transitions on these lines prior to this open which may have confused a slave can initialise that slave to a known state. The STOP condition should be detected by the slave.
1-31	Unused and MUST be set to 0

nHandle — *byRef nHandle AS INTEGER*
The handle for this interface will be returned in this variable if it was successfully opened. This handle is subsequently used to read/write and close the interface.

Related Commands: I2CCLOSE, I2CWRITEREAD$, I2CWRITEREG8, I2CWRITEREG16, I2CWRITEREG32, I2CREADREG8, I2CREADREG16, I2CREADREG32

```
//Example :: I2cOpen.sb (See in BL600CodeSnippets.zip)
DIM handle
DIM rc : rc=I2cOpen(100000,0,handle)

IF rc!= 0 THEN
    PRINT "\nFailed to open I2C interface with error code "; INTEGER.h' rc
ELSE
    PRINT "\nI2C open success \nHandle is ";handle
ENDIF
```

Expected Output:

```
I2C open success
Handle is 0
```

I2COPEN is a core function.

I2cClose

SUBROUTINE

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

This subroutine is used to close a I2C port which had been opened with I2COPEN.

This routine is safe to call if it is already closed.

I2CCLOSE(handle)

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

handle — *byVal handle AS INTEGER*
This is the handle value that was returned when I2COPEN was called which identifies the I2C interface to close.

Interactive Command: No

Related Commands: I2COPEN, I2CWRITEREAD$, I2CWRITEREG8, I2CWRITEREG16, I2CWRITEREG32, I2CREADREG8, I2CREADREG16, I2CREADREG32

```
//Example :: I2cClose.sb (See in BL600CodeSnippets.zip)
DIM handle
DIM rc : rc=I2cOpen(100000,0,handle)

IF rc!= 0 THEN
    PRINT "\nFailed to open I2C interface with error code "; INTEGER.h' rc
ELSE
    PRINT "\nI2C open success \nHandle is ";handle
ENDIF

I2cClose(handle)  //close the port
I2cClose(handle)  //no harm done doing it again
```

I2CCLOSE is a core subroutine.

I2cWriteREG8

SUBROUTINE

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

This function is used to write an 8 bit value to a register inside a slave which is identified by an 8 bit register address.

Note a 'handle' parameter is NOT required as this function is used to interact with the main interface. In the future, a new version of this function will be made available if more than one I2C interface is made available, most likely made available by bit-bashing gpio.

I2CWRITEREG8(nSlaveAddr, nRegAddr, nRegValue)

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

nSlaveAddr *byVal nSlaveAddr AS INTEGER*
This is the address of the slave in range 0 to 127.

nRegAddr *byVal nRegAddr AS INTEGER*
This is the 8 bit register address in the addressed slave in range 0 to 255.

nRegValue *byVal nRegValue AS INTEGER*
This is the 8 bit value to written to the register in the addressed slave.
Please note only the lowest 8 bits of this variable are written.

Interactive Command: No

Related Commands: I2COPEN, I2CCLOSE, I2CWRITEREAD$, I2CWRITEREG8, I2CWRITEREG16, I2CWRITEREG32, I2CREADREG8, I2CREADREG16, I2CREADREG32

```
//Example :: I2cWriteReg8.sb (See in BL600CodeSnippets.zip)

//**Please ensure that nSlaveAddr is the slave address of your I2C peripheral**
DIM rc, handle, nSlaveAddr, nRegAddr, nRegVal


//--- Open I2C Peripheral
rc=I2cOpen(100000,0,handle)
IF rc!= 0 THEN
    PRINT "\nFailed to open I2C interface with error code "; INTEGER.H' rc
ELSE
    PRINT "\nI2C open success"
ENDIF


//--- Write 'nRegVal' to register 'nRegAddr'
nSlaveAddr=0x6f :  nRegAddr = 23 : nRegVal = 0x63
rc = I2cWriteReg8(nSlaveAddr, nRegAddr, nRegVal)
IF rc!= 0 THEN
    PRINT "\nFailed to Write to slave/register "; INTEGER.H'rc
ELSE
    PRINT "\n";nRegVal; " written successfully to register ";nRegAddr
ENDIF

I2cClose(handle)  //close the port
```

Expected Output:

```
I2C open success
99 written successfully to register 23
```

I2CWRITEREG8 is a core function.

I2cReadREG8

SUBROUTINE

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

This function is used to read an 8 bit value from a register inside a slave which is identified by an 8 bit register address.

Note a 'handle' parameter is NOT required as this function is used to interact with the main interface. In the future, a new version of this function will be made available if more than one I2C interface is made available, most likely made available by bit-bashing gpio.

I2CREADREG8(nSlaveAddr, nRegAddr, nRegValue)

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

nSlaveAddr — *byVal nSlaveAddr AS INTEGER*
This is the address of the slave in range 0 to 127.

nRegAddr — *byVal nRegAddr AS INTEGER*
This is the 8 bit register address in the addressed slave in range 0 to 255.

nRegValue — *byRef nRegValue AS INTEGER*
The 8 bit value from the register in the addressed slave will be returned in this variable.

Interactive Command: No

Related Commands: I2COPEN, I2CCLOSE, I2CWRITEREAD$, I2CWRITEREG8, I2CWRITEREG16, I2CWRITEREG32, I2CREADREG8, I2CREADREG16, I2CREADREG32

```
//Example :: I2cReadReg8.sb (See in BL600CodeSnippets.zip)

//**Please ensure that nSlaveAddr is the slave address of your I2C peripheral**
DIM rc, handle, nSlaveAddr, nRegAddr, nRegVal


//--- Open I2C Peripheral
rc=I2cOpen(100000,0,handle)
IF rc!= 0 THEN
    PRINT "\nFailed to open I2C interface with error code "; INTEGER.H' rc
ELSE
    PRINT "\nI2C open success"
ENDIF


//---Read value from address 0x34
nSlaveAddr=0x6f :  nRegAddr = 23
rc = I2cReadReg8(nSlaveAddr, nRegAddr, nRegVal)
```

```
IF rc!= 0 THEN
    PRINT "\nFailed to Read from slave/register "; INTEGER.H'rc
ELSE
    PRINT "\nValue read from register is ";nRegVal
ENDIF

I2cClose(handle)  //close the port
```

Expected Output:

```
I2C open success
Value read from register is 99
```

I2CREADREG8 is a core function.

I2cWriteREG16

SUBROUTINE

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

This function is used to write a 16 bit value to 2 registers inside a slave and the first register is identified by an 8 bit register address supplied.

Note a 'handle' parameter is NOT required as this function is used to interact with the main interface. In the future, a new version of this function will be made available if more than one I2C interface is made available, most likely made available by bit-bashing gpio.

I2CWRITEREG16(nSlaveAddr, nRegAddr, nRegValue)

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

nSlaveAddr — *byVal nSlaveAddr AS INTEGER*
This is the address of the slave in range 0 to 127.

nRegAddr — *byVal nRegAddr AS INTEGER*
This is the 8 bit start register address in the addressed slave in range 0 to 255.

nRegValue — *byVal nRegValue AS INTEGER*
This is the 16 bit value to be written to the register in the addressed slave.
Please note only the lowest 16 bits of this variable are written.

Interactive Command: No

Related Commands: I2COPEN, I2CCLOSE, I2CWRITEREAD$, I2CWRITEREG8, I2CWRITEREG16, I2CWRITEREG32, I2CREADREG8, I2CREADREG16, I2CREADREG32

```
//Example :: I2cWriteReg16.sb (See in BL600CodeSnippets.zip)

//**Please ensure that nSlaveAddr is the slave address of your I2C peripheral**
DIM rc, handle, nSlaveAddr, nRegAddr, nRegVal

//--- Open I2C Peripheral
rc=I2cOpen(100000,0,handle)
IF rc!= 0 THEN
    PRINT "\nFailed to open I2C interface with error code "; INTEGER.H' rc
ELSE
    PRINT "\nI2C open success"
ENDIF

//--- Write 'nRegVal' to register 'nRegAddr'
nSlaveAddr=0x6f :  nRegAddr = 0x34 : nRegVal = 0x4210
rc = I2cWriteReg16(nSlaveAddr, nRegAddr, nRegVal)
IF rc!= 0 THEN
    PRINT "\nFailed to Write to slave/register "; INTEGER.H'rc
ELSE
    PRINT "\n";nRegVal; " written successfully to register ";nRegAddr
ENDIF

I2cClose(handle)  //close the port
```

Expected Output:

```
I2C open success
16912 written successfully to register 52
```

I2CWRITEREG16 is a core function.

I2cReadREG16

SUBROUTINE

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

This function is used to read a 16 bit value from two registers inside a slave which is identified by an 8 bit register address.

Note a 'handle' parameter is NOT required as this function is used to interact with the main interface. In the future, a new version of this function will be made available if more than one I2C interface is made available, most likely made available by bit-bashing gpio.

I2CREADREG16(nSlaveAddr, nRegAddr, nRegValue)

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

nSlaveAddr — *byVal nSlaveAddr AS INTEGER*
This is the address of the slave in range 0 to 127.

nRegAddr — *byVal nRegAddr AS INTEGER*
This is the 8 bit start register address in the addressed slave in range 0 to 255.

nRegValue — *byRef nRegValue AS INTEGER*
The 16 bit value from two registers in the addressed slave will be returned in this variable.

Interactive Command: No

Related Commands: I2COPEN, I2CCLOSE, I2CWRITEREAD$, I2CWRITEREG8, I2CWRITEREG16, I2CWRITEREG32, I2CREADREG8, I2CREADREG16, I2CREADREG32

```
//Example :: I2cReadReg16.sb (See in BL600CodeSnippets.zip)

//**Please ensure that nSlaveAddr is the slave address of your I2C peripheral**
DIM rc, handle, nSlaveAddr, nRegAddr, nRegVal

//--- Open I2C Peripheral
rc=I2cOpen(100000,0,handle)
IF rc!= 0 THEN
    PRINT "\nFailed to open I2C interface with error code "; INTEGER.H' rc
ELSE
    PRINT "\nI2C open success"
ENDIF

//---Read value from address 0x34
nSlaveAddr=0x6f :  nRegAddr = 0x34
rc = I2cReadReg16(nSlaveAddr, nRegAddr, nRegVal)
IF rc!= 0 THEN
    PRINT "\nFailed to Read from slave/register "; INTEGER.H'rc
ELSE
    PRINT "\nValue read from register is ";nRegVal
ENDIF

I2cClose(handle)   //close the port
```

Expected Output:

```
I2C open success
Value read from register is 16912
```

I2CREADREG16 is a core function.

I2cWriteREG32

SUBROUTINE

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

This function is used to write a 32 bit value to 4 registers inside a slave and the first register is identified by an 8 bit register address supplied.

Note a 'handle' parameter is NOT required as this function is used to interact with the main interface. In the future, a new version of this function will be made available if more than one I2C interface is made available, most likely made available by bit-bashing gpio.

I2CWRITEREG32(nSlaveAddr, nRegAddr, nRegValue)

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

nSlaveAddr — *byVal nSlaveAddr AS INTEGER*
This is the address of the slave in range 0 to 127.

nRegAddr — *byVal nRegAddr AS INTEGER*
This is the 8 bit start register address in the addressed slave in range 0 to 255.

nRegValue — *byVal nRegValue AS INTEGER*
This is the 32 bit value to be written to the register in the addressed slave.

Interactive Command: No

Related Commands: I2COPEN, I2CCLOSE, I2CWRITEREAD$, I2CWRITEREG8, I2CWRITEREG16, I2CWRITEREG32, I2CREADREG8, I2CREADREG16, I2CREADREG32

```
//Example :: I2cWriteReg32.sb (See in BL600CodeSnippets.zip)

//**Please ensure that nSlaveAddr is the slave address of your I2C peripheral**
DIM handle
DIM nSlaveAddr, nRegAddr,nRegVal
DIM rc : rc=I2cOpen(100000,0,handle)

IF rc!= 0 THEN
    PRINT "\nFailed to open I2C interface with error code ";INTEGER.h' rc
ELSE
    PRINT "\nI2C open success"
ENDIF

nSlaveAddr = 0x6f :  nRegAddr = 0x56 : nRegVal = 0x4210FEDC
rc = I2cWriteReg32(nSlaveAddr, nRegAddr, nRegVal)
IF rc!= 0 THEN
    PRINT "\nFailed to Write to slave/register "; INTEGER.H'rc
ELSE
```

```
    PRINT "\n";nRegVal; " written successfully to register ";nRegAddr
ENDIF

I2cClose(handle)  //close the port
```

Expected Output:

```
I2C open success
1108410076 written successfully to register 86
```

I2CWRITEREG32 is a core function.

I2cReadREG32

FUNCTION

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

This function is used to read a 32 bit value from four registers inside a slave which is identified by a starting 8 bit register address.

Note a 'handle' parameter is NOT required as this function is used to interact with the main interface. In the future, a new version of this function will be made available if more than one I2C interface is made available, most likely made available by bit-bashing gpio.

I2CREADREG32(nSlaveAddr, nRegAddr, nRegValue)

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

nSlaveAddr — *byVal nSlaveAddr AS INTEGER*
This is the address of the slave in range 0 to 127.

nRegAddr — *byVal nRegAddr AS INTEGER*
This is the 8 bit start register address in the addressed slave in range 0 to 255.

nRegValue — *byRef nRegValue AS INTEGER*
The 32 bit value from four registers in the addressed slave will be returned in this variable.

Interactive Command: No

Related Commands: I2COPEN, I2CCLOSE, I2CWRITEREAD$, I2CWRITEREG8, I2CWRITEREG16, I2CWRITEREG32, I2CREADREG8, I2CREADREG16, I2CREADREG32

```
//Example :: I2cReadREG32.sb (See in BL600CodeSnippets.zip)
```

```
//**Please ensure that nSlaveAddr is the slave address of your I2C peripheral**
DIM handle
DIM nSlaveAddr, nRegAddr,nRegVal
DIM rc : rc=I2cOpen(100000,0,handle)

IF rc!= 0 THEN
    PRINT "\nFailed to open I2C interface with error code ";INTEGER.h' rc
ELSE
    PRINT "\nI2C open success"
ENDIF


//---Read value from address 0x56
nSlaveAddr = 0x6f :  nRegAddr = 0x56
rc = I2cReadReg32(nSlaveAddr, nRegAddr, nRegVal)
IF rc!= 0 THEN
    PRINT "\nFailed to read from slave/register"
ELSE
    PRINT "\nValue read from register is "; nRegVal
ENDIF

I2cClose(handle)  //close the port
```

Expected Output:

```
I2C open success
Value read from register is 1108410076
```

I2CREADREG16 is a core function.

I2cWriteRead

SUBROUTINE

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

This function is used to write from 0 to 255 bytes and then immediately after that read 0 to 255 bytes in a single transaction from the addressed slave. It is a 'free-form' function that allows communication with a slave which has a 10 bit address.

Note a 'handle' parameter is NOT required as this function is used to interact with the main interface. In the future, a new version of this function will be made available if more than one I2C interface is made available, most likely made available by bit-bashing gpio.

I2CWRITEREAD(nSlaveAddr, stWrite$, stRead$, nReadLen)

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

nSlaveAddr — *byVal nSlaveAddr AS INTEGER*
This is the address of the slave in range 0 to 127.

stWrite$* — *byRef stWrite$ AS STRING
This string contains the data that must be written first. If the length of this string is 0 then the write phase is bypassed.

stRead$* — *byRef stRead$ AS STRING
This string will be written to with data read from the slave if and only if nReadLen is not 0.

nReadLen — *byRef nReadLen AS INTEGER*
On entry this variable contains the number of bytes to be read from the slave and on exit will contain the actual number that were actually read. If the entry value is 0, then the read phase will be skipped.

Interactive Command: No

Related Commands: I2COPEN, I2CCLOSE, I2CWRITEREAD$, I2CWRITEREG8, I2CWRITEREG16, I2CWRITEREG32, I2CREADREG8, I2CREADREG16, I2CREADREG32

```
//Example :: I2cWriteRead.sb (See in BL600CodeSnippets.zip)

//**Please ensure that nSlaveAddr is the slave address of your I2C peripheral**
DIM rc
DIM handle
DIM nSlaveAddr
DIM stWrite$, stRead$, nReadLen

rc=I2cOpen(100000,0,handle)
IF rc!= 0 THEN
  PRINT "\nFailed to open I2C interface with error code ";integer.h' rc
ELSE
  PRINT "\nI2C open success"
ENDIF

//Write 2 bytes and read 0
nSlaveAddr=0x6f :  stWrite$ = "\34\35" : stRead$="" : nReadLen = 0
rc = I2cWriteRead(nSlaveAddr, stWrite$, stRead$, nReadLen)
IF rc!= 0 THEN
  PRINT "\nFailed to WriteRead "; integer.h'rc
ELSE
  PRINT "\nWrite = ";StrHexize$(stWrite$);" Read = ";StrHexize$(stRead$)
ENDIF

//Write 3 bytes and read 4
nSlaveAddr=0x6f :  stWrite$ = "\34\35\43" : stRead$="" : nReadLen = 4
rc = I2cWriteRead(nSlaveAddr, stWrite$, stRead$, nReadLen)
IF rc!= 0 THEN
  PRINT "\nFailed to WriteRead "; integer.h'rc
ELSE
  PRINT "\nWrite = ";StrHexize$(stWrite$);" Read = ";StrHexize$(stRead$)
ENDIF

//Write 0 bytes and read 8
nSlaveAddr=0x6f :  stWrite$ = "" : stRead$="" : nReadLen = 8
```

```
rc = I2cWriteRead(nSlaveAddr, stWrite$, stRead$, nReadLen)
IF rc!= 0 THEN
  PRINT "\nFailed to WriteRead "; integer.h'rc
ELSE
  PRINT "\nWrite = ";StrHexize$(stWrite$);" Read = ";StrHexize$(stRead$)
ENDIF

I2cClose(handle)  //close the port
```

Expected Output:

```
I2C open success
Write = 3435 Read =
Write = 343543 Read = 1042D509
Write =  Read = 2B322380ED236921
```

I2CWRITEREAD is a core function.

SPI Interface

This section describes all the events and routines used to interact with the SPI peripheral available on the platform.

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

The three signal lines are called SCK, MOSI and MISO, where the first two are outputs and the last is an input.

A very good introduction to SPI can be found at http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus and the reader is encouraged to refer to it before using the api described in this section.

It is possible to configure the interface to operate in any one of the 4 modes defined for the SPI bus which relate to the phase and polarity of the SCK clock line in relation to the data lines MISO and MOSI. In addition, the clock frequency can be configured from 125,000 to 8000000 and it can be configured so that it shifts data in/out most significant bit first or last.

Note: A dedicated SPI Chip Select (CS) line is not provided and it is up to the developer to dedicate any spare gpio line for that function if more than one SPI slave is connected to the bus. The SPI interface in this module assumes that prior to calling SPIREADWRITE, SPIREAD or SPIWRITE functions the slave device has been selected via the appropriate gpio line.

SPI Events

The API provided in the module is synchronous and so there is no requirement for events.

SpiOpen

FUNCTION

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

SPIOPEN (nMode, nClockHz, nCfgFlags, nHande)

Returns: INTEGER Indicates success of command:

0	Opened successfully
0x5200	Driver not found
0x5207	Driver already open
0x5225	Invalid Clock Frequency Requested
0x521D	Driver resource unavailable
0x522B	Invalid mode

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

nMode — *byVal nMode AS INTEGER*
This is the mode, as in phase and polarity of the clock line, that the interface shall operate at. Valid values are 0 to 3 inclusive:

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

nClockHz — *byVal nClockHz AS INTEGER*
This is the clock frequency to use, and can be one of 125000, 250000, 500000, 1000000, 2000000, 4000000 or 8000000.

nCfgFlags — *byVal nCfgFlags AS INTEGER*
This is a bit mask used to configure the SPI interface. All unused bits are allocated as *for future use* and MUST be set to 0. Used bits are as follows:-

Bit	Description
0	If set then the least significant bit is clocked in/out first.
1-31	Unused and MUST be set to 0

nHandle — *byRef nHandle AS INTEGER*
The handle for this interface will be returned in this variable if it was successfully opened. This handle is subsequently used to read/write and close the interface.

Related Commands: SPICLOSE, SPIREADWRITE, SPIWRITE, SPIREAD

SPIOPEN is a core function.

On the following page is an example which demonstrates usage of all the SPI related functions for this module.

SPI Example

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

//The SPI slave used here is the Microchip 25A512
//See http://ww1.microchip.com/downloads/en/DeviceDoc/22237C.pdf

DIM rc
DIM h    //handle
DIM rl   //readlen
DIM rd$,wr$,p$
DIM wren


//-------------------------------------------------------------
//Get eeprom Status Register
//-------------------------------------------------------------
FUNCTION EepromStatus()
  GpioWrite(13,0)
  wr$="\05\00" : rd$="" : rc=SpiReadWrite(wr$,rd$)
  GpioWrite(13,1)
ENDFUNC StrGetChr(rd$,1)

//-------------------------------------------------------------
//Wait for WR bit in status flag to reset
//-------------------------------------------------------------
SUB WaitWrite()
  DO
    GpioWrite(13,0)
    wr$="\05\00" : rd$="" : rc=SpiReadWrite(wr$,rd$)
    GpioWrite(13,1)
  UNTIL ((StrGetChr(rd$,1)&1)==0)
ENDSUB

//-------------------------------------------------------------
//Enable writes in eeprom
//-------------------------------------------------------------
SUB EnableWrite()
  GpioWrite(13,0)
  wr$="\06" : rd$="" : rc=SpiWrite(wr$)
  GpioWrite(13,1)
ENDSUB

//-------------------------------------------------------------
// Configure the Chip Select line using SIO13 as an output
//-------------------------------------------------------------
rc= GpioSetFunc(13,2,1)
// ensure CS is not enabled
GpioWrite(13,1)

//-------------------------------------------------------------
//open the SPI
//-------------------------------------------------------------
rc=SpiOpen(0,125000,0,h)


//.............................................................
//Write DEADBEEFBAADC0DE 8 bytes to memory at location 0x0180
//.............................................................
EnableWrite()
```

```
wr$="\02\01\80\DE\AD\BE\EF\BA\AD\C0\DE"
PRINT "\nWriting to location 0x180  ";StrHexize$(wr$)
GpioWrite(13,0)
rc=SpiWrite(wr$)
GpioWrite(13,1)
WaitWrite()

//..............................................................
//Read from written location
//..............................................................
wr$="\03\01\80\00\00\00\00\00\00\00\00"
rd$=""
GpioWrite(13,0)
rc=SpiReadWrite(wr$,rd$)
GpioWrite(13,1)
PRINT "\nData at location 0x0180 is ";StrHexize$(rd$)

//..............................................................
//Prepare for reads from location 0x180 and then read 4 and then 8 bytes
//..............................................................
wr$="\03\01\80"
GpioWrite(13,0)
rc=SpiWrite(wr$)
rd$=""
rc=SpiRead(rd$,4)
PRINT "\nData at location 0x0180 is ";StrHexize$(rd$)
rd$=""
rc=SpiRead(rd$,8)
GpioWrite(13,1)
PRINT "\nData at location 0x0184 is ";StrHexize$(rd$)


//--------------------------------------------------------------
//close the SPI
//--------------------------------------------------------------
SpiClose(h)
```

Expected Output:

```
Writing to location 0x180  020180DEADBEEFBAADC0DE
Data at location 0x0180 is 000000DEADBEEFBAADC0DE
Data at location 0x0180 is DEADBEEF
Data at location 0x0184 is BAADC0DEFFFFFFFF
```

SpiClose

SUBROUTINE

This subroutine is used to close a SPI port which had been opened with SPIOPEN.

This routine is safe to call if it is already closed.

SPICLOSE(handle)

Exceptions	▪	Local Stack Frame Underflow

- Local Stack Frame Overflow

Arguments:

handle — *byVal handle AS INTEGER*
This is the handle value that was returned when SPIOPEN was called which identifies the SPI interface to close.

Interactive Command: No

Related Commands: SPICLOSE, SPIREADWRITE, SPIWRITE, SPIREAD

```
//Example :: See SpiExample.sb
```

SPICLOSE is a core subroutine.

SpiReadWrite

FUNCTION

This function is used to write data to a SPI slave and at the same time read the same number of bytes back. Every 8 clock pulses result in one byte being written and one being read.

Note a 'handle' parameter is NOT required as this function is used to interact with the main interface. In the future, a new version of this function will be made available if more than one SPI interface is made available.

SPIREADWRITE(stWrite$, stRead$)

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

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

Arguments:

stWrite$* — *byRef stWrite$ AS STRING
This string contains the data that must be written.

stRead$* — *byRef stRead$ AS STRING
While the data in stWrite$ is being written, the slave sends data back and that data is stored in this variable. Note that on exit this variable will contain the same number of bytes as stWrite$.

Interactive Command: No

Related Commands: SPICLOSE, SPIREADWRITE, SPIWRITE, SPIREAD

```
//Example :: See SpiExample.sb
```

SPIWRITEREAD is a core function.

SpiWrite

FUNCTION

This function is used to write data to a SPI slave and any incoming data will be ignored.

Note a 'handle' parameter is NOT required as this function is used to interact with the main interface. In the future, a new version of this function will be made available if more than one SPI interface is made available.

SPIWRITE(stWrite$)

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

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

Arguments:

stWrite$* *byRef stWrite$ AS STRING
This string contains the data that must be written.

Interactive Command: No

Related Commands: SPICLOSE, SPIREADWRITE, SPIWRITE, SPIREAD

```
//Example :: See SpiExample.sb
```

SPIWRITE is a core function.

SpiRead

FUNCTION

This function is used to read data from a SPI slave.

Note a 'handle' parameter is NOT required as this function is used to interact with the main interface. In the future, a new version of this function will be made available if more than one SPI interface is made available.

SPIREAD(stRead$, nReadLen)

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow

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

Arguments:

stRead$* *byRef stRead$ AS STRING
This string will contain the data that is read from the slave.

nReadLen *byVal nReadLen AS INTEGER*
This specifies the number of bytes to be read from the slave.

Interactive Command: No

Related Commands: SPICLOSE, SPIREADWRITE, SPIWRITE, SPIREAD

```
//Example :: See SpiExample.sb
```

SPIREAD is a core function.

Cryptographic Functions

This section describes cryptographic functions that can be used to encrypt and decrypt data, over and above and in addition to any crypting applied at the transport layer.

In cryptography there are many algorithms which could be symmetric or assymetric. Each function described in this section will detail the type and modes catered for.

AesSetKeyIV

FUNCTION

This function is used to initialise a context for AES encryption and decription using the mode, key and initialisation vector supplied. The modes that are catered for is EBC and CBC with a block size of 128 bits.

AESSETKEYIV (mode, blockSize,key$, initVector$)

Returns: INTEGER

Will be 0x0000 if the context was created successfully. Otherwise an appropriate resultcode will be returned which will convery the reason it failed.

Arguments:

mode — BYVAL mode AS INTEGER
This shall be as follows:-
0x100 for EBC mode
0x101 for EBC mode but data is XORed with same initVector$ everytime
0x200 for CBC mode

blockSize — BYVAL blockSize AS INTEGER
Must always be set to16, which is the size in bytes.

*key$* — BYREF *key*$ AS STRING
This string specifies the key to use for encryption and decryption and MUST be exactly 16 bytes long

*initVector$* — BYREF *initVector*$ AS STRING
If mode is 0x101 or 0x200, then this string MUST be supplied and it shall be 16 bytes long. It is left to the caller to ensure a sensible value is supplied. For example, providing a string where all bytes is 0 is going to be of no value.

Interactive Command: NO

```
//Example :: AesSetKeyIv.sb (See in BL600CodeSnippets.zip)
DIM key$, initVector$
DIM rc
//Create context for EBC mode, 128 bit
key$="\00\01\02\03\04\05\06\07\08\09\0A\0B\0C\0D\0E\0F"
initVector$=""   //EBC does not require initialisation vector
rc=AesSetKeyIv(0x100,16,key$,initVector$)
IF rc==0 THEN
  PRINT "\nEBC context created successfully"
ELSE
```

```
  PRINT "\nFailed to create EBC context"
ENDIF
//Create context for EBC mode with XOR, 128 bit
key$="\00\01\02\03\04\05\06\07\08\09\0A\0B\0C\0D\0E\0F"
initVector$="\FF\01\FF\03\FF\05\FF\07\FF\09\FF\0B\FF\0D\FF\0F"
rc=AesSetKeyIv(0x101,16,key$,initVector$)
IF rc==0 THEN
  PRINT "\nEBC-XOR context created successfully"
ELSE
  PRINT "\nFailed to create EBC-XOR context"
ENDIF
//Create context for CBC mode, 128 bit
key$="\00\01\02\03\04\05\06\07\08\09\0A\0B\0C\0D\0E\0F"
initVector$="\FF\01\FF\03\FF\05\FF\07\FF\09\FF\0B\FF\0D\FF\0F"
rc=AesSetKeyIv(0x200,16,key$,initVector$)
IF rc==0 THEN
  PRINT "\nCBC context created successfully"
ELSE
  PRINT "\nFailed to create CBC context"
ENDIF
```

Expected Output:

AESSETKEYIV is a core language function.

AesEncrypt

FUNCTION

This function is used to encrypt a string up to 16 bytes long using the context that was precreated using the most recent call of the function AesSetKeyIv.

For all modes, AesSetKeyIV is called only once which means in CBC mode the cyclic data is kept in the context object that was created by AesSetKeyIV.

On the BL600, which has AES 128 **encryption** hardware assist, the function has been timed to take roughly 125 microseconds.

AESENCRYPT (inData$,outData$)

Returns: INTEGER

Will be 0x0000 if the data was encrypted successfully. Otherwise an appropriate resultcode will be returned which will convey the reason it failed. ALWAYS check this.

Arguments:

inData$* BYREF *inData$ AS STRING
This string is up to 16 bytes long and should contain the data to encrypt

outData$* BYREF *outData$ AS STRING
On exit, if the function was successful, then this string will contain the encrypted cypher data. If unsuccessful, then string will be 0 bytes long.

Interactive Command: NO

```
//Example :: AesEncrypt.sb (See in BL600CodeSnippets.zip)
DIM key$, initVector$
DIM inData$, outData$
DIM rc
//Create context for EBC mode, 128 bit
key$="\00\01\02\03\04\05\06\07\08\09\0A\0B\0C\0D\0E\0F"
initVector$=""   //EBC does not require initialisation vector
rc=AesSetKeyIv(0x100,16,key$,initVector$)
IF rc==0 THEN
  PRINT "\nEBC context created successfully"
ELSE
  PRINT "\nFailed to create EBC context"
ENDIF
inData$="303132333435363738393A3B3C3D3E3F"
inData$=StrDehexize$(inData$)
rc=AesEncrypt(inData$,outData$)
IF rc==0 THEN
  PRINT "\nEncrypt OK"
ELSE
  PRINT "\nFailed to encrypt"
ENDIF
PRINT "\ninData  = "; strhexize$(inData$)
PRINT "\noutData = "; strhexize$(outData$)
```

Expected Output:

AESENCRYPT is a core language function.

AesDecrypt

FUNCTION

This function is used to decrypt a string of exactly 16 bytes using the context that was precreated using the most recent call of the function AesSetKeyIv.

For all modes, AesSetKeyIV is called only once which means in CBC mode the cyclic data is kept in the context object that was created by AesSetKeyIV.

On the BL600, which does **not** have AES 128 decryption hardware assist, the function has been timed to take roughly 570 microseconds.

AESDECRYPT (inData$,outData$)

Returns: INTEGER

Will be 0x0000 if the data was decrypted successfully. Otherwise an appropriate resultcode will be returned which will convey the reason it failed. ALWAYS check this.

Arguments:

inData$* — BYREF *inData$ AS STRING
This string MUST be eactly 16 bytes long and should contain the data to decrypt

outData$* — BYREF *outData$ AS STRING
On exit, if the function was successful, then this string will contain the decrypted plaintext data. If unsuccessful, then string will be 0 bytes long.

Interactive Command: NO

```
//Example :: AesDecrypt.sb (See in BL600CodeSnippets.zip)
DIM key$, initVector$
DIM inData$, outData$, c$[3]
DIM rc
//Create context for CBC mode, 128 bit
key$="\00\01\02\03\04\05\06\07\08\09\0A\0B\0C\0D\0E\0F"
initVector$="\FF\01\FF\03\FF\05\FF\07\FF\09\FF\0B\FF\0D\FF\0F"
rc=AesSetKeyIv(0x200,16,key$,initVector$)
IF rc==0 THEN
  PRINT "\nCBC context created successfully"
ELSE
  PRINT "\nFailed to create EBC context"
ENDIF
//encrypt some data
inData$="303132333435363738393A3B3C3D3E3F"
inData$=StrDehexize$(inData$)
rc=AesEncrypt(inData$,c$[0])
IF rc==0 THEN
  PRINT "\nEncrypt OK"
ELSE
  PRINT "\nFailed to encrypt"
ENDIF
PRINT "\ninData  = "; strhexize$(inData$)
PRINT "\noutData = "; strhexize$(c$[0])
//encrypt same data again
rc=AesEncrypt(inData$,c$[1])
IF rc==0 THEN
  PRINT "\nEncrypt OK"
ELSE
  PRINT "\nFailed to encrypt"
ENDIF
PRINT "\ninData  = "; strhexize$(inData$)
```

```
PRINT "\noutData = "; strhexize$(c$[1])
//ecrypt same data again
rc=AesEncrypt(inData$,c$[2])
IF rc==0 THEN
  PRINT "\nEncrypt OK"
ELSE
  PRINT "\nFailed to encrypt"
ENDIF
PRINT "\ninData  = "; strhexize$(inData$)
PRINT "\noutData = "; strhexize$(c$[2])
//Rereate context for CBC mode, 128 bit
key$="\00\01\02\03\04\05\06\07\08\09\0A\0B\0C\0D\0E\0F"
initVector$="\FF\01\FF\03\FF\05\FF\07\FF\09\FF\0B\FF\0D\FF\0F"
rc=AesSetKeyIv(0x200,16,key$,initVector$)
IF rc==0 THEN
  PRINT "\nCBC context created successfully"
ELSE
  PRINT "\nFailed to create EBC context"
ENDIF
//now decrypt the data
rc=AesDecrypt(c$[0],outData$)
IF rc==0 THEN
  PRINT "\n**Decrypt OK**"
ELSE
  PRINT "\nFailed to decrypt"
ENDIF
PRINT "\ninData  = "; strhexize$(c$[0])
PRINT "\noutData = "; strhexize$(outData$)
//now decrypt the data
rc=AesDecrypt(c$[1],outData$)
IF rc==0 THEN
  PRINT "\n**Decrypt OK**"
ELSE
  PRINT "\nFailed to decrypt"
ENDIF
PRINT "\ninData  = "; strhexize$(c$[1])
PRINT "\noutData = "; strhexize$(outData$)
//now decrypt the data
rc=AesDecrypt(c$[2],outData$)
IF rc==0 THEN
  PRINT "\n**Decrypt OK**"
ELSE
  PRINT "\nFailed to decrypt"
ENDIF
PRINT "\ninData  = "; strhexize$(c$[2])
PRINT "\noutData = "; strhexize$(outData$)
```

Expected Output:

AESDECRYPT is a core language function.

File I/O Functions

A portion of module's flash memory is dedicated to a file system which is used to store smartBASIC applications and user data files.

Due to the internal requirement, set by the smartBASIC runtime engine (because applications are interpreted in-situ), compiled application files have to be stored entirely in one continguous memory block. This means the file system is currently restricted so that it is NOT possible for an application to open a file and then write to it. To store application data so that they are non-volatile, use the functions described in the section "Non-Volatile Memory Management Routines"

This means any and all user data files need to be preloaded using the commands:-

AT+FOW
AT+FWR or AT+FWRH
AT+FCL

which are described in the section "Interactive Mode Commands".

The utility UwTerminal helps with downloading such files, but not strictly required.

This section describes all the functions that are available to an application to interact with data files in read mode.

With the use of READ, FTELL and FSEEK downloading configuration files (like say digital certificates) can be a very useful and convenient way of making an app behave in custom manner from data derived from these data files as demonstrated by the example application listed in the description of FOPEN.

FOPEN

FUNCTION

This function is used to open a file in mode specified by the 'mode$' string parameter. When the file is opened the file pointer is set to 0 which effectively means that a read operation will happen from the beginning of the file and then after the read the file pointer will be adjusted to offset equal to the size of the read.

Function FSEEK is provided to move that file pointer to an offset relative to the beginning, or current position or from the end of the file and function FTELL is provided to obtain the current position as an offset from the beginning of the file.

FOPEN (filename$, mode$)

Returns: INTEGER

A non-zero integer representing an opaque handle to the file that was opened. If the file failed to open, like for example because the mode specified writing to the file which is not allowed on certain platforms, then the returned value will be 0.

Arguments:

*filename$* **BYREF filename$ AS STRING**
This string specifies the name of the file to open.

*mode$* **BYVAL mode$ AS STRING**
Must always be set to "r"
This string specifies the mode the file should be opened, and for this module, as only reading is allowed must always be specified as "r".

Interactive Command: NO

```
//Example :: FileIo.sb (See in BL600CodeSnippets.zip)
//
// First download a file into the module by submitting the following
// commands manually (wait for a 00 response after each command) :-
//
//     at+fow "myfile.dat"
//     at+fwr "Hello"
//     at+fwr " World. "
//     at+fwr " This is something"
//     at+fwr " in a file which we can read"
//     at+fcl
//
// You can check you have the file in the file system by submitting
// the command AT+DIR and you should see myfile.dat listed
//
DIM handle,fname$,flen,frlen,data$,fpos,rc

fname$="myfile.dat" : handle = fopen(fname$,"r")
IF handle != 0 THEN
```

```
  //determine the size of the file
  flen = filelen(handle)
  print "\nThe file is ";flen;" bytes long"
  //get the current position in the file (should be 0)
  rc = ftell(handle,fpos)
  print "\nCurrent position is ";fpos
  //read the first 11 bytes from the file
  frlen = fread(handle,data$,11)
  print "\nData from file is : ";data$
  //get the current position in the file (should be 11)
  rc = ftell(handle,fpos)
  print "\nCurrent position is ";fpos
  //reposition the file pointer to 6 so that we can read 5 bytes again
  rc = fseek(handle,6,0)
  //get the current position in the file
  rc = ftell(handle,fpos)
  //read 5 bytes
  frlen = fread(handle,data$,5)
  print "\nData from file is : ";data$
  //reposition to the start of 'is'
  rc = fseek(handle,19,0)
  //read until a 'w' is encountered : w = ascii 0x77
  frlen = freaduntil(handle,data$,0x77,32)
  print "\nData from file is : ";data$
  //finally close the file, which on exit will set the handle to 0
  fclose(handle)
ELSE
  print "\nFailed to open file ";fname$
ENDIF
```

FOPEN is a core language function.

FCLOSE

FUNCTION

This function is used to close a file previously opened with FOPEN. It takes a handle parameter as a reference and will on exit set that handle to 0 which signifies an invalid file handle.

FCLOSE (fileHandle)

Returns: N/A as it is a subroutine

Arguments:

fileHandle **BYREF fileHandle AS INTEGER**
The handle of the file to be closed. On exit it will be set to 0

Interactive Command: NO

```
//See the full and detailed example in the FOPEN section
```

FCLOSE is a core language function.

FREAD

FUNCTION

This function is used to read X bytes of data from a file previously opened with FOPEN and will return the actual number of bytes read.

FREAD (fileHandle, data$, maxReadLen)

Returns: INTEGER

The actual number of bytes read from the file. Will be 0 if read from end of file is attempted.

Arguments:

fileHandle **BYVAL fileHandle AS INTEGER**
The handle of the file to be read from

*data$* **BYREF data$ AS STRING**
The data read from file is returned in this string

maxReadLen **BYVAL maxReadLen AS INTEGER**
The max number of bytes to read from the file

Interactive Command: NO

```
//See the full and detailed example in the FOPEN section
```

FREAD is a core language function.

FREADUNTIL

FUNCTION

This function is used to read X bytes or until (and including) a match byte is encountered, whichever comes earlier, from a file previously opened with FOPEN and will return the actual number of bytes read (includes the match byte if encountered).

FREADUNTIL (fileHandle, data$, matchByte, maxReadLen)

Returns: INTEGER

The actual number of bytes read from the file. Will be 0 if read from end of file is attempted.

Arguments:

fileHandle	BYVAL fileHandle AS INTEGER The handle of the file to be read from
data$	BYREF data$ AS STRING The data read from file is returned in this string
matchByte	BYVAL matchByte AS INTEGER Read until this matching byte is encountered or the max number of bytes are read. Whichever condition is asserted first.
maxReadLen	BYVAL maxReadLen AS INTEGER The max number of bytes to read from the file

Interactive Command: NO

```
//See the full and detailed example in the FOPEN section
```

FREADUNTIL is a core language function.

FILELEN

FUNCTION

This function is used determine the total size of the file in bytes.

FILELEN (fileHandle)

Returns: INTEGER

The total number of bytes read from the file specified by the handle. Will be 0 if an invalid handle is supplied.

Arguments:

fileHandle	BYVAL fileHandle AS INTEGER The handle of a file for which the total size is to be returned.

Interactive Command: NO

```
//See the full and detailed example in the FOPEN section
```

FILELEN is a core language function.

FTELL

FUNCTION

This function is used determine the current file position in the open file specified by the handle. It will be a value from 0 to N where N is the size of the file.

FTELL (fileHandle, curPosition)

Returns: INTEGER

The total number of bytes read from the file specified by the handle. Will be 0 if an invalid handle is supplied.

Arguments:

fileHandle — BYVAL fileHandle AS INTEGER
The handle of a file for which the total size is to be returned.

curPosition — BYREF curPosition AS INTEGER
This will be updated with the current file position for the file specified by the fileHandle.

Interactive Command: NO

```
//See the full and detailed example in the FOPEN section
```

FTELL is a core language function.

FSEEK

FUNCTION

This function is used to move the file pointer of the open file specified by the handle supplied. The offset is relative to the beginning of the file or the current position or the end of the file which is specified by the 'whence' parameter.

FSEEK (fileHandle, offset, whence)

Returns: INTEGER

Will be 0 if successful

Arguments:

fileHandle — BYVAL fileHandle AS INTEGER
The handle of a file for which the file pointer is to be moved

offset — BYVAL offset AS INTEGER
This is the offset relative to the position defined by the 'whence' parameter.

whence — BYVAL whence AS INTEGER
This parameter specifies from which position the offset is to be calculated. It shall be 1 to specify from the current position, 2 from the end of the while and then for all other values from the beginning of the file.
When the start position is 'end of file' then a positive 'offset' value is used to calculate backwards from the end of file. Hence supplying a negative value has no meaning.

Interactive Command: NO

```
//See the full and detailed example in the FOPEN section
```

FSEEK is a core language function.

Non-Volatile Memory Management Routines

These commands provide access to the non-volatile memory of the module and provide the ability to use non-volatile storage for individual records.

NvRecordGet

FUNCTION

NVRECORDGET reads the value of a user record as a string from non-volatile memory.

NVRECORDGET (*recnum, strvar$*)

Returns: INTEGER, the number of bytes that were read into ***strvar$***. A negative value is returned if an error was encountered:

Error	Description
-1	***Recnum*** is not in valid range or is unrecognised.
-2	Failed to determine the size of the record.
-3	The raw record is less than 2 bytes long (possible flash corruption).
-4	Insufficient RAM.
-5	Failed to read the data record.

Exceptions:

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

recnum *byVal recnum AS INTEGER*
The record number to be read, in the range 1 to n, where n is the maximum number of records allowed by the specific module.

strvar$* *byRef strvar$ AS STRING
The string variable that will contain the data read from the record.

Interactive Command: NO

```
//Example :: NvRecordGet.sb (See in BL600CodeSnippets.zip)
DIM r$
PRINT NvRecordGet(100,r$);" bytes read"
PRINT "\n";r$
```

Expected Output (When no data present in record):

```
0 bytes read
```

NVRECORDGET is a module function.

NvRecordGetEx

FUNCTION

NVRECORDGETX reads the value of a user record as a string from non-volatile memory and if it does not exist or an error occurred, then the specified default string is returned.

NVRECORDGETEX (*recnum, strvar$, strdef*)

Returns: INTEGER, the number of bytes that are read into ***strvar$***.

Exceptions
- Local Stack Frame Underflow
- Local Stack Frame Overflow
- Out of Memory

Arguments:

recnum — *byVal recnum AS INTEGER*
The record number that is to be read, in the range *1* to *n*, where n is the maximum number of records allowed by the specific module.

strvar$* — *byRef strvar$ AS STRING
The string variable that will contain the data read from the record.

strdef$* — *byVal strdef$ AS STRING
The string variable that will supply the default data if the record does not exist.

Interactive Command: NO

```
//Example :: NvRecordGetEx.sb (See in BL600CodeSnippets.zip)
DIM r$
PRINT NvRecordGetEx(100,r$,"default");" bytes read"
PRINT "\n";r$
```

Expected Output:

```
7 bytes read
default
```

NVRECORDGETEX is a module function.

NvRecordSet

FUNCTION

NVRECORDSET writes a value to a user record in non-volatile memory.

NVRECORDSET (*recnum, strvar$*)

Returns: INTEGER Returns the number of bytes written.

If an invalid record number is specified then -1 is returned. There are a limited number of user records which can be written to, depending on the specific module.

Exceptions:

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

recnum *byVal recnum AS INTEGER*
The record number that is to be read, in the range *1* to *n*, where n depends on the specific module.

strvar$* *byRef strvar$ AS STRING
The string variable that will contain the data to be written to the record.

WARNING: You should minimise the number of writes. Each time a record is changed, empty flash is used up. The flash filing system does not overwrite previously used locations. Eventually there will be no more free memory and an automatic defragmentation will occur. This operation takes much longer than normal as a lot of data may need to be re-written to a new flash segment. This sector erase operation could affect the operation of the radio and result in a connection loss.

Interactive Command: NO

```
//Example :: NvRecordSet.sb (See in BL600CodeSnippets.zip)
DIM w$, r$, rc : w$ = "HelloWorld"
PRINT NvRecordSet(500,w$);" bytes written\n"
PRINT NvRecordGetEx(500,r$,"default");" bytes read\n"
PRINT "\n";r$
```

Expected Output:

```
10 bytes written
10 bytes read

HelloWorld
```

NVRECORDSET is a module function.

NvCfgKeyGet

FUNCTION

NVCFGKEYGET reads the value of a built-in configuration key. See AT+CFG for a list of configuration keys.

NVCFGKEYGET (keyId*, value*)

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

Exceptions:

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

keyId — *byVal keyId AS INTEGER*
The configuration key that is to be read, in the range 1 to n, where n depends on the specific module and the full list is described for the AT+CFG command.

value — *byRef value AS INTEGER*
The integer variable that will be updated with the value of the configuration key if it exists.

Interactive Command: see AT+CFG

```
//Example :: NvCfgKeyGet.sb (See in BL600CodeSnippets.zip)
DIM v : v = 0                    //initial the value just in case the key does not
exist
PRINT NvCfgKeyGet(100,v)
PRINT "\n";v
```

Expected Output:

```
0
33031
```

NVCFGKEYGET is a module function.

NvCfgKeySet

FUNCTION

NVCFGKEYSET writes a value to a pre-existing configuration key. See AT+CFG for a complete list of configuration keys. If a key does not exist, calling this function will not create a new one. The set of configuration keys are created at firmware build time. If you wish to create a database of non-volatile configuration keys for your own application use the NvRecordSet/Get() commands.

NVCFGKEYSET (keyId*, value*)

Returns: INTEGER

An integer result code. The most typical value is 0x0000, which indicates a successful operation.

Exceptions:

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

keyId *byVal keyId AS INTEGER*
The configuration key that is to be read, in the range 1 to n, where n depends on the specific module and the full list is described for the AT+CFG command.

value *byVal value AS INTEGER*
If the configuration key 'keyId' exists then it is updated with the new value.

WARNING: You should minimise the number of writes, as each time a record is changed, empty flash is used up. The flash filing system does not overwrite previously used locations. At some point there will be no more free memory and an automatic defragmentation will occur. This operation takes much longer than normal as a lot of data may need to be re-written to a new flash segment. This sector erase operation could affect the operation of the radio and result in a connection loss.

Interactive Command: NO

```
//Example :: NvCfgKeyGet.sb (See in BL600CodeSnippets.zip)
DIM rc, r, w : w=0x8107
PRINT "\n";NvCfgKeySet(100,w)
PRINT "\n";NvCfgKeyGet(100,r)
PRINT "\nValue for 100 is ";r
```

Expected Output:

```
0
0
Value for 100 is 33031
```

NVCFGKEYSET is a module function.

Input/Output Interface Routines

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

GpioSetFunc

FUNCTION

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

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

GPIOSETFUNC (*nSigNum, nFunction, nSubFunc)*

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

Arguments:

nSigNum **byVal nSigNum AS INTEGER.**
The signal number as stated in the pinout table of the module.

nFunction **byVal nFunction AS INTEGER.**
Specifies the configuration of the GPIO pin as follows:
1 := DIGITAL_IN
2 := DIGITAL_OUT
3 := ANALOG_IN
4 := ANALOG_REF (not currently available on the BL600 module)
5 := ANALOG_OUT (not available in the BL600 module)

nSubFunc **byVal nSubFunc INTEGER.**
Configures the pin as follows:

If nFunction == DIGITAL_IN

Bits 0..3

- 1 - pull down resistor (weak)
- 2 - pull up resistor (weak)
- 3 - pull down resistor (strong)
- 4 - pull up resistor (strong)

Else :- No pull resistors

Bits 4, 5

- 4 - When in deep sleep mode, awake when this pin is LOW
- 5 - 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

Bits 0..3

- 0 = Initial output to LOW
- 1 = Initial output to HIGH
- 2 = Output will be PWM (Pulse Width Modulated Output). See function GpioConfigPW() for more configuration. The duty cycle is set using function GpioWrite().
- 3 = Output will be FREQUENCY. The frequency is set using function GpioWrite() where 0 will switch off the output any value in range 1..4000000 will generate 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=Diconnect, 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.
  For BL600 : 10 bit adc and 2/3rd scaling
  0x13 := For BL600 : 10 bit adc, 1/3rd scaling
  0x11 := For BL600 : 10 bit adc, unity scaling

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

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

Interactive Command: NO

```
//Example :: GpioSetFunc.sb (See in BL600CodeSnippets.zip)
PRINT GpioSetFunc(3,1,2)     //Digital In Gpio pin 3, weak pull up resistor
PRINT GpioSetFunc(4,3,0)     //Analog In Gpio pin 4, default settings
PRINT GpioSetFunc(5,1,0x12)  //internal pull up on gpio5 and wake from deep sleep
                             //when there is transition from high to low
```

Expected Output:

```
000
```

GPIOSETFUNC is a Module function.

GpioConfigPwm

FUNCTION

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

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

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

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

The nMaxPeriodus can also be viewed as defining the resolution of the PWN output in the sense that the duty cycle can be varied from 0 to nMaxPeriodus. 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, a frequency of 2Khz etc.

On exit the function will return with the actual frequency in the nMinFreqHz parameter.

GPIOCONFIGPWM (*nMinFreqHz, nMaxPeriodus)*

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

nMinFreqHz **byRef nMinFreqHz AS INTEGER.**
On entry this variable contains the minimum frequency desired for the PWM output. On exit, if successful, it contains the actual frequency of the PWM output.

nMaxPeriodus **byVal *nMaxPeriodus* INTEGER.**
This specifies the duty cycle resolution and the value to set to get a 100% duty cycle.

Interactive Command: NO

```
// Example :: GpioConfigPWM() (See in BL600CodeSnippets.zip)
DIM rc
DIM nFreqHz, nMaxValUs
// we want a minimum frequency of 500Hz so that we can use a 100Hz low pass filter to
// create an analogue output which has a 100Hz bandwidth
nFreqHz = 500
// we want a resolution of 1:1000 in the generated analogue output
nMaxValUs = 1000

PRINT GpioConfigPWM(nFreqHz,nMaxValUs)

PRINT "\nThe actual frequency of the PWM output is ";nFreqHz;"\n"
// now configure SIO2 pin as a PWM output
PRINT GpioSetFunc(2,2,2)    //3rd parameter is subfunc == PWM output
// Set PWM output to 0%
GpioWrite(2,0)
// Set PWM output to 50%
GpioWrite(2,(nMaxValUs/2))
// Set PWM output to 100%
GpioWrite(2,nMaxValUs)         // any value >= nMaxValUs will give a 100% duty cycle
// Set PWM output to 33.333%
// Set PWM output to 50%
GpioWrite(2,(nMaxValUs/3))
```

Expected Output:

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

GPIOCONFIGPWM is a Module function.

GpioRead

FUNCTION

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

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

GPIOREAD (*nSigNum)*

Returns: INTEGER, the value from the signal. If the signal number is invalid, then it will return value 0. For digital pins, the value will be 0 or 1. For ADC pins it will be a value in the range 0 to M where M is the max 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.

Interactive Command: NO

```
//Example :: GpioRead.sb (See in BL600CodeSnippets.zip)
DIM signal
signal = GpioRead(3)
PRINT signal
```

Expected Output:

```
1
```

GPIOREAD is a Module function.

GpioWrite

SUBROUTINE

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

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

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

GPIOWRITE (*nSigNum, nNewValue)*

Arguments:

nSigNum byVal *nSigNum* INTEGER.
The signal number as stated in the pinout table of the module.

nNewValue byVal *nNewValue* INTEGER.
The value to be written to the port. If the pin is configured as digital then 0 will clear the pin

and a non-zero value will set it.
If the pin is configured as analogue, then the value is written to the pin.
If the pin is configured as a PWM then this value sets the duty cycle.
If the pin is configured as a FREQUENCY then this value sets the frequency.

Interactive Command: NO

```
//Example :: GpioWrite.sb (See in BL600CodeSnippets.zip)
DIM rc,dutycycle,freqHz,minFreq
//set sio pin 1 to an output and initialise it to high
PRINT GpioSetFunc(1,2,0);"\n"
//set sio pin 5 to PWM output
minFreq = 500
PRINT GpioConfigPWM(minFreq,1024);"\n"   //set max pwm value/resolution to 1:1024
PRINT GpioSetFunc(5,2,2);"\n"
PRINT GpioSetFunc(7,2,3);"\n\n"          //set sio pin 7 to Frequency output

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

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

Expected Output:

```
0000
```

GPIOWRITE is a Module function.

GPIO Events

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

Tgenerate these events.

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

GpioBindEvent

FUNCTION

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

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

GPIOBINDEVENT (*nEventNum, nSigNum, nPolarity)*

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

Arguments:

nEventNum — byVal *nEventNum* INTEGER.
The GPIO event number (in the range of 0 - N) which will result in the event EVGPIOCHANn being thrown to the *smart* BASIC runtime engine.

nSigNum — byVal *nSigNum* INTEGER.
The signal number as stated in the pinout table of the module.

nPolarity — byVal *nPolarity* INTEGER.
States the transition as follows:

0 - Low to high transition
1 - High to low transition
2 - Either a low to high or high to low transition

Interactive Command: NO

```
//Example :: GpioBindEvent.sb (See in BL600CodeSnippets.zip)
FUNCTION Btn0Press()
    PRINT "\nHello"
ENDFUNC 0

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

PRINT "\nPress button 0"

WAITEVENT
```

Expected Output:

```
0
Press button 0
Hello
```

GPIOBINDEVENT is a Module function.

GpioUnbindEvent

FUNCTION

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

GPIOUNBINDEVENT (*nEventNum)*

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

Arguments:

nEventNum byVal *nEventNum* INTEGER.
The GPIO event number (in the range of 0 - N) which will be disabled so that it no longer generates run-time events in *smart* BASIC.

Interactive Command: NO

```
//Example :: GpioUnbindEvent.sb (See in BL600CodeSnippets.zip)
FUNCTION Btn0Press()
    PRINT "\nHello"
ENDFUNC 1

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

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

ONEVENT EVGPIOCHAN0 CALL Btn0Press
ONEVENT EVTMR0      CALL Tmr0TimedOut

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

WAITEVENT
```

Expected Output:

GPIOUNBINDEVENT is a Module function.

GpioAssignEvent

FUNCTION

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

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

GPIOASSIGNEVENT (*nEventNum, nSigNum, nPolarity)*

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

Arguments:

nEventNum — byVal *nEventNum* INTEGER.
The GPIO event number (in the range of 0 - N) which will result in the event EVDETECTCHANn being thrown to the *smart* BASIC runtime engine.

Note: For BL600 only nEventNum = 0 is valid

nSigNum — byVal *nSigNum* INTEGER.
The signal number as stated in the pinout table of the module.

nPolarity — byVal *nPolarity* INTEGER.
States the transition as follows:

- 0 - Low to high transition
- 1 - High to low transition
- 2 - Either a low to high or high to low transition (Not available in BL600)

Interactive Command: NO

```
//Example :: GpioAssignEvent.sb (See in BL600CodeSnippets.zip)
 FUNCTION Btn0Press()
     PRINT "\nHello"
 ENDFUNC 0

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

 PRINT "\nPress button 0"

 WAITEVENT
```

Expected Output:

```
0
Press button 0
Hello
```

GPIOASSIGNEVENT is a Module function.

GpioUnAssignEvent

FUNCTION

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

GPIOUNASSIGNEVENT (*nEventNum)*

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

Arguments:

nEventNum byVal *nEventNum* INTEGER.
The GPIO event number (in the range of 0 - N) which will be disabled so that it no longer generates run-time events in *smart* BASIC.

Note: For BL600 only nEventNum = 0 is valid.

Interactive Command: NO

```
//Example :: GpioUnAssignEvent.sb (See in BL600CodeSnippets.zip)
FUNCTION Btn0Press()
    PRINT "\nHello"
ENDFUNC 1

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

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

ONEVENT EVDETECTCHAN0 CALL Btn0Press
ONEVENT EVTMR0        CALL Tmr0TimedOut

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

Expected Output:

GPIOUNASSIGNEVENT is a Module function.

User Routines

As well as providing a comprehensive range of built-in functions and subroutines, *smart* BASIC provides the ability for users to write their own, which are referred to as 'user' routines as opposed to 'built-in' routines.

These are often used to perform frequently repeated tasks in an application and to write event and message handler functions. An application with user routines is highly modular, allowing reusable functionality.

SUB

A subroutine is a block of statements which constitute a user routine which does not return a value but takes arguments.

SUB routinename (arglist)
EXITSUB
ENDSUB

A SUB routine MUST be defined before the first instance of it being called. It is good practice to define SUB routines and functions at the beginning of an application, immediately after global variable declarations.

A typical example of a subroutine block would be

```
SUB somename(arg1 AS INTEGER arg2 AS STRING)
  DIM S AS INTEGER
  S = arg1
  IF arg1 == 0 THEN
    EXITSUB
  ENDIF
ENDSUB
```

Defining the routine name

The function name can be any valid name that is not already in use as a routine or global variable.

Defining the arglist

The arguments of the subroutine may be any valid variable types, i.e. INTEGER or STRING.

Each argument can be individually specified to be passed either as byVal or byRef. By default, simple variables (INTEGER) are passed by value (byVal) and complex variables (STRING) are passed by reference (byRef).

However, this default behaviour can be varied by using the #SET directive during compilation of an application.

```
#SET 1,0   'Default Simple arguments are BYVAL
#SET 1,1   'Default Simple arguments are BYREF
#SET 2,0   'Default Complex arguments are BYVAL
#SET 2,1   'Default Complex arguments are BYREF
```

When a value is passed by value to a routine, any modifications to that variable will not reflect back to the calling routine. However, if a variable is passed by reference then any changes in the variable will be reflected back to the caller on exit.

The SUB statement marks the beginning of a block of statements which will consist of the body of a user routine. The end of the routine is marked by the ENDSUB statement.

ENDSUB

This statement ends a block of statements belonging to a subroutine. It MUST be included as the last statement of a SUB routine, as it instructs the compiler that there is no more code for the SUB routine. Note that any variables declared within the subroutine lose their scope once ENDSUB is processed.

EXITSUB

This statement provides an early run-time exit from the subroutine.

FUNCTION

A statement beginning with this token marks the beginning of a block of statements which will consist of the body of a user routine. The end of the routine is marked by the ENDFUNC statement.

A function is a block of statements which constitute a user routine that returns a value. A function takes arguments, and can return a value of type simple or complex.

FUNCTION routinename (arglist) AS vartype
EXITFUNC arithemetic_expression_or_string_expression
ENDFUNC arithemetic_expression_or_string_expression

A function MUST be defined before the first instance of its being called. It is good practice to define subroutines and functions at the beginning of an application, immediately after variable declarations. A typical example of a function block would be:

```
FUNCTION somename(arg1 AS INTEGER arg2 AS STRING) AS INTEGER
  DIM S AS INTEGER
  S = arg1
  IF arg1 == 0 THEN
    EXITFUNC arg1*2
  ENDIF
ENDFUNC arg1 * 4
```

Defining the routine name

The function name can be any valid name that is not already in use. The return variable is always passed as byVal and shall be of type varType.

Return values are defined within zero or more optional EXITFUNC statements and ENDFUNC is used to mark the end of the block of statements belonging to the function.

Defining the return value

The variable type AS varType for the function may be explicitly stated as one of INTEGER or STRING prior to the routine name. If it is omitted, then the type is derived in the same manner as in the DIM statement for declaring variables. Hence, if function name ends with the $ character then the type will be a STRING. Otherwise, it is an INTEGER.

Since functions return a value, when used, they must appear on the right hand side of an expression statement or within a *[ ]* index for a variable. This is because the value has to be 'used up' so that the underlying expression evaluation stack does not have 'orphaned' values left on it.

Defining the arglist

The arguments of the function may be any valid variable type, i.e. INTEGER or STRING.

Each argument can be individually specified to be passed either as byVal or byRef. By default, simple variables (INTEGER) are passed byVal and complex variables (STRING) are passed byRef. However, this default behaviour can be varied by using the #SET directive.

SET 1,0 Default Simple arguments are BYVAL
SET 1,1 Default Simple arguments are BYREF
SET 2,0 Default Complex arguments are BYVAL
SET 2,1 Default Complex arguments are BYREF

Interactive Command: NO

ENDFUNC

This statement marks the end of a function declaration. Every function must include an ENDFUNC statement, as it instructs the compiler that here is no more code for the routine.

ENDFUNC arithemetic_expression_or_string_expression

This statement marks the end of a block of statements belonging to a function. It also marks the end of scope on any variables declared within that block.

ENDFUNC must be used to provide a return value, through the use of a simple or complex expression.

```
FUNCTION doThis$( byRef s$ as string) AS STRING
  S$=S$+" World"
ENDFUNC S$ + "world"

FUNCTION doThis( byRef v as integer) AS INTEGER
  v=v+100
ENDFUNC v * 3
```

EXITFUNC

This statement provides a run-time exit point for a function before reaching the ENDFUNC statement.

EXITFUNC arithemetic_expression or string expression

EXITFUNC can be used to provide a return value, through the use of a simple or complex expression. It is usually invoked in a conditional statement to facilitate an early exit from the function.

```
FUNCTION doThis$( byRef s$ as string) AS STRING
  S$=S$+" World"
  IF a==0 THEN
    EXITFUNC S$ + "earth"
  ENDIF
ENDFUNC S$ + "world"
```

6. BLE Extensions Built-in Routines

Bluetooth Low Energy (BLE) extensions are specific to the BL600 *smart*BASIC BLE module and provide a high level managed interface to the underlying Bluetooth stack.

MAC Address

To address privacy concerns there are 4 types of MAC addresses in a BLE device which can change as often as required. For example, an iPhone will regularly change it's BLE MAC address and it always exposes only it's resolvable random address.

To manage this, the usual 6 octet MAC address is qualified on-air by a single bit which qualifies the MAC address as **public** or **random**. If public, then the format is as defined by the IEEE organisation. If random, then it can be up to 3 types and this qualification is done using the upper 2 bits of the most significant byte of the random MAC address. The exact details and format of how the specification requires this to be managed is not relevant for the purpose of how BLE functionality as exposed in this module and only how various API functions in smartBASIC expect MAC addresses to be provided is detailed here.

Where a MAC address is expected as a parameter (or provided as a response) it will always be a STRING variable. This variable SHALL be 7 octets long where the first octet is the address type and the the rest of the 6 octets is the usual MAC address in big endian format (so that most significant octet of the address is at offset 1), whether public or random.

The address type is :-

0 for Public
1 for Random Static
2 for Random Private Resolvable
3 for Random Private Non Resolvable
All other values are illegal

For example, to specify a public address which has the MAC potion as 112233445566 then then STRING variable shall contain 7 octets 00112233445566 and a variable can be initialised using a constant string by escaping as follows: DIM addr : addr="\00\11\22\33\44\55\66". Likewise a static random address will be 01C12233445566 (upper 2 bits of MAC portion == 11), a resolvable random address will be 02412233445566 (upper 2 bits of MAC portion ==01) and a non-resolvable address will be 03112233445566 (upper 2 bits of MAC portion ==00).

Please note: The MAC address portion in smartBASIC is always in big endian format. If you sniff on-air packets, the same 6 packets will appear little endian format, hence reverse order – and you will NOT see 7 bytes, but a bit in the packet somewhere which specifies it to be public or random.

Events and Messages

EVBLE_ADV_TIMEOUT

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

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

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

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

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

ONEVENT  EVBLE_ADV_TIMEOUT  CALL   HndlrBleAdvTimOut

WAITEVENT
```

Expected Output:

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

EVBLEMSG

The BLE subsystem is capable of informing a *smart* BASIC application when a significant BLE related event has occurred and it does so by throwing this message (as opposed to an EVENT, which is akin to an interrupt and has no context or queue associated with it). The message contains two parameters. The first parameter, to be called **msgID** subsequently, identifies what event was triggered and the second parameter, to be called **msgCtx** subsequently, conveys some context data associated with that event. The *smart* BASIC application will have to 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 will pend all messages until they are handled. Only messages that have handlers associated with them will get inserted into the queue. This is to prevent messages that will not get handled from filling that queue. The list of triggers and associated context parameter follows:

MsgID	Description
0	A connection has been established and msgCtx is the connection handle.
1	A disconnection event and msgCtx identifies the handle.
2	Immediate Alert Service Alert. The 2nd parameter contains new alert level.
3	Link Loss Alert. The 2nd parameter contains new alert level.
4	A BLE Service Error. The 2nd parameter contains the error code.
5	Thermometer Client Characteristic Descriptor value has changed. (Indication enable state and msgCtx contains new value, 0 for disabled, 1 for enabled)

MsgID	Description
6	Thermometer measurement indication has been acknowledged.
7	Blood Pressure Client Characteristic Descriptor value has changed. (Indication enable state and msgCtx contains new value, 0 for disabled, 1 for enabled)
8	Blood Pressure measurement indication has been acknowledged.
9	Pairing in progress and display Passkey supplied in msgCtx.
10	A new bond has been successfully created.
11	Pairing in progress and authentication key requested. msgCtx is key type.
12	Heart Rate Client Characteristic Descriptor value has changed. (Notification enable state and msgCtx contains new value, 0 for disabled, 1 for enabled)
14	Connection parameters update and msgCtx is the conn handle.
15	Connection parameters update fail and msgCtx is the conn handle.
16	Connected to a bonded master and msgCtx is the conn handle.
17	A new pairing has replaced old key for the connection handle specified.
18	The connection is now encrypted and msgCtx is the conn handle.
19	The supply voltage has dropped below that specified in the most recent call of SetPwrSupplyThreshMv() and msgCtx is the current voltage in milliVolts.
20	The connection is no longer encrypted and msgCtx is the conn handle
21	The device name characteristic in the GAP service of the local gatt table has been written by the remote gatt client.

Note: Message ID 13 is reserved for future use

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

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

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

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

```
ENDFUNC 0

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

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

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

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

WAITEVENT
```

Expected Output (When connection made with BL600):

Expected Output (When no connection made):

EVDISCON

This event is thrown when there is a disconnection. It comes with 2 parameters. Parameter 1 is the connection handle and Parameter is 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' can be determined in provided in this document here.

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

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

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

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

ONEVENT EVBLEMSG    CALL HndlrBleMsg
ONEVENT EVDISCON    CALL HndlrDiscon

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

WAITEVENT
```

Expected Output:

```
Adverts Started

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

EVCHARVAL

This event is thrown when a characteristic has been written to by a remote GATT client. It comes with three parametera which are the characteristic handle that was returned when the characteristic was registered using the function BleCharCommit() the Offset and Length of the data from the characteristic value

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

DIM hMyChar,rc,at$,conHndl


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

```
     //commit service
     rc=BleSvcCommit(1,BleHandleUuid16(0x18EE),hSvc)
     //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)

     rc=BleScanRptInit(scRpt$)
     //Add 1 service handle to scan report
     //rc=BleAdvRptAddUuid16(scRpt$,hSvc,-1,-1,-1,-1,-1)
     //commit reports to GATT table - adRpt$ is empty
     rc=BleAdvRptsCommit(adRpt$,scRpt$)
     rc=BleAdvertStart(0,addr$,20,300000,0)
ENDFUNC rc


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


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


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

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


     CloseConnections()
ENDFUNC 1


ONEVENT  EVCHARVAL CALL HandlerCharVal
ONEVENT  EVBLEMSG  CALL HndlrBleMsg
```

```
IF OnStartup()==0 THEN
    rc = BleCharValueRead(hMyChar,at$)
    PRINT "\nValue of the characteristic is ";at$
    PRINT "\nSend a new value to write to the characteristic\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF


WAITEVENT

PRINT "\nExiting..."
```

Expected Output:

EVCHARHVC

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

```
// Example :: EVCHARHVC charHandle

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

EVCHARCCCD

This event is thrown when the client writes to the CCCD descriptor of a characteristic. It comes with two parameters, the first is the characteristic handle returned when the characteristic was registered with BleCharCommit() and the second is the new 16 bit value in the updated CCCD attribute.

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

DIM hMyChar,rc,at$,conHndl


//==============================================================================
// Initialise and instantiate service, characteristic, start adverts
//==============================================================================
FUNCTION OnStartup()
    DIM rc, hSvc, metaSuccess, at$, attr$, adRpt$, addr$, scRpt$
```

```
    attr$="Hi"
    DIM svcUuid : svcUuid=0x18EE
    DIM charUuid : charUuid = BleHandleUuid16(1)
    DIM charMet : charMet = BleAttrMetaData(1,1,20,0,metaSuccess)
    DIM hSvcUuid : hSvcUuid = BleHandleUuid16(svcUuid)
    DIM mdCccd : mdCccd = BleAttrMetadata(1,1,2,0,rc)   //CCCD metadata for char

    //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 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  EVGPIOCHAN1 CALL HndlrBtn0Pr


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


WAITEVENT

PRINT "\nExiting..."
```

EVCHARSCCD

This event is thrown when the client writes to the SCCD descriptor of a characteristic. It comes with two parameters, the first is the characteristic handle that was returned when the characteristic was registered using the function BleCharCommit()and the second is the new 16 bit value in the updated SCCD attribute.

The SCCD is used to manage broadcasts of characteristic values.

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

```
DIM hMyChar,rc,at$,conHndl


//==============================================================================
// Initialise and instantiate service, characteristic, start adverts
//==============================================================================
FUNCTION OnStartup()
    DIM rc, hSvc, at$, attr$, adRpt$, addr$, scRpt$
    attr$="Hi"
    DIM charMet : charMet = BleAttrMetaData(1,0,20,0,rc)
    DIM mdSccd : mdSccd = BleAttrMetadata(1,1,2,0,rc)

    //Commit svc with handle 'hSvcUuid'
    rc=BleSvcCommit(1,BleHandleUuid16(0x18EE),hSvc)
    //initialise char, read enabled, accept signed writes, broadcast capable
    rc=BleCharNew(0x03,BleHandleUuid16(1),charMet,0,mdSccd)
    //commit char initialised above, with initial value "hi" to service
'hMyChar'
    rc=BleCharCommit(hSvc,attr$,hMyChar)
    rc=BleAdvRptInit(adRpt$,0x02,0,20)
    //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 HndlrCharSccd(BYVAL charHandle, BYVAL nVal) AS INTEGER
    DIM value$
    IF charHandle==hMyChar THEN
        IF nVal & 0x01 THEN
            PRINT "\nBroadcasts have been enabled by client"
        ELSE
            PRINT "\nBroadcasts have been disabled by client"
        ENDIF
    ELSE
        PRINT "\nThis is for some other characteristic"
    ENDIF
ENDFUNC 1


ONEVENT  EVBLEMSG    CALL HndlrBleMsg
ONEVENT  EVCHARSCCD  CALL HndlrCharSccd
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 "\n--- Press button 0 to exit\n"
ELSE
    PRINT "\nFailure OnStartup"
ENDIF

WAITEVENT

PRINT "\nExiting..."
```

EVCHARDESC

This event is thrown when the client writes to writable descriptor of a characteristic which is not a CCCD or SCCD as they are catered for with their own dedicated messages. It comes with two parameters, the first is the characteristic handle that was returned when the characteristic was registered using the function BleCharCommit() and the second is an index into an opaque array of handles managed inside the characteristic handle. Both parameters are supplied as-is as the first two parameters to the function BleCharDescRead().

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

DIM hMyChar,rc,at$,conHndl, hOtherDescr

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

```
// Initialise and instantiate service, characteristic, start adverts
//==============================================================================
Sub 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=BleSvcCommit(1,BleHandleUuid16(0x18EE),hSvc)
    //initialise char, read/write enabled, accept signed writes
    rc=BleCharNew(0x4A,BleHandleUuid16(1),charMet,0,0)
    //Add another descriptor
    attr$="descr_value"
    rc=BleCharDescAdd(0x2999,attr$,BleAttrMetadata(1,1,20,0,rc))
    //commit char initialised above, with initial value "hi" to service 'hMyChar'
    attr2$="char value"
    rc=BleCharCommit(hSvc,attr2$,hMyChar)
    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
ENDSUB

//==============================================================================
// 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 HndlrCharDesc(BYVAL charHandle, BYVAL hDesc) AS INTEGER
    IF charHandle == hMyChar THEN
        PRINT "\n  ::Char Handle: ";charHandle
        PRINT "\n  ::Descriptor Index: ";hDesc
        PRINT "\nThe new descriptor value is then read using the function
BleCharDescRead()"
    ELSE
        PRINT "\nThis is for some other characteristic"
    ENDIF
ENDFUNC 1

ONEVENT  EVBLEMSG     CALL HndlrBleMsg
ONEVENT  EVCHARDESC   CALL HndlrCharDesc
ONEVENT  EVGPIOCHAN1  CALL HndlrBtn0Pr

OnStartup()
PRINT "\nWrite to the User Descriptor with UUID 0x2999"
PRINT "\n--- Press button 0 to exit\n"


WAITEVENT

PRINT "\nExiting..."
```

EVVSPRX

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

EVVSPTXEMPTY

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

EVNOTIFYBUF

When in a connection and attribute data is sent to the GATT Client using a notify procedure (for example using the function BleCharValueNotify() or when a Write_with_no_response is sent by the Gatt Client to a remote server they are stored in temporary buffers in the underlying stack. There is finite number of these temporary buffers and if they are exhausted the notify function or the write_with_no_resp command 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 smartBASIC application can handle this event to retrigger the data pump for sending data using notifies or writes_with_no_resp commands.

Note that when sending data using Indications, this event is not thrown because those messages have to be confirmed by the client which will result in a EVCHARHVC message to the smartBASIC application. Likewise, writes which are acknowledged also do not consume these buffers.

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

DIM hMyChar,rc,at$,conHndl,ntfyEnabled
```

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

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


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

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


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

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


//==============================================================================
// CCCD descriptor written handler
//==============================================================================
FUNCTION HndlrCharCccd(BYVAL charHandle, BYVAL nVal) AS INTEGER
    DIM value$,tx$
    IF charHandle==hMyChar THEN
        IF nVal THEN
            PRINT " : Notifications have been enabled by client"
            ntfyEnabled=1
            tx$="Hello"
            rc=BleCharValueNotify(hMyChar,tx$)
        ELSE
            PRINT "\nNotifications have been disabled by client"
            ntfyEnabled=0
        ENDIF
    ELSE
        PRINT "\nThis is for some other characteristic"
    ENDIF
ENDFUNC 1


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


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


WAITEVENT

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

Expected Output:

Miscellaneous Functions

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

BleTxPowerSet

FUNCTION

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

The actual value is determined by scanning through the value list (4, 0, -4, -8, -12, -16, -20, -30 , -55) so that the highest value in the list which is less than the desired value is selected, unless the desired value is less than -55 and in that case -55 will be set.

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

At any time SYSINFO(2008) will return the actual transmit power setting. Or when in command mode use the command AT I 2008.

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 scanning through the following values (4, 0, -4, -8, -12, -16, -20, -30 , -55) such that the highest value in the table which is less than the desired value is selected, unless the desired value is less than -55 and in that case -55 will be set.

Interactive Command: NO

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

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

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

Expected Output:

BLETXPOWERSET is an extension function.

BleTxPwrWhilePairing

FUNCTION

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

The actual value is determined by scanning through the value list (4, 0, -4, -8, -12, -16, -20, -30 , -55) so that the highest value in the list which is less than the desired value is selected, unless the desired value is less than -55 and in that case -55 will be set.

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

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

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. The actual value is determined by scanning through the following values (4, 0, -4, -8, -12, -16, -20, -30 , -55) such that the highest value in the table which is less than the desired value is selected, unless the desired value is less than -55 and in that case -55 will be set.

Interactive Command: NO

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

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

Expected Output:

```
```

BLETXPOWERSET is an extension function.

BleConfigDcDc

SUBROUTINE

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

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

BLECONFIGDCDC(nNewState)

Returns: None

Arguments:

nNewState byVal *nNewState* AS INTEGER.
Configure the internal DC to DC converter as follows:
0 = OFF
2 = AUTO
Any other value = ON

Interactive Command: NO

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

BLECONFIGDCDC is an extension function.

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 4 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 3 fields. The first field is 1 octet in length and contains the number of octets that follow it that belong to that record. The second field is again a single octet and is a tag value which identifies the type of payload that starts at the next octet. Hence the payload data is 'length – 1'. A special NULL AD record consists of only one field, that is, the length field, when it contains just the 00 value.

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

The reader is encouraged to refer to the "Supplement to the Bluetooth Core Specification, Version 1, Part A" which has the latest list of all AD records. You will need to register as at least an Adopter, which is free, to gain access to this information. It is available at
https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=245130

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 will result in scan and connection requests being serviced.

> **Note:** nAdvTimeout in the BL600 is rounded up to the nearest 1000 msec.

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 whitelist has been enabled but the Flags AD in the advertising report is set for Limited and/or General Discoverability. The solution is to resubmit a new advert report which is made up so that the nFlags argument to BleAdvRptInit() function is 0.
The BT 4.0 spec disallows discoverability when a whitelist is enabled during advertisement see Volume 3, Sections 9.2.3.2 and 9.2.4.2.

Arguments:

nAdvType

byVal *nAdvType* AS INTEGER.
Specifies the advertisement type as follows:

0	ADV_IND	invites connection requests
1	ADV_DIRECT_IND	invites connection from addressed device
2	ADV_SCAN_IND	invites scan request for more advert data
3	ADV_NONCONN_IND	will not accept connections / active scans

peerAddr$

byRef *peerAddr$* AS STRING
It can be an empty string that is omitted if the advertisement type is not ADV_DIRECT_IND. This is only required when nAdvType == 1. When not empty, a valid address string is exactly 7 octets long for example "\00\11\22\33\44\55\66", where the first octet is the address type and the rest of the 6 octets is the usual MAC address in big endian format (so that most significant octet of the address is at offset 1), whether public or random. The address type is 0 for Public, 1 for Random Static, 2 for Random Private Resolvable and 3 for Random Private Non Resolvable and all other values are illegal.

nAdvInterval

byVal *nAdvInterval* *AS* INTEGER.
The interval between two advertisement events (in milliseconds).
An advertisement event consists of a total of 3 packets being transmitted in the 3 advertising channels.
The range of this interval is between 20 and 10240 milliseconds.

nAdvTimeout

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 will fail. 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.

nFilterPolicy

byVal *nFilterPolicy* AS INTEGER.
Specifies the filter policy for the whitelist as follows:

0	Filter Policy	- Any
1	Filter Policy	- Filter Scan Request, Allow Connection Request from Any
2	Filter Policy	- Filter Connection Request, Allow Scan Request from Any
3	Filter Policy	- Filter Scan Request and Connection Request

If the filter policy is not 0, then the whitelist is enabled and filled with all the addresses of all the devices in the trusted device database.

Interactive Command: NO

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

FUNCTION HndlrBlrAdvTimOut()
   PRINT "\nAdvert stopped via timeout"
```

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

//The advertising interval is set to 25 milliseconds. The module will stop
//advertising after 60000 ms (1 minute)
IF BleAdvertStart(0,addr$,25,60000,0)==0 THEN
    PRINT "\nAdverts Started"
    PRINT "\nIf you search for bluetooth devices on your device, you should see
'Laird BL600'"
ELSE
    PRINT "\n\nAdvertisement not successful"
ENDIF


ONEVENT  EVBLE_ADV_TIMEOUT  CALL HndlrBlrAdvTimOut

WAITEVENT
```

Expected Output:

BLEADVERTSTART is an extension function.

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

Interactive Command: NO

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

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

FUNCTION Btn0Press()
    IF BleAdvertStop()==0 THEN
        PRINT "\nAdvertising Stopped"
    ELSE
```

```
        PRINT "\n\nAdvertising failed to stop"
    ENDIF

    PRINT "\nExiting..."
ENDFUNC 0

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

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

ONEVENT  EVBLE_ADV_TIMEOUT  CALL HndlrBlrAdvTimOut
ONEVENT  EVGPIOCHAN0        CALL Btn0Press

WAITEVENT
```

Expected Output:

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

Advertising Stopped
Exiting...
```

BLEADVERTSTOP is an extension function.

BleAdvRptInit

FUNCTION

This function is used to create and initialise an advert report with a minimal set of ADs (advertising records) and store it the string specified. It will not be 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 will contain an advertisement report.

nFlagsAD — byVal *nFlagsAD* AS INTEGER.
Specifies the flags AD bits where bit 0 is set for limited discoverability and bit 1 is set for general discoverability. Bit 2 will be forced to 1 and bits 3 & 4 will be forced to 0. Bits 3 to 7 are reserved for future use by the BT SIG and must be set to 0.

Note: If a whitelist is enabled in the BleAdvertStart() function then both Limited and General Discoverability flags MUST be 0 as per the BT 4.0 specification (Volume 3, Sections 9.2.3.2 and 9.2.4.2)

nAdvAppearance byVal *nAdvAppearance* AS INTEGER.
Determines whether the appearance advert should be added or omitted as follows:

0	Omit appearance advert
1	Add appearance advert as specified in the GAP service which is supplied via the BleGapSvcInit() function.

nMaxDevName byVal *nMaxDevName* AS INTEGER.
The n leftmost characters of the device name specified in The GAP service. If this value is set to 0 then the device name will not be included.

Interactive Command: NO

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

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

Expected Output:

```
Advert report initialised
```

BLEADVRPTINIT is an extension function.

BleScanRptInit

FUNCTION

This function is used to create and initialise a scan report which 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 will contain a scan report.

Interactive Command: NO

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

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

Expected Output:

```
Scan report initialised
```

BLESCANRPTINIT is an extension function.

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.
Uuid1	*byVal uuid1 AS INTEGER* UUID in the range 0 to FFFF, if value is outside that range it will be ignored, so set the value to -1 to have it be ignored and then all further UUID arguments will also be ignored.
Uuid2	*byVal uuid2 AS INTEGER* UUID in the range 0 to FFFF, if value is outside that range it will be ignored, so set the value to -1 to have it be ignored and then all further UUID arguments will also be ignored.
Uuid3	*byVal uuid3 AS INTEGER* UUID in the range 0 to FFFF, if value is outside that range it will be ignored, so set the value to -1 to have it be ignored and then all further UUID arguments will also be ignored.
Uuid4	*byVal uuid4 AS INTEGER* UUID in the range 0 to FFFF, if value is outside that range it will be ignored, so set the value to -1 to have it be ignored and then all further UUID arguments will also be ignored.
Uuid5	*byVal uuid5 AS INTEGER* UUID in the range 0 to FFFF, if value is outside that range it will be ignored, so set the value to -1 to have it be ignored and then all further UUID arguments will also be ignored.
Uuid6	*byVal uuid6 AS INTEGER* UUID in the range 0 to FFFF, if value is outside that range it will be ignored, so set the value to -1 to have it be ignored.

Interactive Command: NO

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

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

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

IF BleAdvRptAddUuid16(advRpt$,0x180F,0x180A, -1, -1, -1, -1)==0 THEN
```

```
    PRINT "\nUUID Service List AD added"
ENDIF

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

Expected Output:

```
UUID Service List AD added
```

BLEADVRPTADDUUID16 is an extension function.

BleAdvRptAddUuid128

FUNCTION

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

BLEADVRPTADDUUID128 (advRpt, nUuidHandle)

Returns: INTEGER, a result code. 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 say the function BleHandleUuid128() or some other function which returns one, like BleVSpOpen()

Interactive Command: NO

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

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

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

Expected Output:

```
00000
```

BLEADVRPTADDUUID128 is an extension function.

BleAdvRptAppendAD

FUNCTION

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

BLEADVRPTAPPENDAD (advRpt, nTag, stData$)

Returns: INTEGER, a result code. 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.

Interactive Command: NO

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

PRINT BleScanRptInit(scnRpt$)

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

Expected Output:

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

BLEADVRPTAPPENDAD is an extension function.

BleGetADbyIndex

FUNCTION

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

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

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 will be copied into the output data parameter ADval$.

rptData$* byREF *rptData$ AS STRING.
This parameter is a string that contains concatenated AD elements which will have been either constructed for an outgoing advert or will have been received in a scan (depends on module variant)

nADTag *byREF nADTag AS INTEGER*
When the nth index is found, the single byte tag value for that AD element is returned in this paramater

ADval$* *byREF ADval$ AS STRING
When the nth index is found, the data excluding single byte the tag value for that AD element is returned in this parameter.

Interactive Command: NO

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

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

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

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

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

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

'//Will fail because there are only 2 AD elements
rc=BleGetADbyIndex(2, fullAD$ , nADTag, ADval$)
IF rc==0 THEN
```

```
     PRINT "\nThird AD element with tag 0x"; INTEGER.H'nADTag ;" is
";StrHexize$(ADval$)
 ELSE
     PRINT "\nError reading AD: "; INTEGER.H'rc
 ENDIF
```

Expected Output:

BLEGETADBYINDEX is an extension function.

BleGetADbyTag

FUNCTION

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

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

BLEGETADBYTAG (rptData$, nADtag, ADval$)

Returns: INTEGER, a result code.
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 will have been either constructed for an outgoing advert or will have been received in a scan (depends on module variant)

nADTag — *byVAL nADTag AS INTEGER*
This parameter specifies the single byte tag value for the AD element that is to returned in the ADval$ parameter. Only the first instance can be catered for. If multiple instances are suspected then use BleAdvADbyIndex() to extract it.

ADval$* — *byREF ADval$ AS STRING
When the nth index is found, the data excluding single byte the tag value for that AT element is returned in this parameter.

Interactive Command: NO

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

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

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

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

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

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

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

Expected Output:

BLEGETADBYTAG is an extension function.

BleAdvRptsCommit

FUNCTION

This function is used to commit one or both advert reports. If the string is empty then that report type is not updated. Both strings can be empty and 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.

Interactive Command: NO

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

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

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

Expected Output:

```
000
```

BLEADVRPTSCOMMIT is an extension function.

Connection Functions

This section describes all the connection manager related routines.

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

Events & Messages

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

MsgId	Description
0	There is a connection and the context parameter contains the connection handle.
1	There is a disconnection and the context parameter contains the connection handle.
14	New connection parameters for connection associated with connection handle.
15	Request for new connection parameters failed for connection handle supplied.
16	The connection is to a bonded master
17	The bonding has been updated with a new long term key

18	The connection is encrypted
20	The connection is no longer encrypted

BleDisconnect

FUNCTION

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

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

BLEDISCONNECT (nConnHandle)

Returns: INTEGER, a result code. 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.

Interactive Command: NO

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

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

ONEVENT  EVBLEMSG           CALL HndlrBleMsg

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

WAITEVENT
```

Expected Output:

```
Adverts Started

New Connection 35800
Disconnected 3580
```

BLEDISCONNECT is an extension function.

BleSetCurConnParms

FUNCTION

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

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

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

Returns: INTEGER, a result code. 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 the actual granted connection interval.

nSlaveLatency — byVal *nSlaveLatency* AS INTEGER.
The number of connection interval polls that the peripheral may ignore. This times the connection interval shall not be greater than the link supervision timeout.

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

For example, a keyboard could set the connection interval to 1000 msec and slave latency to 0. In this case, key presses are reported to the central device once per second, a poor user experience. Instead, the connection interval can be set to e.g. 50 msec and slave latency to 19. If there are no key presses, the power use is the same as before because ((19+1) * 50) equals 1000. When a key is pressed, the peripheral knows that the central device will poll within 50 msec, so it can send that keypress with a latency of 50 msec. A connection interval of 50 and slave latency of 19 means the slave is allowed to NOT acknowledge a poll for up to 19 poll messages from the central device.

Interactive Command: NO

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

FUNCTION HandlerBleMsg(BYVAL nMsgId AS INTEGER, BYVAL nCtx AS INTEGER) AS INTEGER
```

```
    DIM intrvl,sprvTo,sLat

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

ONEVENT  EVBLEMSG    CALL HandlerBleMsg

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

WAITEVENT
```

Expected Output (Unsuccessful Negotiation):

Expected Output (Successful Negotiation):

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

BLESETCURCONNPARMS is an extension function.

BleGetCurConnParms

FUNCTION

This function gets the current connection parameters for the connection identified by the connection handle. Given there are 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 that needs to have the connection parameters changed
nIntervalUs	byRef *nIntervalUs* AS INTEGER. The current connection interval in microseconds
nSuprToutUs	byRef *nSuprToutUs* AS INTEGER. The current link supervision timeout in microseconds for the connection.
nSlaveLatency	byRef *nSlaveLatency* AS INTEGER. This is the current number of connection interval polls that the peripheral may ignore. This value multiplied by the connection interval will not be greater than the link supervision timeout. **Note**: See Note on Slave Latency.

Interactive Command: NO

See previous example

BLEGETCURCONNPARMS is an extension function.

Security Manager Functions

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

Events & Messages

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

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

To submit a passkey, use the function BLESECMNGRPASSKEY.

BleSecMngrPasskey

FUNCTION

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

BLESECMNGRPASSKEY(connHandle, nPassKey)

Returns: INTEGER, a result code. 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.

nPassKey — **byVal *nPassKey* AS INTEGER.**
This is the passkey to submit to the stack. Submit a value outside the range 0 to 999999 to reject the pairing.

Interactive Command: NO

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

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

FUNCTION HandlerBleMsg(BYVAL nMsgId AS INTEGER, BYVAL nCtx AS INTEGER) AS INTEGER
    SELECT nMsgId
        CASE 0
            connHandle = nCtx
            PRINT "\n--- Ble Connection, ",nCtx
        CASE 1
            PRINT "\n--- Disconnected ";nCtx;"\n"
            EXITFUNC 0
        CASE 11
            PRINT "\n +++ Auth Key Request, type=";nCtx
            rc=BleSecMngrPassKey(connHandle,123456)
            IF rc==0 THEN //key is 123456
```

```
                PRINT "\nPasskey 123456 was used"
            ELSE
                PRINT "\nResult Code 0x";integer.h'rc
            ENDIF
        CASE ELSE
    ENDSELECT
ENDFUNC 1

ONEVENT  EVBLEMSG  CALL HandlerBleMsg

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

WAITEVENT
```

Expected Output:

BLESECMNGRPASSKEY is an extension function.

BleSecMngrKeySizes

FUNCTION

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

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

BLESECMNGRKEYSIZES(nMinKeysize, nMaxKeysize)

Returns: INTEGER, a result code. 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.

Interactive Command: NO

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

Expected Output:

```
0
```

BLESECMNGRKEYSIZES is an extension function.

BleSecMngrIoCap

FUNCTION

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

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

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

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

You will need to 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 only (authenticated pairing)

Interactive Command: NO

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

Expected Output:

```
0
```

BLESECMNGRIOCAP is an extension function.

BleSecMngrBondReq

FUNCTION

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

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

BLESECMNGRBONDREQ (nBondReq)

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

Arguments:

nBondReq byVal *nBondReq AS* INTEGER.
0 Disable
1 Enable

Interactive Command: NO

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

Expected Output:

```
Bonding disabled
```

BLESECMNGRBONDREQ is an extension function.

BleAuthenticate

FUNCTION

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

BLEAUTHENTICATE (nConnCtx)

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

Arguments:

nConnCtx byVal *nConnCtx* AS INTEGER.
This is the context value provided in the EVBLEMSG(0) message which informed the stack that a connection had been established.

Interactive Command: NO

See example for BleDisconnect:

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

BLEAUTHENTICATE is an extension function.

GATT Server Functions

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

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

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

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

To wrap up the loose analogy, think of Service as just a carrier bag to hold a group of related Characterisics together where the printing on the carrier bag is a UUID. You will find that from a smartBASIC developer's perspective, a set of characteristics is what you will need to manage and the concept of Service is only required at GATT table creation time.

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

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

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

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

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

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

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

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

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

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

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

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

A Service is an abstraction of some collectivised functionality which, if broken down further into smaller components, would cease to provide the intended behaviour. A couple of examples in the BLE domain that have been adopted by the Bluetooth SIG are Blood Pressure Service and Heart Rate Service. Each have sub-components that map to Characteristics.

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

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

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

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

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

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

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

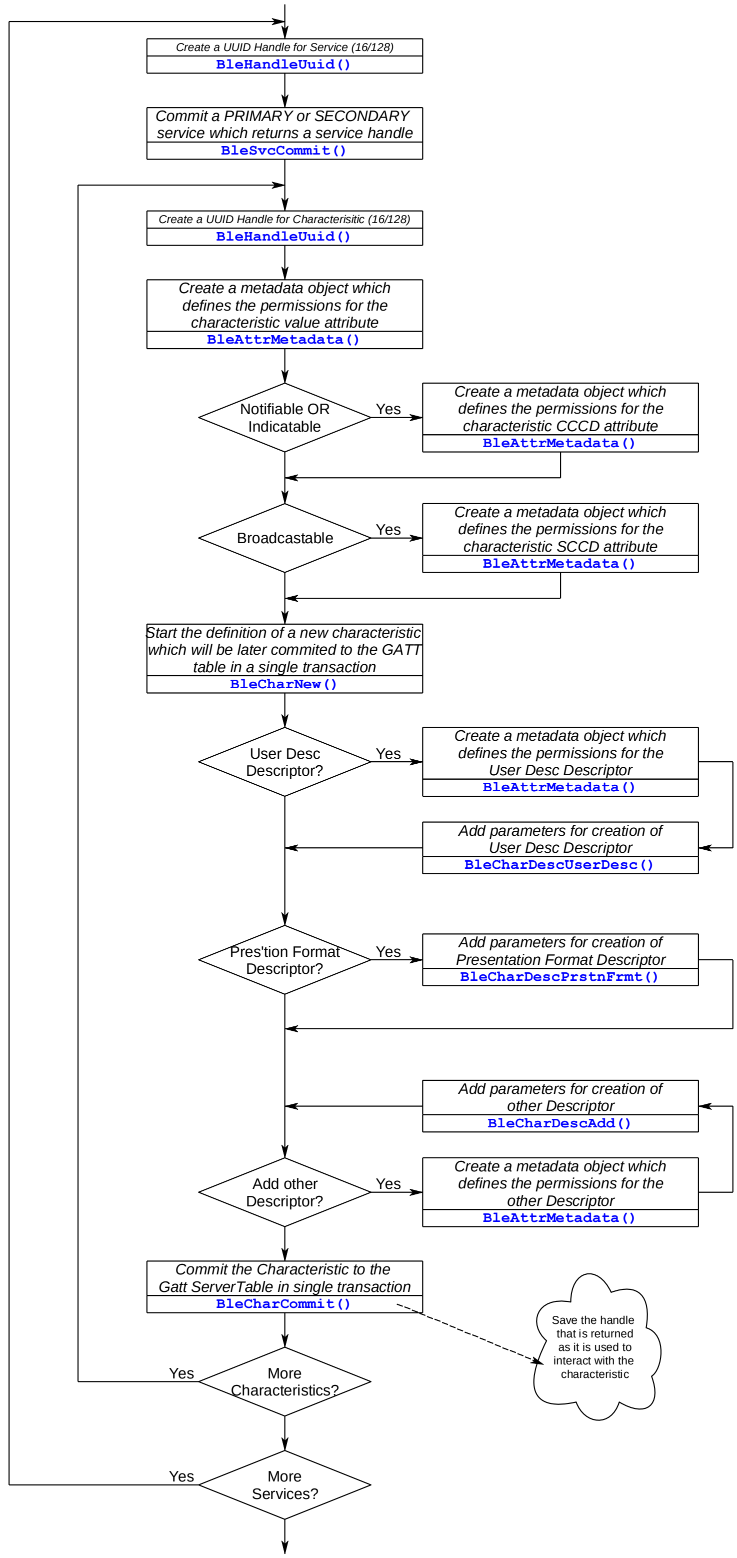
The GATT Client (remote end of the wireless connection) needs to see those carrier bags to determine the groupings and once it has identified the pots it will only need to keep a list of references to the pots it is interested in. Once that list is made at the client end, it can 'throw away the carrier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

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

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

A particular char_handle is in turn used to make something happen to the referenced characteristic (data container) using a *smart*BASIC function and conversely if data is written into that characteristic (data container), by a remote GATT Client, then an event is thrown, in the form of a message, into the *smart*BASIC runtime engine which will get processed <u>if and only if</u> 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 :: ServicesAndCharacteristics.sb (See in BL600CodeSnippets.zip)

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

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

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

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

//---Register Service 1
hUuidS1 = BleHandleUuid16(0x180D)
rc = BleSvcCommit(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)

//---Register Service 2  (can now reuse the service handle)
hUuidS2 = BleHandleUuid16(0x1856)
rc = BleSvcCommit(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)
//===The 2 services are now visible in the gatt table
```

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

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

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

  ENDIF
ENDFUNC 1

//enable characteristic value write handler
OnEvent  EVCHARVAL          call HandlerCharVal

WAITEVENT
```

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

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

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

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

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

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

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

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

Events & Messages

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

BleGapSvcInit

FUNCTION

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

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

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

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

Arguments:

deviceName — byRef *deviceName* AS STRING
The name of the device (e.g. Laird_Thermometer) to store in the 'Device Name' characteristic of the GAP service.

Note: When an advert report is created using BLEADVRPTINIT() this field is read from the service and an attempt is made to append it in the Device Name AD. If the name is too long, that function fails to initialise the advert report and a default name is transmitted. It is recommended that the device name submitted in this call be as short as possible.

nameWritable — byVal *nameWritable* AS INTEGER
If non-zero, the peer device is allowed to write the device name. Some profiles allow this to be made optional.

nAppearance — byVal *nAppearance* AS INTEGER
Field lists the external appearance of the device and updates the Appearance characteristic of the GAP service. Possible values: org.bluetooth.characteristic.gap.appearance.

nMinConnInterval — byVal *nMinConnInterval* AS INTEGER
The preferred minimum connection interval, updates the 'Peripheral Preferred Connection Parameters' characteristic of the GAP service. Range is between 7500 and 4000000 microseconds (rounded to the nearest 1250 microseconds). This must be smaller than nMaxConnInterval.

nMaxConnInterval — byVal *nMaxConnInterval* AS INTEGER
The preferred maximum connection interval, updates the 'Peripheral Preferred Connection Parameters' characteristic of the GAP service. Range is between 7500

	and 4000000 microseconds (rounded to the nearest 1250 microseconds). This must be larger than nMinConnInterval.
nSupervisionTimeout	byVal *nSupervisionTimeout* AS INTEGER The preferred link supervision timeout and updates the 'Peripheral Preferred Connection Parameters' characteristic of the GAP service. Range is between 100000 to 32000000 microseconds (rounded to the nearest 10000 microseconds).
nSlaveLatency	byVal *nSlaveLatency* AS INTEGER The preferred slave latency is the number of communication intervals that a slave may ignore without losing the connection and updates the 'Peripheral Preferred Connection Parameters' characteristic of the GAP service. This value must be smaller than (nSupervisionTimeout/ nMaxConnInterval) -1. i.e. nSlaveLatency < (nSupervisionTimeout / nMaxConnInterval) -1

Interactive Command: NO

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

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

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

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

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

Expected Output:

```
Success
```

BLEGAPSVCINIT is an extension function.

BleGetDeviceName$

FUNCTION

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

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

BLEGETDEVICENAME$ ()

Returns: STRING, the current device name in the local GATT table. It is the same as that supplied in BleGapSvcInit() if the 'nameWritable' parameter was 0, otherwise it can be different. EVBLEMSG event is thrown with 'msgid' == 21 when the GATT client writes a new value.

Arguments: None

Interactive Command: NO

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

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

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

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

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

Expected Output:

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

BLEGETDEVICENAME$ is an extension function.

BleSvcRegDevInfo

FUNCTION

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

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

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

FUNCTION

Returns: INTEGER, a result code. The 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 8 octets long, where:
Byte 0..4 := Manufacturer Identifier
Byte 5..7 := Organisationally Unique Identifier
For the special case of the string being exactly 1 character long and containing "@", the system ID is created from the MAC address if (and only if) an IEEE public address is set. If the address is the random static variety, this characteristic is omitted.

regDataList$* — byVal *regDataList$ AS STRING
The device's regulatory certification data list as defined in the specification. It can be set as an empty string to omit submission.

pnpId$* — byVal *pnpId$ AS STRING
The device's plug and play ID as defined in the specification. Can be set empty to omit submission. Otherwise, it shall be exactly 7 octets long, where:
Byte 0 := Vendor Id Source
Byte 1,2 := Vendor Id (Byte 1 is LSB)
Byte 3,4 := Product Id (Byte 3 is LSB)
Byte 5,6 := Product Version (Byte 5 is LSB)

Interactive Command: NO

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

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

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

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

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

Expected Output:

```
Success
```

BLESVCREGDEVINFO is an extension function.

BleHandleUuid16

FUNCTION

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

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

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

BLEHANDLEUUID16 (nUuid16)

Returns: INTEGER, a nonzero handle shorthand for the UUID. Zero is an invalid UUID handle.

Arguments:

nUuid16 byVal *nUuid16* AS INTEGER
nUuid16 is first bitwise ANDed with 0xFFFF and the result will be treated as an offset into the Bluetooth SIG 128 bit base UUID.

Interactive Command: NO

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

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

Expected Output:

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

BLEHANDLEUUID16 is an extension function.

BleHandleUuid128

FUNCTION

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

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

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

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

BLEHANDLEUUID128 (stUuid$)

Returns: INTEGER, A handle representing the shorthand UUID. If zero, which is an invalid UUID handle, there is either no spare RAM memory to save the 16 byte base or more than 253 custom base UUIDs have been registered.

Arguments:

stUuid$* byRef *stUuid$ AS STRING
Any 16 byte string that was generated using a UUID generation utility that has enough entropy to ensure that it is random. The first byte of the string is the MSB of the UUID – that is, big endian format.

Interactive Command: NO

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

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

Expected Output:

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

BLEHANDLEUUID128 is an extension function.

BleHandleUuidSibling

FUNCTION

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

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

BLEHANDLEUUIDSIBLING (nUuidHandle, nUuid16)

Returns: INTEGER, a handle representing the shorthand UUID and can be zero which is an invalid UUID handle, if nUuidHandle is an invalid handle in the first place.

Arguments:

nUuidHandle byVal *nUuidHandle* AS INTEGER
A handle that was previously created using either BleHandleUui16() or BleHandleUuid128().

nUuid16 byVal *nUuid16* AS INTEGER
A UUID value in the range 0 t0 65535 which will be treated as an offset into the 128 bit base UUID referenced by nUuidHandle.

Interactive Command: NO

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

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

hUuid2 = BleHandleUuidSibling(hUuid1,0x1234)
IF hUuid2 == 0 THEN
    PRINT "\nFailed to create a handle"
ELSE
    PRINT "\nHandle for custom sibling Uuid is ";integer.h';hUuid2;"(";hUuid2;")"
ENDIF
```

```
// hUuid2 now references an object which also points to
// the base uuid = ced9000066924a1287d56f2700004762 (note 0's in byte position 2/3)
// and has the offset = 0x1234
```

Expected Output:

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

BLEHANDLEUUIDSIBLING is an extension function.

BleSvcCommit

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

BleServiceNew

FUNCTION

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

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

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

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

BLESERVICENEW (nSvcType, nUuidHandle, hService)

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

Arguments:

nSvcType byVal *nSvcType* AS INTEGER
This will be 0 for a SECONDARY service and 1 for a PRIMARY service and all other values are reserved for future use and will result in this function failing with an appropriate result code.

nUuidHandle byVal *nUuidHandle* AS INTEGER
This is a handle to a 16 bit or 128 bit UUID that identifies the type of Service function provided by all the Characteristics collected under it. It will have been pre-created using one of the three functions: BleHandleUuid16(), BleHandleUuid128() or BleHandleUuidSibling()

hService byRef hService AS INTEGER
If the Service attribute is created in the GATT table then this will contain a composite handle which references the actual attribute handle. This is then subsequently used when adding Characteristics to the GATT table. If the function fails to install the Service attribute for any reason this variable will contain 0 and the returned result code will be non-zero.

Interactive Command: NO

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

#DEFINE BLE_SERVICE_SECONDARY                          0
#DEFINE BLE_SERVICE_PRIMARY                            1

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

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


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

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

Expected Output:

BLESERVICENEW is an extension function.

BleServiceCommit

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

BLESERVICECOMMIT (hService)

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

Arguments:

hService **byVal hService AS INTEGER**
This handle will have been returned from BleServiceNew().

BleSvcAddIncludeSvc

FUNCTION

Note: This function is currently not available for use on the BL600

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

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

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

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

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

BleSvcAddIncludeSvc (hService)

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

Arguments:

hService **byVal hService AS INTEGER**
This argument will contain a handle that was previously created using the function BleSvcCommit().

Interactive Command: NO

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

#define BLE_SERVICE_SECONDARY                              0
#define BLE_SERVICE_PRIMARY                                1

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

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

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

rc = BleSvcCommit(BLE_SERVICE_PRIMARY,BleHandleUuid16(0x1809),hHtsSvc)

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

BleSvcAddIncludeSvc is an extension function.

BleAttrMetadata

FUNCTION

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

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

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

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

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

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

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

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

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

This function helps write that metadata.

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

Returns: INTEGER, a 32 bit opaque data object to be used in subsequent calls when adding Characteristics to a GATT table.

Arguments:

nReadRights — byVal *nReadRights* AS INTEGER
This specifies the read rights and shall have one of the following values:
0 : No Access
1 : Open
2 : Encrypted with No Man-In-The-Middle (MITM) Protection
3 : Encrypted with Man-In-The-Middle (MITM) Protection
4 : Signed with No Man-In-The-Middle (MITM) Protection (not available)
5 : Signed with Man-In-The-Middle (MITM) Protection (not available)

Note: In early releases of the firmware, 4 and 5 are not available.

nWriteRights

byVal *nWriteRights* AS INTEGER
This specifies the write rights and shall have one of the following values:
0 : No Access
1 : Open
2 : Encrypted with No Man-In-The-Middle (MITM) Protection
3 : Encrypted with Man-In-The-Middle (MITM) Protection
4 : Signed with No Man-In-The-Middle (MITM) Protection (not available)
5 : Signed with Man-In-The-Middle (MITM) Protection (not available)

Note: In early releases of the firmware, 4 and 5 are not available.

nMaxDataLen

byVal *nMaxDataLen* AS INTEGER
This specifies the maximum data length of the VALUE attribute. Range is from 1 to 512 bytes according to the BT specification; the stack implemented in the module may limit it for early versions. At the time of writing the limit is **20 bytes**.

fIsVariableLen

byVal *fIsVariableLen* AS INTEGER
Set this to non-zero only if you want the attribute to automatically shorten it's length according to the number of bytes written by the client. For example, if the initial length is 2 and the client writes only 1 byte, then if this is 0, then only the first byte gets updated and the rest remain unchanged. If this parameter is set to 1, then when a single byte is written the attribute will shorten it's length to accommodate. If the client tries to write more bytes than the initial maximum length, then the client will get an error response.

resCode

byRef resCode AS INTEGER
This variable will be updated with result code which will be 0 if a metadata object was successfully returned by this call. Any other value implies a metadata object did not get created.

Interactive Command: NO

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

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

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

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

//Create the Characteristic object
```

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

Expected Output:

```
Success
```

BLEATTRMETADATA is an extension function.

BleCharNew

FUNCTION

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

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

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

Returns: INTEGER, a result code. The 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 will get added to the GATT table:

BIT	Description
0	Broadcast capable (Sccd Descriptor has to be present)
1	Can be read by the client
2	Can be written by the client without response
3	Can be written
4	Can be Notifiable (Cccd Descriptor has to be present)
5	Can be Indicatable (Cccd Descriptor has to be present)
6	Can accept signed writes
7	Reliable writes

nUuidHandle — byVal *nUuidHandle* AS INTEGER
This specifies the UUID that will be allocated to the Characteristic, either 16 or 128 bits. This variable is a handle, pre-created using one of the following functions: BleHandleUuid16() , BleHandleUuid128() , BleHandleUuidSibling().

mdVal — byVal *mdVal* AS INTEGER
This is the mandatory metadata that is used to define the properties of the Value attribute that will be created in the Characteristic and will have been pre-created using the help of the function BleAttrMetadata().

mdCccd byVal *mdCccd* AS INTEGER

This is an optional metadata that is used to define the properties of the CCCD Descriptor attribute that will be created in the Characteristic and will have been pre-created using the help of the function BleAttrMetadata() or set to 0 if CCCD is not to be created. If nCharProps specifies that the Characteristic is notifiable or indicatable and this value contains 0, this function will abort with an appropriate result code.

mdSccd byVal *mdSccd* AS INTEGER

This is an optional metadata that is used to define the properties of the SCCD Descriptor attribute that will be created in the Characteristic and will have been pre-created using the help of the function BleAttrMetadata() or set to 0 if SCCD is not to be created. If nCharProps specifies that the Characteristic is broadcastable and this value contains 0, this function will abort with an appropriate resultcode.

Interactive Command: NO

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

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

Expected Output:

```
New Characteristic created
```

BLECHARNEW is an extension function.

BleCharDescUserDesc

FUNCTION

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

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

BLECHARDESCUSERDESC(userDesc$, mdUser)

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

Arguments:

userDesc$* byRef *userDesc$ AS STRING
The user description string to initiliase the Descriptor with. If the length of the string exceeds the maximum length of an attribute then this function will abort with an error result code.

mdUser byVal *mdUser* AS INTEGER
This is a mandatory metadata that defines the properties of the User Description Descriptor attribute created in the Characteristic and will have been pre-created using the help of BleAttrMetadata(). If the write rights are set to 1 or greater, the attribute will be marked as writable and the client will be able to provide a user description that overwrites the one provided in this call.

Interactive Command: NO

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

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

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

Expected Output:

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

BLECHARDESCUSERDESC is an extension function.

BleCharDescPrstnFrmt

FUNCTION

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

The BT 4.0 specification states that one or more than 1 presentation format descriptor can occur in a Characteristic and that if more than one then an Aggregate Format description shall be included too.

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

> *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 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.
At the time of writing, the enumeration list is as follows:

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

nExponent — byVal *nExponent* AS INTEGER

Valid range -128 to 127. This value is used with integer data types given by the enumeration in nFormat to further qualify the value so that the actual value is:
*actual value = Characteristic Value * 10 to the power of nExponent.*

nUnit — byVal *nUnit* AS INTEGER

Valid range 0 to 65535. This value is a 16 bit UUID used as an enumeration to specify the units which are listed in the Assigned Numbers document published by the Bluetooth SIG, found at: http://developer.bluetooth.org/gatt/units/Pages/default.aspx

nNameSpace — byVal *nNameSpace* AS INTEGER

Valid range 0 to 255. The value identifies the organization, defined in the Assigned Numbers document published by the Bluetooth SIG, found at:
https://developer.bluetooth.org/gatt/Pages/GattNamespaceDescriptors.aspx

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

Interactive Command: NO

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

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

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

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

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

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

Expected Output:

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

BLECHARDESCPRSTNFRMT is an extension function.

BleCharDescAdd

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

FUNCTION

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

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

BLECHARDESCADD (nUuid16, attr$, mdDesc)

Returns: INTEGER, a result code. The 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 0x2908, 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 will be saved in the Descriptor's attribute

mdDesc — byVal *n* AS INTEGER
This is mandatory metadata that is used to define the properties of the Descriptor attribute that will be created in the Characteristic and will have been pre-created using the help of the function BleAttrMetadata(). If the write rights are set to 1 or greater, then the attribute is marked as writable and so the client will be able to modify the attribute value.

Interactive Command: NO

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

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

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

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

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

```
//++++
DIM mdChrDsc : mdChrDsc = BleAttrMetadata(1,0,20,0,metaSuccess)
DIM attr$ : attr$="some_value1"
rc=BleCharDescAdd(0x2905,attr$,mdChrDsc)

//++++
//Add the other Descriptor 0x29XX  -- second one
//++++
attr$="some_value2"
rc=rc+BleCharDescAdd(0x2906,attr$,mdChrDsc)

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

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

Expected Output:

```
Other descriptors added successfully
```

BLECHARDESCADD is an extension function.

BleCharCommit

FUNCTION

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

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

Please note that in reality, in the GATT table, when a Characteristic is registered there are actually a minimum of 2 attribute handles, one for the Characteristic Declaration and the other for the Value. However there is no need for the *smart* BASIC apps developer to ever access it, so it is not exposed. Access is not required because the Characteristic was created by the application developer and so shall already know its content – which never changes once created.

BLECHARCOMMIT (hService,attr$,charHandle)

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

Arguments:

hService byVal hService AS INTEGER
This is the handle of the service that this Characteristic shall belong to, which in turn was created using the function BleSvcCommit().

attr$* byRef *attr$ AS STRING
This string contains the initial value of the Value attribute in the Characteristic. The content of this string is copied into the GATT table and so the variable can be reused after this function returns.

charHandle byRef *charHandle* AS INTEGER
The composite handle for the newly created Characteristic is returned in this argument. It is zero if the function fails with a non-zero result code. This handle is then used as an argument in subsequent function calls to perform read/write actions, so it is must be placed in a global smartBASIC variable. When a significant event occurs as a result of action by a remote client, an event message is sent to the application which can be serviced using a handler. That message contains a handle field corresponding to this composite characteristic handle. Standard procedure is to 'select' on that value to determine which Characteristic the message is intended for.

See event messages: EVCHARHVC, EVCHARVAL, EVCHARCCCD, EVCHARSCCD, EVCHARDESC.

Interactive Command: NO

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

#DEFINE BLE_SERVICE_SECONDARY                          0
#DEFINE BLE_SERVICE_PRIMARY                            1

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


//------------------------------------------------------------------------------
//Create a Health Thermometer PRIMARY service attribute which has a uuid of 0x1809
//------------------------------------------------------------------------------
rc=BleSvcCommit(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

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

Expected Output:

```
Characteristic Commited
```

BLECHARCOMMIT is an extension function.

BleCharValueRead

FUNCTION

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

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

BLECHARVALUEREAD (charHandle,attr$)

Returns: INTEGER, a result code. The 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.

Interactive Command: NO

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

DIM hMyChar,rc, conHndl

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

    //commit service
    rc=BleSvcCommit(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$,hMyChar)
    //initialise scan report
    rc=BleScanRptInit(scRpt$)
    //Add 1 service handle to scan report
    rc=BleAdvRptAddUuid16(scRpt$,hSvc,-1,-1,-1,-1,-1)
    //commit reports to GATT table - adRpt$ is empty
    rc=BleAdvRptsCommit(adRpt$,scRpt$)
    rc=BleAdvertStart(0,addr$,150,0,0)
ENDFUNC rc


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

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


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


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

ONEVENT EVCHARVAL  CALL HndlrChar
ONEVENT EVBLEMSG  CALL HndlrBleMsg

WAITEVENT

PRINT "\nExiting..."
```

Expected Output:

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

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

BLECHARVALUEREAD is an extension function.

BleCharValueWrite

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

FUNCTION

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

BLECHARVALUEWRITE (charHandle,attr$)

Returns: INTEGER, a result code. The 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.

Interactive Command: NO

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

DIM hMyChar,rc

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

    //commit service
    rc=BleSvcCommit(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)
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$)
    IF BleCharValueWrite(hMyChar,t$)==0 THEN
        PRINT "\nNew characteristic value: ";t$
    ELSE
        PRINT "\nFailed to write new characteristic value"
    ENDIF
ENDFUNC 0


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

ONEVENT EVUARTRX     CALL HndlrUartRx
ONEVENT EVTMR0      CALL HndlrTmr0

WAITEVENT

PRINT "\nExiting..."
```

Expected Output:

BLECHARVALUEWRITE is an extension function.

BleCharValueNotify

FUNCTION

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

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

BLECHARVALUENOTIFY (charHandle,attr$)

Returns: INTEGER, a result code. The 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 containing new value to write to the characteristic and then send as a notification to the client. If there is no connection, this function fails with an appropriate result code.

Interactive Command: NO

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

DIM hMyChar,rc,at$,conHndl


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

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

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

//==============================================================================
// CCCD descriptor written handler
//==============================================================================
FUNCTION HndlrCharCccd(BYVAL charHandle, BYVAL nVal) AS INTEGER
    DIM value$
```

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


ONEVENT  EVBLEMSG    CALL HndlrBleMsg
ONEVENT  EVCHARCCCD  CALL HndlrCharCccd


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

WAITEVENT

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

Expected Output:

BLECHARVALUENOTIFY is an extension function.

BleCharValueIndicate

FUNCTION

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

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

BLECHARVALUEINDICATE (charHandle,attr$)

Returns: INTEGER, a result code. The 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 containing new value to write to the characteristic and then to send as a notification to the client. If there is no connection, this function fails with an appropriate result code.

Interactive Command: NO

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

DIM hMyChar,rc,at$,conHndl


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

    //Commit svc with handle 'hSvcUuid'
    rc=BleSvcCommit(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$,hMyChar)
    rc=BleScanRptInit(scRpt$)
    //Add 1 service handle to scan report
    rc=BleAdvRptAddUuid16(scRpt$,hSvc,-1,-1,-1,-1,-1)
    //commit reports to GATT table - adRpt$ is empty
    rc=BleAdvRptsCommit(adRpt$,scRpt$)
    rc=BleAdvertStart(0,addr$,50,0,0)
ENDFUNC rc


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

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


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

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


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


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

```
    PRINT "\nFailure OnStartup"
ENDIF


WAITEVENT

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

Expected Output:

BLECHARVALUEINDICATE is an extension function.

BleCharDescRead

FUNCTION

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

In most cases a local read will be performed when a GATT client writes to a characteristic descriptor attribute. The write event will be presented asynchronously to the *smart*BASIC application in the form of an EVCHARDESC event and so this function will most often be 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 was returned when BleCharCommit() was called and will have been supplied in the EVCHARDESC event message.

nDescHandle — byVal *nDescHandle* AS INTEGER
This is an index into an opaque array of descriptor handles inside the charHandle and will have been supplied as the second parameter in the EVCHARDESC event message.

nOffset — byVal *nOffset* AS INTEGER
This is the offset into the descriptor attribute from which the data shoud be read and copied into attr$.

nLength	byVal *nLength* AS INTEGER This is the number of bytes to read from the descriptor attribute from offset nOffset and copied into attr$.
nDescUuidHandle	byRef *nDescUuidHandle* AS INTEGER On exit this will be updated with the uuid handle of the descriptor that got updated.
attr$	byRef attr$ AS STRING On exit this string variable contains the new value from the characteristic descriptor.

Interactive Command: NO

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

DIM rc,conHndl,hMyChar

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

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

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


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

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

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

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



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

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

//wait for events and messages
WAITEVENT

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

Expected Output:

BLECHARDESCREAD is an extension function.

GATT Client Functions

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

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

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

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

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

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

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

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

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

Handle	Type	Value	Comments
0x0001	0x2800	UUID of the Service	Primary Service 1 Start
0x0002	0x2803	Properties, Value Handle, Value UUID1	Characteristic 1 Start
0x0003	Value UUID1	Value : 1 to 512 bytes	Actual data
0x0004	0x2803	Properties, Value Handle, Value UUID2	Characteristic 2 Start
0x0005	Value UUID2	Value : 1 to 512 bytes	Actual data
0x0006	0x2902	Value	Descriptor 1(CCCD)
0x0007	0x2903	Value	Descriptor 2 (SCCD)
0x0008	0x2800	UUID of the Service	Primary Service 2 Start
0x0009	0x2803	Properties, Value Handle, Value UUID3	Characteristic 1 Start
0x000A	Value UUID3	Value : 1 to 512 bytes	Actual data
0x000B	0x2800	UUID of the Service	Primary Service 3 Start
0x000C	0x2803	Properties, Value Handle, Value UUID3	Characteristic 3 Start
0x000D	Value UUID3	Value : 1 to 512 bytes	Actual data
0x000E	0x2902	Value	Descriptor 1(CCCD)
0x000F	0x2903	Value	Descriptor 2 (SCCD)
0x0010	0x2904	Value (presentation format data)	Descriptor 3
0x00111	0x2906	Value (valid range)	Descriptor 4 (Range)

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

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

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

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

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

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

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

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

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

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

Events & Messages

The nature of GATT Client operation consists of multiple queries and acting on the responses. Due to the connection intervals being vastly slower than the speed of the cpu, responses can arrive many 10s of milliseconds after the precudure was triggered, which are delivered to an app using an event or message. Since these event/messages are tightly coupled with the appropriate commands, all but one will be described when the command that triggers them is described.

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

EVGATTCTOUT event message

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

Connection Handle

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

DIM rc,conHndl

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


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

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

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

rc = OnStartup()

WAITEVENT
```

Expected Output:

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

BleGattcOpen

FUNCTION

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

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

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

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

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

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

BLEGATTCOPEN (nNotifyBufLen, nFlags)

Returns: INTEGER, a result code. 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 characterstic data. Set to 0 to use the default size.
nFlags	byVal *nFlags* AS INTEGER Bit 0 : Set to 1 to disable automatic indication confirmations if buffer is full then the Handle Value Confirmation will only be sent when BleGattcNotifyRead() is called to read the ring buffer. Bit 1..31 : Reserved for future use and must be set to 0s

Interactive Command: NO

```
//Example :: BleGattcOpen.sb (See in BL600CodeSnippets.zip)
DIM rc
//open the gatt client with default notify/indicate ring buffer size
rc = BleGattcOpen(0,0)
IF rc == 0 THEN
    PRINT "\nGatt Client is now open"
ENDIF
//open the client with default notify/indicate ring buffer size - again
rc = BleGattcOpen(128,1)
IF rc == 0 THEN
    PRINT "\nGatt Client is still open, because already open"
ENDIF
```

Expected Output:

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

BLEGATTCOPEN is an extension function.

BleGattcClose

SUBROUTINE

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

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

BLEGATTCCLOSE ()

Arguments: None

Interactive Command: NO

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

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

Expected Output:

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

BLEGATTCCLOSE is an extension subroutine.

BleDiscServiceFirst / BleDiscServiceNext

FUNCTIONS

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

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

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

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

EVDISCPRIMSVC event message

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

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

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

BLEDISCSERVICEFIRST (connHandle,startAttrHandle,uuidHandle)

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

```
Register a handler for the EVDISCPRIMSVC event message

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

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

Returns: INTEGER, a result code. The typical value is 0x0000, indicating a successful operation and it means an EVDISCPRIMSVC event message WILL be thrown by the smartBASIC runtime engine containing the results. A non-zero return value implies an EVDISCPRIMSVC message will NOT be thrown.

Arguments:

connHandle	byVal *nConnHandle* AS INTEGER This is the connection handle as returned in the on-connect event for the connection on which the remote Gatt Server can be accessed. This will have been returned in the EVBLEMSG event message with msgId == 0 and msgCtx will have been the connection handle.
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() has been 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 WILL be thrown by the smartBASIC runtime engine containing the results. A non-zero return value implies an EVDISCPRIMSVC message will NOT be thrown.

Arguments:

connHandle	byVal *nConnHandle* AS INTEGER This is the connection handle as returned in the on-connect event for the connection on which the remote Gatt Server can be accessed. This will have been returned in the EVBLEMSG event message with msgId == 0 and msgCtx will have been the connection handle.

Interactive Command: NO

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

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

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

```
    IF rc==0 THEN : rc = BleGattcOpen(0,0) : ENDIF
ENDFUNC rc

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

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

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

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

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

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

```
        PRINT "\nScan complete"

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

            EXITFUNC 0
        ENDIF
    ENDIF
endfunc 1

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

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

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

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

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

Expected Output:

BLEDISCSERVICEFIRST and BLEDISCSERVICENEXT are both extension functions.

BleDiscCharFirst / BleDiscCharNext

FUNCTIONS

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

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

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

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

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

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

EVDISCCHAR event message

This event message **WILL** be thrown if either BleDiscCharFirst() or BleDiscCharNext() returns a success. The message contains 5 INTEGER parameters:-

Connection Handle
Characteristic Uuid Handle
Characteristic Properties
Handle for the Value Attribute of the Characteristic
Included Service Uuid Handle

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

'**Characteristic Uuid Handle**' contains the uuid of the characteristic and supplied as a handle.

'**Characteristic Properties**' contains the properties of the characteristic and is a bit mask as follows:-
Bit 0 : Set if BROADCAST is enabled
Bit 1 : Set if READ is enabled
Bit 2 : Set if WRITE_WITHOUT_RESPONSE is enabled
Bit 3 : Set if WRITE is enabled
Bit 4 : Set if NOTIFY is enabled
Bit 5 : Set if INDICATE is enabled
Bit 6 : Set if AUTHENTICATED_SIGNED_WRITE is enabled
Bit 7 : Set if RELIABLE_WRITE is enabled
Bit 15 : Set if the characteristic has extended properties

'**Handle for the Value Attribute of the Characteristic**' is the handle for the value attribute and is the value to store to keep track of important characteristics in a gatt server for later read/write operations.

'**Included Service Uuid Handle**' is for future use and will always be 0.

BLEDISCCHARFIRST (connHandle, charUuidHandle, startAttrHandle,endAttrHandle)

A typical pseudo code for discovering characteristic involves first calling BleDiscCharFirst() with information obtained from a primary services scan and then waiting for the EVDISCCHAR event message and depending on the information returned in that message calling BleDiscCharNext() which in turn will result in another EVDISCCHAR event message and typically is as follows:-

```
Register a handler for the EVDISCCHAR event message

On EVDISCCHAR event message
    If Char Value Handle == 0 then scan is complete
    Else Process information then
         call BleDiscCharNext()
         if BleDiscCharNext() not OK then scan complete
```

```
Call BleDiscCharFirst( --information from EVDISCPRIMSVC )
If BleDiscCharFirst() ok then Wait for EVDISCCHAR
```

Returns: INTEGER, a result code. The typical value is 0x0000, indicating a successful operation and it means an EVDISCCHAR event message WILL be thrown by the smartBASIC runtime engine containing the results. A non-zero return value implies an EVDISCCHAR message will NOT be thrown.

Arguments:

connHandle — byVal *nConnHandle* AS INTEGER
This is the connection handle as returned in the on-connect event for the connection on which the remote Gatt Server can be accessed. This will have been returned in the EVBLEMSG event message with msgId == 0 and msgCtx will have been the connection handle.

charUuidHandle — byVal *charUuidHandle* AS INTEGER
Set this to 0 if you want to scan for any characteristic in the service, otherwise this value will have been generated either by BleHandleUuid16() or BleHandleUuid128() or BleHandleUuidSibling().

startAttrHandle — byVal *startAttrHandle* AS INTEGER
This is the attribute handle from where the scan for characteristic will be started and will have been acquired by doing a primary services scan, which returns the start and end handles of services.

endAttrHandle — byVal *endAttrHandle* AS INTEGER
This is the end attribute handle for the scan and will have been acquired by doing a primary services scan, which returns the start and end handles of services.

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 and it means an EVDISCCHAR event message WILL be thrown by the smartBASIC runtime engine containing the results. A non-zero return value implies an EVDISCCHAR message will NOT be thrown.

Arguments:

connHandle — byVal *nConnHandle* AS INTEGER
This is the connection handle as returned in the on-connect event for the connection on which the remote Gatt Server can be accessed. This will have been returned in the EVBLEMSG event message with msgId == 0 and msgCtx will have been the connection handle.

Interactive Command: NO

```
//Example :: BleDiscCharFirst.Next.sb (See in BL600CodeSnippets.zip)
//
//Remote server has 1 prim service with 16 bit uuid and 8 characteristics where
// 5 uuids are 16 bit and 3 are 128 bit
//  3 of the 16 bit uuid are the same value 0xDEAD and
```

```
//  2 of the 128 bit uuids are also the same 112233445566778899AABBCCDDEEFF
//
// Server created using BleGattcTblDiscChar.sub invoked in _OpenMcp.scr
// using Nordic Usb Dongle PC10000

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


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


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

//==============================================================================
// Ble event handler
//==============================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    DIM uu$
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n- Disconnected"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n- Connected, so scan remote Gatt Table for first service"
        PRINT "\n- and a characeristic scan will be initiated in the event"
        rc = BleDiscServiceFirst(conHndl,0,0)
        IF rc==0 THEN
            //wait for start and end handles for first primary service
            WAITEVENT
            PRINT "\n\nScan for characteristic with uuid = 0xDEAD"
            uHndl = BleHandleUuid16(0xDEAD)
            rc = BleDiscCharFirst(conHndl,uHndl,sAttr,eAttr)
            IF rc == 0 THEN
                //HandlerCharDisc() will exit with 0 when operation is complete
                WAITEVENT
                uu$ = "112233445566778899AABBCCDDEEFF00"
                PRINT "\n\nScan for service with custom uuid ";uu$
                uu$ = StrDehexize$(uu$)
                uHndl = BleHandleUuid128(uu$)
                rc = BleDiscCharFirst(conHndl,uHndl,sAttr,eAttr)
                IF rc==0 THEN
                    //HandlerCharDisc() will exit with 0 when operation is complete
                    WAITEVENT
```

```
                ENDIF
            ENDIF
        ENDIF
        CloseConnections()
    ENDIF
ENDFUNC 1

//==============================================================================
// EVDISCPRIMSVC event handler
//==============================================================================
FUNCTION HandlerPrimSvc(cHndl,svcUuid,sHndl,eHndl) AS INTEGER
    PRINT "\nEVDISCPRIMSVC :"
    PRINT " cHndl=";cHndl
    PRINT " svcUuid=";integer.h' svcUuid
    PRINT " sHndl=";sHndl
    PRINT " eHndl=";eHndl
    IF sHndl == 0 THEN
        PRINT "\nPrimary Service Scan complete"
        EXITFUNC 0
    ELSE
        PRINT "\nGot first primary service so scan for ALL characteristics"
        sAttr = sHndl
        eAttr = eHndl
        rc = BleDiscCharFirst(conHndl,0,sAttr,eAttr)
        IF rc != 0 THEN
            PRINT "\nScan characteristics failed"
            EXITFUNC 0
        ENDIF
    ENDIF
endfunc 1

'//==============================================================================
// EVDISCCHAR event handler
'//==============================================================================
function HandlerCharDisc(cHndl,cUuid,cProp,hVal,isUuid) as integer
    print "\nEVDISCCHAR :"
    print " cHndl=";cHndl
    print " chUuid=";integer.h' cUuid
    print " Props=";cProp
    print " valHndl=";hVal
    print " ISvcUuid=";isUuid
    IF hVal == 0 THEN
        PRINT "\nCharacteristic Scan complete"
        EXITFUNC 0
    ELSE
        rc = BleDiscCharNext(conHndl)
        IF rc != 0 THEN
            PRINT "\nCharacteristics scan abort"
            EXITFUNC 0
        ENDIF
    ENDIF
endfunc 1


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

```
//Register base uuids with the underlying stack, otherwise the services with the
//128bit uuid's will be delivered with a uuid handle == FF000000 == UNKNOWN
uuid$ = "112233445566778899AABBCCDDEEFF00"
uuid$ = StrDehexize$(uuid$)
uHndl = BleHandleUuid128(uuid$)
uuid$ = "1122DEAD5566778899AABBCCDDBEEF00"
uuid$ = StrDehexize$(uuid$)
uHndl = BleHandleUuid128(uuid$)

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

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

Expected Output:

BLEDISCCHARFIRST and BLEDISCCHARNEXT are both extension functions.

BleDiscDescFirst / BleDiscDescNext

FUNCTIONS

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

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

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

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

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

EVDISCDESC event message

This event message **WILL** be thrown if either BleDissDescFirst() or BleDiscDescNext() returns a success. The message contains 3 INTEGER parameters:-

Connection Handle
Descriptor Uuid Handle
Handle for the Descriptor in the remote Gatt Table

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

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

'**Handle for the Descriptor in the remote Gatt Table**' is the handle for the descriptor, and also is the value to store to keep track of important characteristics in a gatt server for later read/write operations.

BLEDISCDESCFIRST (connHandle, descUuidHandle, charValHandle)

A typical pseudo code for discovering descriptors involves first calling BleDiscDescFirst() with information obtained from a characteristics scan and then waiting for the EVDISCDESC event message and depending on the information returned in that message calling BleDiscDescNext() which in turn will result in another EVDISCDESC event message and typically is as follows:-

```
Register a handler for the EVDISCDESC event message

On EVDISCDESC event message
    If Descriptor Handle == 0 then scan is complete
    Else Process information then
         call BleDiscDescNext()
         if BleDiscDescNext() not OK then scan complete

Call BleDiscDescFirst( --information from EVDISCCHAR )
If BleDiscDescFirst() ok then Wait for EVDISCDESC
```

Returns: INTEGER, a result code. The typical value is 0x0000, indicating a successful operation and it means an EVDISCDESC event message WILL be thrown by the smartBASIC runtime engine containing the results. A non-zero return value implies an EVDISCDESC message will NOT be thrown.

Arguments:

connHandle byVal *nConnHandle* AS INTEGER
This is the connection handle as returned in the on-connect event for the connection on which the remote Gatt Server can be accessed. This will have been returned in the EVBLEMSG event message with msgId == 0 and msgCtx will have been the connection handle.

descUuidHandle byVal *descUuidHandle* AS INTEGER
Set this to 0 if you want to scan for any descriptor in the characteristic, otherwise this value will have been generated either by BleHandleUuid16() or BleHandleUuid128() or BleHandleUuidSibling().

charValHandle byVal *charValHandle* AS INTEGER
This is the value attribute handle of the characteristic on which the descriptor scan is to be performed. It will have been acquired from an EVDISCCHAR event

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 WILL be thrown by the smartBASIC runtime engine containing the results. A non-zero return value implies an EVDISCDESC message will NOT be thrown.

Arguments:

connHandle byVal *nConnHandle* AS INTEGER
This is the connection handle as returned in the on-connect event for the connection on which the remote Gatt Server can be accessed. This will have been returned in the EVBLEMSG event message with msgId == 0 and msgCtx will have been the connection handle.

Interactive Command: NO

```
//Example :: BleDiscDescFirst.Next.sb (See in BL600CodeSnippets.zip)
//
//Remote server has 1 prim service with 16 bit uuid and 1 characteristics
// which contains 8 descriptors, that are ...
// 5 uuids are 16 bit and 3 are 128 bit
//  3 of the 16 bit uuid are the same value 0xDEAD and
//  2 of the 128 bit uuids are also the same 112233445566778899AABBCCDDEEFF
//
// Server created using BleGattcTblDiscDesc.sub invoked in _OpenMcp.scr
// using Nordic Usb Dongle PC10000

DIM rc,at$,conHndl,uHndl,uuid$,sAttr,eAttr,cValAttr


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

```
// Initialise and instantiate service, characteristic, start adverts
//==============================================================================
FUNCTION OnStartup()
    DIM rc, adRpt$, addr$, scRpt$
    rc=BleAdvRptInit(adRpt$, 2, 0, 10)
    IF rc==0 THEN : rc=BleScanRptInit(scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvRptsCommit(adRpt$,scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvertStart(0,addr$,50,0,0) : ENDIF
    //open the gatt client with default notify/indicate ring buffer size
    IF rc==0 THEN : rc = BleGattcOpen(0,0) : ENDIF
ENDFUNC rc
//==============================================================================
// Close connections so that we can run another app without problems
//==============================================================================
SUB CloseConnections()
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
ENDSUB

//==============================================================================
// Ble event handler
//==============================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    DIM uu$
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n- Disconnected"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n- Connected, so scan remote Gatt Table for first service"
        PRINT "\n- and a 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:

BLEDISCDESCFIRST and BLEDISCDESCNEXT are both extension functions.

BleGattcFindChar

FUNCTION

This function facilitates a quick and efficient way of locating the details of a characteristic if the uuid is known along with the uuid of the service containing it and the results will be delived in a EVFINDCHAR event message. If the Gatt server table has multiple instances of the same service/characteristic combination then this function will work because in addition to the uuid handles to be searched for, it also accepts instance parameters which are indexed from 0, which means the 4th instance of a characteristic with the same uuid in the 3rd instance of a service with the same uuid will be located with index values 3 and 2 respectively.

Given that the results are returned in an event message, a handler **must** be registered for the EVFINDCHAR event.

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

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

EVFINDCHAR event message

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

Connection Handle
Characteristic Properties
Handle for the Value Attribute of the Characteristic
Included Service Uuid Handle

If the specified instance of the service/characteristic is not present in the remote Gatt Server Table then all parameters will contain 0 apart from the Connection Handle.

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

Bit 0 : Set if BROADCAST is enabled
Bit 1 : Set if READ is enabled
Bit 2 : Set if WRITE_WITHOUT_RESPONSE is enabled
Bit 3 : Set if WRITE is enabled
Bit 4 : Set if NOTIFY is enabled
Bit 5 : Set if INDICATE is enabled
Bit 6 : Set if AUTHENTICATED_SIGNED_WRITE is enabled
Bit 7 : Set if RELIABLE_WRITE is enabled
Bit 15 : Set if the characteristic has extended properties

'**Handle for the Value Attribute of the Characteristic**' is the handle for the value attribute and is the value to store to keep track of important characteristics in a gatt server for later read/write operations.

'**Included Service Uuid Handle**' is for future use and will always be 0.

BLEGATTCFINDCHAR (connHandle, svcUuidHndl, svcIndex,charUuidHndl, charIndex)

A typical pseudo code for finding a characteristic involves calling BleGattcFindChar() which in turn will result in the EVFINDCHAR event message and typically is as follows:-

```
Register a handler for the EVFINDCHAR event message

On EVFINDCHAR event message
    If Char Value Handle == 0 then
         Characteristic not found
    Else
         Characteristic has been found

Call BleGattcFindChar()
If BleGattcFindChar () ok then Wait for EVFINDCHAR
```

Returns: INTEGER, a result code. The typical value is 0x0000, indicating a successful operation and it means an EVFINDCHAR event message WILL be thrown by the smartBASIC runtime engine containing the results. A non-zero return value implies an EVFINDCHAR message will NOT be thrown.

Arguments:

connHandle — byVal *nConnHandle* AS INTEGER
This is the connection handle as returned in the on-connect event for the connection on which the remote Gatt Server can be accessed. This will have been returned in the EVBLEMSG event message with msgId == 0 and msgCtx will have been the connection handle.

svcUuidHndl — byVal *svcUuidHndl* AS INTEGER
Set this to the service uuid handle which will have been generated either by BleHandleUuid16() or BleHandleUuid128() or BleHandleUuidSibling().

svcIndex — byVal *svcIndex* AS INTEGER
This is the instance of the service to look for with the uuid handle svcUuidHndl, where 0 is the first instance, 1 is the second etc

charUuidHndl — byVal *charUuidHndl* AS INTEGER
Set this to the characteristic uuid handle which will have been generated either by BleHandleUuid16() or BleHandleUuid128() or BleHandleUuidSibling().

charIndex — byVal *charIndex* AS INTEGER
This is the instance of the characteristic to look for with the uuid handle charUuidHndl, where 0 is the first instance, 1 is the second etc

Interactive Command: NO

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

```
DIM rc,at$,conHndl,uHndl,uuid$,sIdx,cIdx


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


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


//==============================================================================
// Ble event handler
//==============================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    DIM uu$,uHndS,uHndC
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n- Disconnected"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n- Connected, so scan remote Gatt Table for an instance of char"
        uHndS = BleHandleUuid16(0xDEAD)
        uu$ = "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:

BLEGATTCFINDCHAR is an extension function.

BleGattcFindDesc

FUNCTION

This function facilitates a quick and efficient way of locating the details of a descriptor if the uuid is known along with the uuid of the service and the uuid of the characteristic containing it and the results will be delivered in a EVFINDDESC event message. If the Gatt server table has multiple instances of the same service/characteristic/descriptor combination then this function will work because in addition to the uuid handles to be searched for, it also accepts instance parameters which are indexed from 0, which means the 2nd instance of a descriptor in the 4th instance of a characteristic with the same uuid in the 3rd instance of a service with the same uuid will be located with index values 1, 3 and 2 respectively.

Given that the results are returned in an event message, a handler **must** be registered for the EVFINDDESC event.

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

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

EVFINDDESC event message

This event message **WILL** be thrown if BleGattcFindDesc()returned a success. The message contains 2 INTEGER parameters:-

Connection Handle
Handle of the Descriptor

If the specified instance of the service/characteristic/descriptor is not present in the remote Gatt Server Table then all parameters will contain 0 apart from the Connection Handle.

'**Handle of the Descriptor**' is the handle for the descriptor and is the value to store to keep track of important descriptors in a gatt server for later read/write operations – for example CCCD's to enable notifications and/or indications.

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

A typical pseudo code for finding a descrirptor involves calling BleGattcFindDesc() which in turn will result in the EVFINDDESC event message and typically is as follows:-

```
Register a handler for the EVFINDDESC event message

On EVFINDDESC event message
    If Descriptor Handle == 0 then
        Descriptor not found
    Else
        Descriptor has been found

Call BleGattcFindDesc()
If BleGattcFindDesc() ok then Wait for EVFINDDESC
```

Returns: INTEGER, a result code. The typical value is 0x0000, indicating a successful operation and it means an EVFINDDESC event message WILL be thrown by the smartBASIC runtime engine containing the results. A non-zero return value implies an EVFINDDESC message will NOT be thrown.

Arguments:

connHndl — byVal *connHndl* AS INTEGER
This is the connection handle as returned in the on-connect event for the connection on which the remote Gatt Server can be accessed. This will have been returned in the EVBLEMSG event message with msgId == 0 and msgCtx will have been the connection handle.

svcUuHndl — byVal *svcUuHndl* AS INTEGER
Set this to the service uuid handle which will have been generated either by BleHandleUuid16() or BleHandleUuid128() or BleHandleUuidSibling().

svcIdx — byVal *svcIdx* AS INTEGER
This is the instance of the service to look for with the uuid handle svcUuidHndl, where 0 is the first instance, 1 is the second etc

charUuHndl — byVal *charUuHndl* AS INTEGER
Set this to the characteristic uuid handle which will have been generated either by BleHandleUuid16() or BleHandleUuid128() or BleHandleUuidSibling().

charIdx — byVal *charIdx* AS INTEGER
This is the instance of the characteristic to look for with the uuid handle charUuidHndl, where 0 is the first instance, 1 is the second etc

descUuHndl — byVal *descUuHndl* AS INTEGER
Set this to the descriptor uuid handle which will have been generated either by BleHandleUuid16() or BleHandleUuid128() or BleHandleUuidSibling().

descIdx — byVal *descIdx* AS INTEGER
This is the instance of the descriptor to look for with the uuid handle charUuidHndl, where 0 is the first instance, 1 is the second etc

Interactive Command: NO

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

DIM rc,at$,conHndl,uHndl,uuid$,sIdx,cIdx,dIdx


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

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


//==============================================================================
// Ble event handler
//==============================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    DIM uu$,uHndS,uHndC,uHndD
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n- Disconnected"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n- Connected, so scan remote Gatt Table for ALL services"
        uHndS = BleHandleUuid16(0xDEAD)
        uu$ = "112233445566778899AABBCCDDEEFF00"
        uu$ = StrDehexize$(uu$)
        uHndC = BleHandleUuid128(uu$)
        uu$ = "1122C0DE5566778899AABBCCDDEEFF00"
        uu$ = StrDehexize$(uu$)
        uHndD = BleHandleUuid128(uu$)
        sIdx = 2
        cIdx = 1
        dIdx = 1 // handle will be 37
        rc = BleGattcFindDesc(conHndl,uHndS,sIdx,uHndC,cIdx,uHndD,dIdx)
        IF rc==0 THEN
            //BleDiscCharFirst() will exit with 0 when operation is complete
            WAITEVENT
        ENDIF
        sIdx = 1
        cIdx = 3
        dIdx = 4 //does not exist
        rc = BleGattcFindDesc(conHndl,uHndS,sIdx,uHndC,cIdx,uHndD,dIdx)
        IF rc==0 THEN
            //BleDiscCharFirst() will exit with 0 when operation is complete
            WAITEVENT
        ENDIF
        CloseConnections()
    ENDIF
ENDFUNC 1

'//==============================================================================
'//==============================================================================
function HandlerFindDesc(cHndl,hndl) as integer
    print "\nEVFINDDESC "
    print " cHndl=";cHndl
    print " dscHndl=";hndl
    IF hndl == 0 THEN
        PRINT "\nDid NOT find the descriptor"
    ELSE
        PRINT "\nFound the descriptor at handle ";hndl
        PRINT "\nSvc Idx=";sIdx;" Char Idx=";cIdx;" desc Idx=";dIdx
```

```
    ENDIF
endfunc 0


//==============================================================================
// Main() equivalent
//==============================================================================
ONEVENT  EVBLEMSG           CALL HndlrBleMsg
OnEvent  EVFINDDESC         call HandlerFindDesc

//Register base uuids with the underlying stack, otherwise the services with the
//128bit uuid's will be delivered with a uuid handle == FF000000 == UNKNOWN
uuid$ = "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:

BLEGATTCFINDDESC is an extension function.

BleGattcRead / BleGattcReadData

FUNCTIONS

If the handle for an attribute is known then these functions are used to read the content of that attribute from a specified offset in the array of octets in that attribute value.

Given that the success or failure of this read operation is returned in an event message, a handler **must** be registered for the EVATTRREAD event.

Depending on the connection interval, the read of the attribute may take many 100s of milliseconds, and while this is in progress it is safe to do other non Gatt related operations like for example servicing sensors and displays or any of the onboard peripherals.

BleGattcRead is used to trigger the procedure and BleGattcReadData is used to read the data from the underlying cache when the EVATTRREAD event message is received with a success status.

EVATTRREAD event message

This event message **WILL** be thrown if BleGattcRead() returns a success. The message contains 3 INTEGER parameters:-

Connection Handle
Handle of the Attribute
Gatt status of the read operation.

'**Gatt status of the read operation**' is one of the following values, where 0 implies the read was successfully expedited and the data can be obtained by calling BlePubGattClientReadData().

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

BLEGATTCREAD (connHndl, attrHndl, offset)

A typical pseudo code for reading the content of an attribute calling BleGattcRead() which in turn will result in the EVATTRREAD event message and typically is as follows:-

```
Register a handler for the EVATTRREAD event message

On EVATTREAD 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 WILL be thrown by the smartBASIC runtime engine containing the results. A non-zero return value implies an EVATTRREAD message will NOT be thrown.

Arguments:

connHndl — byVal *connHndl* AS INTEGER
This is the connection handle as returned in the on-connect event for the connection on which the remote Gatt Server can be accessed. This will have been returned in the EVBLEMSG event message with msgId == 0 and msgCtx will have been the connection handle.

attrHndl — byVal *attrHndl* AS INTEGER
Set this to the handle of the attribute to read and will be a value in the range 1 to 65535

offset — byVal *offset* AS INTEGER
This is the offset from which the data in the attribute is to be read.

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 will have been returned in the EVBLEMSG event message with msgId == 0 and msgCtx will have been the connection handle.

attrHndl — byRef *attrHndl* AS INTEGER
The handle for the attribute that was read is returned in this variable. Will be the same as the one supplied in BleGattcRead, but supplied here so that the code can be stateless.

offset — byRef *offset* AS INTEGER
The offset into the attribute data that was read is returned in this variable. Will be the

same as the one supplied in BleGattcRead, but supplied here so that the code can be stateless.

attrData$* byRef *attrData$ AS STRING
The attribute data which was read is supplied in this parameter.

Interactive Command: NO

```
//Example :: BleGattcRead.sb (See in BL600CodeSnippets.zip)
//
//Remote server has 3 prim services with 16 bit uuid. First service has one
//characteristic whose value attribute is at handle 3 and has read/write props
//
// Server created using BleGattcTblRead.sub invoked in _OpenMcp.scr
// using Nordic Usb Dongle PC10000

DIM rc,at$,conHndl,uHndl,nOff,atHndl


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


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


//==============================================================================
// Ble event handler
//==============================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    DIM uHndA
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n- Disconnected"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n- Connected, so read attibute handle 3"
        atHndl = 3
        nOff = 0
        rc=BleGattcRead(conHndl,atHndl,nOff)
```

```
        IF rc==0 THEN
            WAITEVENT
        ENDIF
        PRINT "\nread attibute handle 300 which does not exist"
        atHndl = 300
        nOff = 0
        rc=BleGattcRead(conHndl,atHndl,nOff)
        IF rc==0 THEN
            WAITEVENT
        ENDIF
        CloseConnections()
    ENDIF
ENDFUNC 1

'//==============================================================================
'//==============================================================================
function HandlerAttrRead(cHndl,aHndl,nSts) as integer
    dim nOfst,nAhndl,at$
    print "\nEVATTRREAD "
    print " cHndl=";cHndl
    print " attrHndl=";aHndl
    print " status=";integer.h' nSts
    if nSts == 0 then
        print "\nAttribute read OK"
        rc = BleGattcReadData(cHndl,nAhndl,nOfst,at$)
        print "\nData    = ";StrHexize$(at$)
        print " Offset= ";nOfst
        print " Len=";strlen(at$)
        print "\nhandle = ";nAhndl
    else
        print "\nFailed to read attribute"
    endif
endfunc 0


//==============================================================================
// Main() equivalent
//==============================================================================
ONEVENT  EVBLEMSG           CALL HndlrBleMsg
OnEvent  EVATTRREAD         call HandlerAttrRead

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

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

Expected Output:

BLEGATTCREAD and BLEGATTREADDATA are extension functions.

BleGattcWrite

FUNCTION

If the handle for an attribute is known then this function is used to write into an attribute starting at offset 0. The acknowledgement will be returned via a EVATTRWRITE event message.

Given that the success or failure of this write operation is returned in an event message, a handler **must** be registered for the EVATTRWRITE event.

Depending on the connection interval, the write to the attribute may take many 100s of milliseconds, and while this is in progress it is safe to do other non Gatt related operations like for example servicing sensors and displays or any of the onboard peripherals.

EVATTRWRITE event message

This event message **WILL** be thrown if BleGattcWrite() returns a success. The message contains 3 INTEGER parameters:-

Connection Handle
Handle of the Attribute
Gatt status of the write operation.

'**Gatt status of the write operation**' is one of the following values, where 0 implies the write was successfully expedited.

```
0x0000  Success
0x0001  Unknown or not applicable status
0x0100  ATT Error: Invalid Error Code
0x0101  ATT Error: Invalid Attribute Handle
0x0102  ATT Error: Read not permitted
0x0103  ATT Error: Write not permitted
0x0104  ATT Error: Used in ATT as Invalid PDU
0x0105  ATT Error: Authenticated link required
0x0106  ATT Error: Used in ATT as Request Not Supported
0x0107  ATT Error: Offset specified was past the end of the attribute
```

```
0x0108  ATT Error: Used in ATT as Insufficient Authorisation
0x0109  ATT Error: Used in ATT as Prepare Queue Full
0x010A  ATT Error: Used in ATT as Attribute not found
0x010B  ATT Error: Attribute cannot be read or written
                   using read/write blob requests
0x010C  ATT Error: Encryption key size used is insufficient
0x010D  ATT Error: Invalid value size
0x010E  ATT Error: Very unlikely error
0x010F  ATT Error: Encrypted link required
0x0110  ATT Error: Attribute type is not a supported grouping attribute
0x0111  ATT Error: Encrypted link required
0x0112  ATT Error: Reserved for Future Use range #1 begin
0x017F  ATT Error: Reserved for Future Use range #1 end
0x0180  ATT Error: Application range begin
0x019F  ATT Error: Application range end
0x01A0  ATT Error: Reserved for Future Use range #2 begin
0x01DF  ATT Error: Reserved for Future Use range #2 end
0x01E0  ATT Error: Reserved for Future Use range #3 begin
0x01FC  ATT Error: Reserved for Future Use range #3 end
0x01FD  ATT Common Profile and Service Error:
                   Client Characteristic Configuration Descriptor (CCCD)
                   improperly configured
0x01FE  ATT Common Profile and Service Error:
                   Procedure Already in Progress
0x01FF  ATT Common Profile and Service Error:
                   Out Of Range
```

BLEGATTCWRITE (connHndl, attrHndl, attrData$)

A typical pseudo code for writing to an attribute which will result in the EVATTRWRITE event message and typically is as follows:-

```
Register a handler for the EVATTRWRITE event message

On EVATTWRITE event message
    If Gatt_Status == 0 then
         Attribute was written successfully
    Else
         Attribute could not be written

Call BleGattcWrite()
If BleGattcWrite() ok then Wait for EVATTRWRITE
```

Returns: INTEGER, a result code. 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 will have been returned in the EVBLEMSG event message with msgId == 0 and msgCtx will have been the connection handle.

attrHndl	byVal *attrHndl* AS INTEGER The handle for the attribute that is to be written to.
attrData$	byRef *attrData$* AS STRING The attribute data to write.

Interactive Command: NO

```
//Example :: BleGattcWrite.sb (See in BL600CodeSnippets.zip)
//
//Remote server has 3 prim services with 16 bit uuid. First service has one
//characteristic whose value attribute is at handle 3 and has read/write props
//
// Server created using BleGattcTblWrite.sub invoked in _OpenMcp.scr
// using Nordic Usb Dongle PC10000

DIM rc,at$,conHndl,uHndl,atHndl


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


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


//==============================================================================
// Ble event handler
//==============================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    DIM uHndA
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n- Disconnected"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n- Connected, so write to attibute handle 3"
        atHndl = 3
        at$="\01\02\03\04"
        rc=BleGattcWrite(conHndl,atHndl,at$)
        IF rc==0 THEN
```

```
            WAITEVENT
        ENDIF
        PRINT "\nwrite to attibute handle 300 which does not exist"
        atHndl = 300
        rc=BleGattcWrite(conHndl,atHndl,at$)
        IF rc==0 THEN
            WAITEVENT
        ENDIF
        CloseConnections()
    ENDIF
ENDFUNC 1

'//==============================================================================
'//==============================================================================
function HandlerAttrWrite(cHndl,aHndl,nSts) as integer
    dim nOfst,nAhndl,at$
    print "\nEVATTRWRITE "
    print " cHndl=";cHndl
    print " attrHndl=";aHndl
    print " status=";integer.h' nSts
    if nSts == 0 then
        print "\nAttribute write OK"
    else
        print "\nFailed to write attribute"
    endif
endfunc 0


//==============================================================================
// Main() equivalent
//==============================================================================
ONEVENT  EVBLEMSG           CALL HndlrBleMsg
OnEvent  EVATTRWRITE        call HandlerAttrWrite

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

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

Expected Output:

BLEGATTCWRITE is an extension function.

BleGattcWriteCmd

FUNCTION

If the handle for an attribute is known then this function is used to write into an attribute at offset 0when no acknowledgment response is expected. The signal that the command has actually been transmitted and that the remote link layer has acknowledged is by the EVNOTIFYBUF event.

Note that the acknowledgement received for the BleGattcWrite() command is from the higher level GATT layer, not to be confused with the link layer ack in this case.

> *All packets are acknowledged at link layer level. If a packet fails to get through then that condition will manifest as a connection drop due to the link supervision timeout.*

Given that the transmission and link layer ack of this write operation is indicated in an event message, a handler **must** be registered for the EVNOTIBUF event.

Depending on the connection interval, the write to the attribute may take many 100s of milliseconds, and while this is in progress it is safe to do other non Gatt related operations like for example servicing sensors and displays or any of the onboard peripherals.

EVNOTIFYBUF event

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

BLEGATTCWRITECMD (connHndl, attrHndl, attrData$)

A typical pseudo code for writing to an attribute which will result in the EVNOTIFYBUF event is as follows:-

```
Register a handler for the EVNOTIFYBUF event message

On EVNOTIFYBUF event message
    Can now send another write command
```

```
Call BleGattcWriteCmd()
If BleGattcWrite() ok then Wait for EVNOTIFYBUF
```

Returns: INTEGER, a result code. 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 will have been returned in the EVBLEMSG event message with msgId == 0 and msgCtx will have been the connection handle.

attrHndl — byVal *attrHndl* AS INTEGER
The handle for the attribute that is to be written to.

attrData$* — byRef *attrData$ AS STRING
The attribute data to write.

Interactive Command: NO

```
//Example :: BleGattcWriteCmd.sb (See in BL600CodeSnippets.zip)
//
//Remote server has 3 prim services with 16 bit uuid. First service has one
//characteristic whose value attribute is at handle 3 and has read/write props
//
// Server created using BleGattcTblWriteCmd.sub invoked in _OpenMcp.scr
// using Nordic Usb Dongle PC10000

DIM rc,at$,conHndl,uHndl,atHndl


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


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


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

```
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    DIM uHndA
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n- Disconnected"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n- Connected, so write to attribute handle 3"
        atHndl = 3
        at$="\01\02\03\04"
        rc=BleGattcWriteCmd(conHndl,atHndl,at$)
        IF rc==0 THEN
            WAITEVENT
        ENDIF
        PRINT "\n- write again to attribute handle 3"
        atHndl = 3
        at$="\05\06\07\08"
        rc=BleGattcWriteCmd(conHndl,atHndl,at$)
        IF rc==0 THEN
            WAITEVENT
        ENDIF
        PRINT "\n- write again to attribute handle 3"
        atHndl = 3
        at$="\09\0A\0B\0C"
        rc=BleGattcWriteCmd(conHndl,atHndl,at$)
        IF rc==0 THEN
            WAITEVENT
        ENDIF
        PRINT "\nwrite to attribute handle 300 which does not exist"
        atHndl = 300
        rc=BleGattcWriteCmd(conHndl,atHndl,at$)
        IF rc==0 THEN
            PRINT "\nEven when the attribute does not exist an event will occur"
            WAITEVENT
        ENDIF
        CloseConnections()
    ENDIF
ENDFUNC 1

'//==============================================================================
'//==============================================================================
function HandlerNotifyBuf() as integer
  print "\nEVNOTIFYBUF Event"
endfunc 0  '//need to progress the WAITEVENT

//==============================================================================
// Main() equivalent
//==============================================================================
ONEVENT  EVBLEMSG           CALL HndlrBleMsg
OnEvent  EVNOTIFYBUF        call HandlerNotifyBuf

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

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

Expected Output:

BLEGATTCWRITECMD is an extension function.

BleGattcNotifyRead

FUNCTION

A Gatt Server has the ability to notify or indicate the value attribute of a characteristic when enabled via the Client Characeristic Configuration Descriptor (CCCD). This means data will arrive from a Gatt Server at any time and so has to be managed so that it can synchronised with the smartBASIC runtime engine.

Data arriving via a notification does not require Gatt acknowledgements, however indications require them. This Gatt Client manager saves data arriving via a notification in the same ring buffer for later extraction using the command BleGattcNotifyRead() and for indications an automatic gatt acknowledgement is sent when the data is saved in the ring buffer. This acknowledgment happens even if the data was discarded because the ring buffer was full. If however it is required that the data NOT be acknowledged when it is discarded on a full buffer then set the flags parameter in the BleGattcOpen() function where the Gatt Client manager is opened.

In the case when an ack is NOT sent on data discard, the Gatt Server will be throttled and so no further data will be notified or indicated by it until BleGattNotifyRead() is called to extract data from the ring buffer to create space and it will trigger a delayed acknowledgement.

When the Gatt Client manager is opened using BleGattcOpen() it is possible to specify the size of the ring buffer. If a value of 0 is supplied then a default size is created. SYSINFO(2019) in a smartBASIC application or the interactive mode command AT I 2019 will return the default size. Likewise SYSINFO(2020) or the command AT I 2020 will return the maximum size.

Data that arrives via notifications or indications get stored in the ring buffer and at the same time a EVATTRNOTIFY event is thrown to the smartBASIC runtime engine. This is an event, in the same way an incoming UART receive character generates an event, that is, no data payload is attached to the event.

EVATTRTOTIFY event message

This event **WILL** be thrown when an notification or an indication arrives from a gatt server . The event contains no parameters. Please note that if one notification/indication arrives or many, like in the case of UART events, the same event mask bit is asserted. The paradigm being that the smartBASIC application is informed that it needs to go and service the ring buffer using the function BleGattcNotifyRead.

BLEGATTCNOTIFYREAD (connHndl, attrHndl, attrData$, discardCount)

A typical pseudo code for handling and accessing notification/indication data is as follows:-

```
Register a handler for the EVATTRNOTIFY event message

On EVATTRNOTIRY event
    BleGattcNotifyRead()  //to actually get the data
    Process the data

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

Returns: INTEGER, a result code. The typical value is 0x0000, indicating data was successful read.

Arguments:

connHndl	byRef *connHndl* AS INTEGER On exit this will be the connection handle of the gatt server that sent the notification or indication.
attrHndl	byRef *attrHndl* AS INTEGER On exit this will be the handle of the characteristic value attribute in the notification or indication.
attrData$	byRef *attrData$* AS STRING On exit this will be the data of the characteristic value attribute in the notification or indication. It is always from offset 0 of the source attribute.
discardedCount	byRef *discardedCount* AS INTEGER On exit this should contain 0 and it signifies the total number of notifications or indications that got discared because the ring buffer in the gatt client manager was full. If non-zero values are encountered, it is recommended that the ring buffer size be increased by using BleGattcClose() when the gatt client was opened using BleGattcOpen().

Interactive Command: NO

```
//Example :: BleGattcNotifyRead.sb (See in BL600CodeSnippets.zip)
//
// Server created using BleGattcTblNotifyRead.sub invoked in _OpenMcp.scr
// using Nordic Usb Dongle PC10000
//
// Charactersitic at handle 15 has notify   (16==cccd)
// Charactersitic at handle 18 has indicate (19==cccd)

DIM rc,at$,conHndl,uHndl,atHndl



//==============================================================================
// Initialise and instantiate service, characteristic, start adverts
//==============================================================================
FUNCTION OnStartup()
    DIM rc, adRpt$, addr$, scRpt$
```

```
    rc=BleAdvRptInit(adRpt$, 2, 0, 10)
    IF rc==0 THEN : rc=BleScanRptInit(scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvRptsCommit(adRpt$,scRpt$) : ENDIF
    IF rc==0 THEN : rc=BleAdvertStart(0,addr$,50,0,0) : ENDIF
    //open the gatt client with default notify/indicate ring buffer size
    IF rc==0 THEN : rc = BleGattcOpen(0,0) : ENDIF
ENDFUNC rc


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


//==============================================================================
// Ble event handler
//==============================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\n\n- Disconnected"
        EXITFUNC 0
    ELSEIF nMsgID==0 THEN
        PRINT "\n- Connected, so enable notification for char with cccd at 16"
        atHndl = 16
        at$="\01\00"
        rc=BleGattcWrite(conHndl,atHndl,at$)
        IF rc==0 THEN
            WAITEVENT
        ENDIF
        PRINT "\n- enable indication for char with cccd at 19"
        atHndl = 19
        at$="\02\00"
        rc=BleGattcWrite(conHndl,atHndl,at$)
        IF rc==0 THEN
            WAITEVENT
        ENDIF
    ENDIF
ENDFUNC 1

'//==============================================================================
'//==============================================================================
function HandlerAttrWrite(cHndl,aHndl,nSts) as integer
    dim nOfst,nAhndl,at$
    print "\nEVATTRWRITE "
    print " cHndl=";cHndl
    print " attrHndl=";aHndl
    print " status=";integer.h' nSts
    if nSts == 0 then
        print "\nAttribute write OK"
    else
        print "\nFailed to write attribute"
    endif
endfunc 0

'//==============================================================================
```

```
'//==============================================================================
function HandlerAttrNotify() as integer
    dim chndl,aHndl,att$,dscd
    print "\nEVATTRNOTIFY Event"
    rc=BleGattcNotifyRead(cHndl,aHndl,att$,dscd)
    print "\n  BleGattcNotifyRead()"
    if rc==0 then
        print " cHndl=";cHndl
        print " attrHndl=";aHndl
        print " data=";StrHexize$(att$)
        print " discarded=";dscd
    else
        print " failed with ";integer.h' rc
    endif
endfunc 1

//==============================================================================
// Main() equivalent
//==============================================================================
ONEVENT  EVBLEMSG           CALL HndlrBleMsg
OnEvent  EVATTRWRITE        call HandlerAttrWrite
OnEvent  EVATTRNOTIFY       call HandlerAttrNotify

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

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

Expected Output:

BLEGATTCNOTIFYREAD is an extension function.

Attribute Encoding Functions

Data for Characteristics are stored in Value attributes, arrays of bytes. Multibyte Characteristic Descriptors content is stored similarly. Those bytes are manipulated in *smart* BASIC applications using STRING variables.

The Bluetooth specification stipulates that multibyte data entities are stored communicated in little endian format and so all data manipulation is done similarly. Little endian means that a multibyte data entity will be stored so that lowest significant byte is position at the lowest memory address and likewise when transported, the lowest byte will get on the wire first.

This section describes all the encoding functions which allow those strings to be written to in smaller bytewise subfields in a more efficient manner compared to the generic STRXXXX functions that are made available in *smart* BASIC.

Note: CCCD and SCCD Descriptors are special cases; they have just 2 bytes which are treated as 16 bit integers. This is reflected in smartBASIC applications so that INTEGER variables are used to manipulate those values instead of STRINGS.

BleEncode8

FUNCTION

This function overwrites a single byte in a string at a specified offset. If the string is not long enough, then it will be extended with the new extended block uninitialized and then the byte specified is overwritten.

If the nIndex is such that the new string length exceeds the maximum attribute length, this function fails. The maximum attribute length can be obtained using the function SYSINFO(n) where n is 2013. The Bluetooth specification allows a length between 1 and 512.

BLEENCODE8 (attr$,nData, nIndex)

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

Arguments:

attr$* — byRef *attr$ AS STRING
This argument is the string that will be written to an attribute

nData — byVal nData AS INTEGER
The least significant byte of this integer is saved. The rest is ignored.

nIndex — byVal *nIndex* AS INTEGER
This is the zero-based index into the string attr$ where the new fragment of data is written to. If the string attr$ is not long enough to accommodate the index plus the length of the fragment, it is extended. If the extended length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function fails.

Interactive Command: NO

```
//Example :: BleEncode8.sb (See in BL600CodeSnippets.zip)

DIM rc
DIM attr$

attr$="Laird"

PRINT "\nattr$=";attr$

//Remember: - 4 bytes are used to store an integer on the BL600

//write 'C' to index 2 -- '111' will be ignored
rc=BleEncode8(attr$,0x11143,2)
//write 'A' to index 0
rc=BleEncode8(attr$,0x41,0)
//write 'B' to index 1
rc=BleEncode8(attr$,0x42,1)
//write 'D' to index 3
rc=BleEncode8(attr$,0x44,3)
//write 'y' to index 7 -- attr$ will be extended
rc=BleEncode8(attr$,0x67, 7)

PRINT "\nattr$ now = ";attr$
```

Expected Output:

```
attr$=Laird
attr$ now = ABCDd\00\00g
```

BLEENCODE8 is an extension function.

BleEncode16

FUNCTION

This function overwrites two bytes in a string at a specified offset. If the string is not long enough, then it is extended with the new extended block uninitialized and then the bytes specified are overwritten.

If the nIndex is such that the new string length exceeds the maximum attribute length, this function fails. The maximum attribute length can be obtained using the function SYSINFO(n) where n is 2013. The Bluetooth specification allows a length between 1 and 512.

BLEENCODE16 (attr$,nData, nIndex)

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

Arguments:

attr$* byRef *attr$ AS STRING
This argument is the string that will be written to an attribute

nData byVal nData AS INTEGER
The two least significant bytes of this integer is saved. The rest is ignored.

nIndex byVal *nIndex* AS INTEGER
This is the zero based index into the string attr$ where the new fragment of data is written. If the string attr$ is not long enough to accommodate the index plus the length of the fragment, it is extended. If the extended length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function fails.

Interactive Command: NO

```
//Example :: BleEncode16.sb (See in BL600CodeSnippets.zip)

DIM rc, attr$
attr$="Laird"
PRINT "\nattr$=";attr$

//write 'CD' to index 2
rc=BleEncode16(attr$,0x4443,2)
//write 'AB' to index 0 - '2222' will be ignored
rc=BleEncode16(attr$,0x22224241,0)
//write 'EF' to index 3
rc=BleEncode16(attr$,0x4645,4)

PRINT "\nattr$ now = ";attr$
```

Expected Output:

```
attr$=Laird
attr$ now = ABCDEF
```

BLEENCODE16 is an extension function.

BleEncode24

FUNCTION

This function overwrites three bytes in a string at a specified offset. If the string is not long enough, then it will be extended with the new extended block uninitialized and then the bytes specified are overwritten.

If the nIndex is such that the new string length exceeds the maximum attribute length, this function fails. The maximum attribute length can be obtained using the function SYSINFO(n) where n is 2013. The Bluetooth specification allows a length between 1 and 512.

BLEENCODE24 (attr$,nData, nIndex)

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

Arguments:

attr$* byRef *attr$ AS STRING
This argument is the string that will be written to an attribute.

nData byVal nData AS INTEGER
The three least significant bytes of this integer is saved. The rest is ignored.

nIndex	byVal *nIndex* AS INTEGER This is the zero based index into the string attr$ where the new fragment of data is written. If the string attr$ is not long enough to accommodate the index plus the length of the fragment, it is extended. If the extended length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function will fail.

Interactive Command: NO

```
//Example :: BleEncode24.sb (See in BL600CodeSnippets.zip)

DIM rc
DIM attr$ : attr$="Laird"

//write 'BCD' to index 1
rc=BleEncode24(attr$,0x444342,1)
//write 'A' to index 0
rc=BleEncode8(attr$,0x41,0)
//write 'EF'to index 4
rc=BleEncode16(attr$,0x4645,4)

PRINT "attr$=";attr$
```

Expected Output:

```
attr$=ABCDEF
```

BLEENCODE24 is an extension function.

BleEncode32

FUNCTION

This function overwrites four bytes in a string at a specified offset. If the string is not long enough, then it is extended with the new extended block uninitialized and then the bytes specified are overwritten.

If the nIndex is such that the new string length exceeds the maximum attribute length, this function fails. The maximum attribute length can be obtained using the function SYSINFO(n) where n is 2013. The Bluetooth specification allows a length between 1 and 512.

BLEENCODE32(attr$,nData, nIndex)

Returns:	INTEGER, a result code. The typical value is 0x0000, indicating a successful operation.
Arguments:	
attr$	byRef *attr$* AS STRING This argument is the string that will be written to an attribute
nData	byVal nData AS INTEGER The four bytes of this integer is saved. The rest is ignored.
nIndex	byVal *nIndex* AS INTEGER This is the zero based index into the string attr$ where the new fragment of data is written. If the string attr$ is not long enough to accommodate the index plus the length

of the fragment, it is extended. If the extended length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function fails.

Interactive Command: NO

```
//Example :: BleEncode32.sb (See in BL600CodeSnippets.zip)

DIM rc
DIM attr$ : attr$="Laird"

//write 'BCDE' to index 1
rc=BleEncode32(attr$,0x45444342,1)
//write 'A' to index 0
rc=BleEncode8(attr$,0x41,0)

PRINT "attr$=";attr$
```

Expected Output:

```
attr$=ABCDE
```

BLEENCODE32 is an extension function.

BleEncodeFLOAT

FUNCTION

This function overwrites four bytes in a string at a specified offset. If the string is not long enough, it is extended with the new extended block uninitialized and then the byte specified is overwritten.

If the nIndex is such that the new string length exceeds the maximum attribute length, this function fails. The maximum attribute length can be obtained using the function SYSINFO(n) where n is 2013. The Bluetooth specification allows a length between 1 and 512.

BLEENCODEFLOAT (attr$, nMatissa, nExponent, nIndex)

Returns: INTEGER, a result code. The 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 that the range is not +/- 2048 because after encoding the following 2 byte values have special meaning:

0x07FFFFFF	NaN (Not a Number)
0x08000000	NRes (Not at this resolution)
0x07FFFFFE	+ INFINITY
0x08000002	- INFINITY
0x08000001	Reserved for future use

nExponent byVal nExponent AS INTEGER
This value must be in the range -128 to 127 or the function fails.

nIndex byVal *nIndex* AS INTEGER
This is the zero based index into the string attr$ where the new fragment of data is written. If the string attr$ is not long enough to accommodate the index plus the length of the fragment, it is extended. If the extended length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function fails.

Interactive Command: NO

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

DIM rc
DIM attr$ : attr$=""

//write 1234567 x 10^-54 as FLOAT to index 2
PRINT BleEncodeFLOAT(attr$,123456,-54,0)

//write 1234567 x 10^1000 as FLOAT to index 2 and it will fail
//because the exponent is too large, it has to be < 127
IF BleEncodeFLOAT(attr$,1234567,1000,2)!=0 THEN
  PRINT "\nFailed to encode to FLOAT"
ENDIF

//write 10000000 x 10^0 as FLOAT to index 2 and it will fail
//because the mantissa is too large, it has to be < 8388600
IF BleEncodeFLOAT(attr$,10000000,0,2)!=0 THEN
  PRINT "\nFailed to encode to FLOAT"
ENDIF
```

Expected Output:

```
0
Failed to encode to FLOAT
Failed to encode to FLOAT
```

BLEENCODEFLOAT is an extension function.

BleEncodeSFLOATEX

FUNCTION

This function overwrites two bytes in a string at a specified offset as short 16 bit float value. If the string is not long enough, it is extended with the extended block uninitialized. Then the bytes are overwritten.

If the nIndex is such that the new string length exceeds the maximum attribute length, this function fails. The maximum attribute length can be obtained using the function SYSINFO(n) where n is 2013. The Bluetooth specification allows a length between 1 and 512.

BLEENCODESFLOATEX(attr$,nData, nIndex)

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

Arguments:

attr$* byRef *attr$ AS STRING
This argument is the string that will be written to an attribute

nData byVal nData AS INTEGER
The 32 bit value is converted into a 2 byte IEEE-11073 16 bit SFLOAT consisting of a 12 bit signed mantissa and a 4 bit signed exponent. This means a signed 32 bit value always fits in such a FLOAT enitity, but there will be a loss in significance to 12 from 32.

nIndex byVal *nIndex* AS INTEGER
This is the zero based index into the string attr$ where the new fragment of data is written. If the string attr$ is not long enough to accommodate the index plus the length of the fragment, it is extended. If the new length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function fails.

Interactive Command: NO

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

DIM rc, mantissa, exp
DIM attr$ : attr$=""

//write 2,147,483,647 as SFLOAT to index 0
rc=BleEncodeSFloatEX(attr$,2147483647,0)
rc=BleDecodeSFloat(attr$,mantissa,exp,0)
PRINT "\nThe number stored is ";mantissa;" x 10^";exp
```

Expected Output:

```
The number stored is 214 x 10^7
```

BLEENCODESFLOAT is an extension function.

BleEncodeSFLOAT

FUNCTION

This function overwrites two bytes in a string at a specified offset as short 16 bit float value. If the string is not long enough, it is extended with the new block uninitialized. Then the byte specified is overwritten.

If the nIndex is such that the new string length exceeds the maximum attribute length, this function fails. The maximum attribute length can be obtained using the function SYSINFO(n) where n is 2013. The Bluetooth specification allows a length between 1 and 512.

BLEENCODESFLOAT(attr$, nMatissa, nExponent, nIndex)

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

Arguments:

attr$* byRef *attr$ AS STRING
This argument is the string that will be written to an attribute

nMatissa — byVal n AS INTEGER
This must be in the range -2046 to +2046 or the function fails. The data is written in little endian so the least significant byte is at the lower memory address.

Note that the range is not +/- 2048 because after encoding the following 2 byte values have special meaning:

0x07FF	NaN (Not a Number)
0x0800	NRes (Not at this resolution)
0x07FE	+ INFINITY
0x0802	- INFINITY
0x0801	Reserved for future use

nExponent — byVal n AS INTEGER
This value must be in the range -8 to 7 or the function fails.

nIndex — byVal *nIndex* AS INTEGER
This is the zero based index into the string attr$ where the new fragment of data is written. If the string attr$ is not long enough to accommodate the index plus the length of the fragment, it is extended. If the new length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function fails.

Interactive Command: NO

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

DIM rc
DIM attr$ : attr$=""

SUB Encode(BYVAL mantissa, BYVAL exp)
    IF BleEncodeSFloat(attr$,mantissa,exp,2)!=0 THEN
        PRINT "\nFailed to encode to SFLOAT"
    ELSE
        PRINT "\nSuccess"
    ENDIF
ENDSUB

Encode(1234,-4)      //1234 x 10^-4
Encode(1234,10)      //1234 x 10^10 will fail because exponent too large
Encode(10000,0)      //10000 x 10^0 will fail because mantissa too large
```

Expected Output:

```
Success
Failed to encode to SFLOAT
Failed to encode to SFLOAT
```

BLEENCODESFLOAT is an extension function.

BleEncodeTIMESTAMP

FUNCTION

This function overwrites a 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 will be represented as "\14\0D\05\05\0A\1F\18"

If the nIndex is such that the new string length exceeds the maximum attribute length, this function fails. The maximum length of an attribute as implemented can be obtained using the function SYSINFO(n) where n is 2013. The Bluetooth specification allows a length between 1 and 512.

Note: When the attr$ string variable is updated, the two byte year field is converted into a 16 bit integer. Hence \14\0D gets converted to \DD\07

BLEENCODETIMESTAMP (attr$, timestamp$, nIndex)

Returns: INTEGER, a result code. The 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 an exactly 7 byte string as described above. For example 5 May 2013 10:31:24 is entered "\14\0D\05\05\0A\1F\18"

nIndex — **byVal** ***nIndex*** **AS INTEGER**
This is the zero based index into the string attr$ where the new fragment of data is written. If the string attr$ is not long enough to accommodate the index plus the length of the fragment it is extended. If the new length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function fails.

Interactive Command: NO

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

DIM rc, ts$
DIM attr$ : attr$=""

//write the timestamp <5 May 2013 10:31:24>
ts$="\14\0D\05\05\0A\1F\18"
PRINT BleEncodeTimestamp(attr$,ts$,0)
```

Expected Output:

```
0
```

BLEENCODETIMESTAMP is an extension function.

BleEncodeSTRING

FUNCTION

This function overwrites a substring at a specified offset with data from another substring of a string. If the destination string is not long enough, it is extended with the new block uninitialized. Then the byte is overwritten.

If the nIndex is such that the new string length exceeds the maximum attribute length, this function fails. The maximum length of an attribute as implemented can be obtained using the function SYSINFO(n) where n is 2013. The Bluetooth specification allows a length between 1 and 512.

BleEncodeSTRING (attr$,nIndex1 str$, nIndex2,nLen)

Returns:	INTEGER, a result code. The typical value is 0x0000, indicating a successful operation.
Arguments:	
attr$	byRef *attr$* AS STRING This argument is the string that will be written to an attribute
nIndex1	byVal *nIndex1* AS INTEGER This is the zero based index into the string attr$ where the new fragment of data is written If the string attr$ is not long enough to accommodate the index plus the length of the fragment it is extended. If the new length exceeds the maximum allowable length of an attribute (see SYSINFO(2013)), this function fails.
str$	byRef *str$* AS STRING This contains the source data which is qualified by the nIndex2 and nLen arguments that follow.
nIndex2	byVal *nIndex2* AS INTEGER This is the zero based index into the string str$ from which data is copied. No data is copied if this is negative or greater than the string
nLen	byVal *nLen* AS INTEGER This species the number of bytes from offset nIndex2 to be copied into the destination string. It is clipped to the number of bytes left to copy after the index.

Interactive Command: NO

```
//Example :: BleEncodeString.sb (See in BL600CodeSnippets.zip)
DIM rc, attr$, ts$ : ts$="Hello World"
//write "Wor" from "Hello World" to the attribute at index 2
rc=BleEncodeString(attr$,2,ts$,6,3)
PRINT attr$
```

Expected Output:

```
\00\00Wor
```

BLEENCODESTRING is an extension function.

BleEncodeBITS

FUNCTION

This function overwrites some bits of a string at a specified bit offset with data from an integer which is treated as a bit array of length 32. If the destination string is not long enough, it is extended with the new extended block uninitialized. Then the bits specified are overwritten.

If the nIndex is such that the new string length exceeds the maximum attribute length, this function fails. The maximum length of an attribute as implemented can be obtained using the function SYSINFO(n) where n is 2013. The Bluetooth specification allows a length between 1 and 512; hence the (nDstIdx + nBitLen) cannot be greater than the max attribute length times 8.

BleEncodeBITS (attr$,nDstIdx, srcBitArr , nSrcIdx, nBitLen)

Returns: INTEGER, a result code. The 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 will be copied. No data is copied if this index is negative or greater than 32.

nBitLen byVal *nBitLen* AS INTEGER
This species the number of bits from offset nSrcIdx to be copied into the destination bit array represented by the string attr$. It will be clipped to the number of bits left to copy after the index nSrcIdx.

Interactive Command: NO

```
//Example :: BleEncodeBits.sb (See in BL600CodeSnippets.zip)
DIM attr$, rc, bA: bA=b'1110100001111
rc=BleEncodeBits(attr$,20,bA,7,5) : PRINT attr$ //copy 5 bits from index 7 to attr$
```

Expected Output:

```
\00\00\A0\01
```

BLEENCODEBITS is an extension function.

Attribute Decoding Functions

Data in a Characteristic is stored in a Value attribute, a byte array. Multibyte Characteristic Descriptors content are stored similarly. Those bytes are manipulated in smartBASIC applications using STRING variables.

Attibute data is stored in little endian format.

This section describes decoding functions that allow attribute strings to be read from smaller bytewise subfields more efficiently than the generic STRXXXX functions that are made available in *smart* BASIC.

Please note that CCCD and SCCD Descriptors are special cases as they are defined as having just 2 bytes which are treated as 16 bit integers mapped to INTEGER variables in smartBASIC.

BleDecodeS8

FUNCTION

This function reads a single byte in a string at a specified offset into a 32bit integer variable with sign extension. If the offset points beyond the end of the string then this function fails and returns zero.

BLEDECODES8 (attr$,nData, nIndex)

Returns: INTEGER, 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.

Interactive Command: NO

```
//Example :: BleDecodeS8.sb (See in BL600CodeSnippets.zip)

DIM chrHandle,v1,svcHandle,rc
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
DIM attr$ : attr$="\00\01\02\03\04\85\86\87\88\89"
DIM uuid : uuid = 0x1853


//create random service just for this example
rc=BleSvcCommit(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=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:

BLEDECODES8 is an extension function.

BleDecodeU8

FUNCTION

This function reads a single byte in a string at a specified offset into a 32bit integer variable without sign extension. If the offset points beyond the end of the string, this function fails.

BLEDECODEU8 (attr$,nData, nIndex)

Returns: INTEGER, the number of bytes extracted from the attribute string. Can be less than the size expected if the nIndex parameter is positioned towards the end of the string.

Arguments:

attr$* — byRef *attr$ AS STRING
This references the attribute string from which the function reads.

nData — byRef nData AS INTEGER
This references an integer to be updated with the 8 bit data from attr$, without sign extension.

nIndex — byVal *nIndex* AS INTEGER
This is the zero based index into the string attr$ from which data is read. If the string attr$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.

Interactive Command: NO

```
//Example :: BleDecodeU8.sb (See in BL600CodeSnippets.zip)

DIM chrHandle,v1,svcHandle,rc
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
DIM attr$ : attr$="\00\01\02\03\04\85\86\87\88\89"
DIM uuid : uuid = 0x1853

rc=BleSvcCommit(1, BleHandleUuid16(uuid),svcHandle)

rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)
```

```
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:

BLEDECODEU8 is an extension function.

BleDecodeS16

FUNCTION

This function reads two bytes in a string at a specified offset into a 32bit integer variable with sign extension. If the offset points beyond the end of the string then this function fails.

BLEDECODES16 (attr$,nData, nIndex)

Returns: INTEGER, the number of bytes extracted from the attribute string. Can be less than the size expected if the nIndex parameter is positioned towards the end of the string.

Arguments:

attr$* — byRef *attr$ AS STRING
This references the attribute string from which the function reads.

nData — byRef nData AS INTEGER
This references an integer to be updated with the 2 byte data from attr$, after sign extension.

nIndex — byVal *nIndex* AS INTEGER
This is the zero based index into the string attr$ from which data is read. If the string attr$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.

Interactive Command: NO

```
//Example :: BleDecodeS16.sb (See in BL600CodeSnippets.zip)

DIM chrHandle,v1,svcHandle,rc
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
DIM attr$ : attr$="\00\01\02\03\04\85\86\87\88\89"
DIM uuid : uuid = 0x1853

rc=BleSvcCommit(1, BleHandleUuid16(uuid),svcHandle)

rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)
```

```
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:

```

```

BLEDECODES16 is an extension function.

BleDecodeU16

This function reads two bytes from a string at a specified offset into a 32bit integer variable without sign extension. If the offset points beyond the end of the string then this function fails.

BLEDECODEU16 (attr$,nData, nIndex)

FUNCTION

Returns: INTEGER, the number of bytes extracted from the attribute string. Can be less than the size expected if the nIndex parameter is positioned towards the end of the string.

Arguments:

attr$* — byRef *attr$ AS STRING
This references the attribute string from which the function reads.

nData — byRef nData AS INTEGER
This references an integer to be updated with the 2 byte data from attr$, without sign extension.

nIndex — byVal *nIndex* AS INTEGER
This is the zero based index into the string attr$ from which data is read. If the string attr$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.

Interactive Command: NO

```
//Example :: BleDecodeU16.sb (See in BL600CodeSnippets.zip)

DIM chrHandle,v1,svcHandle,rc
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
DIM attr$ : attr$="\00\01\02\03\04\85\86\87\88\89"
DIM uuid : uuid = 0x1853
```

```
rc=BleSvcCommit(1, BleHandleUuid16(uuid),svcHandle)

rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)

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:

```

```

BLEDECODEU16 is an extension function.

BleDecodeS24

FUNCTION

This function reads three bytes in a string at a specified offset into a 32bit integer variable with sign extension. If the offset points beyond the end of the string, this function fails.

BLEDECODES24 (attr$,nData, nIndex)

Returns: INTEGER, the number of bytes extracted from the attribute string. Can be less than the size expected if the nIndex parameter is positioned towards the end of the string.

Arguments:

attr$* — byRef *attr$ AS STRING
This references the attribute string from which the function reads.

nData — byRef nData AS INTEGER
This references an integer to be updated with the 3 byte data from attr$, with sign extension.

nIndex — byVal *nIndex* AS INTEGER
This is the zero based index into the string attr$ from which data is read. If the string attr$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.

Interactive Command: NO

```
//Example :: BleDecodeS24.sb (See in BL600CodeSnippets.zip)

DIM chrHandle,v1,svcHandle,rc
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
DIM attr$ : attr$="\00\01\02\03\04\85\86\87\88\89"
DIM uuid : uuid = 0x1853

rc=BleSvcCommit(1, BleHandleUuid16(uuid),svcHandle)

rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)

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:

BLEDECODES24 is an extension function.

BleDecodeU24

FUNCTION

This function reads three bytes from a string at a specified offset into a 32bit integer variable *without* sign extension. If the offset points beyond the end of the string then this function fails.

BLEDECODEU24 (attr$,nData, nIndex)

Returns: INTEGER, the number of bytes extracted from the attribute string. Can be less than the size expected if the nIndex parameter is positioned towards the end of the string.

Arguments:

attr$* byRef *attr$ AS STRING
This references the attribute string from which the function reads.

nData byRef nData AS INTEGER
This references an integer to be updated with the 3 byte data from attr$, without sign extension.

nIndex	byVal *nIndex* AS INTEGER This is the zero based index into the string attr$ from which data is read. If the string attr$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.

Interactive Command: NO

```
//Example :: BleDecodeU24.sb (See in BL600CodeSnippets.zip)

DIM chrHandle,v1,svcHandle,rc
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
DIM attr$ : attr$="\00\01\02\03\04\85\86\87\88\89"
DIM uuid : uuid = 0x1853

rc=BleSvcCommit(1, BleHandleUuid16(uuid),svcHandle)

rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)

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:

BLEDECODEU24 is an extension function.

BleDecode32

FUNCTION

This function reads four bytes in a string at a specified offset into a 32bit integer variable. If the offset points beyond the end of the string, this function fails.

BLEDECODE32 (attr$,nData, nIndex)

Returns: INTEGER, the number of bytes extracted from the attribute string. Can be less than the size expected if the nIndex parameter is positioned towards the end of the string.

Arguments:

attr$* — byRef *attr$ AS STRING
This references the attribute string from which the function reads.

nData — byRef nData AS INTEGER
This references an integer to be updated with the 3 byte data from attr$, after sign extension.

nIndex — byVal *nIndex* AS INTEGER
This is the zero based index into the string attr$ from which data is read. If the string attr$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.

Interactive Command: NO

```
//Example :: BleDecode32.sb (See in BL600CodeSnippets.zip)

DIM chrHandle,v1,svcHandle,rc
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
DIM attr$ : attr$="\00\01\02\03\04\85\86\87\88\89"
DIM uuid : uuid = 0x1853

rc=BleSvcCommit(1, BleHandleUuid16(uuid),svcHandle)

rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)

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:

```

```

BLEDECODE32 is an extension function.

BleDecodeFLOAT

FUNCTION

This function reads four bytes in a string at a specified offset into a couple of 32bit integer variables. The decoding results in two variables, the 24 bit signed mantissa and the 8 bit signed exponent. If the offset points beyond the end of the string, this function fails.

BLEDECODEFLOAT (attr$, nMatissa, nExponent, nIndex)

Returns: INTEGER, the number of bytes extracted from the attribute string. Can be less than the size expected if the nIndex parameter is positioned towards the end of the string.

Arguments:

attr$* — byRef *attr$ AS STRING
This references the attribute string from which the function reads.

nMantissa — byRef nMantissa AS INTEGER
This is updated with the 24 bit mantissa from the 4 byte object.

If nExponent is 0, you MUST check for the following special values:

`0x007FFFFF`	NaN (Not a Number)
`0x00800000`	NRes (Not at this resolution)
`0x007FFFFE`	+ INFINITY
`0x00800002`	- INFINITY
`0x00800001`	Reserved for future use

nExponent — byRef nExponent AS INTEGER
This is updated with the 8 bit mantissa. If it is zero, check nMantissa for special cases as stated above.

nIndex — byVal *nIndex* AS INTEGER
This is the zero based index into the string attr$ from which data is read. If the string attr$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.

Interactive Command: NO

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

DIM chrHandle,v1,svcHandle,rc, mantissa, exp
```

```
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
DIM attr$ : attr$="\00\01\02\03\04\85\86\87\88\89"
DIM uuid : uuid = 0x1853

rc=BleSvcCommit(1, BleHandleUuid16(uuid),svcHandle)
rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)

rc=BleCharValueRead(chrHandle,attr$)

//read 4 bytes FLOAT from index 2 in the string
rc=BleDecodeFloat(attr$,mantissa,exp,2)
PRINT "\nThe number read is ";mantissa;" x 10^";exp

//read 4 bytes FLOAT from index 6 in the string
rc=BleDecodeFloat(attr$,mantissa,exp,6)
PRINT "\nThe number read is ";mantissa;"x 10^";exp
```

Expected Output:

```
The number read is 262914*10^-123
The number read is -7829626*10^-119
```

BLEDECODEFLOAT is an extension function.

BleDecodeSFLOAT

FUNCTION

This function reads two bytes in a string at a specified offset into a couple of 32bit integer variables. The decoding results in two variables, the 12 bit signed maintissa and the 4 bit signed exponent. If the offset points beyond the end of the string then this function fails.

BLEDECODESFLOAT (attr$, nMatissa, nExponent, nIndex)

Returns: INTEGER, the number of bytes extracted from the attribute string. Can be less than the size expected if the nIndex parameter is positioned towards the end of the string.

Arguments:

attr$* — **byRef *attr$ AS STRING**
This references the attribute string from which the function reads.

nMantissa — **byRef nMantissa AS INTEGER**
This is updated with the 12 bit mantissa from the 2 byte object.

If the nExponent is 0, you MUST check for the following special values:

0x007FFFFF	NaN (Not a Number)
0x00800000	NRes (Not at this resolution)
0x007FFFFE	+ INFINITY
0x00800002	- INFINITY
0x00800001	Reserved for future use

nExponent byRef nExponent AS INTEGER
This is updated with the 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.

Interactive Command: NO

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

 DIM chrHandle,v1,svcHandle,rc, mantissa, exp
 DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
 DIM attr$ : attr$="\00\01\02\03\04\85\86\87\88\89"
 DIM uuid : uuid = 0x1853

 rc=BleSvcCommit(1, BleHandleUuid16(uuid),svcHandle)
 rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
 rc=BleCharCommit(svcHandle,attr$,chrHandle)

 rc=BleCharValueRead(chrHandle,attr$)

 //read 2 bytes FLOAT from index 2 in the string
 rc=BleDecodeSFloat(attr$,mantissa,exp,2)
 PRINT "\nThe number read is ";mantissa;" x 10^";exp

 //read 2 bytes FLOAT from index 6 in the string
 rc=BleDecodeSFloat(attr$,mantissa,exp,6)
 PRINT "\nThe number read is ";mantissa;"x 10^";exp
```

Expected Output:

```
The number read is 770 x 10^0
The number read is 1926x 10^-8
```

BLEDECODESFLOAT is an extension function.

BleDecodeTIMESTAMP

FUNCTION

This function reads 7 bytes from string an offset into an attribute string. If the offset plus 7 bytes points beyond the end of the string then this function fails.

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 will be represented in the source as "\DD\07\05\05\0A\1F\18" and the year will be translated into a century and year so that the destination string will be "\14\0D\05\05\0A\1F\18"

BLEDECODETIMESTAMP (attr$, timestamp$, nIndex)

Returns: INTEGER, the number of bytes extracted from the attribute string. Can be less than the size expected if the nIndex parameter is positioned towards the end of the string.

Arguments:

attr$* byRef *attr$ AS STRING
This references the attribute string from which the function reads.

timestamp$* byRef *timestamp$ AS STRING
On exit this is an exact 7 byte string as described above. For example 5 May 2013 10:31:24 is stored as "\14\0D\05\05\0A\1F\18"

nIndex byVal *nIndex* AS INTEGER
This is the zero based index into the string attr$ from which data is read. If the string attr$ is not long enough to accommodate the index plus the number of bytes to read, this function fails.

Interactive Command: NO

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

DIM chrHandle,v1,svcHandle,rc, ts$
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
//5th May 2013, 10:31:24
DIM attr$ : attr$="\00\01\02\DD\07\05\05\0A\1F\18"
DIM uuid : uuid = 0x1853

rc=BleSvcCommit(1, BleHandleUuid16(uuid),svcHandle)
rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)

rc=BleCharValueRead(chrHandle,attr$)

//read 7 byte timestamp from the index 3 in the string
rc=BleDecodeTimestamp(attr$,ts$,3)
PRINT "\nTimestamp = "; StrHexize$(ts$)
```

Expected Output:

```
Timestamp = 140D05050A1F18
```

BLEENCODETIMESTAMP is an extension function.

BleDecodeSTRING

FUNCTION

This function reads a maximum number of bytes from an attribute string at a specified offset into a destination string. This function will not fail as the output string can take truncated strings.

BLEDECODESTRING (attr$, nIndex, dst$, nMaxBytes)

Returns: INTEGER, the number of bytes extracted from the attribute string. Can be less than the size expected if the nIndex parameter is positioned towards the end of the string.

Arguments:

attr$	byRef *attr$* AS STRING This references the attribute string from which the function reads.
nIndex	byVal *nIndex* AS INTEGER This is the zero based index into string attr$ from which data is read.
dst$	byRef dst$ AS STRING This argument is a reference to a string that will be updated with up to nMaxBytes of data from the index specified. A shorter string will be returned if there are not enough bytes beyond the index.
nMaxBytes	byVal nMaxBytes AS INTEGER This specifies the maximum number of bytes to read from attr$.

Interactive Command: NO

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

DIM chrHandle,v1,svcHandle,rc, ts$,decStr$
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
//"ABCDEFGHIJ"
DIM attr$ : attr$="41\42\43\44\45\46\47\48\49\4A"
DIM uuid : uuid = 0x1853

rc=BleSvcCommit(1, BleHandleUuid16(uuid),svcHandle)
rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)

rc=BleCharValueRead(chrHandle,attr$)

//read max 4 bytes from index 3 in the string
rc=BleDecodeSTRING(attr$,3,decStr$,4)
PRINT "\nd$=";decStr$

//read max 20 bytes from index 3 in the string - will be truncated
rc=BleDecodeSTRING(attr$,3,decStr$,20)
PRINT "\nd$=";decStr$


//read max 4 bytes from index 14 in the string - nothing at index 14
rc=BleDecodeSTRING(attr$,14,decStr$,4)
PRINT "\nd$=";decStr$
```

Expected Output:

```
d$=CDEF
d$=CDEFGHIJ
d$=
```

BLEDECODESTRING is an extension function.

BleDecodeBITS

FUNCTION

This function reads bits from an attribute string at a specified offset (treated as a bit array) into a destination integer object (treated as a bit array of fixed size of 32). This implies a maximum of 32 bits can be read. This function will not fail as the output bit array can take truncated bit blocks.

BLEDECODEBITS (attr$, nSrcIdx, dstBitArr, nDstIdx,nMaxBits)

Returns: INTEGER, the number of bits extracted from the attribute string. Can be less than the size expected if the nSrcIdx parameter is positioned towards the end of the source string or if nDstIdx will not allow more to be copied.

Arguments:

attr$* — byRef *attr$ AS STRING
This references the attribute string from which to read, treated as a bit array. Hence a string of 10 bytes will be an array of 80 bits.

nSrcIdx — byVal *nSrcIdx* AS INTEGER
This is the zero based bit index into the string attr$ from which data is read. E.g. the third bit in the second byte is index number 10.

dstBitArr — byRef *dstBitArr* AS INTEGER
This argument references an integer treated as an array of 32 bits into which data is copied. Only the written bits are modified.

nDstIdx — byVal *nDstIdx* AS INTEGER
This is the zero based bit index into the bit array dstBitArr where the data is written to.

nMaxBits — byVal *nMaxBits* AS INTEGER
This argument specifies the maximum number of bits to read from attr$. Due to the destination being an integer variable, it cannot be greater than 32. Negative values are treated as zero.

Interactive Command: NO

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

DIM chrHandle,v1,svcHandle,rc, ts$,decStr$
DIM ba : ba=0
DIM mdVal : mdVal = BleAttrMetadata(1,1,50,0,rc)
//"ABCDEFGHIJ"
DIM attr$ : attr$="41\42\43\44\45\46\47\48\49\4A"
DIM uuid : uuid = 0x1853

rc=BleSvcCommit(1, BleHandleUuid16(uuid),svcHandle)
rc=BleCharNew(0x07,BleHandleUuid16(0x2A1C),mdVal,0,0)
rc=BleCharCommit(svcHandle,attr$,chrHandle)

rc=BleCharValueRead(chrHandle,attr$)

//read max 14 bits from index 20 in the string to index 10
rc=BleDecodeBITS(attr$,20,ba,10,14)
```


Europe: +44-1628-858-940
Hong Kong: +852 2923 0610
wireless.support@lairdtech.com
www.lairdtech.com/bluetooth

```
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:

BLEDECODEBITS is an extension function.

Pairing/Bonding Functions

This section describes all functions related to the pairing and bonding manager which manages trusted devices. The database stores information like the address of the trusted device along with the security keys. At the time of writing this manual a maximum of 4 devices can be stored in the database.

The command AT I 2012 or at runtime SYSINFO(2012) returns the maximum number of devices that can be saved in the database

The type of information that can be stored for a trusted device is:

- The MAC address of the trusted device.
- The eDIV and eRAND for the long term key.
- A 16 byte Long Term Key (LTK).
- The size of the long term key.
- A flag to indictate if the LTK is authenticated – Man-In-The-Middle (MITM) protection.
- A 16 byte Indentity Resolving Key (IRK).
- A 16 byte Connection Signature Resolving Key (CSRK)

Whisper Mode Pairing

BLE provides for simple secure pairing with or without man-in-the-middle attack protection. To enhance security while a pairing is in progress the specification has provided for Out-of-Band pairing where the shared secret information is exchanged by means other than the Bluetooth connection. That mode of pairing is currently not exposed.

Laird have provided an additional mechanism for bonding using the standard inbuilt simple secure pairing which is called Whisper Mode pairing. In this mode, when a pairing is detected to be in progress, the transmit power is automatically reduced so that the 'bubble' of influence is reduced and thus a proximity based enhanced security is achieved.

To take advantage of this pairing mechanism, use the function BleTxPwrWhilePairing() to reduce the transmit power for the short duration that the pairing is in progress.

Tests have shown that setting a power of -55 using BleTxPwrWhilePairing() will create a 'bubble' of about 30cm radius, outside which pairing will not succeed. This will be reduced even further if the BL600 module is in a case which affects radio transmissions.

BleBondMngrErase

Note: For firmware versions prior to 1.4.X.Y, this subroutine has a bug. It occurs when the subroutine is called during radio activity.

Workaround when advertising:

1. Stop adverts by calling BleAdvertStop()
2. Call BleBondMngrErase()
3. Restart adverts using BleAdvertStart()

SUBROUTINE

This subroutine deletes the entire trusted device database if the supplied parameter is 0. Other values of the parameter are reserved for future use.

Note: In Interactive Mode, the command AT+BTD* can also be used to delete the database.

BLEBONDMNGRERASE (nFutureUse)

Arguments:

nFutureUse	**byVal nFutureUse AS INTEGER** This shall be set to 0.
Interactive Command:	NO

Workaround for FW **1.3.57.0 and earlier** when there is radio activity:

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

DIM rc

rc=BleAdvertStop()
BleBondMngrErase(0)
```

For FW **1.4.X.Y and newer:**

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

DIM rc

BleBondMngrErase(0)
```

BLEBONDMNGRERASE is an extension function.

BleBondMngrGetInfo

FUNCTION

This function retrieves the MAC address and other information from the trusted device database via an index.

Note: Do not rely on a device in the database mapping to a static index. New bondings will change the position in the database.

BLEBONDMNGRGETINFO (nIndex, addr$, nExtraInfo)

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

Arguments:

nIndex **byVal nIndex AS INTEGER**
This is an index in the range 0 to 1, less than the value returned by SYSINFO(2012).

*addr$* **byRef addr$ AS STRING**
On exit if nIndex points to a valid entry in the database, this variable contains a MAC address exactly 7 bytes long. The first byte identifies public or private random address. The next 6 bytes are the address.

nExtraInfo **byRef nExtraInfo AS INTEGER**
On exit if nIndex points to a valid entry in the database, this variable contains a composite integer value where the lower 16 bits are the eDIV. Bit 16 is set if the IRK (Identity Resolving Key) exists for the trusted device and bit 17 is set if the CSRK (Connection Signing Resolving Key) exists for the trusted device.

Interactive Command: NO

```
//Example :: BleBondMngrGetInfo.sb (See in BL600CodeSnippets.zip)
#define BLE_INV_INDEX            24619
DIM rc, addr$, exInfo
rc = BleBondMngrGetInfo(0,addr$,exInfo)  //Extract info of device at index 1

IF rc==0 THEN
    PRINT "\nMAC address: ";addr$
    PRINT "\nInfo: ";exInfo
ELSEIF rc==BLE_INV_INDEX   THEN
    PRINT "\nInvalid index"
ENDIF
```

Expected Output when valid entry present in database:

```
MAC address: \00\BC\B1\F3x3\AB
Info: 97457
```

Expected Output with invalid index:

```
Invalid index
```

BLEBONDMNGRGETINFO is an extension function.

Virtual Serial Port Service – Managed test when dongle and application availbable

This section describes all the events and routines used to interact with a managed virtual serial port service.

"Managed" means there is a driver consisting of transmit and receive ring buffers that isolate the BLE service from the *smart*BASIC application. This in turn provides easy to use API functions.

Note: The driver makes the same assumption that the driver in a PC makes: If the on-air connection equates to the serial cable, there is no assumption that the cable is from the same source as prior to the disconnection. This is analogous to the way that a PC cannot detect such in similar cases.

The module can present a serial port service in the local GATT Table consisting of two mandatory characteristics and two optional characteristics. One mandatory characteristic is the TX FIFO and the other is the RX FIFO, both consisting of an attribute taking up to 20 bytes. Of the optional characteristics, one is the ModemIn which consists of a single byte and only bit 0 is used as a CTS type function. The other is ModemOut, also a single byte, which is notifiable only and is used to convey an RTS flag to the client.

By default, (configurable via AT+CFG 112), Laird's serial port service is exposed with UUID's as follows:-

The UUID of the service is:	`569a`***`1101`***`-b87f-490c-92cb-11ba5ea5167c`
The UUID of the rx fifo characteristic is:	`569a`***`2001`***`-b87f-490c-92cb-11ba5ea5167c`
The UUID of the tx fifo characteristic is:	`569a`***`2000`***`-b87f-490c-92cb-11ba5ea5167c`
The UUID of the ModemIn characteristic is:	`569a`***`2003`***`-b87f-490c-92cb-11ba5ea5167c`
The UUID of the ModemOut characteristic is:	`569a`***`2002`***`-b87f-490c-92cb-11ba5ea5167c`

Note: Laird's Base 128bit UUID is `569aXXXX-b87f-490c-92cb-11ba5ea5167c` where XXXX is a 16 bit offset. We recommend, to save RAM, that you create a 128 bit UUID of your own and manage the 16 bit space accordingly, akin to what the Bluetooth SIG does with their 16 bit UUIDs.

If command AT+CFG 112 1 is used to change the value of the config key 112 to 1 then Nordic's serial port service is exposed with UUID's as follows:-

The UUID of the service is:	`6e40`***`0001`***`-b5a3-f393-e0a9-e50e24dcca9e`
The UUID of the rx fifo characteristic is:	`6e40`***`0002`***`-b5a3-f393-e0a9-e50e24dcca9e`
The UUID of the tx fifo characteristic is:	`6e40`***`0003`***`-b5a3-f393-e0a9-e50e24dcca9e`

Note: The first byte in the UUID's above is the most significant byte of the UUID.

The 'rx fifo characteristic' is for data that **comes to** the module and the 'tx fifo characteristic' is for data that **goes out** from the module. This means a GATT Client using this service will send data by writing into the 'rx fifo characteristic' and will get data from the module via a value notification.

The 'rx fifo characteristic' is defined with no authentication or encryption requirements, a maximum of 20 bytes value attribute. The following properties are enabled:

- WRITE
- WRITE_NO_RESPONSE

The 'tx fifo characteristic' value attribute is with no authentication or encryption requirements, a maximum of 20 bytes value attribute. The following properties are enabled:

- NOTIFY (The CCCD descriptor also requires no authentication/encryption)

The ‘ModemIn characteristic’ is defined with no authentication or encryption requirements, a single byte attribute. The following properties are enabled:

- WRITE
- WRITE_NO_RESPONSE

The ‘ModemOut characteristic’ value attribute is with no authentication or encryption requirements, a single byte attribute. The following properties are enabled:

- NOTIFY (The CCCD descriptor also requires no authentication/encryption)

For ModemIn, only bit zero is used, which is set by 1 when the client can accept data and 0 when it cannot (inverse logic of CTS in UART functionality). Bits 1 to 7 are for future use and should be set to 0.

For ModemOut, only bit zero is used which is set by 1 when the client can send data and 0 when it cannot (inverse logic of RTS in UART functionality). Bits 1 to 7 are for future use and should be set to 0.

Note: Both flags in ModemIn and ModemOut are suggestions to the peer, just as in a UART scenario. If the peer decides to ignore the suggestion and data is kept flowing, the only coping mechanism is to drop new data as soon as internal ring buffers are full.

Given that the outgoing data is **notified** to the client, the ‘tx fifo characteristic’ has a Client Configuration Characteristic (CCCD) which must be set to 0x0001 to allow the module to send any data waiting to be sent in the transmit ring buffer. While the CCCD value is not set for notifications, writes by the *smart* BASIC application result in data being buffered. If the buffer is full the appropriate write routine indicates how many bytes actually got absorbed by the driver. In the background, the transmit ring buffer is emptied with one or more indicate or notify messages to the client. When the last bytes from the ring buffer are sent, **EVVSPTXEMPTY** is thrown to the *smart*BASIC application so that it can write more data if it chooses.

When GATT Client sends data to the module by writing into the ‘rx fifo characteristic’ the managing driver will immediately save the data in the receive ring buffer if there is any space. If there is no space in the ring buffer, data is discarded. After the ring buffer is updated, event **EVVSPRX** is thrown to the *smart*BASIC runtime engine so that an application can read and process the data.

Similarly, given that ModemOut is **notified** to the client, the ModemOut characteristic has a Client Configuration Characteristic (CCCD) which must be set to 0x0001. By default, in a connection the RTS bit in ModemOut is set to 1 so that the VSP driver assumes there is buffer space in the peer to send data. The RTS flag is affected by the thresholds of 80 and 120 which means the when opening the VSP port the rxbuffer cannot be less than 128 bytes.

It is intended that in a future release it will be possible to register a ‘custom’ service and bind that with the virtual service manager to allow that service to function in the managed environment. This allows the application developer to interact with any GATT client implementing a serial port service, whether one currently deployed or one that the Bluetooth SIG adopts.

VSP Configuration

Given that VSP operation can happen in command mode the ability to configure it and save the new configuration in non-volatile memory is available. For example, in bridge mode, the baudrate of the uart can be specified to something other than the default 9600. Configuration is done using the AT+CFG command and refer to the section describing that command for further details. The configuration id pertinent to VSP are 100 to 116 inclusive

Command & 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. By default, bridge mode is not enabled and the command AT+CFG 100 0x8107 should be supplied either over the UART or the on-air interactive mode.

Note: If $autorun$ file exists in the file system, the bridge mode is always suppressed regardless of the state of the nAutorun input signal.

The main purpose of interactive mode operation is to facilitate the download of an autorun *smart*BASIC application. This allows the module to be soldered into an end product without preconfiguration and then the application can be downloaded over the air once the product has been pre-tested. It is the *smart*BASIC application that is downloaded over the air, NOT the firmware. Due to this principle reason for use in production, to facilitate multiple programming stations in a locality the transmit power is limited to -12dBm. It can be changed by changing the 109 config key using the command AT+CFG.

The default operation of this virtual serial port service is dependent on one of the digital input lines being pulled high externally. Consult the hardware manual for more information on the input pin number. By default it is SIO7 on the module, but it can be changed by setting the config key 100 via AT+CFG.

You can interact with the BL600 over the air via the Virtual Serial Port Service using the iOS “BL600 Serial” app, available free on the Apple App Store.

You may download *smart*BASIC applications using a Windows application, which will be available for free from Laird. The PC must be BLE enabled using a Laird supplied adapter. Contact your local FAE for details.

As most of the AT commands are functional, you may obtain information such as version numbers by sending the command AT I 3 to the module over the air.

Note that the module enters interactive mode only if there is no autorun application or if the autorun application exits to interactive mode by design. Hence in normal operation where a module is expected to have an autorun application the virtual serial port service will not be registered in the GATT table.

If the application requires the virtual serial port functionality then it shall have to be registered programmatically using the functions that follow in subsequent subsections. These are easy to use high level functions such as OPEN/READ/WRITE/CLOSE.

VSP (Virtual Serial Port) Events

In addition to the routines for manipulating the Virtual Serial Port (VSP) service, when data arrives via the receive characteristic it is stored locally in an underlying ring buffer and then an event is generated.

Similarly when the transmit buffer is emptied, events are thrown from the underlying drivers so that user *smart* BASIC code in handlers can perform user defined actions.

The following is a list of events generated by VSP service managed code which can be handled by user code.

EVVSPRX	This event is generated when data has arrived and has been stored in the local ring buffer to be read using BleVSpRead().

EVVSPTXEMPTY	This event is generated when the last byte is transmitted using the outgoing data characteristic via a notification or indication.

Use the iOS BL600 Serial app and connect to your BL600 to test this sample app.

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

DIM tx$,rc,x,scRpt$,adRpt$,addr$,hndl

//handler for data arrival
FUNCTION HandlerBleVSpRx() AS INTEGER
    //print the data that arrived
    DIM n,rx$
    n = BleVSpRead(rx$,20)
    PRINT "\nrx=";rx$
ENDFUNC 1

//handler when VSP tx buffer is empty
FUNCTION HandlerVSpTxEmpty() AS INTEGER
    IF x==0 THEN
        rc = BleVSpWrite("tx buffer empty")
        x=1
    ENDIF
ENDFUNC 1

PRINT "\nDevice name is "; BleGetDeviceName$()

//Open the VSP
PRINT "\n"; BleVSpOpen(128,128,0,hndl)
//Initialise a scan report
PRINT "\n"; BleScanRptInit(scRpt$)
//Advertise the VSP service in the scan report so
//that it can be seen by the client
PRINT "\n"; BleAdvRptAddUuid128(scRpt$,hndl)
adRpt$=""
PRINT "\n"; BleAdvRptsCommit(adRpt$,scRpt$)
addr$=""  //because we are not doing a DIRECT advert
PRINT "\n"; BleAdvertStart(0,addr$,20,300000,0)
//Now advertising so can be connectable

ONEVENT  EVVSPRX       CALL HandlerBleVSpRx
ONEVENT  EVVSPTXEMPTY  CALL HandlerVSpTxEmpty

PRINT "\nUse the iOS 'BL600 Serial' app to test this"

//wait for events and messages
WAITEVENT
```

BleVSpOpen

FUNCTION

This function opens the default VSP service using the parameters specified. The service's UUID is:
`569a`***`1101`***`-b87f-490c-92cb-11ba5ea5167c`

By default, ModemIn and ModemOut characteristics are registered in the GATT table with the Rx and Tx FIFO characteristics. To suppress Modem characteristics in the GATT table, set bit 1 in the nFlags parameter (value 2). If the virtual serial port is already open, this function fails.

BLEVSPOPEN (txbuflen,rxbuflen,nFlags,svcUuid)

Returns: INTEGER, indicating the success of command:

0	Opened successfully
0x604D	Already open
0x604E	Invalid Buffer Size
0x604C	Cannot register Service in Gatt Table while BLE connected

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

txbuflen — *byVal txbuflen AS INTEGER*
Set the transmit ring buffer size to this value. If set to 0, a default value is used by the underlying driver and use BleVspInfo(2) to determine the size.

rxbuflen — *byVal rxbuflen AS INTEGER*
Set the receive ring buffer size to this value. If set to 0, a default value is used by the underlying driver and use BleVspInfo(1) to determine the size.

nFlags — *byVal nFlags AS INTEGER*
This is a bit mask to customise the driver as follows:

Bit 0: Set to 1 to try for reliable data transfer. This uses INDICATE messages if allowed and there is a choice. Some services will only allow NOTIFY and in that case if set to 1 it will be ignored.

Bit1.31 : Reserved for future use. Set to 0

svcUuid — *byRef svcUuid AS INTEGER*
On exit, this variable is updated with a handle to the service UUID which can then be subsequently used to advertise the service in an advert report. Given that there is no BT SIG adopted Serial Port Service the UUID for the service is 128 bit, so an appropriate Advert Data element can be added to the advert or scan report using the function BleAdvRptAddUuid128() which takes a handle of that type.

Related Commands: BLEVSPINFO, BLEVSPCLOSE, BLEVSPWRITE, BLEVSPREAD, BLEVSPFLUSH

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

DIM scRpt$,adRpt$,addr$,vspSvcHndl
```

```
//Close VSP if already open
IF BleVSpInfo(0)!=0 THEN
    BleVSpClose()
ENDIF

//Open VSP
IF BleVSpOpen(128,128,0,vspSvcHndl)==0 THEN
    PRINT "\nVSP service opened"
ELSE
    PRINT "\nFailed"
ENDIF
```

Expected Output:

```
VSP service opened
```

BLEVSPOPEN is an extension function.

BleVSpClose

SUBROUTINE

This subroutine closes the managed virtual serial port which had been opened with BLEVSPOPEN. This routine is safe to call if it is already closed. When this subroutine is invoked both receive and transmit buffers are flushed. If there is data in either buffer when the port is closed, it will be lost.

BLEVSPCLOSE()

Exceptions:

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments: None

Interactive Command: No

Related Commands: BLEVSPINFO, BLEVSPOPEN, BLEVSPWRITE, BLEVSPREAD, BLEVSPFLUSH

Use the iOS "BL600 Serial" app and connect to your BL600 to test this sample app.

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

DIM tx$,rc,scRpt$,adRpt$,addr$,hndl

//handler when VSP tx buffer is empty
FUNCTION HandlerVSpTxEmpty() AS INTEGER
    PRINT "\n\nVSP tx buffer empty"
    BleVspClose()
ENDFUNC 0

PRINT "\nDevice name is "; BleGetDeviceName$()

//Open the VSP, advertise
rc = BleVSpOpen(128,128,0,hndl)
rc = BleScanRptInit(scRpt$)
rc = BleAdvRptAddUuid128(scRpt$,hndl)
```

```
adRpt$=""
rc = BleAdvRptsCommit(adRpt$,scRpt$)
addr$=""
rc = BleAdvertStart(0,addr$,20,300000,0)

//This message will send when connected to client
tx$="send this data and will close when sent"
rc = BleVSpWrite(tx$)

ONEVENT  EVVSPTXEMPTY  CALL HandlerVSpTxEmpty

WAITEVENT

PRINT "\nExiting..."
```

Expected Output:

```

```

BLEVSPCLOSE is an extension subroutine.

BleVSpInfo

FUNCTION

This function is used to query information about the virtual serial port, such as buffer lengths, whether the port is already open or how many bytes are waiting in the receive buffer to be read.

BLEVSPINFO (infoId)

Returns: INTEGER The value associated with the type of uart information requested

Exceptions:

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

infoId *byVal infoId AS INTEGER*
This specifies the information type requested as follows if the port is open:

0: 0 if closed, 1 if open, 3 if open and there is a BLE connection and 7 if the transmit fifo characteristic CCCD has been updated by the client to enable notifies or indications.
1: Receive ring buffer capacity
2: Transmit ring buffer capacity
3: Number of bytes waiting to be read from receive ring buffer
4: Free space available in transmit ring buffer

Related Commands: BLEVSPOPEN, BLEVSPCLOSE, BLEVSPWRITE, BLEVSPREAD, BLEVSPFLUSH

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

DIM hndl, rc

//Close VSP if it is open
BleVSpClose()

rc = BleVSpOpen(128,128,0,hndl)
PRINT "\nVsp State: "; BleVSpInfo(0)
PRINT "\nRx buffer capacity: "; BleVSpInfo(1)
PRINT "\nTx buffer capacity: "; BleVSpInfo(2)
PRINT "\nBytes waiting to be read from rx buffer: "; BleVSpInfo(3)
PRINT "\nFree space in tx buffer: "; BleVSpInfo(4)
BleVspClose()
PRINT "\nVsp State: "; BleVSpInfo(0)
```

Expected Output:

```

```

BLEVSPINFO is an extension subroutine.

BleVSpWrite

FUNCTION

This function is used to transmit a string of characters from the virtual serial port.

BLEVSPWRITE (strMsg)

Returns: INTEGER 0 to N : Actual number of bytes successfully written to local transmit ring buffer.

Exceptions:

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

strMsg *byRef strMsg AS STRING*

The array of bytes to be sent. STRLEN(strMsg) bytes are written to the local transmit ring buffer. If STRLEN(strMsg) and the return value are not the same, it implies that the transmit buffer did not have enough space to accommodate the data.
If the return value does not match the length of the original string, use STRSHIFTLEFT function to drop the data from the string, so subsequent calls to this function only retry with data not placed in the output ring buffer.
Another strategy is to wait for EVVSPTXEMPTY events, then resubmit data.

Interactive Command: No

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

Related Commands: BLEVSPOPEN, BLEVSPCLOSE, BLEVSPINFO, BLEVSPREAD, BLEVSPFLUSH

Use the iOS BL600 Serial app and connect to your BL600 to test this sample app.

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

DIM tx$,rc,scRpt$,adRpt$,addr$,hndl,cnt

//handler when VSP tx buffer is empty
FUNCTION HandlerVSpTxEmpty() AS INTEGER
    cnt=cnt+1
    IF cnt<= 2 THEN
        tx$="then this is sent"
        rc = BleVSpWrite(tx$)
    ENDIF

ENDFUNC 0

rc = BleVSpOpen(128,128,0,hndl)
rc = BleScanRptInit(scRpt$)
rc = BleAdvRptAddUuid128(scRpt$,hndl)
adRpt$=""
rc = BleAdvRptsCommit(adRpt$,scRpt$)
addr$=""
rc = BleAdvertStart(0,addr$,20,300000,0)
PRINT "\nDevice name is "; BleGetDeviceName$()

cnt=1
tx$="send this data and "
rc = BleVSpWrite(tx$)

ONEVENT  EVVSPTXEMPTY  CALL HandlerVSpTxEmpty

WAITEVENT

PRINT "\nExiting..."
```

Expected Output:

```
Device name is LAIRD BL600
Exiting...
```

BLEVSPWRITE is a extension subroutine.

BleVSpRead

FUNCTION

This function is used to read the content of the receive buffer and copy it to the string variable supplied.

BLEVSPREAD(strMsg,nMaxRead)

Returns: INTEGER 0 to N : The total length of the string variable. This means the caller does not need to call strlen() function to determine how many bytes in the string must be processed.

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

strMsg — *byRef strMsg AS STRING*
The content of the receive buffer is copied to this string.

nMaxRead — *byVal nMaxRead AS INTEGER*
The maximum number of bytes to read.

Interactive Command: No

> **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

Related Commands: BLEVSPOPEN, BLEVSPCLOSE, BLEVSPINFO, BLEVSPWRITE, BLEVSPFLUSH

Use the iOS BL600 Serial app and connect to your BL600 to test this sample app.

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

DIM conHndl
//Only 1 global variable because its value is used in more than 1 routine
//All other variables declared locally, inside routine that they are used in.
//More efficient because these local variables only exist in memory
//when they are being used inside their respective routines


//==============================================================================
// Open VSp and start advertising
//==============================================================================
SUB OnStartup()
    DIM rc, hndl, tx$, scRpt$, addr$, adRpt$ : adRpt$="" : addr$=""
    rc=BleVSpOpen(128,128,0,hndl)
    rc=BleScanRptInit(scRpt$)
    rc=BleAdvRptAddUuid128(scRpt$,hndl)
    rc=BleAdvRptsCommit(adRpt$,scRpt$)
    rc=BleAdvertStart(0,addr$,20,300000,0)
    PRINT "\nDevice name is "; BleGetDeviceName$()

    tx$="\nSend me some text \nTo exit the app, just tell me\n"
    rc = BleVSpWrite(tx$)
ENDSUB
```

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

//==============================================================================
//  VSP Rx buffer event handler
//==============================================================================
FUNCTION HandlerVSpRx() AS INTEGER
    DIM rc, rx$, e$ : e$="exit"
    rc=BleVSpRead(rx$,20)
    PRINT "\nMessage from client: ";rx$

    //If user has typed exit
    IF StrPos(rx$,e$,0) > -1 THEN
       EXITFUNC 0
    ENDIF
ENDFUNC 1

//==============================================================================
// BLE event handler
//==============================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\nDisconnected from client"
        EXITFUNC 0
    ENDIF
ENDFUNC 1

ONEVENT  EVVSPRX  CALL HandlerVSpRx
ONEVENT  EVBLEMSG CALL HndlrBleMsg

OnStartup()          //Calls first subroutine declared above

WAITEVENT

CloseConnections()  //Calls second subroutine declared above
PRINT "\nExiting..."
```

Expected Output:

BLEVSPREAD is an extension subroutine.

BleVSpUartBridge

SUBROUTINE

This function creates a bridge between the managed Virtual Serial Port Service and the UART when both are open. Any data arriving from the VSP is automatically transferred to the UART for forward transmission. Any data arriving at the UART is sent over the air.

It should be called either when data arrives at either end or when either end indicates their transmit buffer is empty. The following events are examples: EVVSPRX, EVUARTRX, EVVSPTXEMPTY and EVUARTTXEMPTY.

Given that data can arrive over the UART a byte at a time, a latency timer specified by AT+CFG 116 command may be used to optimise the data transfer over the air. This tries to ensure that full packets are transmitted over the air. Therefore, if a single character arrives over UART, a latency timer is started. If it expires, that single character (or any more that arrive but less than 20) will be forced onwards when that timer expires.

BLEVSPUARTBRIDGE()

Exceptions

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments: None

Interactive Command: No

Related Commands: BLEVSPOPEN, BLEVSPCLOSE, BLEVSPINFO, BLEVSPWRITE, BLEVSPFLUSH

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

DIM conHndl

//==============================================================================
// Open VSp and start advertising
//==============================================================================
SUB OnStartup()
    DIM rc, hndl, tx$, scRpt$, addr$, adRpt$

    rc=BleVSpOpen(128,128,0,hndl)
    rc=BleScanRptInit(scRpt$)
    rc=BleAdvRptAddUuid128(scRpt$,hndl)
    rc=BleAdvRptsCommit(adRpt$,scRpt$)
    rc=BleAdvertStart(0,addr$,20,300000,0)
    rc=GpioBindEvent(1,16,1)      //Channel 1, bind to low transition on GPIO pin 16
    PRINT "\nDevice name is "; BleGetDeviceName$();"\n"

    tx$="\nSend me some text. \nPress button 0 to exit\n"
    rc = BleVSpWrite(tx$)
ENDSUB

//==============================================================================
// Close connections so that we can run another app without problems
//==============================================================================
SUB CloseConnections()
    DIM rc
```

```
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
    BleVspClose()
ENDSUB

//==============================================================================
// BLE event handler - connection handle is obtained here
//==============================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\nDisconnected from client"
        EXITFUNC 0
    ENDIF
ENDFUNC 1

//==============================================================================
//handler to service button 0 pressed
//==============================================================================
FUNCTION HndlrBtn0Pr() AS INTEGER
    //just exit and stop waiting for events
ENDFUNC 0


//==============================================================================
//handler to service an rx/tx event
//==============================================================================
FUNCTION HandlerBridge() AS INTEGER
  // transfer data between VSP and UART ring buffers
  BleVspUartBridge()
ENDFUNC 1


ONEVENT  EVVSPRX             CALL HandlerBridge
ONEVENT  EVUARTRX            CALL HandlerBridge
ONEVENT  EVVSPTXEMPTY        CALL HandlerBridge
ONEVENT  EVUARTTXEMPTY       CALL HandlerBridge
ONEVENT  EVBLEMSG           CALL HndlrBleMsg
ONEVENT  EVGPIOCHAN1        CALL HndlrBtn0Pr


OnStartup()

WAITEVENT

CloseConnections()  //Calls second subroutine declared above
PRINT "\nExiting..."
```

BLEVSPUARTBRIDGE is an extension subroutine.

BleVSpFlush

SUBROUTINE

This subroutine flushes either or both receive and transmit ring buffers.

This is useful when, for example, you have a character terminated messaging system and the peer sends a very long message, filling the input buffer. In that case, there is no more space for an incoming termination character. A flush of the receive buffer is the best approach to recover from that situation.

BLEVSPFLUSH(bitMask)

Exceptions:

- Local Stack Frame Underflow
- Local Stack Frame Overflow

Arguments:

bitMask	*byVal bitMask AS INTEGER* Bit 0 is set to flush the Rx buffer. Bit 1 is set to flush the Tx buffer. Set both bits to flush both buffers.

Interactive Command: No

Related Commands: BLEVSPOPEN, BLEVSPCLOSE, BLEVSPINFO, BLEVSPWRITE, BLEVSPREAD

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

DIM conHndl


//==============================================================================
// Open VSp and start advertising
//==============================================================================
SUB OnStartup()
    DIM rc, hndl, tx$, scRpt$, addr$, adRpt$ : adRpt$="" : addr$=""
    rc=BleVSpOpen(128,128,0,hndl)
    rc=BleScanRptInit(scRpt$)
    rc=BleAdvRptAddUuid128(scRpt$,hndl)
    rc=BleAdvRptsCommit(adRpt$,scRpt$)
    rc=BleAdvertStart(0,addr$,20,300000,0)
    rc=GpioBindEvent(1,16,1)      //Channel 1, bind to low transition on GPIO pin 16
    PRINT "\nDevice name is "; BleGetDeviceName$()

    tx$="\nSend me some text, I won't get it. \nTo exit the app press Button 0\n"
    rc = BleVSpWrite(tx$)
ENDSUB

//==============================================================================
// Close connections so that we can run another app without problems
//==============================================================================
SUB CloseConnections()
    DIM rc
    rc=BleDisconnect(conHndl)
    rc=BleAdvertStop()
    BleVspClose()
    BleVspFlush(2)       //Flush both buffers
ENDSUB
```

```
//==============================================================================
//  VSP Rx buffer event handler
//==============================================================================
FUNCTION HandlerVSpRx() AS INTEGER
    BleVspFlush(0)
    PRINT "\nRx buffer flushed"
ENDFUNC 1

//==============================================================================
//handler to service button 0 pressed
//==============================================================================
FUNCTION HndlrBtn0Pr() AS INTEGER
     //stop waiting for events and exit app
ENDFUNC 0


//==============================================================================
// BLE event handler
//==============================================================================
FUNCTION HndlrBleMsg(BYVAL nMsgId, BYVAL nCtx)
    conHndl=nCtx
    IF nMsgID==1 THEN
        PRINT "\nDisconnected from client"
        EXITFUNC 0
    ENDIF
ENDFUNC 1


ONEVENT  EVVSPRX      CALL HandlerVSpRx
ONEVENT  EVBLEMSG     CALL HndlrBleMsg
ONEVENT  EVGPIOCHAN1   CALL HndlrBtn0Pr


OnStartup()          //Calls first subroutine declared above

WAITEVENT

CloseConnections()  //Calls second subroutine declared above
PRINT "\nExiting..."
```

Expected Output:

BLEVSPFLUSH is an extension subroutine.

7. Other Extension Built-in Routines

This chapter describes non BLE-related extension routines that are not part of the core *smart* BASIC language.

System Configuration Routines

SystemStateSet

FUNCTION

This function is used to alter the power state of the module as per the input parameter.

SYSTEMSTATESET (nNewState)

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

Arguments:

nNewState — byVal *nNewState* AS INTEGER
New state of the module as follows:
0 System OFF (Deep Sleep Mode)

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.

Interactive Command: NO

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

//Put the module into deep sleep
PRINT "\n"; SystemStateSet(0)
```

SYSTEMSTATESET is an extension function.

Miscellaneous Routines

ReadPwrSupplyMv

FUNCTION

This function is used to read the power supply voltage and the value will be returned in millivolts.

READPWRSUPPLYMV ()

Returns: INTEGER, the power supply voltage in millivolts.

Arguments: None

Interactive Command: NO

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

//read and print the supply voltage
PRINT "\nSupply voltage is "; ReadPwrSupplyMv();"mV"
```

Expected Output:

```
Supply voltage is 3343mV
```

READPWRSUPPLYMV is an extension function.

SetPwrSupplyThreshMv

FUNCTION

This function sets a supply voltage threshold. If the supply voltage drops below this then the BLE_EVMSG event is thrown into the run time engine with a MSG ID of BLE_EVBLEMSGID_POWER_FAILURE_WARNING (19) and the context data will be the current voltage in millivolts.

Events & Messages

MsgId	Description
19	The supply voltage has dropped below the value specified as the argument to this function in the most recent call. The context data is the current reading of the supply voltage in millivolts

SETPWRSUPPLYTHRESHMV(nThresh)

Returns: INTEGER, 0 if the threshold is successfully set, 0x6605 if the value cannot be implemented.

Arguments: None

nThreshMv byVal *nThresMv* AS INTEGER
The BLE_EVMSG event is thrown to the engine if the supply voltage drops below this value. Valid values are 2100, 2300, 2500 and 2700.

Interactive Command: NO

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

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:

```

```

SETPWRSUPPLYTHRESHMV is an extension function.

8. Events & Messages

smart BASIC is designed to be event driven, which makes it suitable for embedded platforms where it is normal to wait for something to happen and then respond.

To ensure that access to variables and resources ends up in race conditions, the event handling is done synchronously, meaning the *smart* BASIC runtime engine has to process a WAITEVENT statement for any events or messages to be processed. This guarantees that *smart* BASIC will never need the complexity of locking variables and objects.

There are many subsystems which generate events and messages as follows:-

- Timer events, which generate timer expiry events and are described here.
- Messages thrown from within the user's BASIC application as described here.
- Events related to the UART interface as described here.
- GPIO input level change events as described here.
- BLE events and messages as described here.
- Generic Characteristics events and messages as described here.

9. Module Configuration

There are many features of the module that cannot be modified programmatically which relate to interactive mode operation or alter the behaviour of the smartBASIC runtime engine. These configuration objects are stored in non-volatile flash and are retained until the flash file system is erased via AT&F* or AT&F 1.

To write to these objects, which are identified by a positive integer number, the module must be in interactive mode and the command AT+CFG must be used which is described in detail here.

To read current values of these objects use the command AT+CFG, described here.

Predefined configuration objects are as listed under details of the AT+CFG command.

10. Miscellaneous

Bluetooth Result Codes

There are some operations and events that provide a single byte Bluetooth HCI result code, e.g. the EVDISCON message. The meaning of the result code is as per the list reproduced from the Bluetooth Specifications below. No guarantee is supplied as to its accuracy. Consult the specification for more.

Result codes in `grey` are not relevant to Bluetooth Low Energy operation and are unlikely to appear.

BLE_HCI_STATUS_CODE_SUCCESS	**0x00**
BLE_HCI_STATUS_CODE_UNKNOWN_BTLE_COMMAND	**0x01**
BLE_HCI_STATUS_CODE_UNKNOWN_CONNECTION_IDENTIFIER	**0x02**
BLE_HCI_HARDWARE_FAILURE	0x03
BLE_HCI_PAGE_TIMEOUT	0x04
BLE_HCI_AUTHENTICATION_FAILURE	**0x05**
BLE_HCI_STATUS_CODE_PIN_OR_KEY_MISSING	**0x06**
BLE_HCI_MEMORY_CAPACITY_EXCEEDED	**0x07**
BLE_HCI_CONNECTION_TIMEOUT	**0x08**
BLE_HCI_CONNECTION_LIMIT_EXCEEDED	0x09
BLE_HCI_SYNC_CONN_LIMI_TO_A_DEVICE_EXCEEDED	0x0A
BLE_HCI_ACL_COONECTION_ALREADY_EXISTS	0x0B
BLE_HCI_STATUS_CODE_COMMAND_DISALLOWED	**0x0C**
BLE_HCI_CONN_REJECTED_DUE_TO_LIMITED_RESOURCES	0x0D
BLE_HCI_CONN_REJECTED_DUE_TO_SECURITY_REASONS	0x0E
BLE_HCI_BLE_HCI_CONN_REJECTED_DUE_TO_BD_ADDR	0x0F
BLE_HCI_CONN_ACCEPT_TIMEOUT_EXCEEDED	0x10
BLE_HCI_UNSUPPORTED_FEATURE_ONPARM_VALUE	0x11
BLE_HCI_STATUS_CODE_INVALID_BTLE_COMMAND_PARAMETERS	**0x12**
BLE_HCI_REMOTE_USER_TERMINATED_CONNECTION	**0x13**
BLE_HCI_REMOTE_DEV_TERMINATION_DUE_TO_LOW_RESOURCES	**0x14**
BLE_HCI_REMOTE_DEV_TERMINATION_DUE_TO_POWER_OFF	**0x15**
BLE_HCI_LOCAL_HOST_TERMINATED_CONNECTION	**0x16**
BLE_HCI_REPEATED_ATTEMPTS	0x17
BLE_HCI_PAIRING_NOTALLOWED	0x18
BLE_HCI_LMP_PDU	0x19
BLE_HCI_UNSUPPORTED_REMOTE_FEATURE	**0x1A**
BLE_HCI_SCO_OFFSET_REJECTED	0x1B
BLE_HCI_SCO_INTERVAL_REJECTED	0x1C
BLE_HCI_SCO_AIR_MODE_REJECTED	0x1D
BLE_HCI_STATUS_CODE_INVALID_LMP_PARAMETERS	**0x1E**
BLE_HCI_STATUS_CODE_UNSPECIFIED_ERROR	**0x1F**
BLE_HCI_UNSUPPORTED_LMP_PARM_VALUE	0x20
BLE_HCI_ROLE_CHANGE_NOT_ALLOWED	0x21
BLE_HCI_STATUS_CODE_LMP_RESPONSE_TIMEOUT	**0x22**
BLE_HCI_LMP_ERROR_TRANSACTION_COLLISION	0x23
BLE_HCI_STATUS_CODE_LMP_PDU_NOT_ALLOWED	**0x24**
BLE_HCI_ENCRYPTION_MODE_NOT_ALLOWED	0x25
BLE_HCI_LINK_KEY_CAN_NOT_BE_CHANGED	0x26
BLE_HCI_REQUESTED_QOS_NOT_SUPPORTED	0x27
BLE_HCI_INSTANT_PASSED	**0x28**
BLE_HCI_PAIRING_WITH_UNIT_KEY_UNSUPPORTED	**0x29**
BLE_HCI_DIFFERENT_TRANSACTION_COLLISION	**0x2A**

BLE_HCI_QOS_UNACCEPTABLE_PARAMETER	0x2C
BLE_HCI_QOS_REJECTED	0x2D
BLE_HCI_CHANNEL_CLASSIFICATION_UNSUPPORTED	0x2E
BLE_HCI_INSUFFICIENT_SECURITY	0x2F
BLE_HCI_PARAMETER_OUT_OF_MANDATORY_RANGE	0x30
BLE_HCI_ROLE_SWITCH_PENDING	0x32
BLE_HCI_RESERVED_SLOT_VIOLATION	0x34
BLE_HCI_ROLE_SWITCH_FAILED	0x35
BLE_HCI_EXTENDED_INQUIRY_RESP_TOO_LARGE	0x36
BLE_HCI_SSP_NOT_SUPPORTED_BY_HOST	0x37
BLE_HCI_HOST_BUSY_PAIRING	0x38
BLE_HCI_CONN_REJ_DUETO_NO_SUITABLE_CHN_FOUND	0x39
BLE_HCI_CONTROLLER_BUSY	**0x3A**
BLE_HCI_CONN_INTERVAL_UNACCEPTABLE	**0x3B**
BLE_HCI_DIRECTED_ADVERTISER_TIMEOUT	**0x3C**
BLE_HCI_CONN_TERMINATED_DUE_TO_MIC_FAILURE	**0x3D**
BLE_HCI_CONN_FAILED_TO_BE_ESTABLISHED	**0x3E**

11. Acknowledgements

The following are required acknowledgements to address our use of open source code on the BL600 to implement AES encryption. Laird's implementation includes the following files: aes.c and aes.h.

Copyright (c) 1998-2008, Brian Gladman, Worcester, UK. All rights reserved.

License Terms

The redistribution and use of this software (with or without changes) is allowed without the payment of fees or royalties providing the following:

- Source code distributions include the above copyright notice, this list of conditions and the following disclaimer;
- Binary distributions include the above copyright notice, this list of conditions and the following disclaimer in their documentation;
- The name of the copyright holder is not used to endorse products built using this software without specific written permission.

Disclaimer

This software is provided 'as is' with no explicit or implied warranties in respect of its properties, including, but not limited to, correctness and/or fitness for purpose.

Issue 09/09/2006

This is an AES implementation that uses only 8-bit byte operations on the cipher state (there are options to use 32-bit types if available).

The combination of mix columns and byte substitution used here is based on that developed by Karl Malbrain. His contribution is acknowledged.

INDEX